

The Alliance of Laboratories in Europe for Education, Research and Technology

ALERT Doctoral School 2023

Machine Learning (ML) in Geomechanics

Editors:

Ioannis Stefanou Félix Darve

Preface

From 28th September to 30th September 2023, the ALERT Doctoral School 2023 is scheduled to be held in Aussois and will be dedicated to "Machine Learning in Geomechanics". The School has been organized by Ioannis Stefanou (EC-Nantes) and Félix Darve (Université Grenoble-Alpes). I sincerely thank the organizers and all the contributors to this book for their effort!

Constitutive modelling, numerical modelling of geotechnical structures, and image correlation are central activities of ALERT researchers in Geomechanics. These activities offer several opportunities for Machine Learning to improve their applications. For instance, when constitutive models are developed to include more microstructural features, their complexity and computational expense also increase. Employing ML is undoubtedly a promising approach to speed up the process. I strongly believe that Machine Learning will become a common tool for development in geomechanics in the near future, thus making this school beneficial to the ALERT community.

The three-day school will explain what machine learning entails, with a focus on the most essential methods. These methods will be demonstrated in solving geomechanics problems, with specific sessions devoted to their applications, aiming to exhibit their efficiency and limitations.

As usual, the pdf file of the book can be downloaded for free from the website of ALERT Geomaterials (http://alertgeomaterials.eu/publications/) after the school. On behalf of the ALERT Board of Directors I wish all participants a successful ALERT Doctoral School 2023!

Frédéric Collin Director of ALERT Geomaterials University of Liege

Contents

Foreword
Stefanou & Darve1
Overview of Machine Learning
Stefanou
Introduction to regression methods
Masi
Unsupervised Learning: Basic Concepts and Application to Particle Dynamics
Jakse
Classification Techniques in Machine Learning
Jakse 93
Data Drivan Madaling in Gaamaahaniaa
Karapiperis 115
New Freelideen meeting looming for some sharing
Sun 131
Sun
Artificial Neural Networks: layer architectures, optimizers and automatic differentia-
Gatti 150
Gau
Artificial Neural Networks: advanced topics
Gatti
Physics-informed and thermodynamics-based neural networks
Ması & Steranou
Introduction to Reinforcement Learning with Applications in Geomechanics
Stathas, Gutiérrez-Oribio & Stefanou

Machine Learning in Geomechanics: Foreword

When discussing about artificial intelligence (AI), some basic questions are immediately emerging such as: what AI is? how does it work? Behind this popular term, there is a collection of methods of Applied Mathematics that allow the computer to learn and identify patterns in data. This collection of methods is called Machine Learning (ML) and it is the target of ALERT's 2023 Doctoral School.

In combination with the tremendous increase of the computational power, Machine Learning has led to incredible achievements in many disciplines of science and technology. These achievements were that striking that some researchers believe that ML could become a turning point for humanity, as the discovery of fire was for our far ancestors!

Until the sixties, the scientific development has been characterized by the so-called "linear physics" and by modeling represented by analytical equations solved explicitly by the available mathematical tools giving rise to analytical solutions. The field of problems which can be solved in this way is, of course, precious, but very limited.

Then the numerical revolution, based on powerful numerical methods and computers, allowed to solve numerically a great variety of problems, which can be described by a known system of equations. Many limits of this methodology are known due to the abundance of non-linear processes in nature, chaos and complexity. In any case, Numerical Analysis, another branch of Applied Mathematics, has immensely enlarged the class of problems that can be solved today.

However, the numerical solutions of these sets of non-linear equations can be computationally very intensive or even impossible. Moreover, many problems in engineering are hard to describe by a set of equations. Machine Learning tools provide promising methods for addressing both those problems.

Another aspect of ML algorithms is their ability to solve very complex problems in a "creative" manner. One characteristic example of creativity was shown when the machine won the world champion of the Go game, which was invented a long time ago in China. Differently from chess, in which the computer can predict the game evolution several moves in advance, in Go the number of possible moves is extremely large (higher than the number of atoms in the known universe). Therefore, it is necessary to follow creative strategies. Indeed, the machine, "AlphaGo", has shown that it is able to carry out novel strategies that surprised even the best human players in the world.

All these methods of Machine Learning, give new powerful tools to scientists and engineers and open new perspectives in Geomechanics. The target of this volume is to demystify Machine Learning, to present its main methods and to show some examples of applications in (geo-)mechanics. Most of the chapters of the volume were drafted having in mind to provide a pedagogical introduction to the most important methods of ML and to uncover the fundamental notions behind them.

The volume is organized in ten chapters:

The first chapter, "Overview of Machine Learning", is the introductory chapter of this volume. In this chapter we explain how the machine can learn, we show a classification of the main methods in Machine Learning, we outline some applications of ML in Geomechanics and we highlight its limitations.

The second chapter, "Introduction to regression methods", focuses on regression, which is one of the fundamental pillars of supervised Machine Learning. In this chapter we introduce the essential concepts in regression analysis and methods, by providing hands-on, practical examples.

The target of the third chapter, "Unsupervised Learning: Basic Concepts and Application to Particle Dynamics", is twofold. The first part of this chapter is devoted to the description of the basic concepts of the most popular techniques of unsupervised learning. The second part illustrates an application of unsupervised learning to the discovery of patterns in particles dynamics.

The fourth chapter, "Classification Techniques in Machine Learning", aims at describing what the problem of classification in Machine Learning is and illustrates some of the methods used for solving it, without resorting to Artificial Neural Networks. Hands-on examples are given and Active Learning is discussed.

Chapter five, "Data-Driven Modeling in Geomechanics", presents the theoretical framework of the so called data-driven computational mechanics. Furthermore, it shows some of its applications for the solution of problems involving Cauchy and Cosserat continua with elastic and inelastic materials, which, naturally, represent common descriptions of geomaterials.

The sixth chapter, "Non-Euclidean machine learning for geomechanics", is intended to provide a concise review on how to train, verify and validate constitutive models enhanced by graph-theoretic data. The use of graph convolutional neural networks for constitutive modeling, material design, and the solution of inverse problems is discussed.

The next two chapters, "Artificial Neural Networks: layer architectures, optimizers and automatic differentiation" and "Artificial Neural Networks: advanced topics" provide a comprehensive introduction to Artificial Neural Networks (ANN). Several hands-on examples are given to help the reader grasp the main ideas and tools of the most important ANN architectures. More advanced topics are also discussed and the connection of ANN with information theory is made.

Chapter nine "Physics-informed and thermodynamics-based neural networks" shows how to inject prior knowledge into deep learning algorithms. Using various examples, we present Physics-Informed Neural Networks for the discovery of partial

differential equations and Thermodynamics-based Artificial Neural Networks for the discovery of constitutive models of complex, inelastic materials.

The last chapter, "Introduction to Reinforcement Learning with Applications in Geomechanics", presents the basic concepts of Reinforcement Learning, which enables the development of software agents that are capable of making optimal decisions in dynamic and uncertain environments. The chapter closes with two applications of Reinforcement Learning in Geomechanics.

We deeply thank all the authors of this volume for their comprehensive contributions and their effort to present complex notions in a pedagogical manner. We hope that the chapters provide a valuable introduction to Machine Learning in Geomechanics.

> Ioannis Stefanou Félix Darve

Overview of Machine Learning

Ioannis Stefanou

Nantes Université, Ecole Centrale Nantes, CNRS, GeM, UMR 6183, F-44000, Nantes, France

1 What machine learning is?

One of the best methods for making a puppy to learn its name is to play the name game:

- 1. Take your puppy to a quiet place and gain its attention by calling its name in a happy and reassuring tone (e.g. LoupI look at me or LoupI come here!).
- 2. As soon as you've gained its attention, reward it with a treat to encourage its good behavior.
- 3. Repeat this process several times until your puppy has learned to respond to its name.

In the above iterative process the puppy learns from experience that reacting when you call its name is a good thing. By repeating this process, the puppy recognizes the pattern between similar, but not identical vocal signals of its name (training data) and associates them with a specific behavior (task).

Many cognitive processes of humans involve a similar way for associating patterns with specific reactions, see for instance how we learn to drive a car, speak a foreign language or even understanding how a geomaterial behaves!

Machine learning (ML) is an anthropomorphic extension of the above process of "learning" to a computer program, i.e. to the machine. By learning, the machine will progressively improve its performance in executing a specific task. According to Tom Mitchel [Mit97], ML can be defined as follows:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

The whole set of examples that the computer will use during learning is the *training set*, while each training example is called a training instance or sample. The whole

process of learning is called *training*. The training set is usually distinguished into *training data*, *validation data* and *test data*. More details about those sets and their importance will be given in section 5 and in Chapters 2 and 7.

A popular example showing the power of ML was *AlphaGo*, a ML program, which won the human world champion of Go (see also [SHM⁺16] and Chapter 10 for *Reinforcement Learning*). Go is a strategy board game for two players in which the aim is to cover more territory than the opponent (see Figure 1). Despite its relatively simple rules, Go is extremely complex. The *state-space complexity* of Go, i.e. the number of legal game positions reachable from the initial position of the game [All94], is of the order of 10^{170} [TF07], which is much higher than that of chess, which is of the order of 10^{44} [Sha50], and certainly higher than the number of atoms in the universe. This makes Go a great challenge for ML algorithms.

Applying the above mentioned definition in a ML program for Go, the performance, P, will be measured by the ability of the machine to win its opponent by gaining experience, E, by executing multiple times the task, T, which is playing the game with itself. Experience can be also obtained by processing existing games of other (human) players.



Figure 1: Eighteen times Go world champion Lee Sedoll playing with the machine (courtesy: Lee Jin-man / AP, The New Yorker,

https://www.newyorker.com/tech/annals-of-technology/ alphago-lee-sedol-and-the-reassuring-future-of-humans-and-machines).

Another recent popular example of machine learning is *ChatGPT* chatbot developed by OpenAI. ChatGPT was released in November 2022 and provoked a massive public reaction. Let us see what it answers when we ask it to describe itself and what it knows about ALERT Geomaterials.

Ioannis: Describe your self.

ChatGPT: I am ChatGPT, an AI language model developed by OpenAI. I have been trained on a vast amount of text data from the internet, books, and other sources. My purpose is to assist users in generating human-like text responses and providing information on a wide range of topics.

I don't have a physical form or personal experiences like humans do, but I can understand and generate text in multiple languages. I strive to provide helpful and accurate responses to the best of my abilities, drawing upon the knowledge I have acquired during my training.

I can engage in conversations, answer questions, offer explanations, and assist with various tasks. However, please keep in mind that while I strive for accuracy, I may occasionally generate incorrect or incomplete information. It's always a good idea to verify important details from reliable sources.

Overall, I am here to assist and provide information to the best of my abilities. Feel free to ask me anything you'd like to know or engage in a conversation on any topic of your interest.

Ioannis: How do you work?

ChatGPT: *As an AI language model, my functioning can be divided into two main phases: training and inference.*

Training: During the training phase, large amounts of text data are used to train the model. In the case of GPT models like ChatGPT, this training involves a process called unsupervised learning. The model learns to predict what comes next in a given sequence of text, based on the patterns and relationships it discovers in the training data. By doing this over a vast dataset, the model gradually learns grammar, facts, reasoning abilities, and the nuances of human language.

Inference: Once the training is complete, the model moves into the inference phase, where it can generate responses and interact with users. Here's a simplified overview of how it works during inference:

Input: When you provide a prompt or a message, the model receives it as input. The input can be a question, a statement, or any text you want to generate a response to.

Tokenization: The model breaks down the input text into smaller units called tokens. These tokens can represent words, subwords, or characters. Tokenization helps the model understand the structure and meaning of the input.

Processing: The tokenized input is then passed through several layers of neural networks. These layers are designed to understand the relationships between the tokens and capture the context and meaning of the text.

Generation: Once the input is processed, the model generates a response. It predicts the most likely next tokens based on the context and the patterns it has learned during training. The generated tokens form the model's response.

Iteration: The process of tokenization, processing, and generation can happen in an iterative manner. The generated response can be further used as input, creating a back-and-forth conversation or a continuous interaction.

It's important to note that while ChatGPT can generate impressive responses, it may sometimes produce incorrect or nonsensical answers. It doesn't have real-world experiences or access to current information beyond its knowledge cutoff date. It's always a good idea to fact-check and verify important information from reliable sources.

Ioannis: What is ALERT Geomaterials in summary?

ChatGPT: In summary, ALERT Geomaterials is an international scientific association dedicated to the study of the mechanics and behavior of geomaterials, including soil, rock, and other geological materials. It brings together researchers and professionals in the field of geomechanics and geotechnical engineering to promote collaboration, knowledge exchange, and scientific advancements. ALERT Geomaterials organizes workshops, conferences, and training courses, providing a platform for researchers to share their findings and discuss topics related to soil and rock mechanics, slope stability, geotechnical engineering, experimental testing, and numerical modeling. The association also publishes a journal focused on energy geotechnics and geoenvironmental engineering. Overall, ALERT Geomaterials plays a crucial role in fostering research and innovation in the field of geomaterials.

Impressive! At this point it becomes quite tempting to let ChatGPT to finish this chapter or even the whole book! However, I assure you, we took the necessary time to entirely draft this volume word by word. Read carefully also ChatGPT's answers. Are all of them correct?

It is straightforward to see how the above definition for ML applies to ChatGPT and to many other applications of ML, such as in speech, handwriting, face and object recognition, voice-to-text and vice versa, translation, text auto-correction and autocompletion, spam filtering, computer games, self-driving cars, medicine, forecasting, banking, security, marketing, control problems, engineering and, of course, to (geo-)mechanics.

ML is an evolving field of knowledge and involves a plethora of methods and combinations of those. In the next section we will try to categorize and classify them in groups.

Having described and defined what ML is, it is natural to ask what *Artificial Intelligence (AI)* is? ML and AI are closely related and the latter is considered to include the former. The exact definition of AI seems to be a bit foggy for the time being and depends on how we define the terms "artificial" and "intelligence". To the author's

opinion, it is easier to describe the characteristics of AI, rather than give a unique and exact definition of the term. Another example, in a totally different domain, that definitions are hard to make is what is justice. It seems easier and more important to describe the characteristics of justice (e.g. equality for all) rather than give a precise definition of the term.

2 Classification of ML methods

There are numerous ML methods in the literature. Therefore, it is useful to classify them into different categories. Here we follow the classification of Géron [Gé19], who categorizes ML to: *supervised learning* vs *unsupervised learning*, *batch learning* vs *online learning* and *instance-based learning* vs *model-based learning* methods. Of course, this is a rough classification and one method can combine different categories, as shown in Figure 2.



Figure 2: Classification of ML methods. Method 1 is a supervised, batch, instancebased method, while Method 2 has components belonging to different categories.

2.1 Supervised vs unsupervised ML

In *supervised learning* the training set includes the desired solutions/predictions, which are called *labels*.

For instance, imagine that your training set is hundreds of images with each one containing either a dog or a cat and that the training task is to distinguish the photos of dogs from those of cats. This is called a *classification* problem (see Chapter 4). If we give to the computer the information which of the photos show dogs and which cats,

or in other words, if we label the training set then the learning is called supervised. If the training is successful, then the computer will have learned to identify cats from dogs even in a new set of photos that it haven't processed during its training.

On the contrary, if the training set is unlabeled, then the learning is called *unsupervised learning*. In this case the computer will eventually understand the pattern of the two different animals shown in the photos and it will be able to distinguish dogs from cats in an unsupervised manner (see also *clustering* in Chapter 3). Of course, the machine won't have learned to call a cat, cat and a dog, dog, because we haven't given this additional information, but it will have identified their differences and separate the data into two different classes.

Another machine learning problem is regression (see Chapters 2, 7, 8 and 9). In regression the training set contains one or several numerical inputs, also called *features*, and the task is to predict one or several numerical outputs, also called *predictors* that depend on the inputs. Consider as an example the prediction of the stress response of a geomaterial, which as we know depends on several input parameters, such as the applied strain, available information about the evolution of its microstructure (e.g. the position and the velocity of the grains of a sand obtained by a Discrete Element Method (DEM) analysis), history and/or other features. As the training set contains both the output and the input, the training for predicting the stresses based on the above mentioned features is supervised. However, the identification/extraction by the machine of a representation of the most important features of the microstructure that are related to the prediction of the stress response is unsupervised. Examples of unsupervised methods in ML are feature extraction, anomaly detection, dimensionality *reduction*, in which the aim is to reduce the the size of the training set without loosing important information with respect to a specific task or measure and data compression (among others).

In Table 1, we provide a list of important supervised and unsupervised methods in ML. However, not all ML methods can be categorized to supervised and unsupervised. A notable example is *Reinforcement Learning (RL)*, which does not require labeled data or a even training set. For more details on RL we refer to Chapter 10. Finally, when labels are not available for all the samples in the training set, we refer to *semisupervised and unsupervised and unsupervised and unsupervised and unsupervised semisupervised methods* are a combination of supervised and unsupervised algorithms. An example of semisupervised learning is *Active Learning* (see Chapter 4 for more details).

2.2 Batch vs online ML

Another manner to classify ML algorithms is based on whether they can improve their predictions by providing them with new data that may become available after the first training.

In *batch learning* the machine has to be trained over a fixed training data set, without being able to add more data to the training set. Therefore, batch ML methods cannot

Stefanou 11

ML Method	Supervised	Unsupervised	Online	Presented in
				this volume
Linear regression	\checkmark			\checkmark
Logistic regression	✓			✓
Polynomial regression	✓			✓
Lasso, Ridge	✓			√
k-Nearest neighbors	✓			√
Support vector				
machines (SVM)	\checkmark			\checkmark
Decision trees	√			
Random forests	√			
Artificial Neural				
Networks (ANN)	\checkmark		\checkmark	\checkmark
Autoencoders		√	√	✓
Clustering		√		✓
Principal Component				
Analysis (PCA)		\checkmark	\checkmark^*	
Locally Linear				
Embedding (LLE)		\checkmark		
Reinforcement				
Learning (RL)			✓	✓

Table 1: Classification of some important methods in ML. The asterisk denotes that not all variations of the method fall into this category.

improve their performance in a specific task with providing them with more data after the end of the training.

On the contrary, ML methods that support *online learning* allow to modify and increase the initial training set. Consequently, they are more flexible and suitable for large training sets, as they can be partially loaded into the memory of the computer and used whenever needed. Old data, over which the computer has been already trained, can be also erased to save space. Online learning is also very convenient when a constant flow of information exists, contrary to batch learning, in which the machine must be retrained over the whole data set. In other words, in batch learning, when new data become available the training set has to be updated and the training has to be repeated from scratch. This can be fine for some applications, but in many others it could have a very high computational and data storage cost.

Not all ML algorithms support online learning, see Table 1.

2.3 Instance-based vs model-based ML

One more way to categorize ML methods is based on the way data is learned in order to make predictions for data outside the training set. Two ML categories can be distinguished, *instance-based* (also known as *memory-based* or *lazy*) learning and *model-based* learning (also known as *physics-based* in some applications).

In instance-based learning, training data is simply interpolated in a high (usually) dimensional space. Then, new predictions are made based on how close or how similar new data are to those used for training. Similarity is measured on the basis of a distance measure, depending on the data and the problem at hand. In this sense data is "memorized" by the machine, thus the term memory-based. The quality of the predictions of data outside the training set is determined by how well new data are represented by the data of the training set. Instance-based approaches are straightforward to apply in any data set and they don't require any particular knowledge about the structure of the data or other characteristics that they might have. The hope of the user of instance-based methods is the machine to eventually identify by itself the hidden patterns in the data and give correct predictions even for data outside the training set. Examples of instance-based methods are *Artificial Neural Networks*, *Decision trees*, *Random forests*, *k-Nearest neighbors* and many *clustering* techniques, *Locally Linear Embedding* (*LLE*) and *Principal Component Analysis* (*PCA*).

Despite the versatility and the many advantages of instance-based ML methods, they have an important drawback. In physics and engineering, we know that data have to respect at least some fundamental principles, like for instance the conservation of mass, of the energy and of the linear and angular momentum. Therefore, instance-/memory-based predictions that do not respect these conservation laws are unacceptable and can be even dangerous for applications. Data in this case has to be processed and learned by the machine under a model that includes the laws of physics.

Model-based machine learning adopts a model with some model parameters, which

are optimized during training in order to optimally represent the data of the training set. Then the model is used to make predictions for unseen input data. Of course, if the adopted model is poor, the predictions will be poor as well.

Both instance-based and model-based approaches can introduce bias. The former because of limited data over which they were trained and the latter because of the model chosen. To fix the ideas, an example of instance-based and model-based ML is given in Figure 3. The training set contains the shear stress at failure (*predictor*) for a given normal stress (*feature*) of a series of experimental tests of a frictional interface. An instance-based method could give very poor predictions for unseen data, while a model-based prediction will be as good as the model is for describing the data. In this simple example, a Coulomb model was adopted for the model-based approach.



Figure 3: Instance-based vs model-based ML for predicting failure (slip) of a frictional interface based on experimental data (fictitious).

3 ML and Geomechanics

The applications of ML in science are nowadays numerous and increasing. The same holds for applications of ML in geomechanics. It is out of the scope of this chapter to provide a comprehensive literature review, but it is worth mentioning some research directions in *constitutive modeling*, *geotechnics*, *geophysics* and *image corelation* that can be a starting point for the interested reader.

Geomaterials are among the most complex materials to study and model. The main

reason is that their mechanical response is governed by multiphysics couplings at multiple spatial and temporal scales, which result in a macroscopic strongly non-linear and dissipative response.

Empirical constitutive models are often used to describe the mechanical behavior of geomaterials. Classification and regression methods can be used for choosing the appropriate constitutive model that fits the best the experimental data (see [MPRP22, MH19, GBL+21, SKOM23, PEW20, ZYJ21] and references therein). Another challenging task is to capture the macroscopic behavior of geomaterials based on the the behavior of their microstructure and its evolution due to loading. Multiscale approaches can be employed for this purpose, requiring the solution of a boundary value problem (BVP) of an elementary cell of the microstructure at each point and time increment of the macroscopic analysis. Then, based on homogenization, the intrinsic constitutive behavior of the microstructure can be upscaled to the macroscopic level. For this purpose, mixed numerical schemes, such as the FE^2 method [Fey03, LVRSHOO19, EBC⁺16] and FEM×DEM [NMCDD⁺11, NCCD14] are often employed, depending on the nature of the microstructure. However, the computational cost of these methods is extremely high for real-scale applications, if not prohibitive. ML and ANN are one of the most promising ways to speed-up this multiscale process.

Recent works have shown that ANN can successfully encapsulate several aspects of the constitutive behavior of the underlying microstructure and provide the necessary information to the macroscopic scale with reduced calculation cost (see [GGW91, LS03, MBC+19, LW19, HXFD20, VS21, ZHX21, WSD19, RKVDM21, BDMJ22, WXW23, SBV⁺21, ZZJ⁺23, PABT⁺21] to mention few). Going a step further, ANN can be designed in such a way to respect, by construction, the laws of physics [KKL $^+21$, RPK19], symmetries [HWS20] and thermodynamics [MSVMB21, MS22, MS23, HBG⁺21] (see also Chapters 6 and 9). Once trained, these approaches can tremendously speedup the solution of difficult multiscale problems, they can guarantee the respect of the thermodynamic restrictions in their predictions and enable the extraction of the hidden state variables of the material. The latter can, in turn, shed light on the importance and the role of specific micromechanisms to the overall macroscopic behavior of complex (geo-)materials. The above collection of ML approaches is enriched by the so-called "data-driven." methods, which present an alternative formulation, whereby optimal material states are sought within a dataset that most closely satisfy momentum and energy conservation principles [KSOA20, KO16, KO18, KOA21] (see also Chapter 5).

Moving to applications of ML in geotechnics, according to the recent review of Baghbani et al. [BCCR22] (see also [ZLL+21]), more than 1200 articles can be found in the literature starting from the early 90's. According to the same source, a net burst of production of scientific articles is observed after 2017. Notice, that more than half of these works use ANN. Several areas of geotechnical engineering are covered (see Figure 4), such as frozen soils and soil thermal properties, rock mechanics, subgrade soils and pavements, landslides (see [TCL⁺22] for a recent review), liquefaction, slope sta-

bility, shallow foundations, piles, tunneling and tunnel boring machines, dams, and unsaturated soils, among others. In 2018, the increasing interest of the geotechnical community in ML led [ZL22] to the creation of a new technical committee (TC) in the International Society for Soil Mechanics and Geotechnical Engineering (ISSMGE), entitled as TC309 "Machine Learning and Big Data" (https://www.issmge.org/committees/technical-committees/impact-on-society/machine-learning, see also TC304).



ANN FIS ANFIS SVM LSTM ResNet CNN GAN Other Al methods

Figure 4: Number of published articles using ML for geotechnical applications (slightly modified from [BCCR22]).

ML has also promising applications in geophysics. For instance, it finds applications in geophysical exploration, reservoir engineering and drilling (see [SYR⁺21] and references therein). Moreover, it was used for creating synthetic accelerograms based on numerical simulations or databases of real earthquake signals [GC20, LGBC22]. ML was used as well in an attempt for earthquake prediction —the holy grail in seismology— showing that earthquake-like events in the laboratory could be predicted by identification by the machine of seismic precursor patterns [JRLPN⁺21, BSMM21, LTG⁺22, RY23]. RL was also used for controlling earthquake-like events [PS21] (see Chapter 10) for which more exact mathematical theories show that are controllable [Ste19, GOTSP].

Given the striking advances of ML in self-driving cars, object and face recognition [Bal15], it is natural to expect ML methods to also find numerous applications in image processing in experimental geomechanics and geotechnics. *Convolutional Neural Networks (CNNs)* (see Chapters 7 and 8 for more details) is the basic ingredient of most machine learning techniques used in image and video processing, without forgetting more traditional compression techniques that are also considered as ML (e.g. PCA). Boukhtache et al. [BAB⁺21] presents a review of Digital Image Correlation (DIC) with deep learning. We also refer to [CZX⁺23, DXD⁺23, BAB⁺23] for some recent developments. Accuracy is one of the main issues for those methods in order to

outperform the current state of the art (see ALERT Doctoral School 2022 [AMHD22]). Focusing on granular materials, Stefano Buitrón et al. [CJM⁺23] propose a CNN to automatically distinguish properly segmented digital grains with up to 90% of accuracy, while Cheng et al. [CWX23] present a machine learning-based strategy to estimate the contact force chains of uniformly sized spherical granular materials using particle kinematics and inter-particle contact evolution data measured by X-ray micro-tomography.

4 Libraries for ML

Today, many libraries exist for machine learning (see https://en.wikipedia.org/ wiki/Machine_learning#Software for an updated list). Most of the existing libraries provide a Python interface.

Some general purpose libraries for ML that are extensively used in this doctoral school are Numpy [HMW⁺20], Pandas [McK10], SciPy [VGO⁺20] and Scikit-learn [PVG⁺11]. For *Artificial Neural Networks*, TensorFlow [MAP⁺15] and PyTorch [PGM⁺19] are equally popular today (see Figure 5). Both offer parallelization and GPU support for training large ANN and handling large collections of data. In this doctoral school we use PyTorch (see Chapter 7 for more details). As always, the best library is the library that we know the best, provided that it is open-source and it allows us do what we want!



Figure 5: Normalized Google searches over time for TensorFlow and PyTorch ANN libraries.

5 Bias in ML and limitations

The success of ML algorithms in performing complicated and complex tasks makes them convenient tools for many applications. However, we have to be aware of their limitations. In other words, we have to use the right screwdriver for the right screw! In section 2.3 (see Figure 3), we showed how the choice of the ML method and the quality of the data can lead to biased predictions, even in a very simple problem.

Applications of ML algorithms have demonstrated gender [Kel19], racial [DF18, Per21], hiring [VVNR21] and other biases [Var23]. ML methods are algorithms which should be used with care and knowledge of the the underlying limitations. This is not different from the application of other methods. For instance, we know how wrong Finite Element predictions can be when the appropriate finite elements are not used, when convergence analyses are not performed or when we want to model a softening material without regularizing the underlying mathematical problem. The inappropriate use of specific methods has traditionally led to spectacular failures with uncountable casualties and economic loss. ML methods will not be an exception, unless we understand them better and use them with caution.

As far as it concerns ML and geomechanics, luckily, we have at our disposal established and undeniable principles that have to be respected in any application. Conservation principles, the laws of thermodynamics and other physics at various spatiotemporal scales should be incorporated into the ML algorithms in order to assure adequate and safe predictions. Physics-based approaches in ML (see Chapters 5, 6 and 9) gain more and more attention from the scientific community and can become the natural environment for marrying the established know-how of decades of research in geomechanics with ML. This could give fresh ideas and an opportunity to push further the current state of the art in our fields. The incorporation of physics in ML could eventually inspire new ML methods in other domains too, which today suffer from inevitable bias (see social sciences where data are always limited).

Another limitation of most of the available ML methods today is their greediness for data. We do not need thousand of photos to teach to a child what a car is! With ML though we need tons of data! In many applications there is abundance of data (see for instance the data that are produced, but not saved, at each increment at each Gauss point during a non-linear Finite Element analysis in a geomechanics problem). In some other applications though, data may not be enough (see for example X-ray scans of thousand of specimens [TLA⁺20]).

Noise in the data, overfitting and underfitting are some other points that we have to pay attention to. Data with a lot of noise can make hard the learning process and render the predictions unreliable and of poor generalization (see also Chapters 2 and 7 to 9). Data preparation to assure good quality is of paramount importance then.

The choice of a ML method allowing to fit data in a very high dimensional space can lead to overfitting. In this case the training data can be very well represented, but predictions for unseen data can be far off. The opposite happens with underfitting,

where the space for fitting is too low for identifying and reproducing the inherent patterns in the data. A classical example of overfitting and underfitting in polynomial regression is shown in Figure 6. Thankfully, there are ways for estimating in practice how well a ML model performs. This is mainly achieved by splitting the training set into training, validation and test data. *Regularization* methods can also help in avoiding this problem. For more details we refer to Chapters 2, 4 and 7.



Figure 6: Data points of a synthetic heartbeat, interpolated by two polynomial functions. The high degree polynomial fits exactly the data points, but overfits the signal. The low degree polynomial fits poorly the data points and underfits the signal. Both regressions are not acceptable, because they either predict a superhuman heartbeat or a dead person!

6 What to expect from this volume?

This volume aims at explaining what Machine Learning is, what its main methods are and how they can be used for solving problems in (geo-)mechanics. Most of the chapters were written having in mind to provide a pedagogical introduction to the most important methods in machine learning and the fundamental notions behind them.

It is not possible to cover all the available ML methods in the existing literature and, without any doubt, many important methods were inevitably left out. For instance we won't discuss about genetic algorithms, principal component analysis and related methods, particle swarm optimization, fuzzy logic algorithms, ML methods based on control theory and many others. We hope, however, to have provided a good selection of ML methods for an introductory course.

By the end of this school we expect the students to have:

- demystified and understood what ML is;
- be conscious of the fundamental notions of the most important ML methods;
- used ML in simple examples, got aware of pitfalls and understood the need for physics- and (geo-)mechanics-based ML methods for solving problems in (geo-)mechanics.

The courses are addressed to undergraduate and graduate level. The minimum requirements for accessing them are:

- knowledge of Python programming language¹;
- basic concepts in mathematics (calculus, elements of differential calculus and of numerical analysis).
- have some nice problems in mind that could combine ML and geomechanics!

Updated versions of the chapters of this volume and the python scripts supporting this volume and hands-on sessions are available at:

https://github.com/alert-geomaterials/2023-doctoral-school.

Acknowledgments

The author would like to thank all the authors of the chapters of this volume for their time and effort to provide comprehensive and detailed lecture notes for the ALERT 2023 Doctoral School.

Moreover, the author would like to acknowledge the support of the European Research Council (ERC) under the European Union's Horizon2020 research and innovation program (Grant agreement no. 757848 CoQuake).

References

- [All94] L. Victor Allis. Searching for solutions in games and artificial intelligence. 1994. OCLC: 1110390182.
- [AMHD22] Edward Ando, Benjy Marks, Ryan Hurley, and Joshua A Dijksman. *Advanced Experimental Geomechanics*. Doctoral School lecture notes. The Alliance of Laboratories in Europe for Education, Research and Technology, 2022.
- [BAB⁺21] S. Boukhtache, K. Abdelouahab, F. Berry, B. Blaysat, M. Grédiac, and F. Sur. When Deep Learning Meets Digital Image Correlation. *Optics and Lasers in Engineering*, 136:106308, January 2021.

¹For learning Python, we propose the book "Python Crash Course: A Hands-On, Project-Based Introduction to Programming" by Eric Matthes and/or many excellent tutorials that can be found on the internet

- [BAB⁺23] S. Boukhtache, K. Abdelouahab, A. Bahou, F. Berry, B. Blaysat, M. Grédiac, and F. Sur. A lightweight convolutional neural network as an alternative to DIC to measure in-plane displacement fields. *Optics and Lasers in Engineering*, 161:107367, February 2023.
- [Bal15] Stephen Balaban. Deep learning and face recognition: the state of the art. page 94570B, Baltimore, Maryland, United States, May 2015.
- [BCCR22] Abolfazl Baghbani, Tanveer Choudhury, Susanga Costa, and Johannes Reiner. Application of artificial intelligence in geotechnical engineering: A state-of-the-art review. *Earth-Science Reviews*, 228:103991, May 2022.
- [BDMJ22] Sébastien Becker, Emilie Devijver, Rémi Molinier, and Noël Jakse. Unsupervised topological learning approach of crystal nucleation. *Scientific Reports*, 12(1):3195, February 2022.
- [BSMM21] Gregory C. Beroza, Margarita Segou, and S. Mostafa Mousavi. Machine learning and earthquake forecasting—next steps. *Nature Communications*, 12(1):4761, August 2021.
- [CJM⁺23] Stefano Buitrón Cevallos, Alex X. Jerves, Utkarsh Mital, David A. Medina, V. Santiago Quinteros, Maurizio Mulas, and Øyvind Torgersrud. Towards a more accurate characterization of granular media 2.0: Involving AI in the process. *Computers and Geotechnics*, 160:105510, August 2023.
- [CWX23] Zhuang Cheng, Jianfeng Wang, and Wei Xiong. A machine learningbased strategy for experimentally estimating force chains of granular materials using X-ray micro-tomography. *Géotechnique*, pages 1– 13, February 2023.
- [CZX⁺23] Xiangnan Cheng, Shichao Zhou, Tongzhen Xing, Yicheng Zhu, and Shaopeng Ma. Solving digital image correlation with neural networks constrained by strain-displacement relations. *Optics Express*, 31(3):3865, January 2023.
- [DF18] Julia Dressel and Hany Farid. The accuracy, fairness, and limits of predicting recidivism. *Science Advances*, 4(1):eaao5580, January 2018.
- [DXD⁺23] Xiaocen Duan, Hongwei Xu, Runfeng Dong, Feng Lin, and Jianyong Huang. Digital image correlation based on convolutional neural networks. *Optics and Lasers in Engineering*, 160:107234, January 2023.
- [EBC⁺16] A P Van Den Eijnden, P Bésuelle, F. Collin, René Chambon, and Jacques Desrues. Modeling the strain localization around an underground gallery with a hydro-mechanical double scale model ; effect of anisotropy. *Computers and Geotechnics*, 2016.

- [Fey03] Frédéric Feyel. A multilevel finite element method (FE2) to describe the response of highly non-linear structures using generalized continua. Computer Methods in Applied Mechanics and Engineering, 192(28-30):3233–3244, 2003. ISBN: 0045-7825.
- [GBL⁺21] Zhenfei Guo, Ruixiang Bai, Zhenkun Lei, Hao Jiang, Da Liu, Jianchao Zou, and Cheng Yan. CPINet: Parameter identification of pathdependent constitutive model with automatic denoising based on CNN-LSTM. *European Journal of Mechanics - A/Solids*, 90:104327, November 2021.
- [GC20] Filippo Gatti and Didier Clouteau. Towards blending Physics-Based numerical simulations and seismic databases using Generative Adversarial Network. *Computer Methods in Applied Mechanics and Engineering*, 372:113421, December 2020.
- [GGW91] J Ghaboussi, J H Garrett, and X Wu. Knowledge Based Modeling of Material Behavior with Neural Networks. *Journal of Engineering Mechanics*, 117(1):132–153, 1991.
- [GOTSP] Diego Gutiérrez-Oribio, Georgios Tzortzopoulos, Ioannis Stefanou, and Franck Plestan. Earthquake Control: An Emerging Application for Robust Control. Theory and Experimental Tests. Accepted for publication in IEEE Transactions on Control Systems Technology. arXiv: 2203.00296.
- [Gé19] Aurélien Géron. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media, 2019.
- [HBG⁺21] Quercus Hernandez, Alberto Badías, David González, Francisco Chinesta, and Elías Cueto. Deep learning of thermodynamics-aware reduced-order models from data. *Computer Methods in Applied Mechanics and Engineering*, 379:113763, June 2021.
- [HMW⁺20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. Publisher: Springer Science and Business Media LLC.
- [HWS20] Yousef Heider, Kun Wang, and WaiChing Sun. SO(3)-invariance of informed-graph-based deep neural network for anisotropic elastoplastic materials. *Computer Methods in Applied Mechanics and Engineering*, 363:112875, 2020.

- 22 Machine Learning (ML) in Geomechanics
- [HXFD20] Daniel Z Huang, Kailai Xu, Charbel Farhat, and Eric Darve. Learning Constitutive Relations from Indirect Observations Using Deep Neural Networks. *Journal of Computational Physics*, page 109491, 2020. Publisher: Elsevier.
- [JRLPN⁺21] Paul A. Johnson, Bertrand Rouet-Leduc, Laura J. Pyrak-Nolte, Gregory C. Beroza, Chris J. Marone, Claudia Hulbert, Addison Howard, Philipp Singer, Dmitry Gordeev, Dimosthenis Karaflos, Corey J. Levinson, Pascal Pfeiffer, Kin Ming Puk, and Walter Reade. Laboratory earthquake forecasting: A machine learning competition. *Proceedings of the National Academy of Sciences*, 118(5):e2011362118, February 2021.
- [Kel19] Leo Kelion. Apple's 'sexist' credit card investigated by US regulator. BBC, 2019.
- [KKL⁺21] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, May 2021.
- [KO16] Trenton Kirchdoerfer and Michael Ortiz. Data-driven computational mechanics. Computer Methods in Applied Mechanics and Engineering, 304:81–101, 2016. arXiv: 1510.04232 Publisher: Elsevier B.V.
- [KO18] Trenton Kirchdoerfer and Michael Ortiz. Data-driven computing in dynamics. International Journal for Numerical Methods in Engineering, 113(11):1697–1710, 2018. Publisher: Wiley Online Library.
- [KOA21] K Karapiperis, M Ortiz, and J.E. Andrade. Data-Driven nonlocal mechanics: Discovering the internal length scales of materials. *Computer Methods in Applied Mechanics and Engineering*, 386:114039, December 2021. Publisher: Elsevier B.V.
- [KSOA20] K. Karapiperis, L. Stainier, M. Ortiz, and J.E. Andrade. Data-Driven Multiscale Modeling in Mechanics. *Journal of the Mechanics and Physics of Solids*, (December):104239, 2020. Publisher: Elsevier Ltd.
- [LGBC22] Fanny Lehmann, Filippo Gatti, Michaël Bertin, and Didier Clouteau. Machine learning opportunities to conduct high-fidelity earthquake simulations in multi-scale heterogeneous geology. *Frontiers in Earth Science*, 10:1029160, November 2022.
- [LS03] Marek Lefik and Bernhard A. Schrefler. Artificial neural network as an incremental non-linear constitutive model for a finite element code. *Computer Methods in Applied Mechanics and Engineering*, 192(28-30):3265–3283, 2003. Publisher: Elsevier ISBN: 3949827560.

- [LTG⁺22] Laura Laurenti, Elisa Tinti, Fabio Galasso, Luca Franco, and Chris Marone. Deep learning for laboratory earthquake prediction and autoregressive forecasting of fault zone stress. *Earth and Planetary Science Letters*, 598:117825, November 2022.
- [LVRSHOO19] Oriol Lloberas Valls, Marcelo Raschi Schaw, Alfredo Edmundo Huespe, and Xavier Oliver Olivella. Reduced Finite Element square techniques (RFE2): towards industrial multiscale FE software. In COM-PLAS 2019: XV International Conference on Computational Plasticity: Fundamentals and Applications, pages 157–169. International Centre for Numerical Methods in Engineering (CIMNE), 2019.
- [LW19] Zeliang Liu and C. T. Wu. Exploring the 3d architectures of deep material network in data-driven multiscale mechanics. *Journal of the Mechanics and Physics of Solids*, 127:20–46, 2019. arXiv: 1901.04832 Publisher: Elsevier.
- [MAP⁺15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.
- [MBC⁺19] M Mozaffar, R Bostanabad, W Chen, K Ehmann, Jian Cao, and M A Bessa. Deep learning predicts path-dependent plasticity. *Proceedings of the National Academy of Sciences*, 116(52):26414–26420, 2019. Publisher: National Acad Sciences.
- [McK10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.
- [MH19] Lukas Morand and Dirk Helm. A mixture of experts approach to handle ambiguities in parameter identification problems in material modeling. *Computational Materials Science*, 167:85–91, September 2019.
- [Mit97] Tom M. Mitchell. *Machine learning*. McGraw-Hill series in Computer Science. McGraw-Hill, New York, nachdr. edition, 1997.
- [MPRP22] Armando Marques, André Pereira, Bernardete Ribeiro, and Pedro A. Prates. On the Identification of Material Constitutive Model Param-

eters Using Machine Learning Algorithms. *Key Engineering Materials*, 926:2146–2153, July 2022.

- [MS22] Filippo Masi and Ioannis Stefanou. Multiscale modeling of inelastic materials with Thermodynamics-based Artificial Neural Networks (TANN). *Computer Methods in Applied Mechanics and Engineering*, 398:115190, August 2022.
- [MS23] Filippo Masi and Ioannis Stefanou. Evolution TANN and the identification of internal variables and evolution equations in solid mechanics. *Journal of the Mechanics and Physics of Solids*, page 105245, February 2023.
- [MSVMB21] Filippo Masi, Ioannis Stefanou, Paolo Vannucci, and Victor Maffi-Berthier. Thermodynamics-based Artificial Neural Networks for constitutive modeling. *Journal of the Mechanics and Physics of Solids*, 147:104277, February 2021. arXiv: 2005.12183.
- [NCCD14] Trung Kien Nguyen, Gaël Combe, Denis Caillerie, and Jacques Desrues. FEM × DEM modelling of cohesive granular materials: Numerical homogenisation and multi-scale simulations. *Acta Geophysica*, 62(5):1109–1126, August 2014.
- [NMCDD⁺11] Gaël Nitka Michałand Combe, Cristian Dascalu, Jacques Desrues, Michał Nitka, Gaël Combe, Cristian Dascalu, and Jacques Desrues. Two-scale modeling of granular materials: a DEM-FEM approach. *Granular Matter*, 13(3):277–281, March 2011.
- [PABT⁺21] Grace C. Y. Peng, Mark Alber, Adrian Buganza Tepole, William R. Cannon, Suvranu De, Savador Dura-Bernal, Krishna Garikipati, George Karniadakis, William W. Lytton, Paris Perdikaris, Linda Petzold, and Ellen Kuhl. Multiscale Modeling Meets Machine Learning: What Can We Learn? Archives of Computational Methods in Engineering, 28(3):1017–1037, May 2021.
- [Per21] Sidney Perkowitz. The Bias in the Machine: Facial Recognition Technology and Racial Disparities. *MIT Case Studies in Social and Ethical Responsibilities of Computing*, February 2021.
- [PEW20] Mehdi Pouragha, Mahdad Eghbalian, and Richard Wan. A mote on applicability of Artificial Intelligence to constitutive modeling of geomaterials. *Journal of Machine Learning for Modeling and Computing*, 1(2):157–170, 2020.
- [PGM⁺19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chin-

tala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

- [PS21] Efthymios Papachristos and Ioannis Stefanou. Controlling earthquake-like instabilities using artificial intelligence, April 2021. arXiv:2104.13180 [physics].
- [PVG⁺11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [RKVDM21] I.B.C.M. Rocha, P. Kerfriden, and F.P. Van Der Meer. On-the-fly construction of surrogate constitutive models for concurrent multiscale mechanical analysis through probabilistic machine learning. *Journal* of Computational Physics: X, 9:100083, January 2021.
- [RPK19] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019. Publisher: Elsevier Inc.
- [RY23] Nurafiqah Syahirah Md Ridzwan and Siti Harwani Md. Yusoff. Machine learning for earthquake prediction: a review (2017–2021). *Earth Science Informatics*, 16(2):1133–1149, June 2023.
- [SBV⁺21] Xiao Sun, Bahador Bahmani, Nikolaos N. Vlassis, WaiChing Sun, and Yanxun Xu. Data-driven discovery of interpretable causal relations for deep learning material laws with uncertainty propagation. Springer Berlin Heidelberg, 2021. arXiv: 2105.09980 Publication Title: Granular Matter ISSN: 1434-7636.
- [Sha50] Claude E. Shannon. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, March 1950.
- [SHM⁺16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.
- [SKOM23] Robin Schulte, Cavid Karca, Richard Ostwald, and Andreas Menzel. Machine learning-assisted parameter identification for constitu-

tive models based on concatenated loading path sequences. *European Journal of Mechanics - A/Solids*, 98:104854, March 2023.

- [Ste19] Ioannis Stefanou. Controlling Anthropogenic and Natural Seismicity: Insights From Active Stabilization of the Spring-Slider Model. *Journal of Geophysical Research: Solid Earth*, 124(8):8786–8802, August 2019.
- [SYR⁺21] Anirbid Sircar, Kriti Yadav, Kamakshi Rayavarapu, Namrata Bist, and Hemangi Oza. Application of machine learning and artificial intelligence in oil and gas industry. *Petroleum Research*, 6(4):379– 391, December 2021.
- [TCL⁺22] Faraz S. Tehrani, Michele Calvello, Zhongqiang Liu, Limin Zhang, and Suzanne Lacasse. Machine learning and landslide studies: recent advances and applications. *Natural Hazards*, 114(2):1197–1245, November 2022.
- [TF07] John Tromp and Gunnar Farnebäck. Combinatorics of Go. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, H. Jaap Van Den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630, pages 84–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. Series Title: Lecture Notes in Computer Science.
- [TLA⁺20] Alessandro Tengattini, Nicolas Lenoir, Edward Andò, Benjamin Giroud, Duncan Atkins, Jerome Beaucour, and Gioacchino Viggiani. NeXT-Grenoble, the Neutron and X-ray tomograph in Grenoble. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 968:163939, July 2020.
- [Var23] P. S. Varsha. How can we manage biases in artificial intelligence systems – A systematic literature review. *International Journal of Information Management Data Insights*, 3(1):100165, April 2023.
- [VGO⁺20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy

1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

- [VS21] Nikolaos Napoleon Vlassis and Waiching Sun. Component-based machine learning paradigm for discovering rate-dependent and pressure-sensitive level-set plasticity models. *Journal of Applied Mechanics*, pages 1–13, October 2021.
- [VVNR21] Alexis Megan Votto, Rohit Valecha, Peyman Najafirad, and H. Raghav Rao. Artificial Intelligence in Tactical Human Resource Management: A Systematic Literature Review. *International Journal of Information Management Data Insights*, 1(2):100047, November 2021.
- [WSD19] Kun Wang, WaiChing Sun, and Qiang Du. A cooperative game for automated learning of elasto-plasticity knowledge graphs and models with AI-guided experimentation. *Computational Mechanics*, 64(2):467–499, August 2019.
- [WXW23] Mengmeng Wu, Zhangqi Xia, and Jianfeng Wang. Constitutive modelling of idealised granular materials using machine learning method. *Journal of Rock Mechanics and Geotechnical Engineering*, 15(4):1038–1051, April 2023.
- [ZHX21] Xu-Hui Zhou, Jiequn Han, and Heng Xiao. Learning nonlocal constitutive models with neural networks. *Computer Methods in Applied Mechanics and Engineering*, 384:113927, October 2021.
- [ZL22] Wengang Zhang and Zhongqiang Liu. Editorial for machine learning in geotechnics. *Acta Geotechnica*, 17(4):1017–1017, April 2022.
- [ZLL⁺21] Wengang Zhang, Hongrui Li, Yongqin Li, Hanlong Liu, Yumin Chen, and Xuanming Ding. Application of deep learning algorithms in geotechnical engineering: a short critical review. *Artificial Intelligence Review*, 54(8):5633–5673, December 2021.
- [ZYJ21] Pin Zhang, Zhen-Yu Yin, and Yin-Fu Jin. State-of-the-Art Review of Machine Learning Applications in Constitutive Modeling of Soils. Archives of Computational Methods in Engineering, 28(5):3661– 3686, August 2021.
- [ZZJ+23] Ning Zhang, Annan Zhou, Yin-Fu Jin, Zhen-Yu Yin, and Shui-Long Shen. An enhanced deep learning method for accurate and robust modelling of soil stress–strain response. Acta Geotechnica, February 2023.

Introduction to regression methods

Filippo Masi

The University of Sydney, Australia

Regression is one of the fundamental pillars of supervised Machine Learning. In this Chapter we uncover the essential concepts in regression analysis and methods, by providing (hands-on) practical examples, designed for graduate students and researchers seeking to gain a solid understanding.

In particular, we first delve into linear regression methods, exploring both closed-form solutions and the general optimization framework provided by the gradient descent. Then, we introduce the necessary pre- and post-processing steps for building and testing models: feature scaling, hold-out and cross-validation.

Moving on, we focus on nonlinear regression methods, and, in particular, polynomial regression. Through extensive examples, we then demonstrate the benefits associated with regularization techniques (LASSO, Ridge, Elastic Net) and the possibility of constructing interpretable and parsimonious models.

Lastly, we introduce Bayesian approaches with focus on linear and Gaussian process regression. The advantages and drawbacks of the latter are compared with ordinary regression methods.

1 Introduction

Regression is a statistical technique used to explore and quantify the relationship between dependent variables (often called outcomes) and independent variables (often called predictors). In other words, regression involves fitting a model to some data, under some error function. Two distinct purposes characterize regression. The first one, and the primary, is to identify a model that can be used for predictions and forecasting. While, the second is to infer causal relationships between independent and dependent variables, under certain circumstances.

To grasp the main core of regression, consider a scenario where we seek to understand the influence of relative density and effective confining pressure on the shear strength and critical state behavior of a sand sample. Figure 1(a) presents 25 drained monotonic triaxial compression tests of Karlsruhe fine sand with different initial relative densities I_{D0} and effective confining pressures [WT].

In this framework, regression allows to identify the intrinsic relationship between independent variables \mathbf{x} (that is, relative density and effective pressure) and a dependent variable \mathbf{y} (shear strength or critical state line), by assuming

$$y = f_{\theta}(\mathbf{x}) + \epsilon \tag{1}$$

where f_{θ} represents a function, with θ denoting its parameters (unknown), and ϵ is an error term^{*}. With this objective in mind, we continue by formulating an objective



Figure 1: Drained monotonic triaxial compression tests of Karlsruhe fine sand with different initial relative densities I_{D0} and effective confining pressures [WT] (a). Non-linear regression of the failure envelope – locus of peak deviatoric stress – (b) and critical state line (c) using a basis of polynomial and trigonometric functions.

function that measures the difference between the predictions of the model (1) and the values of the dependent variable. Via the minimization of the latter, we can determine an optimal fit and thus identify the parameters θ . By doing so, we obtain two models, $f_{\theta}(\mathbf{x})$, able to generate predictions of the failure envelope and critical state, as depicted in Figure 1(b-c).

This process is the core of regression and enables us to uncover relationships between variables (data), make predictions, and gain insights from data.

^{*}The error term ϵ is an irreducible source of error. For instance, it may arise whenever there exist variables other than \mathbf{x} – and independent of \mathbf{x} – that have some not negligible effect on the dependent variable y.
This Chapter provides the tools necessary for carrying such analyses by means of a comprehensive introduction to regression methods, using a "hands-on approach" that is, providing essential theoretical aspects and prioritizing understanding through concrete examples. After studying this Chapter, we hope that the reader will be able to:

- · Understand the fundamental notions related to regression methods and gradient descent;
- Solve a regression problem and select the best (class of) method(s) depending on the particular task (linear, nonlinear, regularized, or Bayesian regression);
- Understand the importance of validating and testing a regression model (and in general any Machine Learning model) and how to do so;
- Grasp the importance of regularization techniques to uncover parsimonious and simple models from data sets;
- · And understand the benefits and drawbacks of Bayesian approaches as compared to ordinary regression methods.

All codes used in this Chapter are available on ALERT Geomaterial GitHub (repository alert-geomaterials/2023-doctoral-school) and updated versions of this Chapter can be found at filippo-masi.github.io.

As it follows, scalar values and functions are represented in *lowercase italic* font, while lowercase bold font denotes vectors and uppercase bold font denotes matrices.

2 Linear regression

In linear regression [Man82], the idea is to select a linear function $f_{\theta}(\cdot)$, in equation (1), whose parameters θ are identified to fit data by means of the minimization of some error metric between the predictions, i.e., the outputs of the function, and the dependent variables. As we will see, linear regression can be formulated as a simple solution of a linear system.

To fix the ideas, let us start by considering a *training* data set, with n data points or snapshots, composed of p dependent variables \mathbf{Y} and m independent variables \mathbf{X}

$$\mathbf{Y} \equiv \begin{bmatrix} | & | & | & | \\ \mathbf{y}^{(1)} & \mathbf{y}^{(2)} & \cdots & \mathbf{y}^{(n)} \\ | & | & | & | \end{bmatrix}, \qquad \mathbf{X} \equiv \begin{bmatrix} | & | & | & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(n)} \\ | & | & | & | \end{bmatrix}, \quad (2)$$

. .

where $\mathbf{y}^{(k)}$ and $\mathbf{x}^{(k)}$ are the vectors collecting the p dependent variables and the m independent variables, respectively, at the k-th snapshot of the data set.

In the following, we will consider for simplicity one-dimensional dependent variables, i.e., p = 1, and we will refer to the vector collecting the snapshots of the dependent variable as y. However the entire framework developed herein can be easily formulated in higher dimensions.

To find a best fit line through the training data points, we thus assume a linear model of the form

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_m x_m, \tag{3}$$

where \hat{y} is the value predicted by the model, x_k is the k-th independent variable value, and θ_k is the k-th model parameter – including the *bias* term, θ_0 , and the model *weights*, $\theta_1, \theta_2, \dots, \theta_m$. The above equation can be formulated more concisely as

$$\hat{y} = f_{\theta} \left(\mathbf{x} \right) \equiv \theta^{\mathrm{T}} \mathbf{x}^{*}, \tag{4}$$

where $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \dots, \theta_m]^T$ is the vector collecting the model parameters, $\mathbf{x}^* = [x_0, x_1, x_2, \dots, x_m]^T$ is the augmented feature vector with $x_0 = 1$, $f_{\boldsymbol{\theta}}(\cdot)$ is often called *hypothesis function* – a linear function, in this case –, and the superscript T denotes the transpose operator. To simplify notation, in the following we write interchangeably \mathbf{x} for both \mathbf{x} and \mathbf{x}^* .

The next step is to identify the parameters θ such that the linear regression model, $f_{\theta}(\mathbf{x})$, best fits the data set (\mathbf{X}, \mathbf{y}) . This operation is called *training*. Training identifies the parameters value that optimize a measure of the *goodness-of-fit*, i.e., how well (or poorly) a model fits the data. Various objective functions, often referred to as loss functions, can be adopted and their choice strongly determines the subsequent model. Two standard error metrics of hypothesis $f_{\theta}(\cdot)$ on a set \mathbf{X} are often considered which are associated with the ℓ_1 and ℓ_2 norms, respectively defined as the mean absolute error (MAE) and the root mean square error (RMSE):

$$E_1(\mathbf{X}, f_{\boldsymbol{\theta}}) \equiv \frac{1}{n} \sum_{k=1}^n \left| f_{\boldsymbol{\theta}}(\mathbf{x}^{(k)}) - y^{(k)} \right|$$
(5a)

$$E_2(\mathbf{X}, f_{\boldsymbol{\theta}}) \equiv \left(\frac{1}{n} \sum_{k=1}^n \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(k)}) - y^{(k)}\right)^2\right)^{1/2}.$$
(5b)

One can also broadly define the error based on the ℓ_r norm, and namely

$$E_r\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) \equiv \left(\frac{1}{n} \sum_{k=1}^n \left| f_{\boldsymbol{\theta}}(\mathbf{x}^{(k)}) - y^{(k)} \right|^r \right)^{1/r}.$$
(6)

The higher the norm index r, the more the error metrics focuses on large values and neglect small ones, thus the best fit model intrinsically depends on r. In most cases, the differences between models based on different norms are small. However, when there are outliers[†] in the data, the choice of norm can have a significant impact: for instance, the RMSE is more sensitive to outliers than the MAE.

In linear regression, the most common choice is the root mean square error, but in practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value of θ that minimizes a function also

[†]An outlier is an observation that lies an abnormal distance from the other values present in a data set.

minimizes its square root). The MSE of a linear regression hypothesis f_{θ} on a set X is given by

$$MSE(\mathbf{X}, f_{\boldsymbol{\theta}}) \equiv E_2^2(\mathbf{X}, f_{\boldsymbol{\theta}}) = \frac{1}{n} \sum_{k=1}^n \left(\boldsymbol{\theta}^{\mathsf{T}} \mathbf{x}^{(k)} - y^{(k)} \right)^2.$$
(7)

Once the loss function is defined, training requires to find the parameters that minimize that particular loss. This requires differentiation with respect to θ to identify the value of the latter such that a minimum of the error occurs – that is, find those θ for which $\partial MSE(\mathbf{X}, f_{\theta}) / \partial \theta = 0$. Note that, although a zero derivative denotes either a minimum or a maximum, we know this must be a minimum of the error since there is no maximum error, i.e., we can always find a model that has a larger error.

Following the above procedure, the values of the parameters for which the error is minimum, $\hat{\theta}$, are the solution of the following linear system of equations

$$\mathbf{X}^{\mathrm{T}}\mathbf{X}\hat{\boldsymbol{\theta}}^{\mathrm{T}} = \mathbf{X}^{\mathrm{T}}\mathbf{y}.$$
(8)

If the matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is square and invertible (i.e., it has nonzero determinant), then there exists a unique solution $\hat{\theta}$ which is given by the *normal equation*,

$$\hat{\boldsymbol{\theta}}^{\mathrm{T}} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}.$$
(9)

However, when $\mathbf{X}^T \mathbf{X}$ is singular, the normal equation does not hold. An alternative way to solve equation (4) consists of using the Moose-Penrose pseudoinverse[‡] \mathbf{X}^+ of \mathbf{X} to obtain the general solution

$$\boldsymbol{\theta} = \mathbf{X}^+ \mathbf{y},\tag{10}$$

obtained by leveraging the commutative property $(\mathbf{X}^+)^T = (\mathbf{X}^T)^+$.

We have now all the necessary ingredients for building a linear regression model: is time to move to a simple example.

2.1 Example

Let consider fitting a data set, shown in Figure 2, generated by the function

$$y = \alpha_2 \exp(\alpha_1 x + \mathcal{U}(-1, 1)), \quad \alpha_1 = 1, \ \alpha_2 = e,$$
 (11)

where $\mathcal{U}(-1,1)$ is a uniformly distributed random variable that lies between -1 and 1. The snippet hereinafter generates our data set:

[‡]The pseudoinverse is computed using the matrix factorization technique called singular value decomposition that can decompose the independent variables matrix **X** into the multiplication of three matrices $U\Sigma V^{T}$. The pseudoinverse is computed as $\mathbf{X} = V\Sigma^{+} \mathbf{U}$. To compute the matrix Σ^{+} , the algorithm takes Σ and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix.

```
import numpy as np
np.random.seed(42)
n_snapshots = 200 # set number of snapshots: n
a_1 = 1.; a_2 = np.exp(1) # set coefficients
noise = np.random.uniform(-1,1,(n_snapshots,1)) # uniform noise
X = np.random.uniform(0,4.,(n_snapshots,1)) # independent variable X
y = a_2*np.exp(a_1*X+noise) # dependent variable y
```

At this point, we can use the normal equation (9) and compute the values $\hat{\theta}$, using NumPy's linear algebra module (np.linalg) to calculate the inverse of a matrix and the dot() method for matrix multiplication:

```
X_p = np.c_[np.ones((n_snapshots, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_p.T.dot(X_p)).dot(X_p.T).dot(y)
print(theta_best) # parameters
array([[-16.93873601],
```

[28.05702557]])

The normal equation gives a linear model of the form $\hat{y} \approx -16.9 + 28.0x$. Let's check if the model correctly fits the data set, making predictions as follows

```
X_new = np.array([[0],[4]])
X_b = np.c_[np.ones((2,1)), X_new] # add x0 = 1 to each instance
y_predict = X_b.dot(theta_best)
```

Finally, we can plot the predictions (y_predict) with the training data set, see Figure 2. As one could have imagined from the very beginning, the linear regression model poorly fits the nonlinear data set.



Figure 2: Linear regression: an example and a failure (!). Randomly generated data set (left) – $y = \alpha_2 e^{(\alpha_1 x + \mathcal{U}(-1,1))}$ – and linear regression model predictions (right).

But, as in most cases, data set preparation and transformation may help us in improving the model predictions. In this particular case, one has to simply perform a

change of coordinates, defining $y' = \ln(y)$ and $\alpha'_2 = \ln(\alpha_2)$. Such a transformation (*magically*) performs the linearization of the original data set, which from (11) now becomes

$$y' = \alpha'_2 + \alpha_1 x + \mathcal{U}(-1,1), \quad \alpha_1 = 1, \; \alpha'_2 = 1,$$
 (12)

Let's try to apply the following changes to the original data set, repeating the linear regression fit, and plotting the predictions:

```
y_prime = np.log(y) # change of coordinate, y' = ln(y)
X_b = np.c_[np.ones((n_snapshots, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y_prime)
X_new = np.array([[0],[4]])
X_b = np.c_[np.ones((2,1)), X_new] # add x0 = 1 to each instance
y_predict = X_b.dot(theta_best) # parameters
print(theta_best)
array([[0.99530983],
```

```
[0.98646971]])
```

Done! The linear regression now correctly fits the transformed data set, identifying the following model: $\hat{y}' = 0.98 + 0.99x$. Note that the presence of noise renders impossible to retrieve the exact values of the intercept and the angular coefficient.



Figure 3: Comparison of the linear regression model predictions with the linearized data set, see equation (12).

Alternatively, a simpler (and more concise) way to perform a linear regression fit is to directly compute the pseudoinverse of the independent variable matrix \mathbf{x} , see equation (10). This can be done, in few lines of code, using Scikit-Learn's library [BLB⁺13]:

```
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
regr = linear_model.LinearRegression() # create linear regression object
regr.fit(X, y_prime) # train the model
```

3 Gradient descent

We have seen that linear regression allow us to fit data sets with pre-identified linear models or hypothesis f_{θ} . The advantages are (i) the admission of analytically tractable, best-fit solutions and (ii) the reduced computational complexity. However, the major drawback is the impossibility to fit nonlinear functions, much more abundant in nature than linear ones (cf. Chapters 4 and 7).

To this end, the general theory of nonlinear regression[§]) considers a nonlinear hypothesis function $f_{\theta}(\mathbf{x})$, in contrast with the linear function in equation (3). In this case, if we proceed as we have done for linear regression – that is, requiring that $\partial MSE(\mathbf{X}, f_{\theta}) / \partial \theta = 0$ – we obtain the following nonlinear system of equations

$$\frac{1}{n}\sum_{k=1}^{n} \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(k)}) - y^{(k)} \right) \frac{\partial f_{\boldsymbol{\theta}}}{\partial \boldsymbol{\theta}} = 0.$$
(13)

Unfortunately, there are no analytical methods to solve such a nonlinear system for a general (unspecified) nonlinear hypothesis f_{θ} . And, actually, sometimes equation (13) may even not admit a solution or admit an infinity of solutions.

In such scenarios, the idea to solve the nonlinear system (13) is to resort to iterative approaches which, depending on the good (or bad) initial guess, may converge to the global (or a local) minimum error.

One of the most effective approaches to identify the roots of a nonlinear system of equations is *gradient descent* [BV04]. Gradient descent (GD), also called steepest descent, is an optimization algorithm for finding a local minimum of a differentiable function – that is, a set of optimal parameters θ that minimizes a given (differentiable) loss function.

A quite easy way to grasp the essence of gradient descent is to imagine yourself lost in a foggy mountain, where visibility is so limited that you can only sense the slope beneath your feet. To find your way out quickly, a smart approach would be to head downhill, following the steepest slope[¶]. This is exactly what gradient descent does: it measures the local gradient of the loss function with regard to the parameters θ and tweaks the latter to go in the direction of descending gradient. Once the gradient is zero, we have reached the minimum.

[§]Note, however, that if we formulate the nonlinear regression problem differently, within a reduced special setting, a seemingly linear hypothesis function can actually be formulated (cf. Section 5.

[¶]Disclaimer: if you are truly lost on a mountain, then you should probably stop and wait for a break in the fog, rather than following the steepest descent.

Let's break it down further. At the beginning, θ is filled with random values, or some other predefined values. This process is called *initialization*. Then, in small incremental steps, often called *epochs*, the algorithm tweaks repeatedly the parameters to minimize the value of the loss function $E(\theta, f_{\theta})$ (that must be differentiable for all θ – which also implies that f_{θ} must be differentiable for every θ) according to the regular gradient descent optimizer equation

$$\boldsymbol{\theta}^{\text{next step}} = \boldsymbol{\theta} - \eta \frac{\partial E}{\partial \boldsymbol{\theta}} \left(\boldsymbol{\theta}, f_{\boldsymbol{\theta}} \right), \tag{14}$$

where η is the *learning rate*. In other words, we need to compute how much the loss function E will change if we change θ_i just a little bit, with $i = 1, 2, \dots, m$.

By resorting to the analogy of the foggy mountain, it is like asking "What is the slope of the mountain under my feet if I face east?" and then asking the same question facing north (and so on for all other dimensions i up to m, if you can imagine a universe with more than three dimensions).

After having computed the gradient vector $\partial E/\partial \theta$ which points uphill, we just have to head in the opposite direction to go downhill – that is, subtracting the gradient from the parameters θ , with a weighting factor η . This process continues until the algorithm converges, reaching a (global or local) minimum value, as shown in Figure 4.



Figure 4: Schematic representation of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the parameters approach the minimum.

When optimizing an objective function using GD two important aspects must be considered:

• The size of the incremental steps, which is determined by the **learning rate**, η . The latter is a *hyperparameter*. A hyperparameter is a variable controlling the learning process (in contrast to the model parameters that are determined via training). The learning rate, as such, is crucial in identifying the best value of

the parameters θ within the smallest number of iterations.

Indeed, if η is set too small, the algorithm will require numerous iterations to converge, resulting in a time-consuming computations. Conversely, if η is excessively high, we might overshoot the minimum and end up on the opposite side, potentially at a point where error is higher than before. This scenario could even cause the algorithm to diverge, producing increasingly larger error values and failing to find an optimal solution.

• The **loss function**. Not all loss functions exhibit a smooth, bowl-like shape as the ones depicted in Figure 4. Depending on the choice of loss function and/or the hypothesis function, we may find irregularities such as holes, ridges, plateaus, and many local minima, creating challenges for convergence towards the (global) minimum. Also remember that the loss function must always be differentiable^{||}, otherwise we cannot compute the optimizer equation (14).

Figure 5 highlights some of the primary obstacles that we may encounter with some loss functions and, in most of the cases, for nonlinear hypotheses f_{θ} – but also in neural networks (cf. Chapter 7). When the algorithm starts on the left due to random initialization, it attempts to converge to a local minimum, which may not be as desirable as the global minimum. This same local minimum is a point of discontinuity which renders even more difficult the convergence. On the other hand, if the algorithm begins on the right, it finds a plateau and crossing it takes an extensive amount of time and if we prematurely stop the algorithm we won't succeed in reaching the global minimum.



Figure 5: Gradient descent for a general non convex loss function.

The MSE for linear regression models happens to be always differentiable and convex: great news! This means that any two points on the curve can be connected by a line segment that never intersects the curve. Consequently, there are no local minima, only one single global minimum. Moreover, the MSE function is continuous, exhibiting a slope that changes gradually (differently from the MAE). These characteristics have a significant implication: given sufficient time and an appropriate value of the learning rate, gradient descent is guaranteed to approach the global minimum closely.

^{II}In reality, when dealing with a loss function that is not everywhere differentiable, an ad-hoc subgradient procedure can be used. Instead of calculating the gradient, we use a subgradient that provides a valid lower bound on the slope of the loss function at that particular singular point.

In linear regression, the shape of the MSE loss function always resembles a bowl. However, we should note that the latter can be elongated if the parameters – that is, the features – have different scales. Figure 4, right, demonstrates GD applied to a set with features θ_1 and θ_2 , where the latter has significantly smaller values than the former. Note that, in general, when features have quite different scales, the convergence to the global minimum requires a large number of iterations. Such a drawback can be easily overcome by appropriately scaling the features – that is, by scaling the independent and dependent variables (cf. Section 4).

It's worth noting that training a model, whether linear or not, involves searching for a perfect combination of model parameters that minimizes the loss function across the entire data set. This search takes place within the vast parameter space of the model. Imagine we are trying to train a model with an extensive set of parameters that surpasses the simplicity of the one- and two-parameters models in Figure 4. In this scenario, we are essentially following the same steps as before, but now in a higher-dimensional space. The dimension of this space is equal to the number of parameters, making the search much more challenging. Indeed, finding a needle in a haystack that has 1000 dimensions is significantly more intricate than doing so in just two dimensions. However, in the case of linear regression (unlike artificial neural networks, see Chapter 7), the loss function has optimal properties and the needle (global minimum) always resides comfortably at the bottom.

3.1 Batch gradient descent

Batch gradient descent is the most cumbersome implementation of the GD algorithm. It involves the use of equation (14) for the full training set \mathbf{X} at every step – that is, the whole batch of training data (and this is the reason of its name).

Applying batch GD for a linear regression problem starts with the computation of the partial derivative of the MSE function with respect to the model parameter,

$$\frac{\partial}{\partial \boldsymbol{\theta}} \text{MSE}\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) = \frac{2}{n} \mathbf{X}^{\text{T}} \left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right), \tag{15}$$

and continues by recursively using equation (14) to update the parameters. The recursion stops when we reach the prescribed number of steps (epochs) or the error is smaller than a prescribed threshold.

We can now use the training data set from the previous example and find a linear regression model by means of batch GD:

```
eta = 0.05 # learning rate
n_epochs = 500 # no epochs
theta = np.array([[0.],[0.]]) # initialization
for epoch in range(n_epochs):
    gradients = 2/n_snapshots * X_p.T.dot(X_p.dot(theta) - y_prime) # Eq. (15)
    theta = theta - eta * gradients # optimizer Eq. (14)
print(theta)
[[0.99530187]
```

```
[0.98647279]]
```

It should come as no surprise that we obtain the same result as using the normal equation. However, we might wonder what would happen if we had chosen a different learning rate. In Figure 6, we visualize the predictions of the same linear model within the first 10 iterations using different learning rates, all starting from the same initial point (represented by the dotted line). Similarly, Figure 7 illustrates the evolution of the model parameters, θ_1 and θ_2 , with the contours representing the loss function (MSE) that has a global minimum approximately located at (1, 1).



Figure 6: Batch gradient descent with various learning rates with the same parameters initialization.



Figure 7: Evolution of the model's parameters during batch gradient descent with various learning rates (see Figure 6).

Starting from the left, we observe that with a relatively high learning rate, the algorithm bounces around during each iteration. In this particular scenario, if we select a larger value of the learning rate, the algorithm will diverge and we will never reach the minimum (you can try by yourself with the above code). In the middle, a moderately appropriate learning rate allows gradient descent to smoothly reach the global minimum. On the right, with a very low learning rate, the algorithm will eventually converge to the solution, but it will take a considerable amount of time.

To identify an appropriate learning rate value, we can employ two strategies: first, we can use an adaptive learning rate (see paragraph 3.2) or second, we can fine-tune the

learning rate using grid search** [Gér22].

Despite being much faster than using the normal equation (9) or computing the pseudoinverse (10) for high-dimensional linear regression tasks, batch GD is the most computationally demanding among all gradient descent variants and the most vulnerable to "bad" initialization. In addition, the evaluation of the optimizer equation (14) is often computationally intractable, especially for deep neural networks (cf. Chapter 7). This is primarily due to two reasons: (i) the parameter vector, collecting the parameters θ_k , can be quite large, and (ii) the number of data points *n* can also be large. Thus, utilizing the entire training set for computing the gradient can hinder convergence speed.

As it follows, we will explore two alternative versions of gradient descent that can enhance the convergence speed to the minimum of the loss function: stochastic gradient descent and mini-batch gradient descent.

3.2 Stochastic gradient descent

Stochastic gradient descent is a version of gradient descent that differs from batch gradient descent in how it estimates the gradient in equation (14). Instead of using all ntraining data points, stochastic GD randomly selects a single datum (snapshot) from the training set to estimate the gradient at each iteration. As a result, stochastic GD is faster than batch GD: it only needs to process a single instance at a time allowing efficient memory allocation when dealing with large data sets (see also Chapter 7).

However, due to its stochastic nature, stochastic GD exhibits less smooth behavior compared to batch GD. Instead of smoothly decreasing towards the minimum, the loss function experiences fluctuations, with only an average error decrease over iterations. Consequently, when the algorithm stops, the final parameter values are generally good but not necessarily optimal. Yet, it is worth noting that the inherent randomness in stochastic GD can actually aid in escaping eventual local minima, increasing the chances of finding the global minimum compared to batch GD.

Hereinafter, we implement the stochastic GD for the same above example:

```
eta = 0.05 #learning rate
n_epochs = 20 # no epochs
theta = np.array([[0],[0.]]) #initialization
for epoch in range(n_epochs):
    for i in range(n_snapshots): # iterate over each snapshot
        random_index = np.random.randint(n_snapshots) # pick random snapshot
        xi = X_p[random_index:random_index+1]
        yi = y_prime[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi) # Eq. (15)
        theta = theta - eta * gradients # Eq. (14)
print(theta)
```

^{**}Grid search is a technique used in ML to systematically search for the optimal combination of hyperparameter values for a given model. It involves specifying a set of possible values for each hyperparameter and exhaustively evaluating the model performance using all possible combinations of these values. By evaluating the model accuracy for each combination, grid search helps identify the hyperparameter configuration that yields the best performance.

```
[[0.83905266]
[1.15041967]]
```

The results are shown in Figure 8, where we can observe that the algorithm struggles in converging to the global minimum, rather bouncing all around. This is to the significant dependence of stochastic GD on the learning rate, which in turn is due to inherent stochastic nature of the former.



Figure 8: Stochastic gradient descent and evolution of the parameters, with constant learning rate.

The problem can be alleviated by gradually adapting the learning rate while training. Below, we repeat the learning process using a rather simple learning schedule that linearly decrease the learning rate at each epoch:

```
theta = np.array([[0],[0.]]) #initialization
t0, t1 = 1, 100 # learning schedule hyperparameters
def learning_schedule(t):
    return t0 / (t + t1)
for epoch in range(n_epochs):
    for i in range(n_snapshots):
        random_index = np.random.randint(n_snapshots)
        xi = X_p[random_index:random_index+1]
        yi = y_prime[random_index:random_index+1]
        eta = learning_schedule(epoch * n_snapshots + i) # schedule
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        theta = theta - eta * gradients
    print(theta)
[[0.90607535]
```

```
[1.02276101]]
```

Super effective! In 20 iterations only, the algorithm converges – relatively close – to the global minimum, as depicted in Figure 9. And this should be compared with the 500 iterations that were needed for the batch GD to reach the minimum.

3.3 Mini-batch gradient descent

Once we grasped both batch and stochastic GD, understanding mini-batch GD becomes straightforward if we get inspiration from Aristotle's (Latin) quote: "*In medio stat virtus*" – that is, "*the best option lies between two extremes*".

Masi 43



Figure 9: Stochastic gradient descent with learning schedule.

Instead of computing gradients based on the entire training set (batch GD) or a single datum (stochastic GD), mini-batch GD computes gradients on small random sets of data points known as *mini-batches*. This approach offers the advantage of leveraging optimized matrix operations, resulting in improved performance compared to stochastic GD. At the same time, it avoids the need for large memory allocation required by batch GD. Depending on the size of the mini-batch, the optimization path followed by mini-batch GD is less erratic than its stochastic counterpart, often leading to a more optimal minimum.

Implementing mini-batch GD is relatively simple. We only need to make slight modifications to the stochastic GD algorithm and provide an iterator to extract minibatches:

```
def iterate_minibatches(inputs, targets, batchsize, shuffle=False):
    '''Iterator over mini-batches
    :param inputs, targets: independent and dependent variables
    :param batchsize: mini-batch size
    :param shuffle: shuffle mini-batches
    :return: mini-batches of inputs and targets'''
    # Check if no of samples in inputs and targets are equal
    assert inputs.shape[0] == targets.shape[0]
    if shuffle:
        indices = np.arange(inputs.shape[0]) # Generate array of indices
        np.random.shuffle(indices) # Shuffle the indices
    # Iterate over mini-batches
    for start_idx in range(0, inputs.shape[0] - batchsize + 1, batchsize):
        if shuffle: # subset of shuffled indices
            excerpt = indices[start_idx:start_idx + batchsize]
        else: # slice to select samples
            excerpt = slice(start_idx, start_idx + batchsize)
        yield inputs[excerpt], targets[excerpt] # return mini-batches
eta = 0.05 #learning rate
theta = np.array([[0],[0.]]) #initialization
batch_size = 30 # define mini-batches size
for epoch in range(n_epochs):
    for batch in iterate_minibatches(X_p, y_prime, batch_size, shuffle=True):
       x_batch, y_batch = batch
        gradients = 2/batch_size * x_batch.T.dot(x_batch.dot(theta) - y_batch)
        theta = theta - eta * gradients
```

print(theta)

[[0.94862053] [0.96679924]]

Done: much faster and accurate than the stochastic GD!

Figure 10 depicts the predictions of the model within the first 10 iterations of the algorithm and the parameters evolution. Note that the value of the minimum reached by the mini-batch GD intrinsically depends on the selected size of the mini-batch. The latter represents yet another important hyperparameter of the optimization problem.



Figure 10: Mini-batch gradient descent.

To perform the same task, we could also use Scikit-Learn, and namely the SGDRegressor class, which by default minimizes the MSE. The following code runs for maximum 20 epochs or until the loss drops by less than 0.001 during one epoch (max_iter=20, tol=1e-3). It starts with a learning rate equal to eta and use the following learning schedule: $\eta^{(\text{next step})} = \eta t^{-1/4}$. The option penalty=None avoids the use of regularization strategies (cf. Section 6).

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=20, tol=1e-3, penalty=None, eta0=eta)
for epoch in range(n_epochs):
    for batch in iterate_minibatches(X_p, y_prime, batch_size, shuffle=True):
        x_batch, y_batch = batch
        sgd_reg.partial_fit(X, y_prime.ravel()) # partial_fit mini-batch
print(sgd_reg.intercept_, sgd_reg.coef_)
```

[1.00172812] [1.00508048]

Also note that if we were performing stochastic GD, we would not need to iterate over the mini-batches, and we should replace partial_fit() with fit().

In conclusion, the three alternative algorithms for gradient descent – batch, stochastic, and mini-batch – are powerful tools for solving an optimization problem and they should be preferred to the normal equation and the computation of the pseudoinverse, when dealing with a high-dimensional linear regression problem (with large numbers of data sets and/or large number of parameters).

The main difference among the three alternatives is summarized by Figure 11 which

provides a comparison for the parameters trajectory during training. The mini-batch GD is often preferred as it allows a good compromise in terms of the goodness of the reached minimum and the computational speed, however it is difficult to provide strict recommendations for a general case. As a rule of thumb, we may opt for a mini-batch GD and train the model with different mini-batch sizes – if equal to 1, we will resort to stochastic GD, if equal to the training set size, we will end up with batch GD.



Figure 11: Gradient descent paths comparison in the parameters space.

Finally, it is worth noticing that herein we limited ourselves to the standard optimizer equation (14), however alternative and more effective solutions exist (cf. Chapter 7).

4 Data preprocessing and model validation

In the above examples, we have seen how to perform linear regression given some training data sets. However, we have neglected two important aspects common to all ML models – regression included: *feature scaling* and models *validation*. The former is part of the preprocessing to train a model, while the latter can be interpreted as postprocessing of the trained model.

4.1 Feature scaling

Feature scaling is a crucial step in preparing data for ML. Indeed, all gradient descent algorithms and in general most of the optimization algorithms struggle in dealing with variables with different scales (e.g. Figure 4). In the frame of gradient descent, there are two common methods to address this issue: normalization and standardization. Both techniques transform a feature x into its scaled counterpart \bar{x} , according to

$$\bar{x} = \frac{1}{\alpha} \left(x - \beta \right). \tag{16}$$

Normalization shifts and re-scales the feature values to a range between -1 and 1, by considering $\alpha \equiv \frac{1}{2} (\max(x) - \min(x))$ and $\beta \equiv \alpha + \min(x)$. On the other hand, standardization transforms the feature in a distribution with a zero

e omer nand, standardization transforms the feature in a distribution with a ze

mean and unit variance, by considering $\alpha \equiv \sigma(x)$ and $\beta \equiv \mu(x)$, with μ and σ being the mean and standard deviation, respectively.

Normalization is simple and confines values to a specific range, while standardization is not bound to a range and is less affected by outliers. Whether it is more appropriate to normalize or *standardize* the data depends mainly on the loss function and the statistics of the data themselves – when the latter are uniformly distributed, normalization is to be preferred, in all the other cases, standardization is the best choice.

Hereinafter an explicit implementation of both scaling techniques, depending on how the function is called:

```
def scaling(x, fit=False, transform=False, inverse_transform=False,
            norm=True, param=None):
    ''' scale variable x
        :param x: variable
        :param fit: find scaling parameters
        :param transform: scale x
        :param inverse_transform: inverse scale x
        :param norm: inverse scale x
        :param fit: scaling parameters
        :return: scaling parameters (if fit=False)
                scaled x (if transform=True)
                 inverse scaled x (if inverse_transform=True) '''
    if fit==True:
        if norm==True:
            min_ = np.amin(x); max_ = np.amax(x)
            a = 0.5*(max_-min_)
            b = 0.5*(max_+min_)
        else:
            a = np.std(x)
            b = np.mean(x)
        return [a,b]
    elif transform==True:
        return np.divide(x-param[1],param[0])
    elif inverse_transform==True:
        return np.multiply(x,param[0])+param[1]
```

One could also use directly Scikit-learn's preprocessing module.

4.2 Test and validation of a model

In the examples above, we discovered regression models that best fits some training data $\{\mathbf{X}, \mathbf{y}\}$. However, in the general case, and not only in the frame of regression methods, we need to evaluate how well a (regression) model performs on new data. To this end, we must first define three notions: *interpolation*, *generalization*, and *extrapolation*.

Definition. Interpolation is the ability of predicting values within the range of observed data points.

In the context of regression, it means predicting the value of the dependent variables for values of independent variables that lie within the range of the training data.

Definition. Generalization is the ability of making predictions for new, previously unseen data, drawn from the same distribution as the ones used to train the model.

Machine learning is all about having models that can generalize well. Trained to solve one problem, the model attempts to utilize the patterns/relationships learned from that task to solve the same task, with slight variations. Whilst the existence of a subtle difference between interpolation and generalization, both terms are often adopted interchangeably.

Definition. *Extrapolation is the ability of predicting values beyond the range of the observed, training data points.*

Extrapolation requires a solid understanding of the relationships identified by a model to apply the same outside the familiar data range. It is challenging for current ML models to reliably extrapolate, as they often specialize in specific tasks and struggle with broader applications. Many artificial intelligence methods are inherently interpolative, and constructing extrapolative or "intelligent" algorithms still remains an open challenge (cf. Section 7, Chapters 7 and 9).

From the aforementioned definitions, it is clear that we should **always test models** in order to quantify their generalization capabilities. A cumbersome approach consists of dividing the original data set into two: a *training set* and a *test set*.

The training set is used to train the model, as we have already seen, while the test set is used to assess its performance. The latter is estimated by means of the generalization error, which measures the error of the trained model on the (unseen) test data set. Thus, if a model has low training error but high generalization error, it indicates *overfitting* to the training data. Overfitting is defined as the phenomenon where a model becomes overly complex and excessively fits the training data set, capturing noise and irrelevant patterns instead of learning the true underlying patterns and relationships.

However, evaluating the generalization error multiple times on the same test set and optimizing the model and hyperparameters (such as the learning rate in gradient descent, or the mini-batch size in mini-batch GD) specifically for that set may be suboptimal. Indeed, the model will unlikely perform as well on new data.

4.2.1 Hold-out validation

To mitigate the problem arising from testing a model on the same test set, a common solution is *hold-out validation*. A portion of the training set is reserved as a *validation set*. Multiple candidate models with different hyperparameters are trained on the reduced training set, and the one that performs best on the validation set is selected. The best model is then trained on the full training set, including the validation set, to create the final model. The performance of the latter is then evaluated on the test set to quantify the generalization error.

While hold-out validation is effective, the size of the validation set is crucial. If it is too small, evaluations may be imprecise, leading to the selection of suboptimal models. Conversely, if the validation set is too large, the remaining training set becomes

significantly smaller, making it unfair to compare models trained on a much smaller set.

As a rule of thumb (and a first good guess), we often split data in 80%-20%-20% for training-validating-testing. We can easily do it, by leveraging Scikit-learn:

```
from sklearn.model_selection import train_test_split
# split into training+validation and test sets
X_tv, X_tv, y_train, y_test = train_test_split(X, y, test_size=0.2)
# split into training and validation sets - note: 0.25 * 0.8 = 0.2
X_train, X_val, y_train, y_val = train_test_split(X_tv, y_tv, test_size=0.25)
```

4.2.2 Cross-validation

Repeated hold-out validation addresses the issue related to the choice of the validation set size by using multiple small validation sets. This process is called *k*-fold cross-validation, where the training data set is further partitioned into *k*-folds, which are typically randomly selected portions of the original set. In *k*-fold cross validation, the training data is randomly partitioned into *k* partitions (or folds). Each partition is used to construct a regression model $\hat{y}_{(l)} = f_{\theta_{(l)}}(\mathbf{x}_{(l)})$ for $l = 1, 2, \dots, k$. Then, the final, cross-validated model is constructed by averaging the values of the parameters obtained from each fold – that is, $\theta = 1/k \sum_{l=1}^{k} \theta_{(l)}$.

Despite its great advantage over hold-out validation, cross-validation may increase tremendously the learning process as each model needs to be trained on different validation sets.

We will implement cross-validation both using Scikit-learn (cf. example in paragraph 5.1) and performing the aforementioned steps explicitly (cf. application in paragraph 7.1).

5 Nonlinear regression

Up to this point, we have seen how to train and validate linear models that best fits some data. However, what if the data are more complex and cannot be fitted, with good approximation, by a simple straight line?

In that case, we may simply resort (spoiler alert!) to artificial neural networks (see Chapter 7). But, no worries! A much simpler alternative exists: we can actually fit a linear model to nonlinear data. To this end, we just need to replace the original set of independent variables, **X**, with a library $\phi(\mathbf{x}) = [\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \cdots, \phi_l(\mathbf{x})]^T$ of *l* candidate basis functions, e.g., polynomials, trigonometric or radial basis functions, or any other user-supplied function. By doing so, equation (1) becomes

$$\hat{y} = \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}). \tag{17}$$

Everything is settled and we can move straightforward to an example. This time, we will also take care of preprocessing the data (see Section 4). We will consider as candidate basis functions only polynomial ones: the regression is thus said to be polynomial.

Masi 49

5.1 End-to-end example

Let us consider a data set given by

$$y = \sin(|x|) + \sin(x^2) + \mathcal{N}(0,\sigma),$$
 (18)

where $\mathcal{N}(0, \sigma)$ is a normally distributed random variable with mean zero and standard deviation σ . To generate the data, we use the code below:

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(42)
n_snapshots = 200
noise = np.random.normal(0,0.5,(n_snapshots,1)) # generate normal noise
X = np.random.uniform(-np.pi,np.pi,(n_snapshots,1))
y = np.sin(np.abs(X))+np.sin(X)**2+ noise
```

Now, we proceed by splitting the data set into a training and test set, using Scikit-Learn's train_test_split. Then, we scale both independent and dependent variables based on the statistics of the training data set (see Section 4).

```
from sklearn.model_selection import train_test_split
# split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
# compute scaling params
param_x = scaling(X_train,fit=True) # scaling params for X
param_y = scaling(y_train,fit=True) # scaling params for y
# scale X (training and test sets)
norm_X_train = scaling(X_train,transform=True,param=param_x)
morm_X_test = scaling(X_test,transform=True,param=param_x)
# scale y (training and test sets)
norm_y_train = scaling(y_train,transform=True,param=param_y)
norm_y_test = scaling(y_test,transform=True,param=param_y)
```

At this point, we have to construct the polynomial basis functions, with Scikit-Learn's PolynomialFeatures (we stop at the sixth degree). Then, we can simply perform a linear regression

Not so difficult! The model finds the following best-fit equation

 $\hat{y} = -0.36 + 0.20x + 7.24x^2 - 0.73x^3 - 16.11x^4 + 0.49x^5 + 8.95x^6.$

We can now deploy it to make predictions and compare the latter with the test set

```
x = np.expand_dims(np.linspace(-np.pi,np.pi,200),1)
norm_x = scaling(x,transform=True,param=param_x)
norm_x_poly = poly_features.fit_transform(norm_x)
norm_y_predicted = lin_reg.predict(norm_x_poly)
```

Figure 12, left, compares the predictions with the training and test sets and below we evaluate the MSE of the predictions for the training and test sets:

MSE on training data set: 0.05173368072737507 MSE on test data set: 0.05142192371233674

The model performs quite well on both sets.



Figure 12: Nonlinear regression using polynomial basis functions with: (i) degree six (left) and (ii) different degrees (right). Features are scaled.

However, we skipped two important aspects: (i) hyperparameters selection and (ii) validation of the model.

5.1.1 Hyperparameters selection

The degree of the polynomial basis is actually a hyperparameter. Thus, we cannot a priori say that the analysis is completed – that is, whether or not we found the best model for the data set at hand. To this end, we need to train multiple models with different degrees:

Masi 51

```
degree = [2,6,50,100] # different polynomial degrees
color = ['black','red','darkblue','darkorange']
fig = plt.figure(figsize=(3., 2.))
plt.plot(norm_X_train,norm_y_train,'ko',alpha=0.2)
plt.plot(norm_X_test,norm_y_test,'ko')
for i in range(len(degree)):
    poly_features = PolynomialFeatures(degree=degree[i], include_bias=False)
    norm_X_poly = poly_features.fit_transform(norm_X_train)
    lin_reg = LinearRegression() # linear regression model
    lin_reg.fit(norm_X_poly,norm_y_train) # fit model
    norm_x_poly = poly_features.fit_transform(norm_x)
    norm_y_predicted = lin_reg.predict(norm_x_poly)
    plt.plot(norm_x,norm_y_predicted,'-',color=color[i],
             linewidth=3,label=str(degree[i]))
plt.ylim(-1,1)
plt.xlim(-1,1)
plt.show()
```

The results are depicted in Figure 12, right, and from them we can draw some conclusions: (i) a quadratic model is unable to fit the data and (ii) high-degree -50 and 100, here - polynomial models wiggle around to get as close as possible to the training data points. In other words, the quadratic model is underfitting, while for high polynomial degrees, the model tends to overfit the training set. So, how can we select the best models - that is, the one that will generalize best (cf. Section 4)? To answer let's move to the second part of this example: model validation.

5.1.2 Validation

We have already seen in Section 4 that a crucial aspect in finding models that generalize well is to perform hold-out validation. To this end, we should assess the model's performance by examining the validation error at varying of the size of the training set (and eventual hyperparameters). By repeatedly training the model on subsets of varying sizes from the training set, we can generate informative plots that will guide us in selecting the model with the minimum error on the validation set. Below we implement a function that is designed to plot the training, validation, and test errors of models with different polynomial degrees (up to 40).

Let first have a look at the error on the training and validation sets for a sixth-degree polynomial model, depicted in Figure 13. We start by analyzing the behavior of the training error. When there is a small bunch of training points, the model can perfectly fit those data and results in zero error. However, as more points are added, the model struggles to fit the data precisely due to the noise that we artificially added and the intrinsic non-linearity of the function we are trying to fit. Thus, the error on the training data increases until it reaches a plateau where additional data won't significantly affect the mean error.

An analysis on the validation error however rapidly reveals that things are quite different. When there is a small bunch of training points, the model clearly fails to generalize well and leads to relatively high validation errors. As the model is exposed to more training points, it gradually learns and improves, causing the validation error to decrease and, in this example, to reach approximately the same plateau of the training error.



Figure 13: Training and validation MSE at varying of the training set size for a sixthdegree polynomial model.

We are now ready to look at the big picture, namely Figure 14, where we can see the variation of the training, validation, and (optionally) test errors at varying of both the polynomial degree and the size of the training set.

The training error presents the same aforementioned behavior. Additionally, we can observe that for any polynomial degree smaller than 6, the error is high - that is, the model is underfitting the training data. Instead, as we increase the degree of the model, we observe a slight reduction of the error (caused by overfitting the training data). If we move to the validation error, we observe that for training sets smaller than a certain size, the model does not generalize well (exactly as before). However, in addition, we (re-)discover overfitting. For degrees larger than 20, the model displays high validation error meaning that it overfits the training data and cannot accurately predict any other data different from those contained in the training set (poor generalization). As we have discussed in Section 4, the validation set allows us to pick up the best model(s) as the one(s) that presents the minimum validation error - approximately corresponding to a degree equal to 16, in this example. Once we have selected such model, we can deploy it to make prediction of new/unseen data – that is, the test set. And, indeed, the minimum error for the test set (i.e., the minimum of the generalization error) is located within the region where the validation error is minimum (see Figure 14).



Figure 14: Training (left), validation (middle), and test (right) mean squared errors at varying of the training set size and the polynomial degree.

An alternative to the above laborious evaluation of the model performances at varying of the training set size consists of using Scikit-Learn's *k*-fold cross-validation^{††}. The code below, considering a polynomial regression of degree six, splits randomly the training set into five distinct folds, then it trains and evaluates the model five times, picking a different fold for evaluation every time and training on the other four folds.

^{††}Note that cross_val_score expects an objective function to be maximized, so the scoring function is actually the opposite of the MSE.

The result is an array containing the five evaluation scores:

Mean scores: 0.05772081499906291 Standard deviation scores: 0.0207975583120194

The sixth-degree polynomial model has a MSE of approximately 0.057 ± 0.021 and we would have missed such information if we have just used one validation set.

6 Regularization techniques

6.1 Over- and under-determined systems

In linear and nonlinear regression, we are often confronted with a system of equations, either (4) or (17), that is under- or over-determined [Gen12]. To investigate these scenarios, let us rewrite equation (10) as

$$\mathbf{A}\boldsymbol{\theta} = \mathbf{b},\tag{19}$$

where $\mathbf{A} \equiv \mathbf{X}^{\mathrm{T}}$ is a matrix with *n* rows and *m* columns and $\mathbf{b} \equiv \mathbf{y}^{\mathrm{T}}$ is a matrix with *n* rows and *p* columns. Note that an analogous formulation can also be written for nonlinear regression (13).

With reference to equation (19), we define *overdetermined systems* those systems of equations that have more constraints than unknowns (variables) – i.e., when the matrix \mathbf{A} is tall-skinny – while *underdetermined systems* have more unknowns than constraints – i.e., \mathbf{A} is short-fat, see Figure 15.

Let us translate these notions from linear algebra to regression methods. If the number of data points is larger than the dimension of the independent variables (n > m), then we are dealing with an overdetermined systems of linear equations. Viceversa, if n < m, we are confronted with an underdetermined system. In the former case, there are generally no solutions satisfying the linear system, and instead, approximate solutions are found to minimize a given error. In the latter, there is an infinite number of solutions, and some additional constraints must be enforced in order to select an appropriate solution.

Note that in regression, we are most often dealing with overdetermined systems, even if sometimes overparametrization is preferred [BLLT20]. Analogous underdetermined systems are instead met when dealing with overparametrized artificial neural networks (cf. Chapter 7) [JNM⁺19].

Overdetermined or underdetermined optimization problems for linear systems (19) involve the minimization of the error of the solution, the MSE for instance, plus a constraint (n < m) or a penalty (n > m), which is also known as *regularization*, i.e.,

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\operatorname{MSE} \left(\mathbf{A}\boldsymbol{\theta} - \mathbf{b} \right) + \lambda \, w(\boldsymbol{\theta}) \right), \tag{20}$$

where λ is a weighting parameter and $w(\theta)$ a given function, depending whether the system is over- or under-determined. Below, we will see how ℓ_1 and ℓ_2 norm penalties/constraints can help in solving $\mathbf{A}\boldsymbol{\theta} = \mathbf{b}$.



Figure 15: Overdetermined (left) and underdetermined (right) system.

6.1.1 Overdetermined systems

When dealing with overdetermined systems, there exists no solution that satisfy $\mathbf{A}\boldsymbol{\theta} = \mathbf{b}$. Thus, penalties based on the ℓ_1 and ℓ_2 norms are often considered and the optimization problem (20) reads

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\operatorname{MSE} \left(\mathbf{A}\boldsymbol{\theta} - \mathbf{b} \right) + \lambda_1 ||\boldsymbol{\theta}||_1 + \lambda_2 ||\boldsymbol{\theta}||_2 \right),$$
(21)

where $||\boldsymbol{\theta}||_r = (|\boldsymbol{\theta}|^r)^{1/r}$ denotes the ℓ_r norm and the parameters λ_1 and λ_2 control the penalization of the ℓ_1 and ℓ_2 norms, respectively. The above penalty is crucial for obtain optimal approximate solutions, which are directly affected by the value of the penalty parameters.

To fix the ideas, consider an overdetermined system of equations. We draw **A** and **b** from normal distributions $\mathcal{N}[0, 1)$. We then obtain the parameters $\boldsymbol{\theta}$ by (i) using equation (10) and (ii) solving the optimization problem (21), considering two combinations of penalty parameters: $\{\lambda_1, \lambda_2\} = \{0.1, 0\}, \{0, 0.1\}.$

Figure 16 shows the obtained θ , as a $m \times p$ matrix. The key difference between the two regularization techniques and the ordinary solution without regularization is

that ℓ_1 penalty shrinks the parameters associated with less important variables to zero. Indeed, regularization based on the ℓ_1 norm promotes a parsimonious solution, dominated by zero entries, i.e., sparse. While, regularization based on the ℓ_2 norm keeps the solution values as small as possible. We could also combine both kinds of regularization to promote both sparsity and decrease in the value of the parameters.



Figure 16: Solution of an overdetermined linear system of equations obtained using the pseudoinverse (left), ℓ_1 (middle) and ℓ_2 (right) regularization.

6.1.2 Underdetermined systems

When dealing with undetermined systems, there exists an unlimited set of potential solutions that satisfy the equation $A\theta = b$. In such cases, the objective is to introduce constraints that would ideally result in a single, unique solution among the countless possibilities. Usually the optimization problem (20) is reformulated as

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(\operatorname{MSE} \left(\mathbf{A}\boldsymbol{\theta} - \mathbf{b} \right) + ||\boldsymbol{\theta}||_r \right), \tag{22}$$

where the r denotes the r-norm. Once again, one could use either ℓ_1 or ℓ_2 norm, or combination of them, as for the overdetermined case.

6.2 Regularized regression

It is now time to see how the aforementioned penalties and constraints for the solution of under- and over-determined systems can be employed in regression.

As already mentioned, (non)linear regression deals, most of the times, with overdetermined systems of equations – all examples we have looked up so far are indeed characterized by $n \gg m$. We should not be surprised, at this point, that the same penalty terms that are used for the solution of overdetermined systems have their own

counterparts (and names) in regression. Regularized regression models based on the ℓ_1 and ℓ_2 norms are called LASSO – Least Absolute Shrinkage and Selection Operator Regression –, and Ridge, respectively. The combination of both is often referred to as Elastic net.

6.2.1 LASSO

LASSO regression [Tib96] is the (non)linear regression version with a regularization term proportional to $||\theta||_1$. This forces the learning algorithm to not only fit the data but also shrink the model parameters to zero. The loss function is thus defined as

$$\mathcal{L}^{\text{LASSO}}\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) = \text{MSE}\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) + \lambda_1 ||\boldsymbol{\theta}||_1.$$
(23)

Note that the above loss function is not differentiable at $\theta = 0$, thus we cannot derive a closed form solution analogous to the normal equation. However, we can still use gradient descent provided that we define the following modified gradient, whenever θ is equal to zero,

grad
$$(\boldsymbol{\theta}, \mathcal{L}^{\text{LASSO}}) = \frac{\partial}{\partial \boldsymbol{\theta}} \text{MSE}(\mathbf{X}, f_{\boldsymbol{\theta}}) + \text{sign}(\boldsymbol{\theta}),$$
 (24)

where $sign(\cdot)$ is the sign function.

Below, we perform LASSO regression using Scikit-learn's for the same example we have seen for polynomial regression (cf. Section 5) and we consider polynomials of degree 100:

```
from sklearn.linear_model import Lasso
lambda_1 = 0.0001
lasso_reg = Lasso(alpha=lambda_1,max_iter=100000) # define lambda_1
# Polynomial feature up to degree 100
poly_features = PolynomialFeatures(degree=100, include_bias=False)
lasso_reg.fit(norm_X_poly,norm_y_train) # fit LASSO reg model
norm_x_poly = poly_features.fit_transform(norm_x)
norm_y_predicted_lasso = lasso_reg.predict(norm_x_poly) # make predictions
```

The predictions are shown in Figure 17, left. For the sake of clarity, the plot also shows the solution obtained without regularization (pinv). We can easily observe that LASSO allows to rediscover the underlying function, equation (18), used to generate the data, independently of the additive noise, and without overfitting – in contrast with the ordinary regression model (pinv), with all polynomial degrees up to 100. The reason is that LASSO shrinks most of parameters to zero, hence we discover a quite precise series expansion of the combination of sinusoidal functions within the range $(-\pi, \pi)$.

For the sake of completeness, note that we could also use Scikit-learn's SGDRegressor with the following arguments: penalty="l1" and alpha=lambda_1:

```
eta0=0.01)
sgd_reg.fit(norm_X_poly,norm_y_train.ravel())
norm_y_predicted_sgd = sgd_reg.predict(norm_x_poly) # make predictions
```

6.2.2 Ridge

Ridge regression [HK70] is the (non)linear regression version with a regularization term proportional to $||\theta||_2^2$ that forces the model parameters to be as small as possible. The loss function is defined as

$$\mathcal{L}^{\text{Ridge}}\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) = \text{MSE}\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) + \lambda_2 ||\boldsymbol{\theta}||_2^2.$$
(25)

As with linear regression – and differently from LASSO –, we can perform Ridge regression either by computing a closed-form equation [HK70] or by performing gradient descent. The code hereinafter implements the closed-form solution using Scikitlearn's Ridge:

```
from sklearn.linear_model import Ridge
lambda_2 = lambda_1
ridge_reg = Ridge(alpha=lambda_2,max_iter=100000) # define lambda_2
ridge_reg.fit(norm_X_poly,norm_y_train) # fit Ridge reg model
norm_y_predicted_ridge = ridge_reg.predict(norm_x_poly) # make predictions
```

We plot the predictions in Figure 17(middle). Once more, thanks to regularization, we are able to retrieve the original function, in contrast with the massively overfitted linear model (pinv).

6.2.3 Elastic net

Elastic net is the middle version between LASSO and Ridge. Accordingly, the regression model is regularized with respect to both the ℓ_1 and ℓ_2 norms. The regularization term is a mix of both Ridge and LASSO's controlled by a ratio ϱ :

$$\mathcal{L}^{\text{Elastic net}}\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) = \text{MSE}\left(\mathbf{X}, f_{\boldsymbol{\theta}}\right) + \varrho\lambda ||\boldsymbol{\theta}||_{1} + \frac{1-\varrho}{\varrho}\lambda ||\boldsymbol{\theta}||_{2}^{2}.$$
 (26)

Elastic net is equivalent to Ridge if $\rho = 0$ and to LASSO if $\rho = 1$. Below the code to define and fit an Elastic net model, whose predictions are shown in Figure 17, right:

7 Challenges in generalization and extrapolation

Regression, like other supervised ML methods, is mainly design for making predictions rather than determining existing relationships between some features. However,



Figure 17: Nonlinear regression using polynomials of degree 100. Comparison between standard regression (pinv) – equation (10) – and regularized models: LASSO and stochastic GD (left), Ridge (middle), Elastic net (right). Features are scaled.

by post-processing the model parameters θ , we can, under some circumstances, grasp the underlying relationships buried under the mere data points we are analyzing. In this sense, we may even end up (re)discovering some laws of physics, in the form of governing equations, see e.g. [CLKB19]. To do so, the obtained model needs to be *interpretable* – that is, capable of explaining why data behaves in a certain manner. An interpretable model has, in general, not only good generalization capabilities, but also good expressive ability (extrapolation) [KB22].

Think to the example in Section 5 involving data generated from a combination of trigonometric functions with additive noise. The example highlighted that regression is actually more nuanced than simply choosing a regression model and performing a least-square fit. The selection of the model itself is crucial for achieving better predictions and interpretable descriptions of the data.

In the example, we have seen that if we pick up an "optimal" polynomial degree, then we will obtain a model able to interpolate quite well the test data (cf. Figure 12, left). However, if the polynomial degree is too large (or too small), we will end up having a model that massively overfits (or underfits) the training data – that is, unable to generalize for new data points (cf. Figure 12, right). The ultimate solution was to resort to regularized regression models (e.g., LASSO). In doing so, we succeeded in identifying a good approximation of the true (trigonometric) model hidden under the pile of noisy data (cf. Figure 17).

However, why a regularized regression model seems to have much more predictive power than a straightforward nonlinear regression one, with as many polynomials terms as we are able to count? The answer lies in Occam's razor, a principle of parsimony (*"lex parsimoniae"*) attributed to William of Occam:

"Entia non sunt multiplicanda praeter necessitatem"

i.e. "entities must not be multiplied beyond necessity." In other words, among different competing models that make the same predictions, the simpler one is the more

likely. This principle is extensively used in science, where simpler explanations are preferred. An example is the method of dominant balance, which allows to determine the asymptotic behavior of solutions to an ordinary differential equation by identifying those terms that may be neglected in the limit. Dominant balance is, for instance, when we consider the evolution of shear stress in a sand sample, and we do not model neither its quantum state nor the effects of the relativistic warping of space-time caused by the grains.

Regularized regression adheres to this principle by discouraging the inclusion of numerous polynomial terms – or, in general, of features – thus promoting model simplicity. In addition, regularization helps in avoiding overfitting and results in high degree of stability. Indeed, high-dimensional polynomial models tend to suffer from instability due to multicollinearity that occurs when two or more independent variables have a high correlation with one another, see Figure 12, right.

While some of these benefits could also be obtained with low-degree polynomial models without the need to resort to regularization, these models tend to underfit the training data and exhibit insufficient expressive power. It is also worth mentioning that successful machine learning models often exhibit a certain degree of *benign overfitting* (also called *overparametrization*) [BLLT20]. Therefore, regularized regression models, with their ability to balance complexity and simplicity, offer significant predictive power by selecting relevant features.

In the example involving LASSO, Ridge, and Elastic Net regression (paragraph 6.2), we demonstrated the interpolating power of regularized regression models. However, it is important to further assess their interpretability and potential to provide insights for the identification and/or discovery of patterns and equations hidden within data sets.

7.1 Interpretable models and *where* to find them

This application deals with the investigation of whether it is possible or not to discover interpretable models (governing equations) with extrapolative power using (regularized) regression methods.

In particular, we mimic virtual experiments governed by a very simple equation and consider additive noise simulating virtual statistical errors in a hypothetical data acquisition system. The underlying governing equation is the (one-dimensional) projectile motion – i.e., the solution of the second-order ordinary differential equation $\ddot{x}(t) + g = 0$,

$$x(t) = x(0) + \dot{x}(0)t - \frac{1}{2}gt^2,$$

where x, \dot{x} , and \ddot{x} are the trajectory, velocity, and acceleration of the projectile, respectively; t is the time, and g is the gravitational acceleration. The code hereinafter integrates the differential equation with initial conditions: x(0) = 0 m and $\dot{x} = 5$ m/s. As we are interested in making predictions within the range of values of some training data and outside of it, we generate the data and split them in an interpolation (xi and yi) and extrapolation (xe and ye) sets, depicted in Figure 18(a):

Masi 61

```
import numpy as np
from scipy.integrate import solve_ivp
np.random.seed(42)
def projectile_motion(t, y):
    # equations of motion
    dydt = y[1] # velocity
    dvdt = -g # acceleration
   return [dydt, dvdt]
# Initial conditions
n_snapshots = 200
x0 = 0.0 # meters
v0 = 5.0 # meters per second
\mathbf{g} = 9.81 # acceleration due to gravity
t0 = 0.0 \# initial t
tf = 1.0 \# final t
# time points solution
t = np.linspace(t0, tf, n_snapshots)
# solve ODE
sol = solve_ivp(projectile_motion, [t0, tf], [x0, v0], t_eval=t)
# define independent and dependent variables
X = np.float32(t)
y = np.float32(sol.y[0])
n = n_snapshots//2
xi = X[:n] # train time
xe = X[n:2*n] # test time
ytrain = y[:n] # train x = [0, 0.5]
ytest = y[n:2*n] # test x = [0.5,1]
```

We continue by training a set of n polynomial regression models of degree 20. Every model is trained against dependent variables lying between 0 and 0.5 (xi) and dependent variables (yi) with a normally distributed measurement noise, randomly selected at each loop (fni). In a nested for loop, we store the MSE values between the models predictions and yi for values of the independent variables within the interpolation (mse_int) and extrapolation (mse_ext) range.

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
degree = 20 # polynomial
mse_int = np.zeros((n,degree))
mse_ext = np.zeros((n,degree))
poly_features = PolynomialFeatures(degree=degree, include_bias=False)
xi_poly = poly_features.fit_transform(np.expand_dims(xi,1))
xe_poly = poly_features.fit_transform(np.expand_dims(xe,1))
for deg in range(degree):
   xi_poly_ = xi_poly[:,:deg+1]
    xe_poly_ = xe_poly[:,:deg+1]
    for j in range(n):
        fni = ytrain + 0.3*np.random.normal(0,1,n) # y with normal noise
        lin_reg = LinearRegression() # model
        lin_reg.fit(xi_poly_,fni) # fit for x in [-0.5,1]
        ynai = lin_reg.predict(xi_poly_)
        ynae = lin_reg.predict(xe_poly_) # fit for t in [0.5,1]
        mse_int[j,deg] = mean_squared_error(ytrain,ynai) # MSE train
```

mse_ext[j,deg] = mean_squared_error(ytest,ynae) # MSE val

Figure 18 shows the MSE of the set of models in the interpolation and extrapolation ranges. The results are clear: all the trained models massively overfit the data with measurement noise and cannot generalize. When attempting to extrapolate beyond the range observed in the training data, we are confronted with substantial errors, as shown in Figure 18(c-d). The logarithmic plot in Figure 18(d) illustrates the exponential growth of the error as the polynomial degree increases, reaching magnitudes as high as 10^{10} . This demonstrates the inability of overfitted models to make predictions in the extrapolation range. Thus, we can say that (non-regularized) polynomial regression does not offer an interpretable model for this simple case (except an obvious second-order polynomial).



Figure 18: Regression of the motion of a projectile with measurement noise (a) using 100 different nonlinear regression models with polynomial basis (degree 20). Errors for values within the training, interpolation range (b), and outside (extrapolation) (c-d).

Let us continue by employing cross-validation to mitigate the effects of overfitting. We also extend the set of polynomial regression models – denoted with pinv, diminutive of pseudoinverse – with regularized ones. The code below performs 100-fold cross-validation for the same noisy data sets for the three different regression models (pinv, LASSO, and Ridge) and stores the obtained parameters. Note that, as the measurement noise is randomly select for each k-fold, the training data points may be considered as virtual experimental measurements of the simplified system.

```
from sklearn.linear model import Lasso.Ridge
from sklearn.model_selection import KFold
nfolds = 100
lin_reg = LinearRegression() # linear model
lasso = Lasso(alpha=3e-4) # LASSO model
ridge = Ridge(alpha=3e-4) # Ridge model
k_fold = KFold(nfolds)
# initialize for storing parameters
lasso_params = np.zeros((nfolds,degree+1))
ridge_params = np.zeros((nfolds,degree+1))
lin_params = np.zeros((nfolds,degree+1))
for k, (train, test) in enumerate(k_fold.split(xi_poly, ytrain)):
    fni = ytrain + 0.3*np.random.normal(0,1,n) # y with normal noise
    lin_reg.fit(xi_poly[train], fni[train]) # fit lin
    ridge.fit(xi_poly[train], fni[train]) # fit RIdge
    lasso.fit(xi_poly[train], fni[train]) # fit LASSO
     # store models parameters
    lin_params[k,0] = lin_reg.intercept_
    lin_params[k,1:] = lin_reg.coef_
    ridge_params[k,0] = ridge_intercept_
    ridge_params[k,1:] = ridge.coef_
    lasso_params[k,0] = lasso.intercept_
lasso_params[k,1:] = lasso.coef_
```

Figure 19 plots the values of the parameters θ obtained from the above code, for *k*-th fold (k = 2, 10, and 100). Note how the parameters of the standard regression model explode, with values of the order of 10^5 , independently of the number of folds. The phenomenon is intrinsic to polynomial regression which blows up as the polynomial degree is increased and further enhanced by the intrinsic multicollinearity of the (polynomial) features. In contrast, Ridge and LASSO identify parameters with ranges of values comparable with underlying equation of motion. In addition, LASSO also successfully identifies a parsimonious model, with a (more or less) predominant quadratic term (depending on the number of folds).

In order to proceed to the evaluation of the three cross-validated models, we compute the average of the parameters obtained from the optimization performed at each fold (cf. paragraph 5.1.2):

```
# Compute models based on the mean over the 100-fold cross-validation
lin_theta = np.mean(lin_params[:,:],0)
lasso_theta = np.mean(lasso_params[:,:],0)
ridge_theta = np.mean(ridge_params[:,:],0)
# Set mean params for each model
lin_reg.coef_ = lin_theta[1:]
```



Figure 19: Values of the parameters θ obtained using standard (pinv), LASSO, and Ridge polynomial regression for different cross-validation folds: k = 2 (left), k = 10 (middle) and k = 100 (right).

```
lin_reg.intercept_ = lin_theta[0]
ridge.coef_ = ridge_theta[1:]
ridge.intercept_ = ridge_theta[0]
lasso.coef_ = lasso_theta[1:]
lasso.intercept_ = lasso_theta[0]
# Predictions within interpolation range
fy_lin_xi = lin_reg.predict(xi_poly)
fy_rid_xi = ridge.predict(xi_poly)
# Predictions within extrapolation range
fy_la_xe = lasso.predict(xe_poly)
fy_lin_xe = lin_reg.predict(xe_poly)
fy_rid_xe = ridge.predict(xe_poly)
```

Figure 20 compares the performances of the three models (pinv, LASSO, Ridge) in terms of the mean squared error, while Figure 21 shows the predictions, in the interpolation and extrapolation range. As we might have expected, LASSO excels over all other strategies by providing a model that can interpolate (without overfitting) and, even, extrapolate with good accuracy. We can also observe that Ridge performs better than the standard regression approach, although it rapidly fails in the extrapolation range.

Whilst good generalization and extrapolation capabilities, LASSO has not identified the exact underlying governing equation but just a very good approximation of it,

 $\hat{x}(t) \approx \theta_1 t + \theta_2 t^2 + \theta_3 t^3, \qquad \theta_1 = 4.33, \quad \theta_2 = -2.93, \quad \theta_3 = 1.32.$

Masi 65



Figure 20: Errors of the cross-validated regression models: pinv, LASSO, Ridge within the: (a) interpolation range and (b-c) extrapolation range, where (c) presents a detailed view.

Note that the true equation has $\theta_1 = 5$, $\theta_2 = -4.9$, and $\theta_3 = 0$. This implies that if we ask the model to make predictions for a significantly distant time point, i.e., $t \gg 1$, the accuracy of the extrapolation will not be as good as in the range that we considered here.



Figure 21: Comparison between the projectile motion equation and the predictions of the cross-validated models: piny, LASSO and Ridge.

8 Bayesian regression

All regression methods discussed up to this point fall under the hat of the *frequentist* approaches [Wak13] as they assume that there are enough measurements to say something meaningful about θ and provide a sigle best estimate for a given training set. Such methods provide good models whenever the relationship between the independent and dependent variables is well understood and relatively stable. However, in real case scenarios, we are often confronted with uncertainty or variability in the data and

deterministic methods are not appropriate. To this end, a powerful, alternative tool is provided by Bayesian regression methods.

Bayesian regression is able to provide predictions in the form of probability distributions incorporating uncertainty estimates. This Section introduces the main ideas and novelties characterizing such methodologies as compared to frequentist approaches by focusing on linear Bayesian regression and Gaussian process regression. For more details, we refer to [BN06, Mur18].

8.1 Linear Bayesian regression

In simple linear regression, we aim to find the best-fit line that explains the relationship between two variables. Linear Bayesian regression extends this same concept by considering uncertainty in both the model parameters and the predictions. Instead of relying on a single line (cf. Section 2), it thus provides a distribution of possible lines, accounting for the uncertainty in the data.

Linear Bayesian regression starts by assuming a linear relationship between the independent variables \mathbf{X} and the dependent variable \mathbf{y} of the form

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\epsilon},\tag{27}$$

where θ are the model parameters and ϵ represents an error term assumed to follow a zero-mean Gaussian distribution, $\epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$, where \mathbf{I} is the identity matrix and σ^2 is the variance.

8.1.1 Bayesian inference

Bayesian inference allows to infer a posterior distribution $p(\theta | \mathbf{X}, \mathbf{y})$ – that of the parameters – representing updated beliefs given the observed independent variables **X** and dependent variable **y**. To this end, we use Bayes' formula, i.e.,

$$p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y}) \propto p(\mathbf{X}, \mathbf{y}|\boldsymbol{\theta}) p(\boldsymbol{\theta}),$$
 (28)

where $p(\theta)$ is the prior distribution over θ , incorporating prior beliefs about the parameters values before observing any data, while $p(\mathbf{y}|\mathbf{X}, \theta)$ is the likelihood function, quantifying instead the probability of observing the dependent variable \mathbf{y} given the independent variables \mathbf{X} and the parameters θ .

8.1.2 Prior and posterior

In principle, any prior $p(\theta)$ could be used, however the functional form of most priors, when multiplied by the functional form of the likelihood in (28), results in an posterior $p(\theta|\mathbf{X}, \mathbf{y})$ with no closed-form solution.

In such scenarios, the solution is to resort to approximate Bayesian inference techniques as Monte Carlo methods. But, certain priors are mathematically convenient because they result in posteriors with tractable, well-known densities.
The simplest and most widely used version of linear Bayesian regression is the *normal linear model*, in which y given $\mathbf{X} - p(\mathbf{X}, \mathbf{y}|\boldsymbol{\theta})$ – is distributed Gaussian. In this model, and under a particular choice of the prior for the parameters – namely, when the prior is conjugate, meaning that the prior and posterior share the same functional form – the posterior can be found analytically. With more arbitrarily chosen priors, the posteriors generally have to be approximated.

8.1.3 Posterior predictive

Once we have computed the posterior distribution $p(\theta | \mathbf{X}, \mathbf{y})$, we can make predictions for new, unseen data points. One common approach is to generate posterior predictive samples by drawing parameter values from the posterior distribution and using them to make predictions [BN06].

8.1.4 Example

Let us consider an intuitive example drawn from [BN06] and implemented as in gwgundersen/bayesian-linear-regression. We start by generating 50 snapshots of the independent and dependent variables, according to

$$y = \theta_1 \mathbf{x} + \theta_0,$$

where $\theta_1 = 0.5$ and the bias is $\theta_0 = -0.7$. Then, we perform Bayesian linear regression using as prior a normal-inverse-gamma distribution and consider an increasing number of observations in the training data set.

Figure 22 depicts the evolution of the prior and posterior distributions of the parameters $\boldsymbol{\theta}$ (left column) and six random posterior samples drawn from the vector $\boldsymbol{\theta} = [\theta_0, \theta_1]^T$ (right column). In the top row, the model has seen no data. The prior places high and equal probability on both θ_0 and θ_1 being zero. In the subsequent rows, the model is fit to more data. With more observations, the model inferred posterior variance decreases, and the realizations of $\boldsymbol{\theta}$ from the posterior become more constrained and in agreement with the observed data.

8.2 Gaussian process regression

Before discussing about Gaussian process regression, it is important to distinguish *parametric* from *nonparametric* models and recall the definition of a Gaussian process.

Parametric versus nonparametric model Parametric models are all those regression models we have seen so far. More formally, parametric models assume a well-defined functional relationship between dependent and independent variables and that the distribution of the data can be entirely defined in terms of a finite set of parameters θ . Thus, once we have trained a parametric model, every future prediction is independent of the particular set of new data for which we are making predictions.



Figure 22: Evolution of the prior and posterior distributions of the parameters $\boldsymbol{\theta}$ (left column) and random samples of the vector $\boldsymbol{\theta} = [\theta_0, \theta_1]^T$ (right column) at varying of the number of observation (rows).

Nonparametric models do not prescribe any predetermined functional relationship (or hypothesis f_{θ}) between the dependent and independent variables [HM93]. Nonparametric models thus have the freedom to calculate the probability distribution over all admissible functions that fit the data, rather than, for instance, calculating the probability distribution of parameters of a specific function (cf. linear Bayesian regression). However, even in nonparametric models, we must specify a prior (on the function space), calculate the posterior using the training data, and compute the predictive posterior distribution on the points of interest.

Gaussian process A Gaussian process (GP) is a collection of random variables, any finite number of which has a joint Gaussian distribution. In the context of regression, we can think of a GP as an infinite-dimensional generalization of a multivariate Gaussian distribution. Instead of representing a mean vector and a covariance matrix, a GP is fully characterized by a mean function, denoted as $\mu(\mathbf{x})$, and a covariance function, also known as the kernel function, denoted as $k(\mathbf{x}, \mathbf{x})$, describing the relationship between any given two data points \mathbf{x}_i and \mathbf{x}_j , for $i, j = 1, 2, \dots, m$.

8.2.1 Inference

Gaussian process regression (GPR) is a nonparametric, Bayesian regression method. In particular, GPR uses a Gaussian process prior to infer the distribution of possible functions that could generate the observed data.

To fix the ideas, consider a set of training data $\{\mathbf{X}, \mathbf{y}\} \equiv \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, the objective is to estimate the function $f(\mathbf{X})$ that minimizes $\mathbf{y} - f(\mathbf{X}) = 0$ by placing a Gaussian process over it. Typically, a mean of zero is assumed, leading to the following distribution

$$\mathbf{y} = f(\mathbf{X}) \sim \mathcal{N}(\mathbf{0}, \mathbf{K}),\tag{29}$$

where $\mathbf{K} \equiv k(\mathbf{X}, \mathbf{X})$ denotes the covariance matrix generated by the kernel function k chosen (e.g. periodic, linear, radial basis function) and describes the general shape of $f(\cdot)$. Note that the choice of the kernel is crucial in identifying a good (or a bad) fit of the data.

To predict the values of a new set of dependent variables \mathbf{y}_* , given a new set of independent variables \mathbf{X}_* , we need to estimate the conditional distribution $p(\mathbf{y}_*|\mathbf{X}_*, \mathbf{X})$. According to the properties of GP [Mur18], the joint distribution of the training and test outputs follows a joint Gaussian distribution:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{y}_* \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x} \\ \mathbf{x}_* \end{bmatrix} \right) \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^{\mathrm{T}} & \mathbf{K}_{**} \end{bmatrix} \right), \tag{30}$$

where $\mathbf{K}_* \equiv k(\mathbf{X}, \mathbf{x}_*)$, $\mathbf{K}_{**} \equiv k(\mathbf{x}_*, \mathbf{x}_*)$. While the above joint distribution gives some insight as to how $f(\mathbf{x}_*)$ relates to $f(\mathbf{x})$, at this point no prediction for the new datum \mathbf{x}_* .

To obtain the posterior distribution of the predicted GP realizations $f(\mathbf{x}_*)$, we condition the prior distribution on the training data [Mur18]. This leads to the following mean and covariance predictions for the test point \mathbf{x}_* (see [Mur18] for the detailed derivation):

$$\mu(f(\mathbf{x}_*)) = \mathbf{K}_*^{\mathrm{T}} \mathbf{K}^{-1} \mathbf{y},$$

$$\operatorname{Cov}(f(\mathbf{x}_*)) = \mathbf{K}_{**} - \mathbf{K}_* \mathbf{K}^{-1} \mathbf{K}_*^{\mathrm{T}}.$$
(31)

These equations provide the mean and covariance predictions for the output $f(\mathbf{x}_*)$ given the training data set and a new input \mathbf{x}_* .

8.2.2 Inference in the presence of noise

In real-world scenarios, we often encounter observations with measurement noise. In such scenarios, it is more appropriate to model the training targets \mathbf{y} to be noisy realizations of a Gaussian process $f(\mathbf{X})$,

$$\mathbf{y} = f(\mathbf{X}) \sim \mathcal{N}(\mathbf{0}, \mathbf{K}) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{32}$$

where the noise ϵ is parameterized by a zero-mean Gaussian with positive noise covariance values given by σ^2 , which is a hyperparameter.

By repeating the steps above, we obtain the following joint distribution of the training and new outputs

$$f\left(\begin{bmatrix}\mathbf{x}\\\mathbf{x}_*\end{bmatrix}\right) \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix}\mathbf{K} + \sigma^2 \mathbf{I} & \mathbf{K}_*\\\mathbf{K}_*^{\mathrm{T}} & \mathbf{K}_{**}\end{bmatrix}\right).$$
(33)

As before, but now with the noise term, we can obtain the predictive mean and variance, given by

$$\mu(f(\mathbf{x}_*)) = \mathbf{K}_*^{\mathrm{T}} \left(\mathbf{K} + \sigma^2 \mathbf{I} \right)^{-1} \mathbf{y},$$

$$\operatorname{Cov}(f(\mathbf{x}_*)) = \mathbf{K}_{**} - \mathbf{K}_* \left(\mathbf{K} + \sigma^2 \mathbf{I} \right)^{-1} \mathbf{K}_*^{\mathrm{T}} + \sigma^2.$$
(34)

8.2.3 Example

Let us see how we can perform GRP using Scikit-learn. We consider the same polynomial regression example in Section 5, with data generated from a combination of trigonometric functions with random additive noise. In the code hereinafter, we first fit a Gaussian process on the training data using a radial basis function (RBF) kernel and, additionally, a noise parameter (alpha). Then, we use the kernel to compute the mean predictions (norm_y_mean) for the train and test set and plot the 95% confidence interval (defined as norm_y_mean ± 1.96 norm_y_std).

The predictions are depicted in Figure 23 and compared with the training and test data. Not only the fit in terms of the expected (mean) value is extremely good, but in addition, due to the intrinsic probabilistic nature of GPR, we are also able to compute confidence intervals and thus quantify uncertainties.

In summary, we have seen that Bayesian regression methods offer unique advantages as compared to classical, frequentist ones as far it concerns uncertainty estimation.

Masi 71



Figure 23: Gaussian process regression (with noise) for data generated from a combination of trigonometric functions with random additive noise, see paragraph 5.1: predictions and 95% confidence interval.

However, there exist some drawbacks associated with the former. First, both linear Bayesian and Gaussian process regression have high computational complexity – higher than that of deterministic methods – and may be computationally expensive, especially when dealing with large data sets and not only at training but also at inference. Second, we have seen that deterministic regression methods allow to identify interpretable models that directly relate to the impact of each parameter on the dependent variable predictions – especially in the case of regularized strategies (cf. Section 7). However, this is not usually true for linear Bayesian and Gaussian process regression: indeed, the parameters are expressed in the form of posterior distributions or covariance matrices and their interpretation can be quite challenging.

9 Conclusions

This Chapter provided a comprehensive introduction to regression methods, which serve as one of the foundational pillars of Machine Learning, together with classification methods (cf. Chapters 1 and 4).

We covered linear and nonlinear regression models and their Bayesian counterparts in Sections 2, 5, and 8, respectively. Together, such statistical tools allow to identify relationships between inputs and outputs and deliver predictive models. To do so, the starting point consists of defining a hypothesis – in linear and nonlinear regression – or a kernel – in Gaussian process regression. Then, the process continues with the identification of the hypothesis parameters – in the form of fixed values or posterior distributions – or functionals, obtained through optimization strategies (gradient descent) and/or as closed-form solutions, to finally obtain a predictive model through learning from a training data set.

We also introduced concepts and strategies related to model validation and generalization, as well as features scaling (cf. Section 4). These are extremely important and useful not only for regression methods but, in general, for any Machine Learning approach – e.g., classification methods, artificial neural networks, and dimensionality

reduction techniques.

Then, we emphasized the significance of regularization, particularly in the context of nonlinear regression (cf. Section 6). Regularization techniques play a crucial role in preventing overfitting, where the model becomes too complex and fits noise or irrelevant patterns, leading to poor generalization. Note that regularization strategies go well beyond regression models and are a good ally for enhancing the performance and generalization abilities of Machine Learning models. Additionally, regularization facilitates the identification of interpretable models by promoting simplicity and a small number of dominant parameters. Interpretable models have the potential to reveal hidden physical relationships or equations that may be obscured within vast amounts of data (cf. Section 7).

However, we should pay attention in understanding the limitations of (non) regularized regression models when addressing complex phenomena. For instance, if we were asked to predict occurrences and magnitudes of earthquakes and landslides or characterize the constitutive behavior of intricate materials like sand or clay, the interpretable, but simplistic, approach developed in the case of the projectile motion (cf. paragraph 7.1) will not be quite effective. The reason lies in challenges proper to the above scenarios and related to an intrinsic chaotic nature and a high-dimensional state space.

Every time we are confronted with the forecasting of a complex phenomenon or the description of an intricate system achieving reliable generalization and extrapolation performance becomes increasingly arduous when relying solely on Machine Learning approaches that lack any knowledge bias. And this even in the case of sophisticated neural networks that use regularization strategies (cf. Chapter 7). To address such challenges, it becomes imperative to resort to physics-informed and thermodynamics-based machine learning models (as explored in Chapters 7 and 9) or leverage data-driven computing approaches (presented in Chapter 5). These alternative methodologies incorporate domain knowledge and physical principles into the learning process, resulting in more robust and accurate predictions.

References

- [BLB⁺13] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In ECML PKDD Workshop: Languages for Data Mining and Machine Learning, pages 108–122, 2013.
- [BLLT20] Peter L Bartlett, Philip M Long, Gábor Lugosi, and Alexander Tsigler. Benign overfitting in linear regression. *Proceedings of the National Academy of Sciences*, 117(48):30063–30070, 2020.

- [BN06] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.
- [BV04] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- [CLKB19] Kathleen Champion, Bethany Lusch, J Nathan Kutz, and Steven L Brunton. Data-driven discovery of coordinates and governing equations. *Proceedings of the National Academy of Sciences*, 116(45):22445–22451, 2019.
- [Gen12] James E Gentle. *Numerical linear algebra for applications in statistics*. Springer Science & Business Media, 2012.
- [Gér22] Aurélien Géron. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. "O'Reilly Media, Inc.", 2022.
- [HK70] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.
- [HM93] Wolfgang Hardle and Enno Mammen. Comparing nonparametric versus parametric regression fits. *The Annals of Statistics*, pages 1926–1947, 1993.
- [JNM⁺19] Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. Fantastic generalization measures and where to find them. *arXiv preprint arXiv:1912.02178*, 2019.
- [KB22] J Nathan Kutz and Steven L Brunton. Parsimony as the ultimate regularizer for physics-informed machine learning. *Nonlinear Dynamics*, 107(3):1801–1817, 2022.
- [Man82] John Mandel. Use of the singular value decomposition in regression analysis. *The American Statistician*, 36(1):15–24, 1982.
- [Mur18] Kevin P Murphy. Machine learning: A probabilistic perspective (adaptive computation and machine learning series), 2018.
- [Tib96] Robert Tibshirani. Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1):267–288, 1996.
- [Wak13] Jon Wakefield. *Bayesian and frequentist regression methods*, volume 23. Springer, 2013.
- [WT] Torsten Wichtmann and Theodoros Triantafyllidis. An experimental database for the development, calibration and verification of constitutive models for sand with focus to cyclic loading: part I—tests with monotonic loading and stress cycles.

Unsupervised Learning: Basic Concepts and Application to Particle Dynamics

Noel Jakse

Université Grenoble Alpes, CNRS, Grenoble INP, SIMaP, 38000 Grenoble, France

1 Introduction

Machine learning (ML) approaches attracted significant interest in many scientific fields due to their potential ability to uncover patterns, make predictions, and extract valuable insights from large and complex body of data [EH21]. Web-based data collection platforms, scientific instruments and computer simulations are creating exponentially increasing data stores. they triggered new scientific methods to analyze and organize huge amount of data, giving the possibility to find subtle effects missed previously [HT20, HTTG09].

ML tools can be broadly categorized into three types: supervised, unsupervised, and semi-supervised learning. Each type offers distinct approaches to learning from data, with specific strengths and limitations [EH21, HTFF09]. Supervised learning is one of the most used approach and involves training a model using labeled data, where both input features and corresponding output labels are provided. The model learns to map inputs to their respective outputs, generalizing to make predictions for new, unseen data. Applications of supervised learning include regression and classification tasks that will be be the subject of Chapters 2 and 4 of the present Book.

However, obtaining labeled data can be time-consuming and expensive, especially for large datasets. Unsupervised learning, on the other hand, works with unlabeled data, where the model discovers patterns and structures within the data without prior knowledge of the desired output. It is particularly useful in exploratory data analysis, clustering, dimensionality reduction, and anomaly detection. Finally, semi-supervised learning combines aspects of both supervised and unsupervised learning, leveraging a small amount of labeled data with a large amount of unlabeled data. The method is based on unsupervised techniques to extract features from the unlabeled data, and supervised learning is then applied to improve the trained model using the labeled data. This approach often results in improved model performance compared to purely

supervised or unsupervised learning, especially when labeled data is limited.

Unsupervised learning, at the heart of the present Chapter, is useful in situations where labeled data are scarce, enabling insights from larger datasets. It can be useful to deal with large datasets of unlabelled instance, and is particularly relevant when hidden patterns and structures in data have to be extracted without prior knowledge on any label or output. It plays a vital role in scientific fields such as materials science [Cer19, SMBM19] and geophysics [WT10, KA21], where labeled data are often unavailable, scarce or expensive to calculate or to measure. It enables to analyze large datasets and identify essential relationships, leading to the discovery of new materials or the understanding of complex geological processes.

The Chapter is mainly divided in two parts. The first part is devoted to the comprehensive description of the basic concepts and most popular techniques of unsupervised learning. For the latter, the choice was made to describe two main branches, which are the Clustering and Dimensionality Reduction techniques [HTFF09]. Unsupervised learning has numerous real-world applications in many domains. As an example of applications, the second part of this Chapter illustrates an application of unsupervised learning to the discovery of patterns in particles dynamics. A particular focus is made on large scale molecular dynamics simulations performed with up to 10 million atoms for the purpose of describing the early stages of solidification of a materials so-called Homogeneous nucleation [SCC⁺16, BDMJ22b]. While specific to material science, this example can surely transposed at higher scales with Discrete Particle Dynamics (DPD) modelling in geophysics, which is at the heart of this Book.

2 Basic concepts

2.1 Representation of the data: Feature extraction and selection

Feature extraction and representation are fundamental aspects of unsupervised learning as in most of the ML techniques. The basic idea sketched in Fig. 1 is to transform raw data into a format, often a vector, that can be easily processed and analyzed by ML algorithms, such a representation is so-called hereafter a descriptor. Having say that, the major objective of this transformation is then to identify and extract the most relevant and informative features from the data, intending to capture the underlying patterns and structures while reducing noise and redundancy.

Feature extraction techniques can be divided into two categories: feature construction and feature selection [KKN14]. As mentioned above, feature construction involves creating new features from the original data, often through mathematical transformations or domain-specific knowledge. For example, in image processing, features can be extracted using edge detection or texture analysis. In text analysis, natural language processing techniques like tokenization, stemming, and term frequency-inverse document frequency (TF-IDF) can be employed to extract meaningful features [NQY18] in the so-called Bag of Words representation. In the application treated in the present Chapter regarding atomic scale simulations of homogeneous crystal nucleation, a topological



Figure 1: Typical machine learning flowchart that learns a predictive model from the data.

descriptor is built [BDMJ22a] based on persistence homology (PH) in the framework of the topological data analysis (TDA) [PSG⁺18, Car20].

This poses naturally the question of the dimensionality of the descriptor after extraction. High-dimensional feature spaces can pose challenges for unsupervised learning algorithms, as the increase in dimensions can lead to increased computational complexity, reduced model interpretability, and over-fitting. This phenomenon is known as the "curse of dimensionality" [HTFF09, EH21]. Feature selection addresses this problem by selecting a subset of the original features that are most relevant to the task at hand, in order to reduce the dimensionality of the data. There are three primary methods for feature selection among others. Filter methods evaluate the relevance of individual features based on their statistical properties, independent of any learning algorithm. Examples of filter methods use a specific learning algorithm's performance to guide the feature selection process, by iteratively adding or remove features based on their impact on the model's performance (*i.e.* forward selection, backward elimination, and recursive feature elimination).

Finally, embedded methods integrate feature selection into the model training process, combining the advantages of both filter and wrapper methods. One advantage is the assessment the importance of features during the training phase, automatically selecting the most relevant ones. Emblematic examples of embedded methods include LASSO,

ridge regression, and decision trees.

2.2 Distance and similarity metrics

In order to quantify the relationship between data points in the feature space of dimension D, defining metrics is at the heart of unsupervised learning. These metrics provide a basis for comparing and grouping similar instances in clustering.

The most widely used distance metric, measuring the straight-line distance between two points in Euclidean D-dimensional feature space. Considering two points p and q, it is given by

$$d(p,q) = \sqrt{\sum_{i=1}^{D} (p_i - q_i)^2}.$$
(1)

This applies for continuous features and is sensitive to the scale of the data. A simpler version is the Manhattan distance, also known as the L_1 -norm, which measures the sum of the absolute differences namely

$$d_{L_1}(p,q) = \sum_{i=1}^{N} |p_i - q_i|,$$
(2)

and is less sensitive to outliers compared to Euclidean distance. When the dimensionality of the feature space is high or when the magnitude of the vectors is less relevant, such as in text analysis, the cosine similarity metrics is relevant and measures the angle between two feature vectors \mathbf{p} and \mathbf{q} as

cosine similarity(
$$\mathbf{p}, \mathbf{q}$$
) = $\frac{\mathbf{p} \cdot \mathbf{q}}{\|\mathbf{p}\| \|\mathbf{q}\|} = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} \sqrt{\sum_{i=1}^{n} q_i^2}}.$ (3)

If sets of data have to be compared, the Jaccard similarity metrics can be used for comparing binary or set-based data, measuring the ratio of the intersection to the union of two sets A and B is given by:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$
(4)

Alternatively, Pearson correlation coefficient measures the linear relationship between two variables of size n, ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation) with

$$r = \frac{\sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n} (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^{n} (Y_i - \bar{Y})^2}}.$$
(5)

These are the most common ones, however the metrics to be used should be decided given the specificity of the data at hand. The choice can affect the results of the learning.

3 Unsupervised Learning Techniques

Unsupervised learning comprises a diverse range of methods and algorithms dealing with unlabeled data [EH21]. This section provides an overview of some popular unsupervised learning techniques and their applications, focusing only on *clustering* and *dimensionality reduction* methods. A corresponding, comprehensive tutorial through a JUPYTER notebook attached to this Chapter will explore the most common ones.

What will not be treated in the present Chapter is the combination of powerful deep learning models with unsupervised learning techniques [HWWT13]. These so-called *deep unsupervised techniques* are Autoencoders and Variational Autoencoders (VAEs), Generative Adversarial Networks (GANs), Self-supervised Learning and Deep Clustering.

3.1 Clustering

Clustering is the grouping process of data points based on their features. Many clustering methods were already proposed [HTFF09]. The key notion is the degree of similarity (or dissimilarity) between the individual objects being associated. Indeed, the method attempts to group observations with respect to some similarity criterion. Combinatorial methods were proposed to avoid a modeling through a probability distribution function. However, an exact solution is computable only when the dataset is very small. As a matter of fact, testing every combination of 20 observations into 4 clusters needs 10^{10} comparisons, which is basically unreachable).

Approximations of these methods, were proposed through iterative process. The famous K-means method is a standard combinatorial algorithm, based on the Euclidean distance given by Eq. (1). A standard version of the algorithm given in Al. 1. Given a number K of points in the dataset of size $n X = \{x_1, x_2, \ldots, x_n\}$ are randomly denoted as the initial K centroids, and each observation is assigned to the cluster with the closest centroid. Then, the centroids are updated, and the process is iterated. As an illustration, Figure 2 schematize the application of the K-means to as dataset randomly created as a collection of three distinct blobs ¹ as a guide for the eyes, and with the same color to indicate the fact that the data are unlabelled.

Adding a probabilistic view on this modeling with a soft assignment of each observation to each cluster, leads to Gaussian Mixture Models, with spherical covariance matrices, proportional to the identity matrix. The idea behind is that the data distribution can be well approximated by a mixture of K multivariate Gaussian distributions:

$$\sum_{k=1}^{K} \pi_k \Phi(\mu_k, \Sigma_k), \tag{6}$$

¹see Scikit-learn blobs

Algorithm 1 K-means

Require: Data points $X = \{x_1, x_2, ..., x_n\}$, number of clusters K **Ensure:** Cluster assignments $C = \{c_1, c_2, ..., c_K\}$, centroids $\mu = \{\mu_1, \mu_2, ..., \mu_K\}$ Initialize centroids $\mu_1, \mu_2, ..., \mu_K$ randomly from X **repeat** Assign each point x_i to the nearest centroid: $c_i = \arg \min_k ||x_i - \mu_k||^2$ Update centroids by computing the mean of assigned points: $\mu_k = \frac{\sum_{i=1}^{n} I(c_i=k)x_i}{\sum_{i=1}^{n} I(c_i=k)}$ **until** Convergence (centroids do not change significantly or a maximum number of iterations is reached)

with Φ the multivariate Gaussian distribution, where the cluster $k \in \{1, ..., K\}$ is described by its proportion π_k among the full dataset, its mean μ_k and its covariance matrix Σ_k . Estimation of parameters is classically done using the Expectation Maximization (EM) algorithm [DLR77].

The main issue of most clustering methods is the number of clusters, that has to be set beforehand by a human. To circonvent the arbitrariness of choosing a number of cluster, and when there is a probability density function, the likelihood should be considered. It consists in measuring how close the samples are to the specified clustering distribution. However, this criterion will always be better for a higher number clusters, which lead to the classical problem of over-fitting. If the well-known elbow criterion was largely used to penalize such criterion, it is not related to any theoretical background, and is always seem very objective. Model selection criterion can be used, adapting the likelihood by penalizing it with respect to the dimension (related to the number of clusters) of each model. Usually, Akaïke Information Criterion (AIC) [Aka73] or Bayesian Information Criterion (BIC) [Sch78] are proposed, but the Integrated Completed Likelihood ICL [BCG00] should be preferred when focusing on clustering, because there is also a penalty term about the purity of each cluster based on a Shanon entropy term.

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [EKSX96] is another famous clustering algorithm, known to be more robust. Observations lying in low-density regions are labeled as outliers. No assumptions on the form of the clusters is done, allowing for convex or non convex clusters. Contrarily to *K*-means and GMM, DBSCAN does not directly rely on a given number of clusters, but on hyperparameters that are closely related to the number of clusters: thresholds on the similarity and outliers.

3.2 Dimensionality Reduction

Dimensionality reduction techniques aim to project high-dimensional data into a lowerdimensional space while preserving the underlying structure and relationships in the data at best. These techniques are widely used for visualization, data compression, and noise reduction, for instance. This Section focuses on the *Principal Component*



Figure 2: Schematic representation of a K-means clustering. Initial unlabeled (same color) data points are shown in the left panel, on which the K-means is applied, choosing obviously K = 3 cluster. After convergence, the K-means model is represented on the right panel together with the blue points being the positions of the centroids. The red dashed lines highlights the separation in the three domains given the Euclidean distance metrics: the cluster space. Thus, with this model, each new point can now unambiguously be associated to one of the three clusters given its position in the feature space (here, the x- and y-axis in arbitrary units).

Analysis (PCA) and a non-linear variant, namely the *t-Distributed Stochastic Neighbor Embedding* (t-SNE).

PCA is a widely-used linear technique that projects the data onto a lower-dimensional subspace, maximizing the variance along the new axes. The principal components, which form the basis of the new subspace, are orthogonal and capture the directions of maximum variance in the data. The measure of the variance in the data is based on the calculation of the covariance matrix of the n data points in D-dimensional feature space. For two points x and y, it is given by:

$$\Sigma = \begin{pmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{yx} & \sigma_y^2 \end{pmatrix},\tag{7}$$

where σ_x^2 and σ_y^2 are the variances of x and y, respectively, and σ_{xy} and σ_{yx} are the covariances between x and y. The variances read

$$\sigma_x^2 = \frac{1}{D-1} \sum_{i=1}^{D} (x_i - \bar{x})^2, \tag{8}$$

where \bar{x} is the mean of x, and the same for Y. The covariance between x and Y reads

$$\sigma_{xy} = \sigma_{yx} = \frac{1}{D-1} \sum_{i=1}^{D} (x_i - \bar{x})(y_i - \bar{y}).$$
(9)

Algorithm 2 Principal Component Analysis Require: Data points of dimension $D, X = \{x_1, x_2, \dots, x_n\}$ target dimensionality kEnsure: Transformed data points $Y = \{y_1, y_2, \dots, y_n\}$ Compute the mean vector $\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$ Center the data points by subtracting the mean: $X_{centered} = X - \mu$ Compute the covariance matrix Σ Calculate eigenvectors and eigenvalues of Σ : $(\lambda_1, v_1), \dots, (\lambda_n, v_n)$ Sort eigenvectors by decreasing eigenvalues: v_1, \dots, v_n Select the top k eigenvectors: $V_k = [v_1, \dots, v_k]$ Project the centered data onto the principal components: $Y = X_{centered}V_k$



Figure 3: Principal Component analysis of simple bivariate random Gaussian dataset with two different variances, and having a linear correlation between the two random variables. The left panel shows the point-cloud of the 200 point dataset in the x-y plane together with the eigenvector with the largest eigenvalue in red, and the second one in blue, from the PCA analysis. Center panel displays the dataset in the basis of these two eigenvectors red and blue, respectively denoted PC1₁ and PC₂, and called the first and second principal components. The right panel shows in blue the PCA representation of the point cloud along PC₁ only. The red line is a linear regression model that was learned on the same dataset (see Chapter 2).

The lower-dimensional representation $Y = \{y_1, y_2, \dots, y_n\}$ with target dimensionality k of a dataset $X = \{x_1, x_2, \dots, x_n\}$ of dimensionality D is obtained through Algorithm 2. The n data points are projected on the space of the k eigenvectors whose eigenvalues are the largest and sorted by decreasing values.

Figure 2 illustrates the application of the PCA on a simple arbitrary bivariate random Gaussian dataset with 200 point-cloud using the Scikit-learn PYTHON package ². Interestingly, projection of the data on the first principal component, PC₁, is compared to a linear regression model learned on the same dataset. The good comparison indicates indeed that PC₁ carry the most important part of the information about the dataset.

Contrarily to PCA, t-SNE is a nonlinear dimensionality reduction technique. It is often used for visualizing high-dimensional data, such as images or text embeddings, and

²PCA decomposition module of the Scikit-learn package

is effective at revealing clusters and structures in the data, most often in 2D and 3D for visualization purposes [VdMH08]. t-SNE works by minimizing the divergence between two probability distributions, one representing pairwise similarities in the high-dimensional space

$$p_{j|i} = \frac{\exp(-||x_i - x_j||^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2 / 2\sigma_i^2)}$$
(10)

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$
(11)

using a Gaussian kernel and the other representing pairwise similarities in the lowdimensional space

$$q_{ij} = \frac{(1+||y_i - y_j||^2)^{-1}}{\sum_{k \neq l} (1+||y_k - y_l||^2)^{-1}}$$
(12)

using the Student *t*-statistics. The algorithm consists in minimizing the Kullback-Leibler divergence between the two similarity matrices using gradient descent [HR02, VdMH08]. The perplexity parameter p is a user-defined parameter that balances the focus on local and global structure in the data.

4 Application to particle dynamics

Unsupervised learning has a broad range of applications across various domains, as it can uncover hidden structures and relationships in data without the need for labeled examples. We restrict here to a specific cased in materials science, namely the atomic scale description of homogeneous nucleation, for which the unsupervised learning together with a topological descriptor was applied successfully for the first time very recently [BDMJ22b, BDMJ22a]. Interestingly enough, this concerns particle dynamics and therefore could undoubtedly be applied at larger scales in geophysics.

Crystal nucleation, the early stages where the liquid-to-solid transition occurs upon undercooling, initiates at the atomic level on nanometer length and sub-picoseconds time scales and involves complex multidimensional mechanisms with local symmetry breaking that can hardly be observed experimentally in the very details. In such cases, atomic-level simulations and more particularly molecular dynamics (MD) with a suitable interaction model [AT17] is the dedicated tool. However, reaching statistically meaningful nucleation events, large scale simulations up to million-to billion-atom scale is required [SCC⁺16].

In this Section, an analysis is proposed for the MD simulations performed previously on the solidification of pure zirconium [BDMJ20]. The liquid state above the melting point (approximately $T_M = 2128$ K) was quenched down to the deep undercooled liquid at T = 1250 K, using a simulation box of 1 million atoms. The technical details of the MD simulation are beyond the scope of the present Chapter, and the reader is referred to Ref. [BDMJ20]. During the simulation, the liquid undergoes homogeneous nucleation on this isotherm as shown in Figure 4. In order to unveil their structural



Figure 4: Snapshot of a one-million atom MD simulation of zirconium during nucleation along the T = 1250 K isotherm in the undercooled states at different times after cooling (upper panels). Only atoms having a bcc or distorted bcc crystalline ordering are drawn as detected by the unsupervised topological learning approach (see text). From the snapshots, independent local atomic environment up to the second nearest-neighbors defined by the second minimum of the radial-distribution g(r) (lower right panel), so-called local structures, are sampled by an the algorithm given in the lower left panel (After Ref. [BDMJ22b]).

features during nucleation in such huge simulations, without *a priori*, an unsupervised learning approach founded on topological descriptors loaned from persistent homology concepts was built, and will be described in the following.

4.1 **Topological description of local structures**

Topological data analysis (TDA) [CM21, Mot18] is a growing mathematical field with applications in a wide range of other field such as biology, computer science, physics, and materials science. Persistent homology is an effective and flexible tools to study the underlying topological shapes of a point cloud. In atomic-scale simulations considered here, the point cloud usually corresponds to an atom assembly in a simulation box. Let's consider a dataset as the point-cloud denoted \mathcal{X}_0 in a vector space and a parameter $r \geq 0$. The topological space \mathcal{X}_r is defined by the union of all the balls (Euclidean or with another metric) of radius r, each of them being centered on a each point of the point-cloud. It consists in following the evolution of the topology of \mathcal{X}_r , or more precisely the persistence of its topological features as r grows from 0 about the initial set of points \mathcal{X}_0 , to r big enough so that \mathcal{X}_r has a the topology of a big ball containing all the points.

To encode or quantify the algebraic topology, a dedicated tool is homology [Hat02],



Figure 5: Persistence of a local structure consisting of a central atom and its two neighbourhood shells by means of a barecode description. The local structures with the ball growing as the radius increase are shown on the top. The barecode gives the lifespan of all topological features with colors corresponding to the homological dimension. In the bottom left corner there is the corresponding persistent diagram (with the same corresponding colors for the dimensions).

which derives vector spaces $H_n(\mathcal{X})$, $n \ge 0$, from a space \mathcal{X} , generated by *n*dimensional topological features. 0-dimensional features correspond to connected components, 1-dimensional ones correspond to "holes" in the space, 2-dimensional to cavities and so on. Then persistent homology gives the persistence of these topological features, *i.e.* their lifespan when r is growing. The tracking of this persistence is commonly represented either through a barecode, with each bare corresponding to the lifespan of a specific topological feature, or a persistence diagram (PD) where each topological feature is associated to the point (birth, death) in the plane as can be seen in Figure 5).

While persistent homology in molecular dynamics simulations usually consists in determining the persistence diagram from all atoms of the box as the point cloud. The originality here to define from the PH a topological descriptor of the local environment of each atom as shown in Figure 4. In this case, the point cloud is the local environment and contains a bit less than one hundred atoms for the second neighbor shell. Finally a persistence diagram is obtained for each local structure as a description of its topology, and is associated to its central atom.

Among the various representations of these PDs as a vector, a classical method that has been successfully used to study 3D-shapes [COO15] is chosen here. Each coordinate of the topological vector is associated to a pair of points (x, y) in a persistence diagram

86 Machine Learning (ML) in Geomechanics



Figure 6: TDA-GMM clustering model of undercooled liquid Zirconium during nucleation with 7 clusters C_1 to C_7 , which are closest to the centroids. Evolution of the Integrated Completed Likelihood (ICL) criterion (lower left panel) as a function of number of clusters predicting the optimal number of 7 clusters, trained on the configuration in lower right panel. Adapted from Ref. [BDMJ22a]

D for a fixed level of homology, except the infinite point, and is calculated by

$$m_D(x, y) = \min\{\|x - y\|_{\infty}, d_{\Delta}(x), d_{\Delta}(y)\},$$
(13)

where $d_{\Delta}(\cdot)$ denotes the ℓ^{∞} distance to the diagonal, and those coordinates are sorted by decreasing order. The resulting topological space is high-dimensional with often between 100 and 300 components.

4.2 Clustering local environments during nucleation

For the clustering, a model-based method is used, namely Gaussian Mixture Models (GMM) [HTFF09] and its estimation by an Expectation Maximization (EM) algorithm [DLR77] as mentioned in Section 3.1. The number of clusters is selected by Integrated Criterion Likelihood (ICL, [BCG00]), a refinement for clustering of Bayesian Integrated Likelihood (BIC, [Sch78]).

In a first step, a training set is built from a configuration during the nucleation, where the supercooled liquid coexists with crystalline nuclei, to capture all structural atomic events of interest. This is done by sampling a number of approximately 5000 independent atom centred local structures as described in Figure 4. Using PYTHON packages gudhi [MBGY14] and ripser [TSBO18], the PD of each individual local atomic



Figure 7: analysis of a configuration of undercooled Zr during nucleation with the PCA (left panel) and t-SNE (right panel) after each atom being labelled according to the TDA-GMM model with 7 clusters.

structures of the training set computed up to homological dimensions H_0 , H_1 , and H_2 and the topological feature is calculated using Eq. 13.

In a second step, the unsupervised learning with the Gaussian Mixture Model is performed in the topological space iteratively with cluster numbers 2 to 30. The ICL criterion is computed for each of them giving rise to the curve shown in Figure 6. The optimal number of clusters is chosen from the minimum of ICL, leading to a model with seven clusters numbered C_1 to C_7 whose representative local structure (closest to the centroids) are show in Figure 6. The inferred model from this method is called hereafter TDA-GMM.

Finally, with the learned TDA-GMM as such, each atom of the simulation box is assigned to C_1 to C_7 for any given configuration that enable one to identify and describe the structural properties of the system and morphological properties of the nuclei as can be seen on the specific configuration shown in Figure 6. Moreover, the crystal nucleation and its evolution as a function of time as can be seen strikingly in Figure 4 where only the crystalline local structures (C_1 and C_2) are drawn. All the results and findings can be found in Refs. [BDMJ22b, BDMJ22a], and probably the main physical outcome in these works is that the liquid in the undercooled state appears very heterogeneous and the nucleation is triggered from fluctuations with lowest icosahedral ordering, always present in the liquid at various degrees, the latter being known as incompatible with the long-range ordering of the crystalline state [Fra52, Tur52].

To close this Section, a question arise from the relevance of the TDA-GMM unsupervised modeling and especially the chosen number of clusters (seven for zirconium) in such a high-dimensional topological descriptor space with roughly $D \approx 200$. This can be visualized by performing a dimensionality reduction with the PCA or the t-SNE. Figure 7 displays a two-dimensional analysis for the two principal components of the topological vectors for all atoms in the simulation box shown in Figure 6. They are colored with the cluster they belong to. Interestingly, both methods lead to a similar

representation, in which the clusterized local structures vary as a quasi continuum but do not overlap mainly (except very partially for C_4 and C_5) in the t-SNE representation). The PCA and tSNE distinguishe the liquid on one component and the crystal on the second one. Clusters C_3 , C_4 , and C_5 that are known to retain partially liquid and crystalline orderings are located as expected at the crossover of the two components.

5 Conclusion

This Chapter was devoted to unsupervised learning approaches that emerged as a powerful approach for uncovering hidden patterns, relationships, and structures in data without the need for labeled examples. The key concepts and techniques in unsupervised learning were discussed, focusing only on the basic approach clustering and dimensionality reduction, letting aside association rule mining and unsupervised deep learning methods, which could be treated as topics in themselves. The power of clustering was highlighted through a application in materials science that helped us to monitor efficiently the structural evolution during crystal nucleation of a liquid metal, and uncover hidden correlations without *a priori* in this process from the huge amount of data of these large-scale molecular dynamics simulations with millions of atoms. Such an unsupervised approach is deemed to be sufficiently general to be transposed to discrete particle dynamics at a larger scale in geophysics.

Acknowledgement

We acknowledge the CINES and IDRIS under Project No. INP2227/72914, as well as CIMENT/GRICAD for computational resources. This work was performed within the framework of the Centre of Excellence of Multifunctional Architectured Materials CEMAM-ANR-10-LABX-44-01 funded by the "Investments for the Future" Program. This work has been partially supported by MIAI@Grenoble Alpes (ANR-19-P3IA-0003). Discussions within the French collaborative network in high-temperature thermodynamics GDR CNRS3584 (TherMatHT) and in artificial intelligence in materials science GDR CNRS 2123 (IAMAT) are also acknowledged.

References

- [Aka73] Hirotogu Akaike. Information Theory and an Extension of the Maximum Likelihood Principle, pages 199–213. Springer New York, New York, NY, 1973.
- [AT17] Michael P Allen and Dominic J Tildesley. *Computer simulation of liquids*. Oxford university press, 2017.
- [BCG00] C. Biernacki, G. Celeux, and G. Govaert. Assessing a mixture model for clustering with the integrated completed likelihood. *IEEE Transactions* on Pattern Analysis and Machine Intelligence, 22(7):719–725, 2000.

- [BDMJ20] S. Becker, E. Devijver, R. Molinier, and N. Jakse. Glass-forming ability of elemental zirconium. *Physical Review B*, 102, 2020.
- [BDMJ22a] Sébastien Becker, Emilie Devijver, Rémi Molinier, and Noël Jakse. Physical review e 105, 045304 (2022) unsupervised topological learning for identification of atomic structures. 045304:1–10, 2022.
- [BDMJ22b] Sébastien Becker, Emilie Devijver, Rémi Molinier, and Noël Jakse. Unsupervised topological learning approach of crystal nucleation. *Scientific Reports*, 12:1–9, 2022.
- [Car20] Gunnar Carlsson. Topological methods for data modelling. *Nature Reviews Physics*, 2:697–708, 2020.
- [Cer19] Michele Ceriotti. Unsupervised machine learning in atomistic simulations, between predictions and understanding. *The Journal of chemical physics*, 150(15):150901, 2019.
- [CM21] Frédèric Chazal and Bertrand Michel. An introduction to topological data analysis: Fundamental and practical aspects for data scientists. *Frontiers in Artificial Intelligence*, 4, 2021.
- [COO15] Mathieu Carrière, Steve Y. Oudot, and Maks Ovsjanikov. Stable topological signatures for points on 3d shapes. *Eurographics Symposium on Geometry Processing*, 34:1–12, 2015.
- [DLR77] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.
- [EH21] Bradley Efron and Trevor Hastie. *Computer age statistical inference, student edition: algorithms, evidence, and data science*, volume 6. Cambridge University Press, 2021.
- [EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proc. of 2nd International Conference on Knowledge Discovery and, pages 226–231, 1996.
- [Fra52] Frederick Charles Frank. Supercooling of liquids. Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences, 215(1120):43–46, 1952.
- [Hat02] A. Hatcher. *Algebraic Topology*. Algebraic Topology. Cambridge University Press, 2002.
- [HR02] Geoffrey E Hinton and Sam Roweis. Stochastic neighbor embedding. Advances in neural information processing systems, 15, 2002.
- [HT20] Tony Hey and Anne Trefethen. The fourth paradigm 10 years on. *Informatik Spektrum*, 42:441–447, 2020.

- 90 Machine Learning (ML) in Geomechanics
- [HTFF09] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.
- [HTTG09] Tony Hey, Stewart Tansley, Kristin Tolle, and Jim Gray. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009.
- [HWWT13] Yongzhen Huang, Zifeng Wu, Liang Wang, and Tieniu Tan. Feature coding in image classification: A comprehensive study. *IEEE transactions on pattern analysis and machine intelligence*, 36(3):493–506, 2013.
- [KA21] K. Karapiperis and J.E. Andrade. Nonlocality in granular complex networks: Linking topology, kinematics and forces. *Extreme Mechanics Letters*, 42:101041, 2021.
- [KKN14] Samina Khalid, Tehmina Khalil, and Shamila Nasreen. A survey of feature selection and feature extraction techniques in machine learning. In 2014 science and information conference, pages 372–378. IEEE, 2014.
- [MBGY14] Clément Maria, Jean-Daniel Boissonnat, Marc Glisse, and Mariette Yvinec. The gudhi library: Simplicial complexes and persistent homology. In *The Gudhi Library: Simplicial Complexes and Persistent Homology*, 06 2014.
- [Mot18] Francis C Motta. Topological data analysis: Developments and applications. *Advances in Nonlinear Geosciences*, pages 369–391, 2018.
- [NQY18] Joel Nothman, Hanmin Qin, and Roman Yurchak. Stop word lists in free open-source software packages. In *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*, pages 7–12, 2018.
- [PSG⁺18] Mariam Pirashvili, Lee Steinberg, Francisco Belchi Guillamon, Mahesan Niranjan, Jeremy G. Frey, and Jacek Brodzki. Improved understanding of aqueous solubility modeling through topological data analysis. *Journal* of Cheminformatics, 10:1–14, 2018.
- [SCC⁺16] Gabriele C. Sosso, Ji Chen, Stephen J. Cox, Martin Fitzner, Philipp Pedevilla, Andrea Zen, and Angelos Michaelides. Crystal nucleation in liquids: Open questions and future challenges in molecular dynamics simulations. *Chemical Reviews*, 116:7078–7116, 2016.
- [Sch78] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, March 1978.
- [SMBM19] Jonathan Schmidt, Mário RG Marques, Silvana Botti, and Miguel AL Marques. Recent advances and applications of machine learning in solidstate materials science. *npj Computational Materials*, 5(1):83, 2019.

[TSBO18]	Christopher Tralie, Nathaniel Saul, and Rann Bar-On. Ripser.py: A lean
	persistent homology library for python. Journal of Open Source Software,
	3(29):925, 2018.

- [Tur52] David Turnbull. Kinetics of solidification of supercooled liquid mercury droplets. *The Journal of chemical physics*, 20(3):411–424, 1952.
- [VdMH08] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.
- [WT10] David M. Walker and Antoinette Tordesillas. Topological evolution in dense granular materials: A complex networks perspective. *International Journal of Solids and Structures*, 47(5):624–639, 2010.

Classification Techniques in Machine Learning

Noel Jakse

Université Grenoble Alpes, CNRS, Grenoble INP, SIMaP, 38000 Grenoble, France

1 Introduction

At the turn of the 21st century, machine learning (ML) techniques aroused a growing interest across various scientific disciplines due to their inherent capacity to identify trends, forecast outcomes, and draw invaluable knowledge from vast and intricate datasets [HT20, HTTG09]. Tools designed for collecting data online, scientific apparatus, and computer-simulated environments are generating data at an exponential rate. This has spurred new scientific methodologies for dissecting and structuring enormous volumes of data, while simultaneously detecting subtle effects that were previously hidden. These techniques have become increasingly prevalent in areas such as materials science [SMBM19] and geophysics [KA21].

ML approaches can be primarily grouped into three categories: supervised learning, unsupervised learning, and semi-supervised learning. Each category presents unique ways of deriving knowledge from data, each possessing their own benefits and limitations. Among these, supervised learning is widely adopted, where a model is trained using labeled data, consisting of both input attributes and associated output labels. The model is trained to associate inputs with their corresponding outputs, enabling it to make predictions for novel, unobserved data.

This Chapter is devoted to classification, which has numerous applications across many domains, including spam detection, medical diagnosis, and image recognition to name only some of the most emblematic ones. Like regression, being the subject of Chapter 2, classification belongs to supervised learning techniques. The objective is to build models that can predict the class label of an unseen data, after being trained on a representative set of labeled data. An application of classification of crystalline structure in materials science and more particularly soft matter is briefly mentioned to highlight the power of the method [RLH⁺17, RP18].

Machine learning algorithms generally require large training sets to get a high performance, but labeling such large amounts of data represents substantial human effort

and often costly experimental or numerical resources. Active learning proposes effective ways to reduce the size of training sets while maintaining the same performance. It represent an attractive strategy in scenarios where data labeling is expensive or time-consuming, which is particularly the case for materials exploration in the vast composition space to accelerate their discovery and development [LBXY19].

This Chapter does not deal with Neural Networks and Deep Learning Techniques for Classification. They gained significant momentum in recent years due to their ability to learn complex hierarchical representations from large datasets, and take advantage of the tremendous increase of computer power.

The remaining part of the Chapter provide an overview of the most commonly used classification techniques, their strengths and limitations, as well as important technical aspects like pre-processing, evaluation metrics, and cross-validation. More advanced technique for classification are then presented briefly, namely Ensemble Methods, finally, the basic aspects of active learning is presented.

2 Classification techniques

2.1 General considerations

In this section, some of the most popular classification techniques and their characteristics will be described. The general classification problem can be described as follows. Let's consider a set of n labeled data

$$\{X_i, y_i, i = 1, \dots, n\}.$$
 (1)

Each data instance X_i is labeled with a class y_i . Without loosing generality, let's consider that X_i for all *i* are points in the *d*-dimensional reals space \mathbb{R}^d . For classification, labels take discrete values and two situations are considered:

- binary classification with $y_i \in C = \{-1, 1\}$, thus having a so-called *positive class* and *negative class*. Note that alternately, classes can be defined as $y_i \in C = \{0, 1\}$.
- multi-class classification with K classes when $y_i \in \mathcal{C} = \{1, \dots, K\}$.

It is worth mentioning that if $y_i \in \mathbb{R}$ reduces to the case of regression, which will be the subject of Chapter 2, and needs a model such as in neural networks (see Chapters 7 and 8). The objective of classification is to predict the class value $y_+ \in C$ for each new X_+ not in the training set. For this purpose the training data are used to construct a classifier \hat{c} such that

$$y_{+} = \hat{c}(X_{+}) \tag{2}$$

One of the main aspects of classification is that it relies on labeled data to learn the relationship between input data and class labels that are identified in the training set. This is in contrast with unsupervised learning that deals with unlabeled data and is treated in Chapter 3.



Figure 1: Classification can be subjective. Image generated by the Author with OpenAI DALLE-E with order: realistic painting with white and black horses, pigs, cows, and chicken in a field.

Labeling of data play therefore a crucial role in supervised learning either for classification or regression, as they provide to the machine learning algorithm the relationships between inputs and output labels. The quality and quantity of labeled data can significantly impact the performance of the obtained model, so-called the *classifier*. This is why acquiring an as large, diverse, and representative dataset as possible is often essential.

Finally, classification methods implies categorizing data instances into distinct classes. There are subjective aspects that can influence the model, which stem from human decisions, interpretations, and biases in the choice of the classes, as illustrated in Figure 1. The way a classification problem is formulated and how classes are defined such as, animals, their color, type, etc. often depends on the domain knowledge, goals, and preferences of the researchers. It leads to discrepancies between different formulations of the same problem.



Figure 2: Schematic representation of a typical workflow for classification. A training dataset is build, featurized, scaled and normalized. A classifier is then trained with a chosen learning algorithm. It can then be used to recognize unseen hand written digit. Note that the new digits shown to the classifier for recognition have also to be transformed with the same feature model, scaling and norm used for the training. Example: Optical Recognition of Handwritten Digits Data Set.

2.2 Typical workflow for classification

The general machine learning scheme is described in Chapter 3 devoted to unsupervised learning. In Figure 2 the typical workflow for classification is described schematically. After having clearly defined the problem that needs to be solved including the choice of the labelling, the first crucial step consists in data collection and preprocessing to prepare the training and test data that will be used to train and evaluate the machine learning model. Collection of raw data that can come from various sources, such as databases, APIs, sensors, or web scraping. Then preprocessing typically includes data cleaning (*e.g.*, handling missing values, removing duplicates, or fixing errors). The second step consists in the data transformation such as normalization, standardization, or feature transformation and extraction as well as associated dimensionality reduction (see the Chapter 3). In particular, normalization typically scales features to a range of [0, 1], while standardization scales features to have a mean of 0 and a standard deviation of 1.

With the data prepared as such, the third step is to select an appropriate machine learning algorithm and train the model that best suits the problem formulation and the characteristics of the data. Finally After training the model, it is important to evaluate its performance on a separate dataset (test set) that was not used during training. This helps to assess the model's ability to generalize to new, unseen data. Various evaluation metrics, such as *accuracy*, *precision*, *recall*, *F1-score*, can be used to quantify the classification model's performance, depending on the type of machine learning task as will be seen in Section 2.3. In addition to evaluating the model's performance, it might be necessary to tune the parameters of the learning algorithm so-called *Hyperparameter*. Tuning them involves searching for the optimal combination of their values that yields the best model performance. This typically consists in using techniques such as grid

search, random search, or Bayesian optimization [RW+06, SSL+21].

2.3 Evaluation metrics

2.3.1 Standard metrics

Accuracy is the most straightforward and simple evaluation metric, measuring the proportion of correctly classified instances in the dataset. It is defined as the ratio

$$Accuracy = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}.$$
 (3)

However, accuracy can be misleading in cases of class imbalance, as it may overemphasize the performance on the majority class. A confusion matrix for a binary classification problem can then be considered. The confusion matrix is a table that summarizes the performance of the classifier by showing the counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). A true positive is a case in which the classifier correctly predicts the positive class, while false positive is a case in which the classifier incorrectly predicts the positive class. The same definition holds for the true and false negatives. The accuracy can then be calculated as

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$
(4)

Moreover, this matrix allows for a deeper understanding of the model's strengths and weaknesses of the chosen classes in the model. To do so, *precision* and *recall* metrics can be used to account for false positives and false negatives, respectively. Precision, also known as positive predictive value, is the ratio of true positive predictions to the total number of positive predictions made. Precision can be calculated as:

$$Precision = \frac{TP}{TP + FP}.$$
(5)

Recall, also known as *sensitivity* or *true positive rate*, is the ratio of true positive predictions to the total number of actual positive instances. Recall can be calculated as:

$$\operatorname{Recall} = \frac{TP}{TP + FN} \tag{6}$$

Finally, the *F1-score* is the harmonic mean of precision and recall, providing a balanced measure of both metrics. It can be calculated as:

$$F1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.$$
 (7)

The F1-score ranges from 0 to 1, with 1 being the best possible score, indicating perfect precision and recall.

The Area Under the ROC Curve (AUC-ROC) is an evaluation metric often used for binary classification problems, that at least should be mentioned. It measures

Algorithm 1 Cross-validation

Require: Randomly shuffle the dataset.
Ensure: Divide the dataset into k equally sized (or approximately equal) folds.
for $i = 1, 2,, k$ do
Use fold i as the testing set, and the remaining k-1 folds as the training set.
Train the model on the training set.
Evaluate the model on the testing set, and compute the evaluation metric.
end for
Calculate the average of the evaluation metric values over all k iterations.

the performance of a classifier across different decision thresholds, plotting the true positive rate (recall) against the false positive rate. AUC-ROC provides a single scalar value representing the overall performance of the classifier, with a value of 1 indicating perfect classification and a value of 0.5 corresponding to random chance.

2.3.2 Cross-Validation

Cross-validation or *folding* is a concept used for evaluating the performance of machine learning models in general. It consists in estimating statistically the average of an evaluation metrics of the model, such as accuracy, over multiple training and testing sets. It consists in dividing the dataset into multiple subsets and iteratively train and test the model on different combinations of these subsets so-called *folds*. This process allows for a more reliable estimation of the model's performance, especially when the amount of available data is limited. One advantage is that it reduces the likelihood of going into *overfitting* and provides a better understanding of the model's generalization capabilities.

In most of the cases, the k-fold cross-validation approach is used, for which the dataset is randomly divided into k equally sized subsets, thus leading to k train and test steps as described in Algorithm 1. Each step uses a different fold as the testing set and the remaining k - 1 folds as the training set. The evaluation metric (e.g. accuracy, precision, recall, or F1-score mentioned just above) is computed for each iteration, and the final result is obtained by averaging the metric values over all iterations. Typically, k is set to values such as 5 or 10, as these values were empirically shown to provide a good balance between bias and variance.

Leave-one-out cross-validation (LOOCV) is a limiting case of k-fold cross-validation, where k is set equal to the number of instances in the dataset. In other words, each instance is used as a testing set exactly once. Therefore, LOOCV can be computationally expensive, especially for large datasets, as it requires training and evaluating the model as many times as there are instances in the dataset.

Algorithm 2 k-Nearest Neighbors

```
Require: Training set \mathcal{T}, query point \boldsymbol{x}, number of neighbors k
for each instance \boldsymbol{x}^{(i)} in \mathcal{T} do
Compute d(\boldsymbol{x}, \boldsymbol{x}^{(i)})
end for
Identify the k instances \boldsymbol{x}^{(i_1)}, \dots, \boldsymbol{x}^{(i_k)} with smallest distances to \boldsymbol{x}
\hat{y} = \arg \max_y \sum_{j=1}^k I(y = y^{(i_j)}) return \hat{y}
```

2.4 Standard classification algorithms

2.4.1 *k*-Nearest Neighbors (*k*-NN)

Let's first consider the k-Nearest Neighbors, which is a non-parametric, instance-based classification algorithm, and is among the simplest of all machine learning algorithms, both conceptually and in terms of implementation. Given a query point x in a training set \mathcal{T} with m samples, $\{(x^{(i)}, y^{(i)}), i = 1, ..., m\}$, each with a label y, the objective of k-NN is to assign x the label most commonly associated with its k nearest neighbors in \mathcal{T} .

The algorithm is given in Al. 2 consists in computing the distance $d(x, x^{(i)})$ between x and each training instance $x^{(i)}$, $i = 1, \ldots, m$. The choice of distance function typically depends on the problem under consideration. Common choices include the Euclidean distance and the Manhattan distance as described in Chapter 3. The k instances $x^{(i_1)}, \ldots, x^{(i_k)}$ that are closest to x are identified and the most common label among these k neighbors is assigned to x. The assigned label \hat{y} can be formally expressed as:

$$\hat{y} = \arg\max_{y} \sum_{j=1}^{k} I(y = y^{(i_j)}),$$
(8)

where $I(\cdot)$ is the indicator function, which equals 1 if the condition inside the parenthesis is true and 0 otherwise.

The choice of *hyper-parameter* k is very critical to the performance of the algorithm. A small value of k means that noise will have a higher influence on the result and a large value make it computationally expensive. Usually an odd number is chosen if the number of classes is 2 (binary classification) and another simple approach to select k is set $k = \sqrt{n}$.

As an example, Fig. 3 shows the results on a binary classification task using the k-NN classifier. Let's consider a dataset consisting in n = 600 points distributed in a plane with axis x_1 and x_2 using a Gaussian bivariate distribution of similar variance. Positive and negative class labels are colored in red and blue, respectively. In a first step, the dataset is plotted as a point-cloud and analyzed with Linear Discriminant Analysis (LDA, similar to the Principal Component Analysis described in Chapter 3). It shows indeed that the points of each class follow a Gaussian distribution along x_1 as well as



Figure 3: Classification task on a Gaussian bivariate distribution of n = 600 points with the same variance in two dimensions x_1 and x_2 , using the k-NN classifier. (a) The dataset with labels in red and blue color respectively for the positive and negative classes. Yellow stars are the respective centroids and ellipses materialize the standard deviation of the Gaussian distributions about the centroids. Red and blue lines are the Gaussian distributions of the respective classes projected onto x_1 and x_2 using a Linear Discriminant Analysis. The dashed line is the total distribution. (b) Test set with positive and negative classes in red and blue respectively as well as predictions of the best k-NN classifier, namely k = 10, highlighted by the light red and blue background. (c) F1-score of the test set and training set for k-NN classifiers, trained with k in the range [1, 30]. (d) Predicted probabilities of each points in the test set and the horizontal line represent the decision line with probability p = 0.5 used for attributing class labels on the unseen points.

 x_2 axis. The ellipses, with radii corresponding to the distance of the standard deviation from the centroids, do not overlap, indicating that the classes should be efficiently separated. The LDA further reveals that x_2 is the most relevant one to discriminate the classes, while the two classes are mostly superimposed along x_1 .

Using the Package SCIKIT-LEARN¹ in PYTHON. the dataset $\{X_i, y_i, i = 1, ..., n\}$ is first split into a training set and a test set as the following snippet:

¹https://scikit-learn.org/

The usual size of the training set is 80% of the data points, leaving 20% of them for the test set to measure the performance of the trained model. Here the random state is fixed in order to compare different models on strictly the same training set. The k-NN model is set up with desired hyper-parameters, leaving most of them to the default values, and fixing only k = 10 neighbors corresponding the best classifier shown in Fig. 3(b)

```
from sklearn.neighbors import KNeighborsClassifier
n_neighbors = 10
knn = KNeighborsClassifier(n_neighbors=n_neighbors)
```

The, the model is trained by providing to the learning algorithm inputs X and outputs y of the training set

knn.fit(X_train, y_train)

Once trained, the model can be used to predict the class of unseen data, let's say those of the test set at first from the command

y_pred = knn.predict(X_test)

The performance of the model is measured by comparing the predictions on the text set against the true known values, using the F1-score for instance:

```
from sklearn.metrics import f1_score
Score = f1_score(y_test, y_pred)
```

so that Fig. 3 (b) indicates the value of 0.99, which is close to the perfect value of 1. Prediction made now on the whole (x_1, x_2) plane leads to the colored background corresponding to the obtained classes. The boundary separating the two colored areas is the decision lines, which separates appropriately the two classes horizontally in the vicinity of the dataset as inferred initially from the LDA.

The model with k = 10 was not chosen by chance. A simple *model selection* was performed by iterating on k-values between 1 and 30, training the k-NN for each k, and evaluating the performance using the F1-score. Fig. 3 (c) displays the performance of k-NN as a function of k measured on the test set (blue curve), showing that k = 10

performs the best in that case. The rule of thumb $k = \sqrt{n}$ with n = 240 (the size of the training set) gives a value of 14, which is a bit higher than the best value of k but remains relevant. A *validation test* can be performed (orange curve) by predicting the classes on the training set itself, which can be indicative on how the model learns. It can be seen that the validation score becomes somewhat stationary above k = 15. An increase towards high values is often characteristic to over-fitting, which is typically the case for k = 1. The model learned the training data very well but when it comes to the prediction, it appears to perform the worst.

Finally, Fig. 3(d) describes the mechanism by which the classes are chosen by the algorithm, given the predicted probability obtained as

p = knn.predict_proba(X_test)

For each point to predict, the negative class is chosen if p(X) > 0.5, and positive class is chosen otherwise. The Figure shows that most of the points are classified correctly.

2.4.2 Logistic and softmax regressions

Logistic Regression is a linear model for *binary classification* that uses the logistic function to model the probability of an instance belonging to a particular class. The model is trained by minimizing the negative log-likelihood (cross-entropy loss) using gradient descent or Newton-Raphson method. For prediction, the model computes the probability of the instance belonging to the positive class and assigns the class label based on a threshold, typically 0.5.

The method uses the logistic function, also known as the sigmoid function denoted as $\sigma(z)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{9}$$

The function takes a real-valued input z and maps it to a value between 0 and 1, making it suitable for modeling probabilities. Given a set X of input features $\{x_1, x_2, ..., x_n\}$, the logistic regression model computes the probability P(Y = 1|X) of an instance belonging to the positive class in Y using the following equation:

$$P(Y = 1|X) = \sigma(w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n),$$
(10)

where w_0, w_1, \ldots, w_n are the model parameters (also known as *weights*) that need to be estimated from the training data, and σ denotes the logistic function given by Eq. 9.

Optimisation of the parameters is based on the maximum likelihood estimation (MLE) method. The objective is to find the parameter values that maximize the likelihood of the observed data. For a given dataset with m samples, $\{(x^{(i)}, y^{(i)}), i = 1, ..., m\}$, the likelihood function is defined as:

$$L(w) = \prod_{i=1}^{m} P(Y = y^{(i)} | X = x^{(i)}).$$
(11)
Algorithm 3 Logistic regression

Require: Initialize the model parameters $w_0, w_1, ..., w_n$ with small random values or zeros

repeat

Calculate the predicted probability of the positive class for each instance using the logistic function.

Compute the negative log-likelihood and its gradient with respect to the model parameters.

pdate the model parameters using the gradient descent update rule.

until convergence or the maximum number of iterations is reached.

To simplify the optimization problem using a gradient based method, the negative log-likelihood is minimized instead:

$$J(w) = -\sum_{i=1}^{m} [y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})],$$
(12)

where $\hat{y}^{(i)}$ denotes the predicted probability of the positive class for instance *i*. The corresponding algorithm is described in Al. 3

To classify an unseen instance, its predicted probability of belonging to the positive class is compared to a threshold, typically 0.5 as mention above. If the probability is greater than or equal to the threshold, the instance is classified as positive; otherwise, it is classified as negative, as

$$\hat{c} = \begin{cases} 1 & \text{if } P(Y=1|X) \ge 0.5\\ 0 & \text{otherwise} \end{cases}$$
(13)

For multi-class problems, the classifier reads:

$$\hat{c} = \operatorname{argmax}_{k \in \{1, \dots, K\}} \#\{i : Y_i = k\}$$
(14)

The limitations of this method includes sensitivity to outliers, inability to model nonlinear relationships, and the assumption of class label independence for multi-class problems.

As mentioned above, the extension of logistic regression to handle multi-class classification problems directly is the so-called *softmax regression*. In this scheme, the probability of an instance belonging to each class is estimated using a generalized logistic function for multiple classes. The probability of the instance belonging to class k is calculated using:

$$P(Y = k|X) = \frac{e^{(w_k^T x)}}{\sum_{i=1}^{K} e^{(w_j^T x)}},$$
(15)

where w_k^T represents the weight vector for class k.



Figure 4: Classification tasks using the logistic regression classifier. (a) binary classification with the same dataset and same coloring as in Fig. 3. (b) Multi-class classification using the softmax method with a dataset containing three classes in the two dimensional space colored in red, blue and green respectively for class 1, 2 and 3. In both cases, boundary decisions are materialized by the background colors corresponding to the predictions.

Fig. 4(a) shows the logistic regression for the binary classification using the same dataset as for the k-NN models. The same scheme as described before is applied here (not repeated). The logistic regression model is included in the SCIKIT-LEARN and setup as the following:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression(solver='liblinear',
    multi_class='ovr')
```

The decision curve is linear, as expected, and separates the two classes horizontally as inferred by the LDA with a score of 0.99. Fig. 4(b) shows a multi-class logistic regression with a dataset created with 3 classes using:

The reader is referred to the SCIKIT-LEARN documentation for the parameters of the

make_classification function. The set up of the model is done with the same function as for the binary classification, but with different hyper-parameters.

Here the (x_1, x_2) plane is divided in three area by linear decision boundaries, and a score of 0.92.

2.4.3 Support Vector Machines (SVM)

Support Vector Machines (SVM) represent a class of linear models that aim to find the optimal separating hyperplane between classes by maximizing the margin between support vectors, *i.e.* those instances closest to the decision boundary. They are particularly well suited for high-dimensional and small-to-medium-sized datasets. However, for large datasets, SVMs can be computationally expensive.

let's consider a training set $\{(x^{(i)}, y^{(i)}), i = 1, ..., m\}$, where $x^{(i)} \in \mathbb{R}^n$ represents the feature vector of instance *i*, and $y^{(i)} \in \{-1, 1\}$ is its corresponding class label in binary classification. The goal of SVM is to find the optimal separating hyperplane that maximizes the *margin* between the classes. The equation of the hyperplane can be represented as:

$$\boldsymbol{w}^T \boldsymbol{x} + \boldsymbol{b} = \boldsymbol{0},\tag{16}$$

where $w \in \mathbb{R}^n$ is the weight vector normal to the hyperplane, and b is the bias. The *margin* is defined as the distance between the hyperplane and the closest instances from each class, so-called the *support vectors*. The margin M can be written as:

$$M = \frac{2}{|\boldsymbol{w}|}.\tag{17}$$

Finding the optimal hyperplane given by Eq. 16, consists in solving the following constrained optimization problem:

minimize
$$\frac{1}{2}|\boldsymbol{w}|^2$$
 subject to $y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b) \ge 1, \quad i = 1, ..., m$ (18)

For non-linearly separable data, SVM can be extended using *kernel functions*. Kernel functions allow SVM to operate in a higher-dimensional feature space, where the data may become linearly separable. A common kernel function is the Radial Basis Function (RBF) kernel used in many ML techniques:

$$K(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma |\boldsymbol{x} - \boldsymbol{x}'|^2), \qquad (19)$$

Algorithm 4 Support Vector Machine

Require: Training set $\{(x^{(i)}, y^{(i)}), i = 1, ..., m\}$ Choose a kernel function $K(\cdot, \cdot)$ and its parameters Formulate the constrained quadratic optimization problem:

$$\min_{\boldsymbol{w}, b} \quad \frac{1}{2} \|\boldsymbol{w}\|^2 \\ \text{s.t.} \quad y^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b) \ge 1, \quad i = 1, \dots, m$$

Solve the optimization problem to obtain the optimal weight vector w and bias term b (or the dual variables in the kernel-based case)

for each new instance x do

Compute the decision function:

 $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b$ (or the corresponding kernel-based version)

Classify the instance based on the sign of the decision function:

$$\hat{y} = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) \ge 0\\ -1 & \text{otherwise} \end{cases}$$

end for

where $\gamma > 0$ is a parameter that controls the shape of the decision boundary. The corresponding algorithm is given in Al. 4

It should be noted that SVMs can be extended to multi-class classification problems using the One-vs-Rest (OvR) or One-vs-One (OvO) approaches. In the OvR approach, a separate SVM is trained for each class against all the other classes. In the OvO approach, a separate SVM is trained for each pair of classes. In both cases, the class with the highest score or the most votes is chosen as the predicted class.

Fig. 5 shows SVM classifications for the binary and multi-class classifications using the same datasets as for the k-NN and logistic regression models. The set up of the SVM is done as the following, listing also the available kernels that can be chosen:

```
from sklearn.svm import SVC
kernels = ['linear', 'poly','rbf','sigmoid']
k = 1
svmk = SVC(kernel=kernels[k])
```

The linear version performs similarly to the logistic regression with similar performances. The non-linear kernels gives rise to models quite different performances and prediction as can be seen on the decision boundaries. It appears that the RBF kernel

Jakse 107



Figure 5: Classification tasks using the support vector machines classifiers with various kernel functions. (upper panels) binary classification with the same dataset and same coloring as in Fig. 3. (lower panels) Multi-class classification with the same dataset as in Fig. 4 and the same coloring. In both cases, boundary decisions are materialized by the background colors corresponding to the predictions.

performs the best, while the sigmoid one seems not to be appropriate in the studied cases, either for binary or multi-class classifications. The decision in choosing a given kernel should be based on the dataset at hand.

2.4.4 Improvements and Ensemble Techniques

One significant improvement was brought by the so-called *bootstrapping*, which is a powerful re-sampling technique used in statistics and machine learning [HTFF09]. Given a dataset of size n, it consists in drawing a sample of size n with or without *replacement* (replacement means that the same data point can be picked up more than once), to infer descriptive statistics on this new sample. Repeated many times (each time drawing a new sample from the original dataset and computing the statistic), it results in a distribution of the computed statistic as an approximation of its sampling distribution. Bootstrapping is commonly used when it is difficult to make assumptions about the population or when the sample size is small. It allows us to estimate the uncertainty of our statistics without making any assumptions about the data distribution by creating multiple subsets of the original dataset, with replacement, and train a separate model on each subset. The final prediction is then made by aggregating the predictions of each model. This process helps to reduce variance and improve the model's robustness.

Ensemble techniques enhance predictive performance compared to standard classification techniques as those considered above. More precisely they increase generalizability and robustness of standard methods on which they are based [HTFF09]. The most

common techniques are *bagging*, *boosting*, and *stacking*, are particularly effective at reducing overfitting by introducing diversity, reducing variance and therefore dealing with noise in the data. For instance, *Random Forest* (RF), is a popular technique based on bagging that uses decision trees as its base classifier. Decision Trees recursively split the input space based on feature values to form a tree-like structure. Each node in the tree represents a decision rule based on a feature, and the leaf nodes represent the class labels. In addition to using bootstrap re-sampling for training, Random Forest introduces an additional layer of randomness by selecting a random subset of features for splitting at each node. This process reduces the correlation between the trees, further improving the ensemble's performance.

3 Active Learning in classification

Active learning (AL) is an approach where the learning algorithm actively queries the user (or an oracle) to provide labels for specific instances during the training process. This approach is particularly useful for classification tasks when labeled data is scarce or expensive to obtain, which is often the case in materials science either, experimentally of numerically, and geophysics. Then, AL aims at minimizing the number of labeled examples required to achieve a desired level of performance, focusing on acquiring the most informative and useful examples for the learning process.

Various active learning strategies have been proposed of which common ones include:

- Uncertainty sampling: In uncertainty sampling, the learner selects the instances for which it is least certain about the correct label. This is often quantified using the model's predicted class probabilities, with instances with the highest entropy or lowest maximum probability being chosen for labeling.
- *Query-by-committee:* In query-by-committee, multiple models (the committee) are trained on the current labeled dataset. The learner selects instances for which the committee members disagree the most, measured by the disagreement in their predicted labels or class probabilities.
- *Expected model change:* This strategy selects instances that are expected to have the largest impact on the current model if their true label were known. This can be quantified by estimating the expected change in the model parameters or loss function after incorporating the labeled instance.
- Expected error reduction: In this strategy, instances are selected based on their expected reduction in the model's generalization error. This can be estimated using techniques such as Monte Carlo sampling or Bayesian model averaging.

The typical active learning workflow for classification tasks is described in Al 5. After an initial training with given classification algorithms such as those described above, adding more instances in the training set is done until a criterion is reached. The latter is very problem dependent, but one has to bear in mind that the final dataset has to be as small as possible for a given performance or accuracy that is required. Therefore,

Algorithm 5 Active Learning workflow

Require: Initialize the labeled dataset with a small set of labeled instances, either selected randomly or using prior knowledge.

Train the classification model on the labeled dataset.

repeat

Select a batch of instances from the unlabeled dataset using the chosen active learning strategy.

Query the user to provide labels for the selected instances.

Update the labeled dataset with the newly labeled instances and retrain the model. **until** a stopping criterion is met.



Figure 6: Active learning for classification on a Gaussian bivariate distribution of n = 600 points with the same variance in two dimensions x_1 and x_2 , used above.

a challenging task is to find a trade-off between exploring the space to discover new informative instances and exploiting the current knowledge to improve the model's performance.

Other challenges comes into play in AL. Model complexity is a bottleneck for Active learning such as deep learning networks, due to their non-convex loss surfaces and high computational requirements for training and evaluation. AL assumes that the labeling is "perfect". In practice, labeling mechanisms may be prone to errors or inconsistencies, which can adversely affect the performance of the active learning process.

A working example is illustrated in Fig. 6. The bivariate Gaussian distribution dataset with n = 600 points (see Fig. 3) was considered as a basis of the following simple implementation:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size= 0.2, random_state=42)
initial_idx = np.random.choice(range(len(X_train)),
    size=int(len(X_train) *0.1), replace=False)
X_pool = np.delete(X_train, initial_idx, axis=0)
y_pool = np.delete(y_train, initial_idx, axis=0)
X_initial = X_train[initial_idx]
y_initial = y_train[initial_idx]
model = SVC(kernel='rbf')
model.fit(X_initial, y_initial)
y_pred_test = model.predict(X_test)
initial_accuracy = accuracy_score(y_test, y_pred_test)
#Start the AL loop
n_queries = 40
for i in range(n_queries):
    query_idx = random.sample(range(len(X_pool)), 10)
    query_instances = X_pool[query_idx]
    y_pred = model.predict(query_instances)
    X_initial = np append(X_initial, query_instances, axis=0)
    y_initial = np.append(y_initial, y_pred)
    X_pool = np.delete(X_pool, query_idx, axis=0)
    y_pool = np.delete(y_pool, query_idx, axis=0)
    model fit(X_initial, y_initial)
    y_pred_test = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred_test)
    y_pred_test = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred_test)
```

The goal is to highlight through this basic active learning implementation the importance as well as limitations that led to the various strategies listed above. The question that can be asked is the following: are the 80% of the dataset needed to train the model? In other words, what is the minimum number of points required to reach the same accuracy? An AL simulation was performed with the Support Vector Machine using the RBF Kernel as a classifier. This SVM was shown to give the best performance for this dataset in the previous Section.

An initial training set containing 10% of the full training set (80% of the dataset) is used to train a first SVM model. The latter is iteratively refined by queering each

time 10 additional points and retrain the SVM using the true output, namely extracted from the training set (usually the queried data points are unlabelled, but here labels are taken from the full training set as they were given from an oracle). Fig. 6 show that the performance increases progressively in that case, and reach the one from the full training with a significantly lower number of data points (around 300 instead 480). The averaged accuracy curve shown was obtained by repeating 200 times this process.

Nevertheless, as mentioned above this is not the true story of active learning. In a real scenario, a first model is trained with all the labelled data at hand, and increasing the performance of the model is done by adding new *a priori* unlabelled data for which a label has to be given by an oracle (like a human expert or a reference method to get 'perfect' or 'true' labels). Another idea could be to use the last trained model to predict the labels. This is precisely what is done for the 'predicted labels' curve in Fig. 6 that progressively worsen. This shows that the random sampling for new queries associated to prediction with the last trained model is not efficient. Both the wise sampling of new data points as well as the labelling are crucial and should be designed carefully, taking care for instance on the diversity in the input and output spaces [WLH19].

3.1 Application of classification

An interesting application of classification in materials science is briefly mentioned here [RLH⁺17, RP18]. This work aims at identifying and differentiating relevant ordered structures in data derived from molecular simulations or particle tracking. In contrast to other commonly used structure identification methods, this approach does not necessitate any predetermined description of the structures being sought. Features vectors of structures are first build based on graph theory and adjacent matrices to generates impartial structural data, that enable to measure the crystalline nature of particles located near defects, grain boundaries, and interfaces. A multi-layer perceptron (see the Chapter 7 and 8) to classify structural relationships among particles based on their local environmental topology. This method is used to categorizing particles in a colloidal crystallization simulation and demonstrating that structural elements that go unnoticed when using conventional methods.

4 Conclusion

Classification together with regression are the two pillars of supervised learning, with applications in many scientific fields, including but not limited to, material science and geophysics. Classification techniques like logistic regression, SVMs, and ensemble methods were covered in this Chapter together with, focusing on the most common ones for the sake of comprehensiveness.

Classification problems can benefit significantly from active learning, a technique where the algorithm selectively queries the user (or some other information source) to obtain new data points considered as the most informative. This is all the more true for regression problem, where AL is more emergent. In material science, active learning

can help in predicting properties of new, yet untested materials, thereby speeding up the discovery process at a minimal cost in time and financial means. In the same spirit in geophysics, it can aid in tasks like seismic monitoring or subsurface exploration, where obtaining labeled data can be costly or time-consuming.

Acknowledgement

We acknowledge the CINES and IDRIS under Project No. INP2227/72914, as well as CIMENT/GRICAD for computational resources. This work was performed within the framework of the Centre of Excellence of Multifunctional Architectured Materials CEMAM-ANR-10-LABX-44-01 funded by the "Investments for the Future" Program. This work has been partially supported by MIAI@Grenoble Alpes (ANR-19-P3IA-0003). Discussions within the French collaborative network in high-temperature thermodynamics GDR CNRS3584 (TherMatHT) and in artificial intelligence in materials science GDR CNRS 2123 (IAMAT) are also acknowledged.

References

- [HT20] Tony Hey and Anne Trefethen. The fourth paradigm 10 years on. *Informatik Spektrum*, 42:441–447, 2020.
- [HTFF09] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.
- [HTTG09] Tony Hey, Stewart Tansley, Kristin Tolle, and Jim Gray. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009.
- [KA21] K. Karapiperis and J.E. Andrade. Nonlocality in granular complex networks: Linking topology, kinematics and forces. *Extreme Mechanics Letters*, 42:101041, 2021.
- [LBXY19] Turab Lookman, Prasanna V Balachandran, Dezhen Xue, and Ruihao Yuan. Active learning in materials science with emphasis on adaptive sampling using uncertainties for targeted design. *npj Computational Materials*, 5(1):21, 2019.
- [RLH⁺17] Wesley F Reinhart, Andrew W Long, Michael P Howard, Andrew L Ferguson, and Athanassios Z Panagiotopoulos. Machine learning for autonomous crystal structure identification. *Soft Matter*, 13(27):4733– 4745, 2017.
- [RP18] Wesley F Reinhart and Athanassios Z Panagiotopoulos. Automated crystal characterization with a fast neighborhood graph analysis method. *Soft matter*, 14(29):6083–6089, 2018.

Jakse 113

- [RW⁺06] Carl Edward Rasmussen, Christopher KI Williams, et al. *Gaussian processes for machine learning*, volume 1. Springer, 2006.
- [SMBM19] Jonathan Schmidt, Mário RG Marques, Silvana Botti, and Miguel AL Marques. Recent advances and applications of machine learning in solidstate materials science. *npj Computational Materials*, 5(1):83, 2019.
- [SSL⁺21] Benjamin J Shields, Jason Stevens, Jun Li, Marvin Parasram, Farhan Damani, Jesus I Martinez Alvarado, Jacob M Janey, Ryan P Adams, and Abigail G Doyle. Bayesian reaction optimization as a tool for chemical synthesis. *Nature*, 590(7844):89–96, 2021.
- [WLH19] Dongrui Wu, Chin-Teng Lin, and Jian Huang. Active learning for regression using greedy sampling. *Information Sciences*, 474:90–105, 2019.

Data-Driven Modeling in Geomechanics

Konstantinos Karapiperis

ETH Zürich

The theoretical framework of data-driven computational mechanics presents an alternative formulation of mechanics, whereby optimal material states are sought within a dataset that most closely satisfy momentum and energy conservation principles. We review the framework for the case of simple and nonsimple (polar), elastic and inelastic media, which represent common descriptions for geomaterials. Data mining from experiments and high-fidelity lower-scale simulations (DEM,FEM) are discussed, while remedies for data scarcity (adaptive data sampling) are also highlighted. Representative examples of a flat punch indentation and a rupture through a soil layer are presented, and a link to open-source Python code is provided.

1 Introduction

Predictive models in geomechanics have traditionally relied on continuum modeling via the formulation constitutive equations [DL82, MA91, VA91, OP04, DM04, DN05, BA06, HP06], discrete particle-based models [CS79, Bar94, KAVA18], as well as multiscale techniques that bridge the continuum and discrete scales [CMNN81, NDR05, KRB07, AT09, GZ14, RY11]. Initially informed by macroscopic experiments [RSW58, Ros70], and later by high-fidelity grain-scale resolved experiments [HBD⁺10, AHV⁺12], these models have been succesful in capturing essential aspects of granular materials and, more generally, geomaterials including pressure-dependent elasticity, history-dependence and critical state, fabric evolution, nonlocality. Despite their success, further progress has been hindered by numerous challenges including the uncertainty related to the models at different scales, as well as their complexity and the associated laborious process of calibration.

Recently, a variety of data-driven approaches have been developed in order to tackle the challenges outlined above, most importantly the bias, complexity or inefficiency of these methods, while incorporating information about the underlying mechanics and physics. These include physics-informed neural networks [HRM⁺21] with a built-in structure of elastoplasticity [EPW22, HAV23, VS23], or with incorporated thermodynamics constraints [MSVMB21, HHR22]. Despite the physical basis of these models,

they are often hard to interpret, and could suffer from generalization errors for unseen stress-strain paths. An alternative approach which is not based on learning a constitutive law, but rather relies directly on the raw data is furnished by the framework of Data-Driven Computational Mechanics (DDCM), introduced Ortiz and co-workers. In DDCM, the mechanical problem is reformulated in terms of distances between a material dataset obtained from experiments, and an equilibrium set where the states that satisfy the physics reside. The method has been extended in various directions including inelasticity [EKR⁺19], nonlocality [KOA21], stochasticity [PRO23], fracture [CDLSO20] and breakage mechanics [UGK⁺23], and has been coupled with model-based approches [BS21] and machine-learning techniques [ESOR21, BS22] have also been developed in an effort to boost the efficiency and robustness of the method. The source of the data can be experiments [LCR⁺18] or high-fidelity micromechanical calculations [KSOA20].

The chapter is organized as follows. In Section 2.1, the framework of data-driven mechanics is presented for simple continua, which is then extended to micropolar continua with a microstructure in Section 2.2. The enhancement of the framework to inelasticity is addressed in Section 2.3. Then, the source of data (experiments, micromechanical simulations) is discussed (Section 2.4), focusing also on data scarcity and how it can be efficiently overcome. We finish with representative examples and a link to an open-source code repository (Section 3).

2 Data-Driven Computational Mechanics

2.1 Cauchy Continuum - Elasticity

Let's first restrict our attention to the geometrically linear mechanical problem of a *simple* (nonlinear) elastic body that is discretized into N nodes and M material points (Fig 1). The body is subject to applied forces $\mathbf{f} = {\{\mathbf{f}^{\alpha}\}}_{\alpha=1}^{N}$, and undergoes displacements $\mathbf{u} = {\{\mathbf{u}^{\alpha}\}}_{\alpha=1}^{N}$ at its nodes. The state of each material point is described by a stress-strain pair indicating a point in the local phase space i.e., $\mathbf{z}^{e} = (\boldsymbol{\varepsilon}^{e}, \boldsymbol{\sigma}^{e}) \in Z^{e}$, and the state of the entire system is collectively a point in the global phase space $\mathbf{z} = {\{\mathbf{z}^{e}\}}_{e=1}^{M} \in Z$. The system is subject to the following discretized compatibility and equilibrium constraints:

$$\varepsilon_{ij}^e = \frac{1}{2} \sum_{\alpha} \left(N_{,j}^{e\alpha} u_i^{\alpha} + N_{,i}^{e\alpha} u_j^{\alpha} \right), \qquad e = 1, \dots, M$$
(1)

$$\sum_{e=1}^{M} w_e \sigma_{ij}^e N_{,j}^{e\alpha} = f_i^{\alpha}, \qquad \alpha = 1, \dots, N$$
 (2)

where N^{ea} is the shape function of node α evaluated at the material point e within a finite element approximation scheme, and $\{w_e\}_{e=1}^M$ are elements of volume. The set of global states satisfying the above constraints define the equilibrium set E.

Instead of relying on a constitutive relation of the form $\sigma^e = \sigma^e(\varepsilon^e)$ for closure, the Data-Driven formulation of the problem consists of finding the global state z that satisfies the compatibility and equilibrium constraints and, at the same time, minimizes the distance to a given material data set D. Therefore, the local phase spaces Z_e are equipped with an appropriate metric:

$$|\mathbf{z}^e| = \mathbb{C}^e \,\boldsymbol{\varepsilon}^e \cdot \boldsymbol{\varepsilon}^e + \mathbb{C}^{e^{-1}} \boldsymbol{\sigma}^e \cdot \boldsymbol{\sigma}^e \tag{3}$$

where \mathbb{C}^e is a symmetric positive-definite tensor. Although the purpose of this tensor is numerical, and does not represent actual material behavior, it is typically given as the isotropic linear elasticity tensor:

$$\mathbb{C}^{e}_{ijkl} = \lambda \delta_{ij} \delta_{kl} + \mu (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) \tag{4}$$

Note that this introduces two parameters λ , μ to the problem, the choice of which may generally affect how well the comparibility or equilibrium constraints are satisfied. To avoid this issue, and at the same time, obtain a parameter-free scheme, one can alternatively introduce a nested optimization problem within the definition of the distance as follows (e.g. [KOA21]):

$$|\mathbf{z}^{e}| = \min_{\lambda,\mu>0} \mathbb{C}^{e}(\lambda,\mu) \,\boldsymbol{\varepsilon}^{e} \cdot \boldsymbol{\varepsilon}^{e} + \mathbb{C}^{e^{-1}}(\lambda,\mu) \boldsymbol{\sigma}^{e} \cdot \boldsymbol{\sigma}^{e}$$
(5)

In the following, we will assume that a constant \mathbb{C}^e is used, for the definition of the distance in the phase space. As a result, a metrization of the global phase space Z is induced by means of the norm:

$$|\mathbf{z}| = \sum_{e=1}^{N} w_e |\mathbf{z}^e|$$

The problem is mathematically formulated as:

$$\min_{\mathbf{y}\in D}\min_{\mathbf{z}\in E}|\mathbf{z}-\mathbf{y}|\tag{6}$$

where z denotes the *mechanical state* of the system i.e., the set of stress-strain pairs that satisfy equilibrium and compatibility, and y denotes the *material state* of the system i.e., the set of stress-strain pairs in the dataset.

The compatibility constraints are imposed by means of direct substitution, while the equilibrium constraints are enforced using Lagrange multipliers, resulting in the stationary problem:

$$\delta \left[\sum_{e} w^{e} |z^{e}| \left(\frac{1}{2} \sum_{\alpha} \left(N_{,j}^{e\alpha} u_{i}^{\alpha} + N_{,i}^{e\alpha} u_{j}^{\alpha} \right), \sigma_{ij}^{e} \right) - \sum_{\alpha} \left(\sum_{e} w_{e} \sigma_{ij}^{e} N_{,j}^{e\alpha} - f_{i}^{\alpha} \right) \eta_{i}^{\alpha} \right] = 0$$

$$(7)$$



Figure 1: a) Simple continuum with granular microstructure. b) Illustration of the stress-strain states in the material data set (D), and their projections on the equilibrium set (E) with highlighted iterative procedure leading to a minimum distance solution.

Taking all possible variations $(\delta u_i^{\alpha}, \delta \sigma_{ij}^e, \delta \eta_i^a)$, and manipulating the resulting equations, one obtains a system of Euler-Lagrange equations [KO16]:

$$\sum_{b}\sum_{e}w_{e}C_{ijkl}^{e^{*}}N_{,j}^{e\alpha}N_{,l}^{eb}u_{k}^{b} = \sum_{e}w_{e}C_{ijkl}^{e^{*}}N_{,j}^{e\alpha}\varepsilon_{kl}^{e^{*}}$$
(8)

$$\sum_{b} \sum_{e} w_{e} C_{ijkl}^{e^{*}} N_{,j}^{e\alpha} N_{,l}^{eb} \eta_{k}^{b} = f_{i}^{\alpha} - \sum_{e} w_{e} N_{,j}^{e\alpha} \sigma_{ij}^{e^{*}}$$
(9)

where $\mathbf{z}^{e^*} = (\boldsymbol{\varepsilon}^{e^*}, \boldsymbol{\sigma}^{e^*})$ are the optimal local data points in the data set D^e that result in the closest possible satisfaction of the constraints. Eqs 8 and 9 represent two standard linear elasticity problems, one in terms of \mathbf{u} , and one in terms of $\boldsymbol{\eta}$.

Solution algorithm

Note that the optimal local points $\mathbf{y}^e = (\boldsymbol{\varepsilon}^{e^*}, \sigma^{e^*})$ in the data set D^e are not known a priori, which therefore calls for an iterative solution scheme. The simplest algorithm involves a fixed point iteration, where a fixed material state $\mathbf{y}^{(k)}$ is projected onto E (i.e. Eqs 8, 9 are solved) to obtain the updated mechanical state $\mathbf{z}^{(k)}$, where k denotes the iteration number. Then a search through the data set is carried out to find the closest material state $\mathbf{y}^{(k+1)}$, and the process is repeated until the material states remain unchanged. For more details the interested reader is referred to [KO16, KSOA20].

2.2 Micropolar continuum - Elasticity

The simple or Cauchy continuum is known to have limitations when it comes to modeling geomaterials especially in the failure regime (shear localization), due to the absence of an internal length scale [MV87]. In this section we describe the extension of the data-driven computational mechanics framework to the micropolar continuum, following [KOA21]. We therefore proceed to consider the mechanical problem of a (nonlinear) elastic micropolar body that is discretized into N nodes and M material points, similar to Section 2.1. However, the body is now subject to not only applied

forces $\mathbf{f} = {\{\mathbf{f}^{\alpha}\}_{\alpha=1}^{N}}$ but also moments $\mathbf{m} = {\{\mathbf{m}^{\alpha}\}_{\alpha=1}^{N}}$ (Fig. 2). Its kinematics are described by displacements $\mathbf{u} = {\{\mathbf{u}^{\alpha}\}_{\alpha=1}^{N}}$ and microrotations $\boldsymbol{\theta} = {\{\theta^{\alpha}\}_{\alpha=1}^{N}}$ at its nodes. Analogously, The state of each material point is described by a stress-strain pair ($\varepsilon^{e}, \sigma^{e}$) and couple stress-curvature pair (κ^{e}, μ^{e}), which alltogether constitute a point in the local phase space i.e., $\mathbf{z}^{e} = (\varepsilon^{e}, \kappa^{e}, \sigma^{e}, \mu^{e})$, and the state of the entire system is collectively a point in the global phase space $\mathbf{z} = {\{\mathbf{z}^{e}\}_{e=1}^{M} \in Z}$. The micropolar system is subject to the following discretized compatibility and equilibrium constraints:

$$\varepsilon_{ij}^{e} = \sum_{\alpha} \left(N_{,j}^{e\alpha} u_{i}^{\alpha} + \epsilon_{ijk} N^{e\alpha} \theta_{k}^{\alpha} \right), \qquad e = 1, \dots, M \qquad (10)$$

$$\kappa_{ij}^e = \sum_{\alpha} N_{,j}^{e\alpha} \theta_i^{\alpha}, \qquad e = 1, \dots, M$$
(11)

$$\sum_{e=1}^{M} w_e \sigma_{ij}^e N_{,j}^{e\alpha} = f_i^{\alpha}, \qquad \alpha = 1, \dots, N$$
 (12)

$$\sum_{e=1}^{M} w_e \left(\mu_{ij}^e N_{,j}^{e\alpha} + \epsilon_{ijk} \sigma_{jk}^e N^{e\alpha} \right) = m_i^{\alpha}, \qquad \alpha = 1, \dots, N$$
(13)

where, besides the quantities already introduced in Section ??, ϵ_{ijk} is the third-order permutation tensor. The set of global states satisfying the above constraints define the equilibrium set E.

Analogously, we assume that a data set D is available, where material states reside. The micropolar formulation of the data-driven problem involes finding the global state z that satisfies the compatibility and equilibrium constraints and, at the same time, minimizes the distance to the material data set. To this end, we shall extend the metric introduced in the Cauchy continuum, to account for the additional kinematic and conjugate kinetic measures present in the micropolar continuum:

$$|\mathbf{z}^{e}| = \mathbb{C}^{e} \,\varepsilon^{e} \cdot \varepsilon^{e} + \mathbb{D}^{e} \,\boldsymbol{\kappa}^{e} \cdot \boldsymbol{\kappa}^{e} + \mathbb{C}^{e^{-1}} \boldsymbol{\sigma}^{e} \cdot \boldsymbol{\sigma}^{e} + \mathbb{D}^{e^{-1}} \,\boldsymbol{\mu}^{e} \cdot \boldsymbol{\mu}^{e}$$
(14)

where \mathbb{C}^e , \mathbb{D}^e are the isotropic micropolar elasticity tensors, which once again do not reflect actual material properties, but are introduced for solely distance-inducing purposes:

$$\mathbb{C}^{e}_{ijkl} = \lambda \delta_{ij} \delta_{kl} + (\mu + \kappa) \delta_{ik} \delta_{jl} + (\mu - \kappa) \delta_{il} \delta_{jk}$$
(15)

$$\mathbb{D}^{e}_{ijkl} = \alpha \delta_{ij} \delta_{kl} + (\gamma + \beta) \delta_{ik} \delta_{jl} + (\gamma - \beta) \delta_{il} \delta_{jk}$$
(16)

The Data-Driven problem retains the the same mathematical formulation as in the Cauchy problem:

$$\min_{\mathbf{y}\in D}\min_{\mathbf{z}\in E}|\mathbf{z}-\mathbf{y}|\tag{17}$$



Figure 2: a) Micropolar continuum with granular microstructure. b) Illustration of the stress-strain states in the material data set (D), and their projections on the equilibrium set (E) with highlighted iterative procedure leading to a minimum distance solution.

with the difference that the state space has higher dimensions, due to the presence of the additional conjugate quantities related to the field of microrotations.

The stationary problem reads:

$$\delta \left[\sum_{e} w^{e} |z^{e}| \left(\sum_{\alpha} N_{,j}^{e\alpha} u_{i}^{\alpha} + \sum_{\alpha} \epsilon_{ijk} N^{e\alpha} \theta_{k}^{\alpha}, \sum_{\alpha} N_{,j}^{e\alpha} \theta_{i}^{\alpha}, \sigma_{ij}^{e}, \mu_{ij}^{e} \right)$$
(18)
$$- \sum_{\alpha} \left(\sum_{e} w_{e} \sigma_{ij}^{e} N_{,j}^{e\alpha} - f_{i}^{\alpha} \right) \eta_{i}^{\alpha} - \sum_{\alpha} \left(\sum_{e} w_{e} (\mu_{ij}^{e} N_{,j}^{e\alpha} + \epsilon_{ijk} \sigma_{jk}^{e} N^{e\alpha}) - m_{i}^{\alpha} \right) \zeta_{i}^{\alpha} \right] = 0$$

Taking all possible variations ($\delta u_i^{\alpha}, \delta \theta_i^{\alpha}, \delta \sigma_{ij}^{e}, \delta \mu_{ij}^{e}, \delta \eta_i^{a}, \delta \zeta_i^{a}$), we obtain the following system of coupled Euler-Lagrange equations:

$$\sum_{b}\sum_{e}w_{e}\left(C_{ijkl}^{e}N_{,j}^{e\alpha}N_{,l}^{eb}u_{k}^{b}+C_{ijkl}^{e}N_{,j}^{e\alpha}N^{eb}\epsilon_{klm}\theta_{m}^{b}\right)=\sum_{e}w_{e}C_{ijkl}^{e^{*}}N_{,j}^{e\alpha}\varepsilon_{kl}^{e^{*}}$$
(19)
$$\sum_{b}\sum_{e}w_{e}\left[C_{ijkl}^{e}N^{e\alpha}N_{,l}^{eb}\epsilon_{ijm}u_{k}^{b}+\left(C_{ijkl}^{e}\epsilon_{ijm}\epsilon_{kln}N^{e\alpha}N^{eb}+D_{mjnl}^{e^{*}}N_{,j}^{e\alpha}N_{,l}^{eb}\right)\theta_{n}^{b}\right]$$

$$=\sum_{e}w_e \left(C^e_{ijkl}\epsilon_{ijm}N^{e\alpha}\epsilon^{e^*}_{kl} + D^e_{mjkl}N^{e\alpha}_{,j}\kappa^{e^*}_{kl}\right)$$
(20)

$$\sum_{b}\sum_{e}w_e \left(C^e_{ijkl}N^{eb}_{,j}N^{ea}_{,l}\eta^b_i + C^e_{ijkl}N^{eb}N^{ea}_{,l}\epsilon_{kij}\zeta^b_k\right) = f^{\alpha}_k - \sum_{e}w_e N^{e\alpha}_{,l}\sigma^{e^*}_{kl}$$
(21)

$$\sum_{b}\sum_{e}w_{e}\left[C_{ijkl}^{e}N^{e\alpha}N_{,l}^{eb}\epsilon_{ijm}\eta_{k}^{b}+\left(C_{ijkl}^{e}\epsilon_{ijm}\epsilon_{kln}N^{e\alpha}N^{eb}+D_{mjnl}^{e}N_{,j}^{e\alpha}N_{,l}^{eb}\right)\zeta_{n}^{b}\right]$$

$$= m_m^{\alpha} - \sum_e w_e \left(N_{,l}^{e\alpha} \mu_{ml}^{e^*} + \epsilon_{mkl} N^{e\alpha} \sigma_{kl}^{e^*} \right)$$
(22)

where $\mathbf{z}^{e^*} = (\varepsilon^{e^*}, \kappa^{e^*}, \sigma^{e^*}, \mu^{e^*})$ are the optimal local data points in the data set D^e that result in the closest possible satisfaction of the constraints. Eqs (19,20) and

(21,22) represent two micropolar linear elasticity problems, one in terms of $\mathbf{u}, \boldsymbol{\theta}$, and one in terms of $\boldsymbol{\eta}, \boldsymbol{\zeta}$.

2.3 Extension to inelasticity

Practical applications in geomechanics often concern deformations beyond the elastic regime, which involve history-dependence and irreversibility. For conciseness we will present the extension to inelasticity for the Cauchy problem, but the inelastic extension for the micropolar problem follows analogously [KOA21]. To this end, we attend to a time-discrete formulation, whereby the Data-Driven problem of at time t_{k+1} reads:

$$\min_{\mathbf{y}_{k+1}\in D_{k+1}} \min_{\mathbf{z}_{k+1}\in E_{k+1}} |\mathbf{z}_{k+1} - \mathbf{y}_{k+1}|$$
(23)

where $\mathbf{z}_{k+1} = {\{\mathbf{z}_{k+1}^e\}_{e=1}^M \in \mathbb{Z} \text{ and } \mathbf{z}_{k+1}^e = (\varepsilon_{k+1}^e, \sigma_{k+1}^e).$ The time-dependent constraint set E_{k+1} arises from the time-dependent applied forces \mathbf{f}_{k+1} . Accordingly, the behavior at a material point is described by a material data set D_{k+1}^e of points that is attainable at time t_{k+1} given its past local history of deformation:

$$D_{k+1}^e = \{ (\boldsymbol{\varepsilon}_{k+1}^e, \boldsymbol{\sigma}_{k+1}^e) \,|\, \text{past history} \}$$
(24)

In practical terms, this implies that one has to deal with evolving material data sets. The conceptually simplest yet computationally expensive parametrization of the history relies on keeping a (potentially truncated) memory of the strain history at a material point [EKR⁺19]. This mathematically translates to:

$$D_{k+1}^e = \{ (\boldsymbol{\varepsilon}_{k+1}^e, \boldsymbol{\sigma}_{k+1}^e) \, | \, \{ \boldsymbol{\varepsilon}_l^e \}_{l \le k} \}$$

$$(25)$$

which resembles a data-driven formulation focusing on trajectories – rather than points – in stress-strain space.

An alternative approach relies on enhancing the state space with a suitable set of internal variables q, which represent the evolving internal structure of the material at hand, and encapsulates its history [KSOA20, EKR⁺19]. In this case, the material data set admits the parametrization:

$$D_{k+1}^{e} = \{ (\boldsymbol{\varepsilon}_{k+1}^{e}, \boldsymbol{\sigma}_{k+1}^{e}) \, | \, (\boldsymbol{\varepsilon}_{k}^{e}, \boldsymbol{\sigma}_{k}^{e}, \boldsymbol{q}_{k}^{e}) \}$$
(26)

The internal variable parametrization outlined above can be replaced or enhanced with an energy-based parametrization, whereby the state space is augmented with the free energy \mathcal{A} and dissipation \mathcal{D} , which are related to the state variables ϵ, σ via the principle of conservation of energy and the second principle (Clausius-Plank inequality), stated in a time-discrete setting as:

$$\mathcal{D}_{k+1}^e - \mathcal{D}_k^e = \frac{\boldsymbol{\sigma}_k^e + \boldsymbol{\sigma}_{k+1}^e}{2} : (\boldsymbol{\epsilon}_{k+1}^e - \boldsymbol{\epsilon}_k^e) - (\mathcal{A}_{k+1}^e - \mathcal{A}_k^e) \ge 0$$
(27)

The local material data set at time t_{k+1} is then represented as:

$$D_{k+1}^e = \{ (\boldsymbol{\epsilon}_{k+1}^e, \boldsymbol{\sigma}_{k+1}^e) \, | \, (\boldsymbol{\epsilon}_k^e, \boldsymbol{\sigma}_k^e), (27) \}$$

$$(28)$$

The above relation states that the admissible stress-strain pairs at time t_{k+1} are those that are thermodynamically consistent with the material state at time t_{k} . The special case where $\mathcal{D}_{k+1}^e - \mathcal{D}_k^e = 0$ defines a bounded equilibrium set (or elastic domain) on the augmented state space. More details about the various options for history parametrization can be found in [EKR⁺19, KSOA20]. Note that in the case of the internal variable and energy-based parametrization, the necessary quantities can be obtained directly from lower scale models, e.g., as we will address in Section 2.4.

Regardless of the particular choice of parametrization of the local material data sets, the global material data set then follows as:

$$D_{k+1} = D_{k+1}^1 \times \ldots \times D_{k+1}^M$$
(29)

2.4 Data sampling

So far, we have assumed that material data are available, but we have not addressed the source of the data, or their potential scarcity. In principle, data can be obtained from various sources, either experimental or computational, and potentially combined within the same simulation. The experimental identification of material data sets has been addressed in [LCR⁺18], whereby a database of stress-strain couples is compiled from a given displacement field – obtained though imaging techniques – and known boundary conditions, by solving a distance minimization problem (Fig. 3 a). Alternatively, data can be compiled from high-fidelity lower scale simulations [KSOA20], which for geomechanical problems typically amount to discrete element models in case of granular assemblies, or finite element models of heterogeneous porous materials (Fig. 3 b). This approach gives rise a multiscale interpretation of data-driven computing. In the latter case, homogenization principles shall be used to derive the macroscopic quantities that describe the material states (stress, structural variables, energetical quantities). These principles are reviewed below in the case of discrete assemblies (e.g. granular media) and continuum representative volume elements (e.g. porous rocks).

Homogenization of granular ensemblies

In the case where *discrete* element simulations (e.g. [KHA⁺20]) are used for the purpose of generating the data, then the macroscopic material states (stress, strain, free energy, dissipation, microstructural variables) need to be obtained from the discrete quantities (particle displacements, interparticle forces). Assuming quasi-static conditions, the average stress tensor of the granular assembly is given by [CMNN81]:

$$\boldsymbol{\sigma} = \frac{1}{V} \sum_{c} \mathbf{f}^{c} \otimes \mathbf{l}^{c}$$
(30)



Figure 3: a) Identification of material data sets from in-situ experiment. b) Material data sets extracted from micromechanical RVEs, with examples showing a granular ensemble and a porous rock.

where the summation is performed over all contacts c of all particles in the assembly (unit cell), \mathbf{f}^c is the interparticle force, and \mathbf{l}^c is the branch vector connecting the centroids of contacting particles, and V is the volume of the granular assembly. The average strain ε is obtained from the boundary deformation of the unit cell, and the free energy density due to local deformation at the contacts is given by:

$$\mathcal{A} = \sum_{c} \mathcal{A}^{c} = \frac{1}{2V} \sum_{c} \left(\frac{\|\mathbf{f}_{n}^{c}\|^{2}}{k_{n}} + \frac{\|\mathbf{f}_{t}^{c}\|^{2}}{k_{t}} \right)$$
(31)

where k_n, k_t are the normal and tangential contact stiffnesses at an interparticle frictional contact, respectively, and $\mathbf{f}_n^c, \mathbf{f}_t^c$ are the normal and tangential components of the interparticle force.

The dissipation can be computed incrementally by energy balance:

$$d\mathcal{D} = \boldsymbol{\sigma} : d\boldsymbol{\varepsilon} - d\mathcal{A} \tag{32}$$

or, from the frictional slip at the interparticle contacts,

$$d\mathcal{D} = \sum_{c} d\mathcal{D}^{c} = \frac{1}{V} \sum_{c} \mathbf{f}_{t}^{c} \cdot d\mathbf{u}^{c, \, \text{slip}}$$
(33)

where $d\mathbf{u}^{c, \text{ slip}} = (\mathbf{f}_t^{c,t} - \mathbf{f}_t^{c,t+dt})/k_t$.

Microstructural measures acting as internal variables augmenting the state space (Section 2.3) may similarly be obtained by averaging microscopic quantities. For example, the commonly used contact normal fabric tensor [Oda72]:

$$\boldsymbol{F} = \frac{1}{2N_c} \sum_{c} \boldsymbol{n}^c \otimes \boldsymbol{n}^c \tag{34}$$

where N_c is the number of contacts in the assembly, and n^c is the contact normal vector. Analogous quantities can be obtained for nonsimple (polar) continua, as discussed in [KOA21].

Homogenization of heterogeneous porous geomaterials

In the case where *continuum* microstructural RVEs are used to generate material data sets, then similar principles apply. Assuming quasistatic conditions, the average stress tensor of the micromechanical RVE is given as:

$$\boldsymbol{\sigma} = \frac{1}{V} \int_{\partial V} \mathbf{t} \otimes \mathbf{x} \, dS \tag{35}$$

where t denotes the traction acting on the boundary of the continuum RVE, and \mathbf{x} denotes the position of the material point on the boundary.

The free energy, and dissipation are given simply by averaging, and energy balance respectively:

$$\mathcal{A} = \int_{\mathcal{U}} \mathcal{A}(\mathbf{x}) \, d\mathbf{x} \tag{36}$$

$$d\mathcal{D} = \boldsymbol{\sigma} : d\boldsymbol{\varepsilon} - d\mathcal{A} \tag{37}$$

while continuum texture tensors, similar to Eq. 34, may analogously be defined.

Adaptive sampling

In the above, it was assumed that micromechanical calculations are carried out along predetermined stress paths. In practice, these predetermined paths may not sufficiently cover the state space as needed for a specific application or boundary value problem.



Figure 4: Adaptive sampling procedure for multiscale data-driven computational mechanics.

In this case, we can leverage the formulation of data-driven computational mechanics as a distance minimization problem, and identify, using unsupervised learning, regions in state space with such poor data coverage. As shown in Fig. 4, these regions may then be targeted with additional experiments or lower-scale simulations, in an active learning manner. The result is a better coverage of the phase space for a given application or problem. The interested reader is referred to $[GKS^+23]$ for details on this adaptive sampling technique.

3 Applications

We present here two representative examples that leverage the theoretical and algorithmic depevelopments discussed above. The first one is a 2D flat punch indentation of a model elastic medium, as shown in Fig. 5 a). For simplicity, we restrict our attention to a simple and history-independent material behavior, by generating a dataset of $N = 10^6$ stress-strain pairs via evaluating an isotropic linear elastic law. The prediction was repeated using a sequence of successively larger datasets, until convergence of the predicted response was obtained. This convergence can be verified in Fig. 5 b), where the indenter force-displacement relation is plotted for increasing dataset size. The code needed to reproduce the example is openly available in the author's Github repository (github.com/kkarapiperis/ddcm-2D/). In the second example, shown in Fig. 5 c), a rupture propagates into a model granular material. In this case, the material behavior is furnished by the homogenized response of a discrete element assembly (Section 2.4), following the multiscale interpretation of the framework. The resulting surface deformation profile at the end of the simulation is shown in Fig. 5.



Figure 5: a) Example of a 2D flat punch indentation into an elastic medium. b) Indenter force-displacement curve prediction for increasing size of the dataset. c) Example of a fault rupture through a layer of model granular material. d) Predicted surface displacement profile, and comparison with a fully resolved discrete element simulation.

d), which compares well with a fully resolved discrete element simulation of the same problem.

4 Conclusions

In summary, the framework of data-driven computational mechanics, offers a novel avenue to solving problems in geomechanics, including challenging ones that involve failure and localized deformation. Free from the uncertainty of the classical constitutive modeling approach and the caveats of machine learning models, the data-driven formulation offers an alternative paradigm for computation. The need for large amounts of data represents a potential pitfall of the method, which may be addressed by the use of high-fidelity micromechanical simulations, augmenting data sets available from experiments. Finally, an additional remedy to the problem is furnished by the use of adaptive sampling techniques.

References

- [AHV⁺12] E. Andò, S.A. Hall, G. Viggiani, J. Desrues, and P. Bésuelle. Grainscale experimental investigation of localised deformation in sand: a discrete particle tracking approach. *Acta Geotechnica*, 7(1):1–13, Mar 2012.
- [AT09] J. E. Andrade and X. Tu. Multiscale framework for behavior prediction in granular media. *Mechanics of Materials*, 41(6):652–669, 2009. Advances in the Dynamics of Granular Materials.
- [BA06] R. I. Borja and J. E. Andrade. Critical state plasticity. part vi: Mesoscale finite element simulation of strain localization in discrete granular materials. *Computer Methods in Applied Mechanics and Engineering*, 195(37–40):5115–5140, 2006. John H. Argyris Memorial Issue. Part I.
- [Bar94] J.P. Bardet. Numerical simulations of the incremental responses of idealized granular materials. *International Journal of Plasticity*, 10(8):879–908, 1994.
- [BS21] Bahador Bahmani and WaiChing Sun. A kd-tree-accelerated hybrid data-driven/model-based approach for poroelasticity problems with multi-fidelity multi-physics data. *Computer Methods in Applied Mechanics and Engineering*, 382:113868, 2021.
- [BS22] Bahador Bahmani and WaiChing Sun. Manifold embedding datadriven mechanics. *Journal of the Mechanics and Physics of Solids*, 166:104927, 2022.

- [CDLSO20] Pietro Carrara, Laura De Lorenzis, Laurent Stainier, and Michael Ortiz. Data-driven fracture mechanics. *Computer Methods in Applied Mechanics and Engineering*, 372:113390, 2020.
- [CMNN81] J. Christoffersen, M. M. Mehrabadi, and S. Nemat-Nasser. A micromechanical description of granular material behavior. *Journal of Applied Mechanics*, 48(2):339–344, 1981.
- [CS79] P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Géotechnique*, 29(1):47–65, 1979.
- [DL82] F. Darve and S. Labanieh. Incremental constitutive law for sands and clays: Simulations of monotonic and cyclic tests. *International Journal for Numerical and Analytical Methods in Geomechanics*, 6(2):243– 275, 1982.
- [DM04] Y. F. Dafalias and M. T. Manzari. Simple plasticity sand model accounting for fabric change effects. *Journal of Engineering Mechanics*, 130(6):622–634, 2004.
- [DN05] F. Darve and F. Nicot. On incremental non-linearity in granular media: Phenomenological and multi-scale views (part i). *International Journal* for Numerical and Analytical Methods in Geomechanics, 29(14):1387– 1409, 2005.
- [EKR⁺19] R. Eggersmann, T. Kirchdoerfer, S. Reese, L. Stainier, and M. Ortiz. Model-free data-driven inelasticity. *Computer Methods in Applied Mechanics and Engineering*, 350:81–99, 2019.
- [EPW22] Mahdad Eghbalian, Mehdi Pouragha, and Richard Wan. A physicsinformed deep neural network for surrogate modeling in classical elasto-plasticity. *arXiv preprint arXiv:2204.12088*, 2022.
- [ESOR21] Robert Eggersmann, Laurent Stainier, Michael Ortiz, and Stefanie Reese. Model-free data-driven computamechanics enhanced by tensor voting. *Computer Methods in Applied Mechanics and Engineering*, 373:113499, 2021.
- [GKS⁺23] Anna Gorgogianni, Konstantinos Karapiperis, Laurent Stainier, Michael Ortiz, and José E Andrade. Adaptive goal-oriented data sampling in data-driven computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 409:115949, 2023.
- [GZ14] N. Guo and J. Zhao. A coupled fem/dem approach for hierarchical multiscale modelling of granular media. *International Journal for Numerical Methods in Engineering*, 99(11):789–818, 2014.
- [HAV23] Ehsan Haghighat, Sahar Abouali, and Reza Vaziri. Constitutive model characterization and discovery using physics-informed deep learning. *Engineering Applications of Artificial Intelligence*, 120:105828, 2023.

- [HBD⁺10] S.A. Hall, M. Bornert, J. Desrues, Y. Pannier, N. Lenoir, G. Viggiani, and P. Bésuelle. Discrete and continuum analysis of localised deformation in sand using x-ray ct and volumetric digital image correlation. *Géotechnique*, 60(5):315–322, 2010.
- [HHR22] Shenglin Huang, Zequn He, and Celia Reina. Variational onsager neural networks (vonns): A thermodynamics-based variational learning strategy for non-equilibrium pdes. *Journal of the Mechanics and Physics of Solids*, 163:104856, 2022.
- [HP06] G. T. Houlsby and A. M Puzrin. *Principles of Hyperplasticity*. Springer, 2006.
- [HRM⁺21] Ehsan Haghighat, Maziar Raissi, Adrian Moure, Hector Gomez, and Ruben Juanes. A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 379:113741, 2021.
- [KAVA18] R. Kawamoto, E. Andò, G. Viggiani, and J. E. Andrade. All you need is shape: Predicting shear banding in sand with LS-DEM. *Journal of the Mechanics and Physics of Solids*, 111:375–392, 2018.
- [KHA⁺20] Konstantinos Karapiperis, John Harmon, Edward Andò, Gioacchino Viggiani, and José E. Andrade. Investigating the incremental behavior of granular materials with the level-set discrete element method. *Journal of the Mechanics and Physics of Solids*, 144:104103, 2020.
- [KO16] T. Kirchdoerfer and M. Ortiz. Data-driven computational mechanics. Computer Methods in Applied Mechanics and Engineering, 304:81– 101, 2016.
- [KOA21] K. Karapiperis, M. Ortiz, and J.E. Andrade. Data-driven nonlocal mechanics: Discovering the internal length scales of materials. *Computer Methods in Applied Mechanics and Engineering*, 386:114039, 2021.
- [KRB07] K. Kamrin, C.H. Rycroft, and M. Z. Bazant. The stochastic flow rule: A multi-scale model for granular plasticity. *Modelling and Simulation* in *Materials Science and Engineering*, 15(4):S449, 2007.
- [KSOA20] K. Karapiperis, L. Stainier, M. Ortiz, and J.E. Andrade. Data-driven multiscale modeling in mechanics. *Journal of the Mechanics and Physics of Solids*, page 104239, 2020.
- [LCR⁺18] Adrien Leygue, Michel Coret, Julien Rethore, Laurent Stainier, and Erwan Verron. Data-based derivation of material response. *Computer Methods in Applied Mechanics and Engineering*, 331:184–196, 2018.
- [MA91] H.-B. Mühlhaus and E.C. Alfantis. A variational principle for gradient plasticity. *International Journal of Solids and Structures*, 28(7):845–857, 1991.

- [MSVMB21] Filippo Masi, Ioannis Stefanou, Paolo Vannucci, and Victor Maffi-Berthier. Thermodynamics-based artificial neural networks for constitutive modeling. *Journal of the Mechanics and Physics of Solids*, 147:104277, 2021.
- [MV87] H. B. Mühlhaus and I. Vardoulakis. The thickness of shear bands in granular materials. *Géotechnique*, 37(3):271–283, 1987.
- [NDR05] F. Nicot, F. Darve, and RNVO Group: Natural Hazards and Vulnerability of Structures. A multi-scale approach to granular materials. *Mechanics of Materials*, 37(9):980–1006, 2005.
- [Oda72] M. Oda. Initial fabrics and their relation to mechanical properties of granular materials. *Soils and Foundations*, 12(1):17–36, 1972.
- [OP04] M Ortiz and A Pandolfi. A variational cam-clay theory of plasticity. Computer Methods in Applied Mechanics and Engineering, 193(27):2645–2666, 2004. Computational Failure Mechanics for Geomaterials.
- [PRO23] Erik Prume, Stefanie Reese, and Michael Ortiz. Model-free data-driven inference in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 403:115704, 2023.
- [Ros70] K. H. Roscoe. The influence of strains in soil mechanics. Géotechnique, 20(2):129–170, 1970.
- [RSW58] K. H. Roscoe, A. N. Schofield, and C. P. Wroth. On the yielding of soils. *Géotechnique*, 8(1):22–53, 1958.
- [RY11] Richard A. Regueiro and Beichuan Yan. Concurrent multiscale computational modeling for dense dry granular materials interfacing deformable solid bodies. In Richard Wan, Mustafa Alsaleh, and Joe Labuz, editors, *Bifurcations, Instabilities and Degradations in Geomaterials*, pages 251–273. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [UGK⁺23] Jacinto Ulloa, Anna Gorgogianni, Konstantinos Karapiperis, Michael Ortiz, and José E Andrade. Data-driven breakage mechanics: Predicting the evolution of particle-size distribution in granular media. *Journal of the Mechanics and Physics of Solids*, page 105328, 2023.
- [VA91] I. Vardoulakis and E.C. Aifantis. A gradient flow theory of plasticity for granular materials. *Acta Mechanica*, 87(3):197–217, 1991.
- [VS23] Nikolaos N Vlassis and WaiChing Sun. Geometric learning for computational mechanics part ii: Graph embedding for interpretable multiscale plasticity. *Computer Methods in Applied Mechanics and Engineering*, 404:115768, 2023.

Non-Euclidean machine learning for geomechanics

WaiChing Sun

Columbia University, New York, USA

This book chapter is intended to provide a concise review on how to train, verify and validate constitutive models enhanced by graph-theoretic data. We begin our discussion by reviewing basic concepts of the various types of graphs and under what situations does a graph may serve as an ideal data structures for representing data. We then outline a few usages on how to use graph convolutional neural networks to perform embedding and how these embedding can be potentially useful for constitutive modeling, material designs, and inverse problems.

1 Motivations

In 1736, Mathematician Leonhard Euler was asked to solve one of the most famous mathematics problems – Is it possible to visit all the seven bridges at Königsberg, Prussia, on both sides of the Preger River without repeating any path. This is the Seven Bridges of Königsberg problem (see Fig. 1). The issue Euler faced was that there has not yet been techniques invented to formulate the problem in proper abstract terms. Euler solved this problem by inventing these abstract terms, which laid the foundations of modern graph theory. His conclusion is that such a route, which is now referred to as the Eulerian path, does not exist in the Seven Bridges of Königsberg problem.

The abstract terms Euler invented the vertex(or node) set \mathbb{V} , which are all the stops a pedestrian may take, and the edge set, which are the seven bridges \mathbb{E} . Together, they form a 2-tuple, which is nowadays referred as an undirected graph, or simply a graph, which is often denoted as $\mathbb{G}(\mathbb{V}, \mathbb{E})$. Here, the graph Euler invented represents the topology of the seven bridges, as whether the Eulerian path exists depends on how the bridges are connected. On the other hand, information such as how long the bridges span, and whether the bridges are safe are irrelevant to the existence of the Eulerian path.

Now let's think of a different case. Consider the polycrystal structure, shown in Fig.



Figure 1: The Seven Bridges of Königsberg problem that leads to the birth of graph theory. Figure obtained from Wikipedia



Figure 2: Polycrystal interpreted as a connectivity graph. The graph is undirected and weighted at the nodes. Figure reproduced from Vlassis et al. (2020).

2. Let's say our goal is to determine the effective Young's modulus of this polycrystal. We are given only (1) the orientation of each crystal in this assemble as well as a database which contains the effective Young's modulii of many other polycrystals formed by the same type of crystals.

What should we do?

In this case, we involve in a different type of problem in which we are no longer concerning finding a sequence from the edge set \mathbb{E} , but we are interested in predictions where we take the micro-structure represented by a graph as the input and output the Young's modulus, i.e.,

$$f: \mathbb{G} \to \mathbb{R}^+. \tag{1}$$

The mapping defined in Eq. (1) may look like a function that maps a vector to a real number. However, unlike a vector, where we can get a sense of proximity between two vectors via the inner product or norm equipped by the vector space, it is not trivial to measure the distance between two graphs properly. This issue makes it difficult to perform any interpolation or extrapolation.

Sun 133

In the history of classical mechanics, our intellectual ancestors often bypass this issue by introducing descriptors that implicitly describe some aspect of the microstructures. For instance, void ratio is one of the most used measures of microstructures in soil plasticity model (Schofield and Wroth, 1968; Mitchell et al., 2005; Zhu et al., 2016; Borja, 2013). With this descriptor, we then hypothesize the existence of a critical state as a function of void ratio and stress. Other material models, such as the bounding surface model with critical state that evolves with fabric tensors (Dafalias and Manzari, 2004), breakage theory (Einav, 2007) can be considered as other instances of this descriptor-driven modeling approach. The central idea is to explain the physics by building models that can corroborate well with experimental observations by supplementing a set of hypotheses formulated around these descriptors. Remarkably, the hierarchy of these hypotheses can also be abstracted as a graph, i.e., a directed graph, in which the direction of the edge is defined to represent a hierarchical relation such as cause-and-effort, and orders of a sequence, as shown in Fig. 3.



(c) The digraph violates Rule No. 4, since there exists a (d) The digraph violates Rule No. 5, since *CN* and A_f are cycle $\phi \rightarrow A_f \rightarrow CN \rightarrow \phi$. not on any paths leading from δ to t.

Figure 3: Directed graph used to represent different models of traction-separation law generated by artificial intelligence. Figure reproduced from Wang and Sun (2019).

However, an important issue central to this formalism is that the set of descriptors incorporated into the constitutive laws could be insufficient to accurately and precisely describe the microstructures. For example, if void ratio are the only physical quantities incorporated into the hardening law of a plasticity model, then the two specimens with completely different fabrics but with the same void ratio will be considered identical. In other words, the capacity of a model to make accurate, robust and precise predictions depends strongly on the expressivity of the set of descriptors.

1.1 Why machine learning?

Consider again the polycrystal structure shown in Fig. 2. What if we can formulate a method in which we can put all the different graphs that represent the microstructures in a vector space such that the microstructures that are close are given position vectors that are close to each other and microstructures that are distant are given position vectors that are far apart? If this can be achieved, we can leverage the properties of the N-dimensional Euclidean space \mathbb{R}^N to perform regression to formulate material models (Vlassis et al., 2020). This technique is often referred as graph embedding in the literature (Hamilton et al., 2017).

In Vlassis et al. (2020), for instance, a modeling framework is formulated to combine unsupervised learning, which embeds high-dimensional data onto simpler geometry with sufficient smoothness, and supervised mapping, which introduces mapping for labeled data (see Fig. 4) which involves Sobolev training, a technique used for training hyperelasticity material models with gradient data (Vlassis and Sun, 2021; Vlassis et al., 2022). This setup can be considered as a standard pipeline for building models not only in mechanics but in almost every applications including but not limited to computational chemistry, biology, recommendation system and games (Chami, 2021) (see Fig. 4).



Figure 4: Overview of a typical machine learning pipeline. Figure reproduced from Chami (2021).

1.2 Organization of this Chapter

This chapter provides a brief review on the state-of-the-art of graph embedding and autoencoder techniques used for computational mechanics. We will begin by reviewing the basic terminology for graphs in Section 2. Then, we explain the concepts of graph autoencoder and graph embedding in Section 3 and provide an example in which internal varialbes are generated from the graph neural network (Section 4).

2 Basic terminology of graphs

In this section, a brief review of several terms of graph theory is provided to facilitate the illustration of the concepts in this current work. More elaborate descriptions can be found in Graham et al. (1989); West et al. (2001); Bang-Jensen and Gutin (2008):

Definition 2.1. A graph is a two-tuple $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ where $\mathbb{V} = \{v_1, ..., v_N\}$ is a non-empty vertex set (also referred to as nodes) and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is an edge set. To define a graph, there exists a relation that associates each edge with two vertices (not necessarily distinct). These two vertices are called the edge's endpoints. The pair of endpoints can either be unordered or ordered.

Definition 2.2. An **undirected graph** is a graph whose edge set $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ connects *unordered* pairs of vertices together.

Definition 2.3. A **loop** is an edge whose endpoint vertices are the same. When all the nodes in the graph are in a loop with themselves, the graph is referred to as allowing self-loops.



Figure 5: Different types of graphs. (a) Undirected (simple) binary graph (b) Directed binary graph (c) Edge-weighted undirected graph (d) Node-weighted undirected graph.

Definition 2.4. Multiple edges are edges having the same pair of endpoint vertices.

Definition 2.5. A simple graph is a graph that does not have loops or multiple edges.

Definition 2.6. Two vertices that are connected by an edge are referred to as **adjacent** or as **neighbors**.

Definition 2.7. The term weighted graph traditionally refers to graph that consists of edges that associate with edge-weight function $w_{ij} : \mathbb{E} \to \mathbb{R}^n$ with $(i, j) \in \mathbb{E}$ that maps all edges in \mathbb{E} onto a set of real numbers. n is the total number of edge weights and each set of edge weights can be represented by a matrix W with components w_{ij} .

In this current work, unless otherwise stated, we will be referring to weighted graphs as graphs weighted at the vertices - each node carries information as a set of weights

that quantify features of microstructures. All vertices are associated with a vertexweight function $f_v : \mathbb{V} \to \mathbb{R}^D$ with $v \in \mathbb{V}$ that maps all vertices in \mathbb{V} onto a set of real numbers, where D is the number of weights - features. The node weights can be represented by a $N \times D$ matrix X with components x_{ik} , where the index $i \in [1, ..., N]$ represents the node and the index $k \in [1, ..., D]$ represents the type of node weight feature.

Definition 2.8. A graph whose edges are unweighted ($w_{\epsilon} = 1 \quad \forall \epsilon \in \mathbb{E}$) can be called a **binary graph**.

To facilitate the description of graph structures, several terms for representing graphs are introduced:

Definition 2.9. The **adjacency matrix** A of a graph \mathbb{G} is the $N \times N$ matrix in which entry α_{ij} is the number of edges in \mathbb{G} with endpoints $\{v_i, v_j\}$, as shown in Eq. 2.

$$\alpha_{ij} = \begin{cases} 1, & v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise.} \end{cases}$$
(2)

Definition 2.10. If the vertex v is an endpoint of edge ϵ , then v and ϵ are **incident**. The **degree** d of a vertex v is the number of incident edges. The **degree matrix** D of a graph \mathbb{G} is the $N \times N$ diagonal matrix with diagonal entries d_i equal to the degree of vertex v_i , as shown in Eq. (3).

$$deg_{ij} = \begin{cases} d_i, & i = j \\ 0, & \text{otherwise.} \end{cases}$$
(3)

Definition 2.11. An **isomorphism** from a graph \mathbb{G} to another graph \mathbb{H} is a bijection g that maps $\mathbb{V}(\mathbb{G})$ to $\mathbb{V}(\mathbb{H})$ and $\mathbb{E}(\mathbb{G})$ to $\mathbb{E}(\mathbb{H})$ such that each edge of \mathbb{G} with endpoints u and v is mapped to an edge with endpoints g(u) and g(v). Applying the same permutation to both the rows and the columns of the adjacency matrix of graph \mathbb{G} results to the adjacency matrix of an isomorphic graph \mathbb{H} .

Definition 2.12. The **unnormalized Laplacian operator** Δ is defined such that:

$$(\Delta f)_i = \sum_{j:(i,j)\in\mathbb{E}} w_{ij}(f_i - f_j) \tag{4}$$

$$= f_i \sum_{j:(i,j)\in\mathbb{E}} w_{ij} - \sum_{j:(i,j)\in\mathbb{E}} w_{ij} f_j.$$
(5)

By writing the equation above in matrix form, the unnormalized Laplacian matrix Δ of a graph \mathbb{G} is the $N \times N$ positive semi-definite matrix defined as $\Delta = D - W$.

In this current work, binary graphs will be used, thus, the equivalent expression is used for the unnormalized Laplacian matrix L, defined as L = D - A with the entries l_{ij} calculated as:

$$l_{ij} = \begin{cases} d_i, & i = j \\ -1, & i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise.} \end{cases}$$
(6)

Definition 2.13. For binary graphs, the symmetric normalized Laplacian matrix L^{sym} of a graph \mathbb{G} is the $N \times N$ matrix defined as:

$$L^{\text{sym}} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}.$$
 (7)

The entries l_{ij}^{sym} of the matrix L^{sym} are shown in Eq. 8.

$$l_{ij}^{\text{sym}} = \begin{cases} 1, & i = j \text{ and } d_i \neq 0\\ -(d_i d_j)^{-\frac{1}{2}}, & i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0, & \text{otherwise.} \end{cases}$$
(8)

3 Graph embedding

Graph embedding is a technique in which one attempts to create a mapping that maps a graph onto a geometry, which in most case is an Euclidean space. This mapping must be constructed in a way such that the distance between any two graphs are preserved to prevent information loss. In fact, one may consider convolutional neural network also as a special case of graph embedding where the grid is regarded as a graph. This special case is simpler in the sense that a node at any point in the grid system has a fixed number of neighbors (2 for one-dimensional cases, 4 for 2D cases,...etc).

However, there are many systems in mechanics, such as granular assembles, knowledge represented in graphs, where the topology is not fixed. In such a case, the topology of the graphs must be considered in the embedding process. Earlier attempts on graph embedding often relies on the eigen-decomposition of the graph Laplacian to obtain vector representations of graphs. However, the major drawback of this approach is that the the eigen-decomposition only makes sense to compare graphs of the same number of nodes. The eigen-decomposition also makes it difficult to focus on specific local effects in the graphs, which can be quite important for applications, such as fracture or strain localization (Goyal and Ferrara, 2018).

An alternative approach, which we will describe in this section, is to introduce a twostep procedure where one first perform node embedding, i.e., mapping each node of the graphs to a vector in the embedding space (see Fig. 6), then introduce a graph pooling to aggregate information of the entire graph into a single encoded feature

vector. This technique is commonly referred as message-passing graph neural network in the literature.

Earlier work of message-passing graph embedding does not necessarily involve any neural network. For instance, in DeepWalk, Perozzi et al. (2014) uses truncated random walk to aggregate relations among nodes in a graph and use a similarity metric to embed data from social networks. Later work, such as graph convolution neural network (Kipf and Welling, 2016), graph attention neural network (Velickovic et al., 2017), and graph isomorphism network (Xu et al., 2018), all introduce neural networks to aggregate information for the node embedding. A key result for this change in design is to improve the expressivity of the embedding. In particular, we want to be able to distinguish graphs that are different by avoiding them to be mapped onto the same encoded feature vectors. As we will see in the latter discussion, the successful embedding with an expressive neural network is the necessary (but not sufficient) condition for the successful downstream tasks, including the building the decoder (i.e. the inverse mapping from the embedding vector space back to the graph), as well as other prediction or classification problems.



Figure 6: Graph autoencoder architecture. Notice that the input and output graphs are not necessarily sharing the same adjacency matrix.

The rest of this section is organized as follows. We first provide an example of representing finite element solution as a weighted graph (Section 3.1). We then introduce the neural network architecture used in Section 3.2. Then, we present the formulation of these learning tasks: Section 3.3 shows how we learn the mapping for building ${}^{h}\tilde{\mathbb{G}}$ and then find the reduced ordered latent space; Section 3.4 shows how to predict the finite element solutions with p based on the latent space.

3.1 Finite element solution represented by graphs

We consider each node of the finite elements as a graph vertex. Graph edges are assigned for each pair of vertices that are nodes of a finite element edge (Vlassis and Sun, 2023). Accordingly, each nodal solution can be stored as the weight of the corresponding vertex such that the finite element solution can be stored as an undirected node-weighted graph. By assuming that we only use the same set of bases for the testing and interpolating functions of all finite elements, we eliminate the need to in-
troduce edge weights for the edge set. This setting simplifies the graph representation of the finite element solutions.

In the following part of this section, we will introduce the mathematical expression of finite element node graph as a foundation for our machine learning model. A finite element node graph is an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i | i = 1, ..., N\}$ is the node set of the graph as v_i corresponding to individual finite element nodes, and $\mathcal{E} = \{(v_{1j}, v_{2j}) | j = 1, ..., M; v_{1j}, v_{2j} \in \mathcal{V}\}$ is the edge set where the existence of each individual edge indicates that node v_{1j} and v_{2j} belong to the same element. N and M indicate the size of the node set and edge set. In order to deal with geometrical features to be incorporated in the machine learning model, we enrich the graph representation as a node-weighted graph: $\mathcal{G}' = (\mathcal{V}, \mathcal{E}, \mathcal{X})$ where $\mathcal{X} = \{\mathbf{x}_i \in \mathbb{R}^D | i = 1, ..., N\}$ is the nodal feature set as \mathbf{x}_i indicates the geometrical feature vector at node v_i with a dimension of D. We may enforce $D \geq 3$ as the manifold reconstruction task requires at least the three spatial coordinates of the finite element point cloud.

3.2 Graph autoencoder architecture

We adopt the graph isomorphism network (GIN) (cf. Xu et al. (2018)) to perform the embedding task. We would like to discuss GIN, because it is a message-passing model capable of discriminating different graph structures identified by the Weisfeiler-Lehma isomorphism test (Weisfeiler and Leman, 1968). Providing that we use a proper graph pooling layer, the embedding of GIN is inherently permutation invariance, which means that the ordering of the nodes will not affect the predictions. More importantly, the fact that GIN passes the isomorphism test enables us to distinguish non-isomorphic subgraphs by mapping them onto different encoded latent vectors and vice versa – a feat that cannot be achieved by the conventional graph convolutional network and GraphSAGE (Xu et al., 2018). These two features combined improve the expressive power of the GIN such that the relationships among finite element nodes can be captured by the neural network. Fig. 7 shows the architecture of the graph autoencoder designed for the finite element problems. The encoder part of this architecture takes in the adjacency matrix and feature matrix of the input graph \mathcal{G}'_{in} : (A_{in}, X_{in}) and produces an encoded feature vector $h_{
m enc}$, which is denoted as a functional expression: $h_{enc} = Enc(X_{in}, A_{in})$. The decoder part of this architecture then utilizes the encoder output to produce a decoded feature matrix X. The adjacency matrix of the output graph A_{out} can be assumed as a prior (since each dataset shares one same A) in order to complete an output graph $\mathcal{G}'_{\mathrm{out}}$: $(A_{\mathrm{out}}, \tilde{X})$, which could be written as $\tilde{X} = \text{Dec}(h_{\text{enc}})$. Problem formulation is generally based on supervision of the decoded output \tilde{X} , which will be the focus of the following sections. The rest of this section will present details about how the layer components shown in Fig. 7 operate.

We first introduce one of the most widely-used architectures called multi-layer perceptron (MLP), which is included as a substructure in our graph autoencoder architecture.



Figure 7: Graph autoencoder architecture. Notice that the input and output graphs are not necessarily sharing the same adjacency matrix.

MLP is a functional approximation expressed as follows:

$$MLP(\boldsymbol{X}) = \boldsymbol{W}^{(K)} \cdot act(\boldsymbol{W}^{(K-1)} \cdot act(...act(\boldsymbol{W}^{(1)} \cdot \boldsymbol{X} + \boldsymbol{b}^{(1)})...) + \boldsymbol{b}^{(K-1)}) + \boldsymbol{b}^{(K)}$$
(9)

where W and b are called the weight matrix and the bias vector of the MLP substructure, respectively. The superscript (K) indicates the K-th layer of the MLP substructure. $act(\cdot)$ is called the activation function of individual layers; here we adopt the rectified linear unit (ReLU) for $act(\cdot)$ such that ReLU(x) = x if x > 0 otherwise ReLU(x) = 0.

We then focus on the GIN convolution layers, which take in some adjacency matrix and feature matrix and output an embedded feature matrix. The matrix formulation of a GIN layer is:

$$\boldsymbol{H}^{(k)} = \mathrm{MLP}^{(k)} \left((\boldsymbol{A} + (1+\epsilon)\boldsymbol{I}) \cdot \boldsymbol{H}^{(k-1)} \right)$$
(10)

where the superscript (k) indicates the k-th layer in the architecture; H is the embedded nodal feature matrix coming from the output of the previous layer with $H^{(1)} = X$ at the input layer. ϵ is a learnable parameter. For consecutive GIN convolution layers, the following layer accepts the same A as the previous layer. For the beginning GIN layer in both the encoder and the decoder, A should be prescribed as either A_{in} or A_{out} .

Our architecture also includes global operations on the graph. The graph global mean pooling operation in the encoder performs the following computation:

$$\boldsymbol{h}_{\text{avg}} = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{h}_i \tag{11}$$

where h_i is the embedded nodal feature of v_i corresponding to the *i*-th row of H, and h_{avg} is the resultant graph feature vector from global mean pooling.

The broadcasting operation in the decoder is a matrix reshape operation that converts a row vector h_{dec} of size ND_{enc} coming from the output of an MLP substructure in the decoder, to an embedded nodal feature matrix of size $N \times D_{enc}$ such that:

$$h_{ij} = (\boldsymbol{h}_{\text{dec}})_{j+(i-1)D_{\text{enc}}} \tag{12}$$

where h_{ij} indicates the *j*-th component of the embedded nodal feature vector h_i after broadcasting.

3.3 Learning problem for the finite element or discrete element simulations

In the previous section, we discuss the strategy to construct the mapping $\mathcal{F}({}^{l}\mathcal{G}'_{j}) = {}^{h}\tilde{\mathcal{G}}'_{j}$ such that ${}^{h}\tilde{\mathcal{G}}'_{j} \sim {}^{l}\mathcal{G}'_{j}$ to establish an augmented data set ${}^{h}\tilde{\mathbb{G}}$. We assume that \mathcal{F} is learned in a supervised manner: we aim to minimize the discrepancy between the nodal feature matrix of training labels and the approximated nodal feature matrix output by the neural network. We intuitively construct the training labels with {}^{h}\mathbb{G}, and thus we required that for each snapshot in {}^{h}\mathbb{G} there exist some snapshot in {}^{l}\mathbb{G} corresponding to the results with the same loading condition, which is summarized as follows:

$$\exists^{lh} \mathbb{G} \subset {}^{l} \mathbb{G} \quad \text{s.t.} \quad {}^{h} \mathbb{G} = \left\{ \mathcal{F}({}^{lh} \mathcal{G}'_{j}) | {}^{lh} \mathcal{G}'_{j} \in {}^{lh} \mathbb{G} \right\}$$
(13)

The subset ${}^{lh}\mathbb{G}$ then constructs the training input set. We next approximate \mathcal{F} with the graph autoencoder proposed in Section 3.2 mainly computing the decoded nodal features as $\tilde{X} = \hat{F}(X)$. The loss function is adopted as the node-wise mean square error of the nodal feature matrix, which leads to the following training objective:

$$\min_{\boldsymbol{\Theta}^{F}} \frac{1}{N_{s}^{lh}} \sum_{i=1}^{N_{s}^{lh}} \left\| \hat{\boldsymbol{F}}(\boldsymbol{X}_{(i)}^{lh}) - \boldsymbol{X}_{(i)}^{h} \right\|_{\text{fro}}^{2}, \quad \hat{\boldsymbol{F}}(\boldsymbol{X}) = \text{Dec}_{F}(\text{Enc}_{F}(\boldsymbol{X}, \boldsymbol{A}^{l})) \quad (14)$$

where Θ^F is the collection of all trainable network parameters of $\hat{F}(\cdot)$. N_s^{lh} is the size of ${}^{lh}\mathbb{G}$ as well as ${}^{h}\mathbb{G}$. The subscript (*i*) indicates the *i*-th sample in ${}^{h}\mathbb{G}$ or ${}^{lh}\mathbb{G}$, while the sample sequence satisfies ${}^{h}\mathcal{G}'_{i} \sim {}^{lh}\mathcal{G}'_{i}$ with ${}^{h}\mathcal{G}'_{i} : (A^{h}, X_{(i)}^{h})$ and ${}^{lh}\mathcal{G}'_{i} : (A^{l}, X_{(i)}^{lh})$. The operator $\|\cdot\|_{\text{fro}}$ indicates the Frobenius norm of a matrix. The subscript *F* of Dec and Enc indicates the decoder and encoder function for the graph autoencoder approximating \mathcal{F} , in order to differentiate from the reconstruction autoencoder mentioned in the following part. The approximated mapping $\hat{F}(\cdot)$ helps us to enrich the high-fidelity dataset as follows:

$${}^{h}\tilde{\mathbb{G}} = \left\{ {}^{h}\tilde{\mathcal{G}}_{j}': (\boldsymbol{A}^{h}, \tilde{\boldsymbol{X}}_{(j)}^{h}) \mid \tilde{\boldsymbol{X}}_{(j)}^{h} = \left\{ \begin{array}{l} \hat{\boldsymbol{F}}(\boldsymbol{X}_{(j)}^{l}), & {}^{l}\mathcal{G}_{j}': (\boldsymbol{A}^{l}, \boldsymbol{X}_{(j)}^{l}) \in {}^{l}\mathbb{G} \setminus {}^{lh}\mathbb{G}, \\ \boldsymbol{X}_{(j)}^{l}, & {}^{l}\mathcal{G}_{j}': (\boldsymbol{A}^{l}, \boldsymbol{X}_{(j)}^{l}) \in {}^{lh}\mathbb{G}. \end{array} \right\}$$
(15)

Here we realize the significance of having a low-fidelity dataset ${}^{l}\mathbb{G}$ and the mapping \mathcal{F} : we are now able to construct a relatively large high-fidelity dataset to improve the reduced ordered model without spending too much effort performing experiments on the high-fidelity scales.

With ${}^{h}\tilde{\mathbb{G}}$ populated from ${}^{l}\mathbb{G}$, we are ready to formulate the graph embedding problem that determines the reduced ordered latent space. The general idea is to construct

a graph autoencoder function $\hat{R}(\cdot)$ whose output approximates its input itself. We still adopt the loss function as the node-wise mean square error for the feature matrix between the labels in ${}^{h}\tilde{\mathbb{G}}$ and the output from $\hat{R}(\cdot)$, which yields the following training objective:

$$\min_{\boldsymbol{\Theta}^{R}} \frac{1}{N_{s}^{hh}} \sum_{i=1}^{N_{s}^{hh}} \left\| \hat{\boldsymbol{R}}(\tilde{\boldsymbol{X}}_{(i)}^{h}) - \tilde{\boldsymbol{X}}_{(i)}^{h} \right\|_{\text{fro}}^{2}, \quad \hat{\boldsymbol{R}}(\boldsymbol{X}) = \text{Dec}_{R}(\text{Enc}_{R}(\boldsymbol{X}, \boldsymbol{A}^{h})) \quad (16)$$

where Θ^R is the collection of all trainable network parameters of $\hat{R}(\cdot)$. N_s^{hh} is the size of ${}^h \tilde{\mathbb{G}}$ as well as ${}^l \mathbb{G}$. The subscript R indicates the encoder and decoder function of the reconstruction autoencoder. The reduced ordered latent space \mathbb{L} is then defined as the space spanned by $h_{\text{enc}} = \text{Enc}_R(X, A^h)$ for arbitrary X coming from an admissible deformed configuration \mathcal{G}' , where $\mathbb{L} \subset \mathbb{R}^{D_{\text{enc}}}$.

3.4 Predicting the high-fidelity results without full-scale simulations

This section presents how we utilize the parametrized load p to predict the actual deformed configuration G'. As we find the reduced ordered latent space \mathbb{L} in the previous section, we may notice that \mathbb{L} is generally entangled, which does not ideally captures the reduced ordered dynamics in the maximum sense. We here propose to construct a response controlling law on \mathbb{L} based on p corresponding to the configuration of interest to disentangle the reduced ordered latent space, denoted as a functional expression $h_{\text{enc}} = f(p)$. We then approximately parametrize the response controlling law function as $\hat{f}(\cdot)$ by a simple feed-forward neural network with MLP architecture. We fit $\hat{f}(\cdot)$ with the mean square error loss between the encoded feature vector obtained from the graph autoencoder and that computed by $\hat{f}(\cdot)$, which yields the following learning objective:

$$\min_{\boldsymbol{\Theta}^{f}} \frac{1}{N_{s}^{hh}} \sum_{i=1}^{N_{s}^{hh}} \left\| \hat{\boldsymbol{f}}(\boldsymbol{p}_{(i)}) - \operatorname{Enc}_{R}(\tilde{\boldsymbol{X}}_{(i)}^{h}, \boldsymbol{A}^{h}) \right\|_{2}^{2}$$
(17)

where Θ^{f} is the collection of all trainable network parameters of $\hat{f}(\cdot)$. $p_{(i)}$ is the loading condition corresponding to the *i*-th snapshot ${}^{h}\tilde{g}'_{i}: (\boldsymbol{A}^{h}, \tilde{\boldsymbol{X}}^{h}_{(i)})$. The operator $\|\cdot\|_{2}$ is the vector Euclidean norm.

After we learn the neural network approximation of $f(\cdot)$, the prediction of highfidelity results is to determine the nodal feature matrix \bar{X} given some loading condition \bar{p} as follows:

$$\bar{\boldsymbol{X}} = \text{Dec}_R(\hat{\boldsymbol{f}}(\bar{\boldsymbol{p}})) \tag{18}$$

In essence, we introduce a graph neural network approach to construct a response surface. Since (1) the data are obtained from the simulations that obey the balance principles and (2) a successful embedding should be capable of preserving the relationships among nodes, we hypothesize that this will give us more accurate and robust

predictions than other alternatives that employ basis functions that directly interpolate the hypersurface in the ambient space \mathbb{R}^{N+P} .

4 Numerical Example: Training of interpretable graph embedding internal variables

In this section, we describe a procedure of embedding field simulation data to construct graph-based internal variables. The content of this section is revised from a section in Vlassis and Sun (2023).

A graph convolutional autoencoder is used to compress the graph structures that carry the plastic deformation distribution of a microstructure. In Section 4.1, we demonstrate the process of generating the plasticity data through finite element method (FEM) simulations and post-processing them into weighted graph structures. In Section 4.2, we showcase the performance capacity of the autoencoder architecture as well as its ability to reproduce the plasticity graph structures. Finally, in Section 4.3, we perform a sensitivity training test for the autoencoder architecture on different FEM meshes for the same microstructure.

4.1 Generation of the plasticity graph database

In this work, the autoencoders used for the generation of the graph-based internal variables and the neural network constitutive models used for the forward predictions are trained on data sets generated by FEM elastoplasticity simulations. To test the autoencoders' capacity to generate encoded feature vectors regardless of the microstructure and plastic strain distribution patterns the FEM mesh represents, we test the algorithm with two microstructures of different levels of complexity. The two microstructures A and B are demonstrated in Fig. 8 (a) and (b) respectively. The outline of the microstructures is a square with a side of 1 mm. This figure also shows the meshing of the two microstructures. The microstructures A and B are discretized by 250 and 186 triangular elements respectively with one integration point each. An investigation of different mesh sizes and the sensitivity of the encoded feature generation is demonstrated in Section 4.3. Each integration point of mesh corresponds to a node in the equivalent graph (also shown in Fig. 8). The integration points of the neighboring elements - elements that share at least one vertex - are connected with an edge in the constructed graph.

The constitutive model selected for the local behavior at the material points was linear elasticity and J2 plasticity with isotropic hardening. The local behavior is predicted with an energy minimization algorithm. The local optimization algorithm is omitted for brevity. The local linear elastic material has a Young's modulus of E = 2.0799MPa and a Poisson ratio of $\nu = 0.3$. The local J2 plasticity has an initial yield stress of 100kPa and a hardening modulus of H = 0.1E. During the simulation, the elastic, plastic, and total strain as well as the hyperelastic energy functional and stress are saved for every integration point.



Figure 8: Two microstructures represented as a finite element mesh and an equivalent node-weighted undirected graph.

To capture varying patterns of distribution of plastic strain, the finite element simulations were performed under various combinations of uniaxial and shear loading. The loading was enforced with displacement boundary conditions applied to all the sides of the mesh for both microstructures A and B. The combinations of displacement boundary conditions are sampled by rotating a loading displacement vector from 0° to 90° whose components are the uniaxial displacements in the two directions for the pure axial displacement cases and the uniaxial and shear displacements for the combined uniaxial and shear loading. The maximum displacement magnitude for axial and shear loading vector components are $u_{goal} = 1.5 \times 10^{-3}$ mm. We sample a total of 100 loading combinations/FEM simulations for each microstructure. During each of these simulations, we record the constitutive response at every material point and post-process it as a node-weighted graph and a volume average response. For every simulation, we record 100 time steps, thus collecting 10000 training sample pairs of graphs and homogenized responses for each microstructure.

4.2 Training of the graph autoencoder

In this section, we demonstrate the training performance of the autoencoder architecture on the two microstructure data sets described in Section 4.1. We also show the capacity of the autoencoder to reproduce the plasticity graphs in the training samples. The dimension of the encoded feature vector in this example is set to $D_{enc} = 16$. An examination of the effect of the encoded feature vector size is described in Section 4.3.

The training curves for the autoencoder's reconstruction loss function is shown in Fig. 9. The autoencoder appears to have similar loss function performance for both microstructures. The autoencoder performs slightly better for microstructure A as it is tasked to learn and reproduce patterns for a seemingly simpler microstructure

Sun 145



Figure 9: Autoencoder reconstruction training loss for microstructures A and B (a and b respectively).



Figure 10: Prediction of the autoencoder architecture of the microstructure A for two loading paths (a and b). The graph node size and color represent the magnitude of the accumulated plastic strain $\overline{\epsilon}_p$. The node-wise predictions for $\overline{\epsilon}_p$ are also demonstrated.

compared to microstructure B. The training loss curves in this figure demonstrate the overall performance of the autoencoder architecture – the encoder and decoder components of the architecture are trained simultaneously. The capacity of the autoencoder to reconstruct the plasticity distribution patterns is explored in Fig. 10 and Fig. 11 for microstructures A and B respectively. In these figures, we showcase the reconstruction capacity of the plastic strain for two different time steps for each microstructure. The time steps selected are from two different loading path combinations resulting in different plasticity graph patterns. We demonstrate how the autoencoder can reproduce these patterns by comparing the internal variable graph data – the autoencoder input – with the graph reconstruction – the autoencoder output. We also show the accuracy of the node-wise prediction of the accumulated plastic strain for these microstructures. It

is noted that the autoencoder predicts the values of the full plastic strain tensor at the nodes. However, these plots show the accumulated plastic strain $\bar{\epsilon}_p$ values calculated from the predicted strain tensor at the nodes for easier visualization.



Figure 11: Prediction of the autoencoder architecture of the microstructure B for two loading paths (a and b). The graph node size and color represent the magnitude of the accumulated plastic strain $\overline{\epsilon}_p$. The node-wise predictions for $\overline{\epsilon}_p$ are also demonstrated.

The autoencoder architecture provides the flexibility of utilizing its two components, the encoder \mathcal{L}_{enc} and the decoder \mathcal{L}_{dec} , separately. In this section, we demonstrate the encoder's ability to process the high-dimensional graph structure in encoded feature vector ζ time histories. In Fig. 12 (a) & (c) and Fig. 13 (a) & (c), we show the predicted encoded feature vector ζ_n for a time step of a loading path for microstructures A and B respectively. These encoded feature vectors specifically correspond to the graphs shown in Fig. 10 and Fig. 11 respectively. In Fig. 12 (b) & (d) and Fig. 13 (b) & (d), we demonstrate the time series of plasticity graphs encoded in time series of encoded feature vectors. It is highlighted that the encoded feature vector values do not change during the elastic/path-independent part of the loading. This is directly attributed to the fact that the plasticity graph is constant (zero plastic strain at the nodes) before yielding for all the time steps. The benefit of separately using the decoder \mathcal{L}_{dec} as a post-processing step to interpret the predicted encoded feature vectors is also explored in the following sections.

4.3 Mesh sensitivity and Encoded Feature Vector dimension

In this section, we investigate the behavior for different dimensionality of the graph data set and the compression of the graph information. In the first experiment, we test the effect of the size of the input plasticity graph that will be reconstructed by the autoencoder. We generate three data sets from finite element simulations with

Sun 147



Figure 12: Prediction of encoded feature vector ζ by the encoder \mathcal{L}_{enc} for microstructure A. (a,c) The encoded feature vector ζ_n for a single time step for the plastic graphs shown in Fig. 10a and Fig. 10b respectively. (b,d) The encoded feature vector ζ history for all the time steps in the loading paths of Fig. 10a and Fig. 10b respectively.

different mesh sizes for microstructure A. All meshes consist of the same triangular elements described in the previous section. The number of elements in the mesh and the corresponding nodes in the post-processed graphs are N = 100, N = 250, and N = 576. For the mesh generation, we start with the N = 100 mesh and refine once to obtain the N = 250 mesh and twice to obtain the N = 576 mesh. The refinement was performed automatically using the meshing software library Cubit (Blacker et al., 1994). The data sets for the meshes of N = 100, N = 250, and N = 576 nodes mesh were generated through the same FEM simulation setup and a subset of the combinations of uniaxial and shear loading paths described in Section 4.1 gathering 2500 training samples of graphs.

The results of the training experiment on the mesh sensitivity are demonstrated in Fig. 14. The figure shows the reconstruction loss for the three mesh sizes. The reconstruction loss exhibits a minor improvement as the number of nodes in the graph increase. This is attributed to the density of the information available for the autoencoder to learn the patterns from – there is a higher resolution of adjacent nodes' features. However, an increase in graph size may increase the duration of the training procedure. Since the benefit of increasing the mesh size is not significant in this set of numerical experiments, we opt for the N = 250 node mesh to use for the rest of this work.



Figure 13: Prediction of encoded feature vector ζ by the encoder \mathcal{L}_{enc} for microstructure B. (a,c) The encoded feature vector ζ_n for a single time step for the plastic graphs shown in Fig. 11a and Fig. 11) respectively. (b,d) The encoded feature vector ζ history for all the time steps in the loading paths of Fig. 11a and Fig. 11b respectively.

In a second numerical experiment, we examine the effect of the size of the encoded feature vector on the capacity of the autoencoder to learn and reconstruct the plasticity distribution patterns. We re-train the autoencoder using the data set of 10000 graphs generated on the N = 250 node mesh for microstructure A as described in Section 4.1. We perform three training experiments selecting different sizes D_{enc} for the encoded feature vector – $D_{enc} = 2$, $D_{enc} = 16$, and $D_{enc} = 32$. All the convolutional filters and the Dense layers have the same size. The only Dense layers that are affected are those around the encoded feature vector whose input and output sizes are modified to accommodate the different sizes of encoded feature vector.

The training performance for these three training experiments is demonstrated in Fig. 15. Compared to encoded feature vector sizes $D_{enc} = 16$ and $D_{enc} = 32$, the $D_{enc} = 2$ autoencoder architecture seems to fail to compress the information as well with a loss performance difference of about two orders of magnitude. The maximum compression achieved for this autoencoder architecture setup appears to be two features. This dimensionality appears to be the smallest feasible encoding limit for this particular data set. It is also possible that more sampling from different loading paths may also increase this minimal dimensionality. Jumping from $D_{enc} = 16$ to $D_{enc} = 32$ encoded feature vector components, only a small improvement in the reconstruction capacity is observed. The reconstruction capacity is also illustrated in Fig. 16. The

Sun 149



Figure 14: Autoencoder reconstruction training loss for microstructure A with the size of the encoded feature vector $D_{enc} = 16$ and graph sizes of N = 100, N = 250, and N = 576 nodes.



Figure 15: Autoencoder reconstruction training loss for microstructure A with the size of the encoded feature vector $D_{enc} = 2$, $D_{enc} = 16$, and $D_{enc} = 32$.

decoder fails to accurately reconstruct the plasticity graph from the $D_{\rm enc} = 2$ encoded feature vector (Fig. 16 a). However, for $D_{\rm enc} = 16$ and $D_{\rm enc} = 32$ the decoder accurately reproduces the plasticity patterns (Fig. 16 b and c). It is expected for dimensions larger than $D_{\rm enc} = 32$ the benefit in reconstruction capacity will me minimal. Thus, the encoded feature vector dimension selected for the rest of this work is $D_{\rm enc} = 16$ for which the dimension reduction capacity is considered adequate and computationally efficient.

It should be noted that the relatively small losses observed in Fig. 14 only indicate that the three reconstructions are independently successful in the sense that the reconstructed graphs are all sufficiently close to the original ones.

4.4 Return mapping algorithm

In this section, we provide the implementation details for the return mapping algorithm to make forward elasoplasticity predictions. The fully implicit stress integration algorithm allows for the incorporation of the graph-based internal variables generated from the autoencoder architecture in the prediction scheme. The return mapping algorithm designed in the principal strain space is described in Algorithm 1.



Figure 16: Comparison of the autoencoder architecture's capacity to reconstruct the accumulated plastic strain pattern of the microstructure A with the size of the encoded feature vector (a) $D_{enc} = 2$, (b) $D_{enc} = 16$, and (c) $D_{enc} = 32$. The graph node size and color represent the magnitude of the accumulated plastic strain $\bar{\epsilon}_p$. The node-wise predictions for $\bar{\epsilon}_p$ are also demonstrated.

This formulation of the return mapping algorithm requires all the strain and stress measures are in principal axes. However, this is not limiting for the choice of strain and stress space formulation of the constitutive law components as the framework allows for coordinate system transformation through automatic differentiation. The automatic differentiation is facilitated with the use of the Autograd library (Maclaurin et al., 2015). Through automatic differentiation and a series of chain rules the constitutive model predictions are expressed in the principal axes, allowing for any invariant formulation of the yield function during training.

The elastoplastic behavior is modeled through a predictor-corrector scheme that integrates the elastic prediction with the corrections by the yield function neural network. It is noted that the elastic update predictions for the hyperelastic energy functional and the plasticity terms encountered in the return mapping algorithm are evaluated as

neural network predictions using the offline trained energy functional, yield function, and kinetic law. The hyperelastic neural network is based on the prediction of an energy functional with interpretable first-order and second-order derivatives (stress and stiffness respectively).

Besides the capacity to predict the values of the approximated functions, these libraries also allow for the automatic evaluation of the approximated function derivatives that are required to perform the return mapping constitutive updates and constructing the local Newton-Raphson tangent matrix as well as the necessary coordinate set transformation chain rules.

Algorithm 1 Return mapping algorithm in strain-space for encoded feature vector internal variable plasticity.

- **Require:** Hyperelastic energy functional $\widehat{\psi}^{e}$ neural network, yield function \widehat{f} neural network, the encoded feature vector neural network $\hat{\zeta}$, and the plastic flow network \widehat{g} .
 - 1. Compute trial elastic strain

Compute $\epsilon_{n+1}^{\text{etr}} = \epsilon_n^{\text{e}} + \Delta \epsilon$. Spectrally decompose $\epsilon_{n+1}^{\text{etr}} = \sum_{A=1}^{3} \epsilon_A^{\text{etr}} \boldsymbol{n}^{\text{tr},A} \otimes \boldsymbol{n}^{\text{tr},A}$.

2. Compute trial elastic stress

Compute $\sigma_A^{\text{tr}} = \partial \hat{\psi}^{\text{e}} / \partial \epsilon_A^{\text{e}}$ for A = 1, 2, 3 and the corresponding $p^{\text{tr}}, q^{\text{tr}}$ at $\epsilon_{n+1}^{\text{e} \text{tr}}$.

3. Check yield condition and perform return mapping if loading is plastic

if $\widehat{f}(p^{\mathrm{tr}}, q^{\mathrm{tr}}, \xi_n) \leq 0$ then

Set $\boldsymbol{\sigma}_{n+1} = \sum_{A=1}^{3} \sigma_A^{\text{tr}} \boldsymbol{n}^{\text{tr},A} \otimes \boldsymbol{n}^{\text{tr},A}$ and exit.

else

Compute encoded feature vector $\boldsymbol{\zeta}_n = \widehat{\boldsymbol{\zeta}}(\boldsymbol{\epsilon}_{\text{histn}}^p)$. Compute plastic flow direction $\frac{\partial \widehat{\vartheta}}{\partial \sigma_A} = \widehat{\boldsymbol{g}}(\boldsymbol{\delta}\boldsymbol{\zeta}_n)$ for A = 1, 2, 3. Solve for $\boldsymbol{\epsilon}_1^{\text{e}}, \boldsymbol{\epsilon}_2^{\text{e}}, \boldsymbol{\epsilon}_3^{\text{e}}$, and $\boldsymbol{\xi}_{n+1}$ such that $\widehat{f}(p, q, \boldsymbol{\xi}_{n+1}) = 0$. Compute $\boldsymbol{\sigma}_{n+1} = \sum_{A=1}^3 \left(\partial \widehat{\psi}^{\text{e}} / \partial \boldsymbol{\epsilon}_A^{\text{e}}\right) \boldsymbol{n}^{\text{tr},A} \otimes \boldsymbol{n}^{\text{tr},A}$ and exit.

The return mapping also incorporates a non-associative flow rule to update the plastic flow direction instead of using the stress gradient of the yield function. This is achieved by incorporating the predictions of the plastic flow \hat{g} network. Through the local iteration scheme, the solution for the true elastic strain values can be retrieved using the solved for discrete plastic multiplier $\Delta \lambda$ and the predicted flow direction as follows:

$$\epsilon_A^{\rm e} = \epsilon_A^{\rm e\,tr} - \Delta \lambda \frac{\partial \vartheta}{\partial \sigma_A} = \epsilon_A^{\rm e\,tr} - \Delta \lambda \widehat{g}_A(\widehat{\delta \zeta}), \quad A = 1, 2, 3.$$
(19)

The return mapping algorithm requires a hyperelastic energy functional neural network $\widehat{\psi}^{e}$, a yield function \widehat{f} , a kinetic law $\widehat{\zeta}$, and a plastic flow \widehat{g} neural network that are pre-trained offline. Given the elastic strain tensor at the current loading step, a

trial elastic stress state is calculated using the hyperelastic energy functional neural network. The yield condition is checked for the trial elastic stress state and the current plastic strain level. If the predicted yield function is positive, the trial stress is the in the elastic region and is the actual stress. The encoded feature vector remains constant. If the yield function is non-positve the trial stress is in the inadmissible stress region and an Newton-Raphson optimization scheme is utilized to correct the stress prediction. The current encoded feature vector is predicted from the time history of plastic strain tensors and is used to predict the current plastic flow directions. The goal of the return mapping algorithm is to solve for the prinicipal elastic strains and the plastic strain such that the predicted yield function is equal to zero and the stress updates is consistent with the plastic flow. The encoded feature vector at every step can be converted back into the corresponding weighted graph via the graph decoder \mathcal{L}_{dec} neural network. This weighted graph can be converted back into information in a finite element mesh and therefore enable us to interpret the microstructure.

It is noted that the return mapping algorithm is formulated via the principle direction is provided in this section for the generalization purpose. This setting is sufficient for isotropic materials. In our numerical examples, we only introduce two-dimensional cases to illustrate the ideas for simplicity. The generalization of the return mapping algorithm for anisotropic materials is straightforward, but the training of the yield function and the plastic flow model in the higher dimensional parametric space is not trivial. This improvement will be considered in the future but is out of the scope of this study.

4.5 Interpretable multiscale plasticity of complex microstructures

In this last section, we demonstrate the capacity of the models to make forward predictions for unseen loading paths and interpret them in the microstructure. The return mapping algorithm does not only predict the strain-stress response of the material but also the plastic strain response and the encoded feature vector variables. These can then be decoded by the graph decoder \mathcal{L}_{dec} to interpret the microstructures' elastoplastic behavior. We provide tests of unseen loading path simulations for both microstructures A and B. The training of the constitutive models used to make the forward predictions is described in Section 4.2.

We first test the models' capacity to make predictions of the plastic state on monotonic data. We demonstrate the result for the predicted stress state in Fig. 17 (a & b) and Fig. 18 (a & b) for microstructures A and B respectively. We also record the homogenized plastic strain tensor of the microstructures. For simplicity, we are demonstrating the predicted accumulated plastic strain measure $\bar{\epsilon}_p$ in Fig. 17 (c) and Fig. 18 (c). Using the trained kinetic law neural network, we can make forward predictions of the encoded feature vectors $\hat{\zeta}$ that are consistent with the current predicted homogenized plastic state. The results of these predictions are shown in 17 (d) and Fig. 18 (d). These predicted curves are a close match to the benchmark data and can closely capture the behavior in the macroscale. We can now interpret this homogenized behavior as the corresponding one in the microscale. Using the trained decoder for each microstructures.

Sun 153



Figure 17: (a,b,c) Prediction of stress invariants p, q, and accumulated plastic strain $\bar{\epsilon}_p$ using the return mapping algorithm for a monotonic loading of microstructure A. (d,e) Prediction of all the encoded feature vector $\boldsymbol{\zeta}$ components and the corresponding decoded internal variable graph for a monotonic loading of microstructure A.



Figure 18: (a,b,c) Prediction of stress invariants p, q, and accumulated plastic strain $\bar{\epsilon}_p$ using the return mapping algorithm for a monotonic loading of microstructure B. (d,e) Prediction of all the encoded feature vector ζ components and the corresponding decoded internal variable graph for a monotonic loading of microstructure B.



Figure 19: Prediction of the plastic flow components for two loading cases for microstructures A and B (a & b respectively). Both the predicted via yield function stress gradient $\frac{\partial \hat{f}}{\partial \sigma_A}$ and the non-associative plastic flow $\frac{\partial \hat{\vartheta}}{\partial \sigma_A}$ predictions are shown.



Figure 20: Prediction of deviatoric stress q, accumulated plastic strain $\overline{\epsilon}_p$, and the encoded feature vector $\boldsymbol{\zeta}$ components using the return mapping algorithm for a cyclic loading of microstructure A.



Figure 21: Prediction of deviatoric stress q, accumulated plastic strain $\overline{\epsilon}_p$, and the encoded feature vector $\boldsymbol{\zeta}$ components using the return mapping algorithm for a cyclic loading of microstructure B.

ture, we can recover the plastic strain distributions as shown in Fig.17 (e) and Fig. 18 (e) for microstructures A and B respectively. It is noted that while only the node-wise prediction of the accumulated plastic strain $\overline{\epsilon}_p$ is shown, the decoded recovers the entire plastic strain tensor ϵ_p . This is done for simplicity of presentation. The node-wise

predictions of the plastic strain are accurate and the decoder can qualitatively capture the general plastic distribution patterns and the plastic strain localization nodes in the microstructure.

We also demonstrate the capacity of the model to predict the plastic flow with the help of the encoded feature vector internal variables. In Fig. 19, we compare the computed plastic flow components using the neural network yield function stress gradient $\frac{\partial \hat{f}}{\partial \sigma_A}$ prediction and the plastic potential stress gradient $\frac{\partial \hat{\vartheta}}{\partial \sigma_A}$ as predicted by the plastic flow \hat{g} network with the benchmark simulations. The results demonstrated are for two blind prediction curves for each microstructure – Fig. 19(a) corresponds to Fig. 17 and Fig. 19(c) corresponds to Fig. 18. The accuracy of the \hat{g} network predictions on the flow is higher than that of the yield function stress gradient. This is attributed to the decoupling of the yielding and hardening from the plastic flow directions allowing for more flexibility of the neural networks to fit these complex laws. Network \hat{g} also utilizes the highly descriptive encoded feature vector $\hat{\zeta}$ input that allows for more refined control of the plastic flow than the volume averaged accumulated plastic strain metric used in the yield function formulation.

Finally, we conduct a similar blind test experiment but with added blind unloading and reloading elastic paths in the loading strains. The results for microstructures A and B are demonstrated in Fig. 20 and Fig. 21 respectively. The model does not have any difficulty recognizing the elastic and plastic regions of the loading path since the behaviors are distinguished with the help of the neural network yield function. This also constrains the evolution of the plastic strain and the encoded feature vector to happen only during the plastic loading. Since the kinetic law neural network is a feedforward architecture, there is no history dependence and no change in the plastic strain corresponds to no change in the encoded feature vector. The decoder architecture is also path-independent so no change in the encoded feature vector corresponds to no change in the respective decoded plastic graph. This is also achieved by the specific way the encoded feature vectors are constructed. The input node features in the auto encoder are the mesh node coordinates and the plastic strains. This specific design ensures that the plastic graph does not evolve during elastic unloading/reloading and prevents any artificial memory effect in the elastic regime. The lack of memory effect in the elastic regime is necessary for the encoded feature vector to be internal variables for rate-independent plasticity models where the history-dependent effect is only triggered once the yield criterion is met. This would not be the case if other integration point data, such as the total strain or stress, are incorporated into the graph encoder.

Note that this switch between path-independent and path-dependent behaviors may also have implications for other neural network constitutive laws. In particular, if a black-box recurrent neural network is used to forecast history-dependent stress-strain responses, then one must ensure that the history-dependent effect is not manifested in the elastic region. For instance, if the LSTM architecture is used, then one must ensure that the forget gate is trained to turn on to filter out any potential artificial influence of the strain history.

5 Conclusion

This book chapter reviews the applications of the message-passing graph neural network for mechanics problems. We provide an example of how we can use graph embedding to create encoder that compress history-dependent spatial pattern into descriptors and state variables for plasticity models.

Acknowledgments.

This book chapter is written while the author is on sabbatical at Stanford University under the support of the UPS Foundation Visiting Professorship. This support is gratefully acknowledged.

References

- Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.
- Ted D Blacker, William J Bohnhoff, and Tony L Edwards. Cubit mesh generation environment. volume 1: Users manual. Technical report, Sandia National Labs., Albuquerque, NM (United States), 1994.
- Ronaldo I Borja. Plasticity, volume 2. Springer, 2013.
- Ines Chami. *Representation Learning and Algorithms in Hyperbolic Spaces*. Stanford University, 2021.
- Yannis F Dafalias and Majid T Manzari. Simple plasticity sand model accounting for fabric change effects. *Journal of Engineering mechanics*, 130(6):622–634, 2004.
- Itai Einav. Breakage mechanics—part i: theory. *Journal of the Mechanics and Physics* of Solids, 55(6):1274–1297, 2007.
- Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.
- Ronald L Graham, Donald E Knuth, Oren Patashnik, and Stanley Liu. Concrete mathematics: a foundation for computer science. *Computers in Physics*, 3(5):106–107, 1989.
- Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.
- Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
- Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML workshop*, volume 238, 2015.

- James Kenneth Mitchell, Kenichi Soga, et al. *Fundamentals of soil behavior*, volume 3. John Wiley & Sons New York, 2005.
- Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.
- Andrew Noel Schofield and Peter Wroth. Critical state soil mechanics, volume 310. McGraw-hill London, 1968.
- Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.
- Nikolaos N Vlassis and WaiChing Sun. Sobolev training of thermodynamic-informed neural networks for interpretable elasto-plasticity models with level set hardening. *Computer Methods in Applied Mechanics and Engineering*, 377:113695, 2021.
- Nikolaos N Vlassis and WaiChing Sun. Geometric learning for computational mechanics part ii: Graph embedding for interpretable multiscale plasticity. *Computer Methods in Applied Mechanics and Engineering*, 404:115768, 2023.
- Nikolaos N Vlassis, Ran Ma, and WaiChing Sun. Geometric deep learning for computational mechanics part i: anisotropic hyperelasticity. *Computer Methods in Applied Mechanics and Engineering*, 371:113299, 2020.
- Nikolaos N Vlassis, Puhan Zhao, Ran Ma, Tommy Sewell, and WaiChing Sun. Molecular dynamics inferred transfer learning models for finite-strain hyperelasticity of monoclinic crystals: Sobolev training and validations against physical constraints. *International Journal for Numerical Methods in Engineering*, 123(17):3922–3949, 2022.
- Kun Wang and WaiChing Sun. Meta-modeling game for deriving theory-consistent, microstructure-based traction–separation laws via deep reinforcement learning. *Computer Methods in Applied Mechanics and Engineering*, 346:216–241, 2019.
- Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *NTI, Series*, 2(9):12–16, 1968.
- Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.
- Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.
- Huaxiang Zhu, Hien NG Nguyen, François Nicot, and Félix Darve. On a common critical state in localized and diffuse failure modes. *Journal of the Mechanics and Physics of Solids*, 95:112–131, 2016.

Artificial Neural Networks: layer architectures, optimizers and automatic differentiation

Filippo Gatti

Université Paris-Saclay CentraleSupélec, ENS Paris-Saclay, CNRS LMPS Laboratoire de Mécanique Paris Saclay UMR 9026

This chapter is meant to provide a basic yet solid understanding of artificial neural networks, to a heterogeneous readership. In particular, the chapter presents three major types of neural networks, namely: feed-forward multi layer perceptrons (\mathcal{MLP}), convolutional neural networks (\mathcal{CNN}) and recurrent neural networks (\mathcal{RNN}) . The chapter describes how deep is the relationship between each of these neural networks and the specific data science task they aim at tackling: from regression to classification, from images to time-series, with practical tutorials on real datasets and mechanics-inspired examples. The neural network optimization is described in its general statistical framework, focusing on the most popular algorithms tailored to efficiently "train" such neural metamodels, through the so called "back-propagation". The chapter explains how in practice the back-propagation occurs, by automatizing the derivative chain rule and by efficiently exploiting the computational graph constructed to predict. The chapter's last sections provide practical recipes on how to efficiently optimize the learning algorithms, by duly initializing the neural network parameters, by adopting design strategies that avoid vanishing or exploding gradients and by pursuing deeper architectures to achieve data disentanglement, parsimony and generalizability. The chapter is featured by several practical examples, with corresponding code snippets, in order to practice the theoretical aspects presented in the main text. For expert readers, this chapter serves as a recap. We defer to the chapter Artificial Neural Networks: advanced topics, for further more technical and theoretical insights on the vast world of artificial neural networks. The chapter is largely inspired, among others, by Stéphane Mallat's Data Science lecture notes at Collège de France, as well as by different lecture notes of CentraleSupélec's engineering curriculum.

1 Why PyTorch?

The scope of these chapters is to provide a basic understanding of the most important notions of artificial neural networks, for a non-expert readership with a strong background in computational geomechanics. Major theoretical aspects are outlined and supported by detailed examples and tutorials, in order to reinforce the learning curve of basic and advanced concepts with the help of practical and commented hands-on sessions, accessible to everybody. This approach follows the same paradigm followed by the scientific community since a decade ago, when the AI revolution started, thanks to the deployment of cumbersome algorithms on large databases on specific hardware, such as GPUs (Graphic Processing Units) and, from 2016, TPUs (Tensor Processing Units). GPU, followed by TPU, took advantage of powerful CPU (Central Processing Units) because of their proneness to solve the same sequence of operations on a continuous stream of data, progressively offloaded from the host (the CPU, accessing large memory but at a slow pace) to the device (GPU or TPU respectively), with limited memory capacity, but leveraging the possibility of performing massive parallel computations on naively parallel problems (such as processing large databases of independent data realizations). AI and machine learning took advantage of the long tradition in computer vision of leveraging GPU computation to render graphical content. However, in order to exploit such techniques for statistical learning, several libraries have been developed, with alternate fortunes. A decade after the start of the AI revolution in 2012 - the year in which Krizhevsky et al. [KSH17] beat the state-of-art reference in image classification - two major libraries have survived and have become extremely popular for the design of neural networks and complex machine learning techniques: PyTorch and TensorFlow. The development of those two libraries has been featured by several extra contributions of prototype libraries (DistBelief, Caffe2, Theano, Keras to cite a few) that have been progressively integrated in either one of two leading libraries, developed by Meta AI and Google Brain Team respectively.

PyTorch was firstly released in 2016, originally developed by Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, as a python interface to the Torch open-source library, initially created by Idiap Research Institute and École polytechnique fédérale de Lausanne (EPFL) and migrated to PyTorch in 2017. Torch provides a LuaJIT interface to deep learning algorithms written in C. LuaJIT leverages the Lua lightweight, high-level, multi-paradigm programming language in the context of tracing just-in-time compilation, to optimize the execution of a program at runtime by tracking frequent linear sequences of code, compiling them to native machine code and executing them. On the contrary, TensorFlow was released in late 2015 by the Google Brain Team, based on its proprietary ancestor called DistBelief, started in 2011. The name TensorFlow was inspired by the archetype on which the library is based, i.e. the multidimensional data array referred to as *tensors*. In 2016, Google even developed an application-specific integrated circuit (ASIC), the TPU, specifi-

cally tailored for TensorFlow applications and oriented toward using or running models on low floating-point precision (8-bit) rather than training them. Both PyTorch and TensorFlow are at their second versions, released in 2023 and 2019 respectively.

From a mere technical standpoint, the main differences between the two libraries is the way they assemble computational graphs. Whilst TensorFlow distinguishes itself for using static graphs (compiled before running them) to perform back-propagation, PyTorch adopts dynamic graph that are assembled at run time. However, in its brand new version 2, recently released, PyTorch introduced an experimental tool to compile graphs. The difference between static and dynamic graphs is that the static graphs are based on fixed tensor shapes and graph connections, that allows to optimize the graph operations before running it. However, certain optimization features cannot be applied to dynamic graphs assembled and modified at runtime, possibly reducing their overall speedup. Nevertheless, the PyTorch's dynamic approach eases the coding part for non-expert users, since memory requirements (often unknown) need not to be defined before hand. Static graphs, on the contrary, allow secure model building with high-level API that enables portability on cloud, in the browser, on-device, multiple CPUs and multiple GPUs or TPUs. Static graphs can be used by other supported languages, such as C++ and Java, facilitating cross-platform deployment. On the contrary, PyTorch has its own API for saving trained model in a serialized way and its own API for distributed training on parallel architectures (DistributedDataParallel or ModelParallel.

As described in THE INTRODUCTORY CHAPTER, TensorFlow dominated the AI scene at the very beginning (2012-2017) and rapidly left the step to PyTorch after release, due to its friendly interface and ease to code, that attired new coming AI practitioners with poor background in machine learning and python coding in general. Therefore, the Google team released TensorFlow v2.0 in 2019, completely turning v1.0 upside down. The most intriguing novelty was the introduction of the "eager" mode, which introduced the possibility of switching from default static graph computation to the "Define-by-Run". Moreover, TensorFlow 2.0 performances on GPU drastically improved, enabling its use to those popular (compared to TPU at least) hardware. This allowed TensorFlow to regain a consistent portion of the market.

All in all, both TensorFlow and PyTorch are incredibly user-friendly and powerful machine learning libraries. TensorFlow is definitely more mature, with advanced features for optimizing complex machine learning algorithms. However, PyTorch seems more adapt to the ALERT school's purpose of providing a first lookup at the basic AI and neural networks fundamentals.

2 Neural networks

Neural Networks (\mathcal{NN}) have shown outstanding capabilities either for classification, either for regression problems. Given a labeled dataset $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, with $\boldsymbol{x}_k \in \mathcal{X} \subset \mathbb{R}^{d_X}$ and $\boldsymbol{y}_k \in \mathcal{Y} \subset \mathbb{R}^{d_Y}$, the problem they aim at solving is the following:

Problem. Empirical loss minimization(\mathcal{P}) Find $\mathbf{h}_{\theta} : \mathcal{X} \to \mathcal{Y}, \ \mathbf{h}_{\theta} \in \mathcal{H}_{\theta}$ such that:

$$\boldsymbol{h}_{\hat{\boldsymbol{\theta}}}\left(\mathcal{D}_{XY}\right) = \operatorname*{arg\,min}_{\boldsymbol{h}_{\boldsymbol{\theta}}\in\mathcal{H}_{\boldsymbol{\theta}}} L_{\mathcal{D}_{XY}}(\boldsymbol{h}_{\boldsymbol{\theta}})$$

where \mathcal{H}_{θ} is the chosen class of classifiers or regression functions (i.e., the neural network architecture), $L_{\mathcal{D}_{XY}}$ the empirical loss (corresponding to the sample average of i.i.d. realizations) defined as:

$$L_{\mathcal{D}_{XY}}\left(\boldsymbol{h}_{\boldsymbol{\theta}}\right) = \frac{1}{N} \sum_{i=1}^{N} l\left(\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{k}\right), \boldsymbol{y}_{k}\right), \quad \left(\boldsymbol{x}_{i}, \boldsymbol{y}_{i}\right) \in \mathcal{D}_{XY}$$
(1)

with $l(\boldsymbol{h}(\boldsymbol{x}), \boldsymbol{y}) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$ being a measure of the "distance" between "true" label and prediction, such that $l(\boldsymbol{y}, \boldsymbol{y}) = 0$. $L_{\mathcal{D}_{XY}}(\boldsymbol{h}_{\theta})$ is an approximation of the "true" loss $L(\boldsymbol{h}_{\theta}) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p}[l(\boldsymbol{h}_{\theta}(\boldsymbol{X}), \boldsymbol{Y})]$ that cannot be computed directly since $p(\boldsymbol{X}, \boldsymbol{Y})$ is not known a priori. The theory of information exposed in Section 1.6 draws the analogy between empirical loss $L_{\mathcal{D}_{XY}}$ and the negative log-likelihood of the conditional probability distribution $p_{\theta}(\boldsymbol{y}|\boldsymbol{x})$, induced by parametrization of the $\mathcal{MLP} h_{\theta}$ by identifying l with the negative log-likelihood of the conditional probability $p_{\theta}(\boldsymbol{y}|\boldsymbol{x})$, defined as follows:

$$l(y, h_{\theta}(\boldsymbol{x})) = -\ln p_{\theta}(y|\boldsymbol{x}) = \mathcal{NLL}(y, h_{\theta})$$
⁽²⁾

In other the negative log-likelihood of . Minimizing $L_{\mathcal{D}_{XY}}$ corresponds therefore to minimize the

A \mathcal{NN} is a particular statistical model $h_{\theta} \in \mathcal{M}_{\theta}$ (see Section 1.3) that depends on a set of weights and biases $\theta \in \Theta$. The most common \mathcal{NN} model is the Multi-Layer Perceptron (\mathcal{MLP}), that consists into a directed acyclic graph (*feed-forward*), with an input layer, several hidden layers and an output layer of *neurons* (the graph's edges) stacked upon each other (see Figure 1).

Every node in one layer is connected to every other node in the next layer. The \mathcal{MLP} gets deeper as many more hidden layers are stacked-up. The fundamental elementary brick of any \mathcal{MLP} is the *neuron*.

Gatti 163



Figure 1: Simple example of Multi-Layer Perceptron (3 hidden layers $h^{(\ell)}$ and 1 output y). The figure was drawn with tikz.net.

2.1 The artificial neuron

Artificial neural networks have been formulated by the pioneering work of Pitts and McCulloch [MP43] and Rosenblatt [ROS57]. A *neuron* (or unit) essentially performs the following non-linear transformation on its input x:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = g(a(\boldsymbol{x})) = g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b) = g\left(\sum_{c=1}^{d_{X}} w_{c} \cdot x_{c} + b\right); \qquad \boldsymbol{\theta} := \{\boldsymbol{w}; b\} \quad (3)$$

where a is the linear *pre-activation*, w the weights and b the bias. g is the nonlinear *activation function* and it can be chosen in the list (non-exhaustive):

• Sigmoid:

$$g(a) = \sigma(a) = \frac{1}{1 + e^{-a}}$$
 (4)

Properties of the Sigmoid function:

- it is bounded in [0; 1]
- it is positive
- it is strictly increasing
- it serves as probability distribution, i.e., $g(a) = P_Y[Y = a | \boldsymbol{x}]$
- it has a "squashing" effect
- it is a smooth version of binary classifier with linear decision boundary (Logistic regression)



Figure 2: Sigmoid activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.

• Hyperbolic Tangent:

$$g(a) = \tanh(a) = \frac{e^{2a} - 1}{e^{2a} + 1}$$
(5)

Properties of the Hyperbolic Tangent:

- it is bounded in [-1, 1]
- it is skew-symmetric
- it is strictly increasing
- it has a "squashing" effect (vanishing gradients, see Section 4.2)
- it is preferred to Sigmoid because it has 0 steady state (tanh (0) = 0) [LeC+98; GB10].
- Rectified Linear Unit [GBB11]:

$$g(a) = ReLU(a) = \max(0, a) \tag{6}$$

Gatti 165



Figure 3: Hyperbolic tangent activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.

Properties of the Rectified Linear Unit:

- closer to real biological activity since indicates that cortical neurons are rarely in their maximum saturation regime [GBB11]
- it is bounded to 0 by below for a < 0
- it is positive $\forall a \in \mathbb{R}$
- it is strictly increasing $\forall a \in \mathbb{R}$
- it does not enforce a sign symmetry or antisymmetry
- it promotes sparsity in combination with a ℓ^1 -norm penalty on the weights (see Section 2.3 and [GBB11])
- the deactivation for a < 0 is robust to noise
- Leaky Rectified Linear Unit:

$$g(a) = LeakyReLU(a) = \begin{cases} a, a \ge 0 & (7) \\ -\alpha \cdot a, a < 0(\alpha > 0) & (8) \end{cases}$$

Properties of Leaky Rectified Linear Unit:

- it is not bounded
- it maintains the sign of a
- it has no "squashing" effect
- Exponential Linear Unit [CUH16]

$$g(a) = ELU(a) = \begin{cases} a, a > 0 & (9) \\ \alpha \cdot (e^a - 1), a \le 0 & (10) \end{cases}$$



Figure 4: ReLU activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.



Figure 5: LeakyReLU activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.

Properties of the Exponential Linear Unit:

- it equals the identity for a > 0
- it is not bounded for a > 0
- it is strictly increasing $\forall a \in \mathbb{R}$
- it has a "squashing" effect for $a \leq 0$
- noise-robust deactivation state $(\pmb{\mu}_w \rightarrow \pmb{0})$ [CUH16] (see Section 4.2)
- Scaled Exponential Linear Unit [Kla+17]

$$g(a) = SELU(a) = s\left(\max(0, a) + \min(0, \alpha (e^a - 1))\right)$$
(11)

Gatti 167



Figure 6: *ELU* activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.

Properties of the Scaled Exponential Linear Unit:

- $-\alpha = 1.6732632423543772848170429916717$
- $-\ s = 1.0507009873554804934193349852946$
- identity for a > 0
- bounded to $-\alpha$ by below for a < 0
- self-normalizing weights $(\mu_w \to 0, \mathbb{C}_{w,w} \to \mathbf{I})$ even for noisy input [Kla+17] (see Section 4.2)



Figure 7: *SELU* activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.

The functions $g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle)$ are *ridge functions*, which is constant on the hyperplane $a(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle = c$ and on each hyperplane $a(\boldsymbol{x}) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = c$ (see

Figure 8).



Figure 8: Examples of linear separation made by a single neuron on an input dataset $\mathcal{X} \subset \mathbb{R}^2$. (a) activation function $g(a) = \sigma(a)$; (b) activation function $g(a) = \tanh(a)$.

From Figure 8, one notices that b makes the ridge h_{θ} translate along the input space, whereas \boldsymbol{w} determines the orientation of the ridge (in-plane rotation). The activation function adds non-linearity and if $g(\cdot) = \sigma(\cdot)$, h_{θ} can be interpreted as the likelihood $p_{\theta}(\boldsymbol{x})$. In a binary classification context with $y \in \mathcal{Y} := \{0, 1\}$ (if $g(\cdot) = \sigma(\cdot)$) or $y \in \mathcal{Y} := \{-1, 1\}$ (if $g(\cdot) = \tanh(\cdot)$), the combination of b, \boldsymbol{w} and g determines decision boundary, i.e., the hyperplane that separates the samples of class y = 0 (or y = -1) if $h_{\theta}(\boldsymbol{x}) > 0.5$ from those labeled as y = 1, if $h_{\theta}(\boldsymbol{x}) \leq 0.5$. In a regression context, the combination of b, \boldsymbol{w} and g represents a non-linear ridge regression.

Example 1. Create a single neuron network in PyTorch

```
import torch
      import torch.nn as nn # library for predefined NN classes
2
      # assure reproductibility
3
      torch.manual_seed(0)
4
\mathbf{5}
      class neuron(nn.Module):
6
        def __init__(self, x_size=28*28, h_channels=10):
7
             super(neuron,self).__init__()
            # hidden nodes in each layer
self.x_size = x_size
8
9
            self.h_channels = h_channels
10
             # This is the 1st hidden layer (784 -> hidden_1)
11
            self.a = nn.Linear(self.x_size, self.h_channels)
self.g = nn.ReLU(inplace=False)
12
13
14
             # Inplace in the code explains how the function
             # should treat the input. Inplace as true replaces
15
             # the input to output in the memory. Though this helps
16
               in memory usage, this creates problems for the code
17
             # being used as the input is always getting replaced as
18
19
             # output. It is better to set in place to false as this
20
             # helps to store input and output as separate storage
```

Gatti 169

```
# spaces in the memory.
# The dropout layer (p=0.2)
self.droput = nn.Dropout(0.2)
21
^{22}
^{23}
^{24}
           def forward(self,x):
                   # flattening the input image
x = x.view(-1,self.x_size)
25
26
                   # adding the hidden layer, for ac
ho = self.droput(self.g1(self.a1(x)))
                                                                 for activation we are using relu
                                                                                                                       activation
27
28
29
                   return ho
30
        x_size
                      28*28
^{31}
                   = 10
        y_size
        h_theta = neuron(x_size, y_size)
32
```

2.2 The Multi-Layer Perceptron

A single neuron is not sufficient when the decision boundary is not linear. For instance, in a classification problem with input $\boldsymbol{x} = (x_1, x_2) \in \{0, 1\}^2$ and associated true label $\boldsymbol{y} = f(\boldsymbol{x})$ a neuron such as the one described in Figure 8 can successfully draw a linear decision boundary and separate the dataset into two clusters if $f = f_{OR}(\boldsymbol{x}) = \{x_1\} \cup \{x_2\}, f = f_{AND1}(\boldsymbol{x}) =$ $\{x_1\} \cap \{\{0, 1\} / \{x_2\}\}$ and $f = f_{AND2}(\boldsymbol{x}) = \{\{0, 1\} / (x_1)\} \cap \{x_2\}$ (see Figures 9b to 9d). However, one neuron cannot linearly separate the data labeled with the function $f = f_{XOR}(\boldsymbol{x}) = \{\{x_1\} \cup \{x_2\}\} / \{\{x_1\} \cap \{x_2\}\}$ as shown in Figure 9d.

Instead of one single neuron, the classifier $h_{\theta}(x)$ can be improved by stacking two layers of neurons one upon each other: a *hidden layer* made of two neurons and an *output layer*, made of one neuron only (the architecture is depicted in Figure 10a). As shown in Figure 10b, $h_{\theta}(x)$ can now successfully separate the boundary induced by f_{XOR} .

But how is that possible? Adding hidden layers reveals to be crucial to raise the complexity of the predictor, thanks to intermediate transformations ϕ : $\boldsymbol{x} \mapsto \mathbb{R}^{d_{\phi}}$ that map the input \boldsymbol{x} into the space of *hidden features* of size d_{ϕ} :

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = g(a(\boldsymbol{\phi}(\boldsymbol{x}))) = g(\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}) \rangle + b) = g\left(\sum_{c=1}^{d_{\phi}} w_c \cdot \phi_c(\boldsymbol{x}) + b\right); \qquad \boldsymbol{\theta} := \{\boldsymbol{w}; b\}$$
(12)

In the case of XOR classification, the architecture in Figure 10a is conceived in order to map \boldsymbol{x} into the intermediate hidden representations, one per hidden neuron, $h_1(\boldsymbol{x}) = f_{AND1}(\boldsymbol{x})$ and $h_2(\boldsymbol{x}) = f_{AND2}(\boldsymbol{x})$. Figure 10b show that the XOR decision boundary is linear in the (h_1, h_2) plane.

It is clear that the choice of ϕ adds flexibility to the predictor. The size and properties of this intermediate representations can be chosen via regression kernels $\Bbbk : \mathcal{X} \times \mathcal{X} \mapsto \mathcal{X}$ such that $\int_{\mathbb{R}^{d_X}} \Bbbk(\boldsymbol{x}_1, \boldsymbol{x}_2) \phi(\boldsymbol{x}_2) d\boldsymbol{x}_2 = \lambda \mathbb{M}^{-1} \phi(\boldsymbol{x}_1)$ with a metric \mathbb{M} such that $\int_{\mathbb{R}^{d_X}} \langle \phi(\boldsymbol{x}), \mathbb{M}^{-1} \phi(\boldsymbol{x}) \rangle d\boldsymbol{x} = 1$ (see Chapter 2 - Introduction to regression methods).

Alternatively, one can stack several layers of neurons one upon each other, so



Figure 9: Solved and unsolved decision boundaries for different binary operations. (a) $f_{OR}(\boldsymbol{x}) = \{x_1\} \cup \{x_2\}$, (b) $f_{AND1}(\boldsymbol{x}) = \{x_1\} \cap \{\{0,1\} / \{x_2\}\}$, (c) $f_{AND2}(\boldsymbol{x}) = \{\{0,1\} / (x_1)\} \cap \{x_2\}$, (d) $f_{XOR}(\boldsymbol{x}) = \{\{x_1\} \cup \{x_2\}\} / \{\{x_1\} \cap \{x_2\}\}$ (unsolved). Reprinted from the video-lecture notes by Hugo Larochelle (https://www.youtube.com/watch?v=iT5P4z6Fzj8&list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=3).



Figure 10: Simple example of Multi-Layer Perceptron (1 hidden layers $h^{(1)}$ and 1 output $y \in \mathbb{R}$). layer. The figure was drawn with tikz.net.

that the final predictor results in a composition of hidden layers $\boldsymbol{h}^{(\ell)}$ and one output function $\boldsymbol{h}^{(o)}$. For instance, in case of one hidden neuron, the $\mathcal{N}\mathcal{N}$

Gatti 171

output reads:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = h^{(o)} \circ \boldsymbol{h}^{(1)}(\boldsymbol{x}) = g^{(o)}(a^{(o)}(\boldsymbol{h}^{(1)}(\boldsymbol{x}))) = g\left(\left\langle \boldsymbol{w}^{(o)}, \boldsymbol{h}^{(1)}(\boldsymbol{x}) \right\rangle + b^{(o)}\right) = g^{(o)}\left(\sum_{c=1}^{d_{h_1}} w_c^{(o)} \cdot h_c^{(1)}(\boldsymbol{x}) + b^{(o)}\right); \quad \boldsymbol{\theta} := \left\{\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(o)}; b^{(1)}, b^{(o)}\right\}$$
(13)

with

$$h_{c}^{(1)} = g^{(1)}(a_{c}^{(1)}(\boldsymbol{x})) = g^{(1)}\left(\left\langle \boldsymbol{w}^{(1)}, \boldsymbol{x} \right\rangle + b^{(1)}\right)$$
$$= g^{(1)}\left(\sum_{c=1}^{d_{X}} w_{c}^{(1)} \cdot x_{c} + b^{(1)}\right)$$
(14)

More in general, \mathcal{MLP} gathers several layers of neurons, with each neuron c in a layer ℓ being fully-connected (a connection means a composition) with all the neurons in the layer $\ell - 1$ - as its input - and with all the neurons in layer $\ell + 1$, as its output. Figure 11 shows an example of fully-connected \mathcal{MLP} , focusing on the connections between a hidden layer ℓ and the successive one layer $\ell + 1$. For each connection c, a weight vector \boldsymbol{w}_c and a bias b_c are adopted. The weight vectors, stacked column-wise, constitute the weight matrix \boldsymbol{W} , with $\boldsymbol{W}_{c::} = \boldsymbol{w}_c$.

Some crucial aspects of a fully-connected (or dense) feed-forward $\mathcal{N}\mathcal{N}$ such as the standard \mathcal{MLP} must be noted:

- the activation function $g^{(\ell)}$ is the same for all neurons belonging to the same layer ℓ ;
- all neurons (or units) belonging to the ℓ^{th} layer receive the same input, i.e., a the vector $\boldsymbol{h}^{(\ell-1)}$ of size $u^{(\ell)1}$, representing the input units to the ℓ^{th} layer of the graph;
- the output of each of the $u^{(\ell+1)}$ neurons the ℓ^{th} layer neuron (or unit) belonging to the ℓ^{th} layer $h_i^{(k)}$ of size ², representing the input units of the graph.

Example 2. Create a single neuron network in PyTorch

```
    # Chapter 7 - Example 2
    import torch
    import torch.nn as nn # library for predefined NN classes
```

 $u^{(\ell)}$ is commonly indicated in the literature as fan_{in} $u^{(k)}$ is commonly indicated in the literature as fan_{in}



Figure 11: Scheme of the last output layer of a fully-connected Multi Layer Perceptron. The figure was drawn with tikz.net.

```
# assure reproductibility
 4
       torch.manual seed(0)
 \mathbf{5}
 6
       class MLP(nn.Module):
 7
            def __init__(self, x_size=28*28, y_size=10):
 8
                 super(MLP,self).__init__()
 9
                 # hidden nodes in each layer
10
                 self.x_size = x_size
                 self.y_size = y_size
11
                 self.hn_1 = 512
12
13
                 self.hn_2 = 512
14
                 # This is the 1st hidden layer (784 -> hidden_1)
15
                 self.a1 = nn.Linear(self.x_size, self.hn_1)
self.g1 = nn.ReLU(inplace=False)
16
17
                  # Inplace in the code explains how the function
18
                  # should treat the input. Inplace as true replaces
19
^{20}
                 # the input to output in the memory. Though this helps
                 # in memory usage, this creates problems for the code
# being used as the input is always getting replaced as
^{21}
22
                 # output. It is better to set in place to false as this
# helps to store input and output as separate storage
23
^{24}
^{25}
                 # spaces in the memory.
^{26}
27
                 ## This is also linear layer but (n_hidden -> hidden_2)
                 self.a2 = nn.Linear(self.hn_1, self.hn_2)
self.g2 = nn.ReLU(inplace=False)
^{28}
29
30
                  # This is the linear output layer with (n_hidden \rightarrow 10)
31
                 self.ao = nn.Linear(self.hn_2, self.y_size)
^{32}
33
                 # The dropout layer (p=0.2)
^{34}
```

Gatti 173

```
self.droput = nn.Dropout(0.2)
35
36
          def forward(self,x):
37
             # flattening the input image
38
             x = x.view(-1,self.x_size)
39
40
             # adding the hidden layer, for activation we are using relu activation
41
            h1 = self.droput(self.g1(self.a1(x)))
42
43
             # adding the hidden layer, for activation we are using relu activation
44
            h2 = self.droput(self.g1(self.a1(h1)))
^{45}
46
47
             # adding the output layer
            ho = self.a3(h2)
48
49
            return ho
50
51
      x_size = 28*28
52
      y_size = 10
53
      h_theta = MLP(x_size, y_size)
54
```

2.3 Possible solutions to the curse of dimensionality

According to [Mai99] (see Theorem 27), the number of hidden neurons N_K demanded to assure that a \mathcal{MLP} with a sole hidden layer approximates a function $f: \mathbf{x} \mapsto y$, provided that at least $f \in W^{k,23}$ and with an accuracy ε is of the order of:

$$N_K \approx \varepsilon^{\frac{1-a_X}{k}} \tag{15}$$

with $d_X = \dim(\mathcal{X})$, and k being the highest order of weak derivative of f. The curse of dimensionality of the Universal Approximation Theorem (expressed in Theorem 24) resides in the fact that the larger is the dimension of the data space \mathcal{X} and the poorer is the regularity of the function (i.e., the lower is the highest order of weak derivative k), the larger would be the number of hidden neurons of a 1-hidden-layer \mathcal{MLP} demanded to approximate the labeling or regression function f. [Bac17] proposed an interesting strategy to break the curse of dimensionality presented in Section 2.1. The strategy consists in several countermeasures that are detailed in the following paragraphs

Reduce the dimensionality In other words, one should select a subset $S \subset \mathcal{X}$ with dimension $d_S < d_X$. In this sense, one possible strategy is the Principal Component Analysis (PCA) or other Reduced Order Methods (ROM). PCA and other linear methods for dimensionality reduction are adopt an linear transform of the type $\boldsymbol{a}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}$, with $\boldsymbol{W} \in \mathcal{M}_{(d_S, d_X)}(\mathbb{R})$, with rank $(\boldsymbol{W}) = d_S < d_X$ (see related chapters for further detail).

Separate the interactions (disentanglement) The separation of interaction is a widely adopted assumption in physics. It postulates that the function

 $^{{}^{3}}W^{k,2}$ is the Sobolev space defined in Equation (234)

 $f: \mathcal{X} \to \mathbb{R}$ is a the sum of functions f_i acting on local subdomains. For instance, the Markov processes assume the separability of the interaction, since they postulate that the transition probability from a state to another strictly depend on those two states only. Future and long-time past interactions are neglected. The separability of interaction can be formulated as follows:

$$\exists \{f_n\}_{n \in I}, f_n : \mathcal{X}_n \to \mathbb{R}, \mathcal{X}_n \subset \mathcal{X} \quad f(\boldsymbol{x}) = \sum_{n \in I} f_n(\boldsymbol{x}) |_{\boldsymbol{x} \in \mathcal{X}_n}, \forall \boldsymbol{x} \in \mathcal{X} \quad (16)$$

The separability of interactions is also called *disentanglement*. Disentangling the data representations consists into mapping the data set into a latent representation (see Section 1.6) spanned by a basis that depends on the so called factor of variation of the data set at stake [KM18]. For instance, a dataset composed of force-displacements values issued from a traction test on a steel sample of Young modulus E, under different temperature values T (within the limits of thermoelasticity), could ideally be represented by a latent representation $\boldsymbol{z} = z_E \boldsymbol{e}_E + z_T \boldsymbol{e}_T$, where z_E only varies whenever the Young's modulus of the sample changes, at fix temperature, whereas z_T changes whenever the same sample undergoes the traction test at different temperature. More in general, assuming that the database can be generated by a set of semantic meaningful *features*, like Young's modulus and temperature in the previous examples, the disentanglement process aims at separating the effect of each factor - in the latent representation - in order to structure the latent space in an interpretable way [Fra+22]. In machine learning, datasets consist of multiple i.i.d. samples, assumption that allows to assume a factorized parametric probability distribu-tion $p_{\theta}(x_1, \ldots, x_N) = \prod_{i=1}^N$ and therefore to maximize the log-likelihood of the observations (see Section 1.3). The idea of disentanglement is to achieve a latent representation of the dataset (to "encode" the data), associated to a parametric probability distribution $q_{\phi}(\boldsymbol{z}|\boldsymbol{x})$, that extract the common features of the dataset and represents them into a disentangled way, e.g., by forcing a factorized probability distribution on $\boldsymbol{z} \sim \prod_{i=1}^{N} q_{\boldsymbol{\phi}}(z_i | \boldsymbol{x})$. However, this strategy is not sufficient to achieve the disentanglement of the latent representation, since all z_i could vary when changing one single factor of variation in the data set [Fra+22].

Adopting the reduction of dimensionality, one can intuitively look for subdomains \mathcal{X}_n of dimension $d_S < d_X$. In this case, the approximation can be done by subdomains, employing the same 1-hidden-layer \mathcal{MLP} with a total amount of neurons $N_K \approx \varepsilon_n^{\frac{1-d_S}{k}}$ (see Equation (15)). The total error would be estimated as

$$\varepsilon = \sum_{n \in I} \varepsilon_n \approx \operatorname{card}(I) \cdot N_K^{\frac{k}{1-d_S}} \qquad N_K \approx \left(\frac{\varepsilon}{\operatorname{card}(I)}\right)^{\frac{1-d_S}{k}} \tag{17}$$

The most intuitive way of separate the interaction is to decompose the function
along each component of the data space, i.e.:

$$\exists \{f_n\}_{n=1}^{d_X}, f_n : \mathbb{R} \to \mathbb{R} \quad f(\boldsymbol{x}) = \sum_{n=1}^{d_X} f_n(w_n \cdot x_n), \forall \boldsymbol{x} \in \mathcal{X}$$

With this strategy, $d_S = \operatorname{rank}(w_n) = 1$.

Promote sparsity

[...] pour ranger les êtres sous des dénominations communes, et générique, il en falloit connoître les proprietés et les différences; il falloit des observations, et des définitions [...] J.J. Rousseau, 1772 [Rou72]

Rousseau, in his "Discours sur l'origine et les fondemens de l'inégalité parmi les hommes", published in 1772, suggested the need of a dictionary to describe the reality. The hidden features that describe the function regularity can be effectively embedded into a dictionary $D = \{(k, v(\boldsymbol{x}))_n\}_{n \leq N_D}$ (k is the key and v the corresponding value), with:

$$h_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right) = \sum_{n \in I} \theta_{n} \left[k_{n}\right] v_{n}\left(\boldsymbol{x}\right), \quad \text{card}\left(\mathbf{I}\right) \le N_{D}$$
(18)

Rousseau intuited that, the size of the dictionary N_D decreases with the knowledge of the phenomenon:

[...] plus les connaissances étoient bornées, et plus le Dictionnaire devint étendu.

J.J. Rousseau, 1772 [Rou72]

As a matter of fact, the dictionary embedding requires the error to decay rapidly with the cardinality of $I \subseteq D$, keeping this cardinality small. In other words, we seek for an error decaying as and $\varepsilon = \operatorname{card}(I)^{-\beta}$. This approach is not related to the notion of regularity of the function as intedend in the Fourier approach, i.e., as the largest order of derivation with continuous derivative. Moreover, the fact that the coefficients $\theta[k_n]$ depend on the key value k_n , the approximation in Equation (18) is non-linear and adaptive: non-linear since the dictionary depends on f, adaptive because the choice of I is abitrary [Cam19]. Moreover, compared to the infinite number of weights $\boldsymbol{w}_n \in \mathbb{Z}^{d_X}$ necessary to approximate f with its Fourier series, with the dictionary embedding approach we hope to promote parsimony by considering card(I) low. In particular, [Bar93] proposes to focus on functions with L^1 first order derivative (see Theorem 28). Penalizing the learning algorithm with a ℓ^1 norm on the weights $\boldsymbol{\theta}$ allows to promote sparsity and achieve parsimony. In other words, the 1-hidden-layer \mathcal{MLP} will learn how to approximate the labeling function with the least amount of hidden features and with an approximation error that is not dependent on the dimension of the input space (or alternatively, for a fixed accuracy) the design

of the $\mathcal{N}\mathcal{N}$ will require an amount of hidden neurons that is independent of the dimension of the data space.

Among the other advantages of sparsity, it must be mentioned the fact that the data variation is better disentangled and therefore it helps separate the interactions (as outlined in Section 2.3) and to adopt the strictly necessary number of keywords for the dictionary in Equation (18). Nevertheless, forcing too much sparsity may limit the predictive performance of the \mathcal{NN} for an equal number of neurons, because it reduces the effective capacity of the model [GBB11].

2.4 How to improve the \mathcal{MLP} accuracy?

Despite the fact that Theorem 24 proves the universal approximation capability of a 1-hidden-layer \mathcal{MLP} , provided that enough neurons are considered, this result is quite hard to exploit in real applications, since the number of neurons can easily become too large to handle fro a computational standpoint. The natural question that arises is: what kind of functions cannot be approximated with an arbitrary accuracy by a \mathcal{MLP} of N_{ℓ} layers? Moreover, how many neurons should be considered for each layer? What is the effect of having a number of layers that is higher than the number of neurons per layer? The answers to these questions were provided by Eldan and Shamir [ES16] showed that it exists an approximately radial function $\varphi(||\mathbf{x}||) : \mathbb{R}^{d_X} \to \mathbb{R}$ that can be approximated by a "small" (bounded number of neurons) 2-hidden-layers \mathcal{MLP} with arbitrary accuracy, but that cannot be approximated by a 1-hiddenlayer \mathcal{MLP} below a certain accuracy, unless the number of neurons N_K grows exponentially with d_X . In particular, this results is valid for any activation function g and with no further constraint on the weights and biases adopted in the \mathcal{MLP} (on the contrary, the Universal Approximation Theorem requires that the high-frequency components $\|\boldsymbol{w}_n\|$ are smaller than a constant). This results proves that increasing the depth of the \mathcal{MLP} widens the approximation capability of the \mathcal{MLP} and that the depth of the \mathcal{MLP} should be privileged with the respect to its layer dimension (but being careful to avoid vanishing gradient problems, as discussed in Section 4.2. The approximately radial function is the inverse Fourier transform of the indicator function on a unit volume euclidean ball $B_{d_X}(\boldsymbol{w})$, of radius R_{d_X} such that $R_{d_X} \cdot B_{d_X}$ has unit volume and it reads:

$$\varphi\left(\boldsymbol{x}\right) = \left(\frac{R_{d_X}}{\|\boldsymbol{x}\|}\right)^{\frac{d_X}{2}} J_{\frac{d_X}{2}}\left(2\pi R_{d_X}\|\boldsymbol{x}\|\right) = \int_{\boldsymbol{w}:\|\boldsymbol{w}\| \le R_d} e^{-2\pi i \langle \boldsymbol{x}, \boldsymbol{w} \rangle} d\boldsymbol{w} \qquad (19)$$

with $J_{\frac{d_X}{2}}(2\pi R_{d_X} || \boldsymbol{x} ||)$ the Bessel function of first kind of order $\frac{d_X}{2}$. $\varphi(\boldsymbol{x})$ has a high-frequency symmetry by revolution, as depicted in Figure 12. Since the function peaks in $\boldsymbol{x} = \boldsymbol{0}$, the choice of training samples $(\boldsymbol{x}_i, \varphi(\boldsymbol{x}_i))$ must be done carefully. In Figure 13, 4000 samples uniformly distributed in $[-10, 10]^2$ were chosen, 70% of which used for training and 30% for testing purposes. According to [ES16], the number of high-frequency components \boldsymbol{w}_n necessary



Figure 12: Function $\varphi(\boldsymbol{x}) = \left(\frac{R_{d_X}}{\|\boldsymbol{x}\|}\right)^{\frac{d_X}{2}} J_{\frac{d_X}{2}}(2\pi R_{d_X}\|\boldsymbol{x}\|)$, defined in [ES16], with $d_X = 2$ and $R_d = 0.2$.



Figure 13: 4000 train and test samples $(x_i, \varphi(x_i))$ to train a \mathcal{MLP} that approximates the function $\varphi(x_i)$ defined in Equation (19). Each instance was uniformly sampled over $[-10, 10]^2$.

to approximate such a function with a one-hidden-layer \mathcal{MLP} (referred as to h_{θ}^1) is very large, which prevents the choice of a parsimonious representations by thresholding the Fourier coefficients such that $\|\boldsymbol{w}_n\| > C$. On the contrary, the revolution symmetry is captured by adding a second hidden layer to the standard 1-hidden-layer \mathcal{MLP} , obtaining a 2-hidden-layers \mathcal{MLP} referred as

to h_{θ}^2 . In order to numerically prove this statement, Figure 14 shows the L^2 error $\|\varphi(\mathbf{x}) - h_{\theta}(\mathbf{x})\|^2$ evolution, along training epochs (for first order gradient descent algorithm with AdamW optimizer, see Section 3.4.5) for both h_{θ}^1 (featured by 10000 neurons in its single hidden layer, with *ReLU* activation functions) and h_{θ}^2 (featured by 100 neurons for each of the 2-hidden layers, with *ReLU* activation functions).



Figure 14: Learning curves for h_{θ}^1 (1-hidden-layer \mathcal{MLP}) and h_{θ}^2 (2-hidden-layers \mathcal{MLP}), representing the evolution of $\|\varphi(x) - h_{\theta}(x)\|^2$ with training epoch. Blue and red curves represent the error evolution $\|\varphi(x) - h_{\theta}^1(x)\|^2$, for train and test datasets respectively. Light blue and orange thick curves represent the error evolution $\|\varphi(x) - h_{\theta}^1(x)\|^2$, for train and test datasets respectively. Light blue and test datasets respectively.

Figure 14 shows the net improvement (one two orders of magnitude) obtained by adding one extra hidden layer to the standard 1-hidden-layer \mathcal{MLP} : not only the error decreases for both train and test dataset, but the number of neuron required per layer is 100 times lower for h_{θ}^2 . Despite the large number of neurons, the $h_{\theta}^1 \quad \mathcal{MLP}$ cannot efficiently minimize the mean-square error, proving the theoretical findings by Eldan and Shamir [ES16].

2.5 From neurons to filters

Given its universal approximation capability (see Theorem 24), the use of \mathcal{MLP} is rather attractive. However, in order to pursue a general criterion of parsimony, some specific *symmetries* can be exploited. For instance: classifying the image should not depend on any affine transformation (i.e., roto-translation). Exploiting symmetries (or more in general *group invariance*) is a valid strategy to face the curse of dimensionality. The statistical model must avoid overparametrization (that leads to overfitting) due to non invariant representations of the dataset with the respect to those symmetries [Cam20].

To better understand this aspect, let us consider a collection of labeled images $\mathcal{D}_{XY} = \{\mathbb{X}_k, y_k\}_{k=1}^N$ of $W \times H$ pixels each. Any image \mathbb{X}_k of the database is composed by a set of pixel and each pixel is associated to a color, expressed by a 3-dimensional vector on the Red Green Blue (RGB) scale. Therefore, $\mathbb{X}_k \in \mathbb{R}^{W \times H \times 3}$. For the sake of simplicity, the image is spanned over a $[0, 1]^2$ domain, by a regular grid of $W \times H$ pixels with a 2D index $\boldsymbol{x} = x_1 \boldsymbol{e}_1 + x_2 \boldsymbol{e}_2, (x_1, x_2) \in [0, 1]^2$ and with the color being represented by $\mathbb{X}(x_1, x_2, :) = \psi(x_1, x_2) = \psi_R(x_1, x_2) \boldsymbol{e}_1 + \psi_G(x_1, x_2) \boldsymbol{e}_2 + \psi_B(x_1, x_2) \boldsymbol{e}_3$ (see the code snippet below and Figure 15).

Example 3. Convert an image to PyTorch tensor

The example that follows parses and converts a digital image X of the geological cross-cut underneath the Kashiwazaki-Kariwa nuclear power plant (see [CGL21]).

```
import torch
1
2
      import torchvision
      from torchvision.transforms import ToTensor
3
4
      from PIL import Image
5
6
      # parse image
      with Image.open('./geology.tiff') as image:
7
          # convert image to RGB
8
          image=image.convert("RGB")
10
11
      # convert image to tensor
     X = ToTensor()(image).unsqueeze(0)
12
13
     print(X.size)
      # >>> torch.Size([1, 3, 188, 257])
14
```

 y_k belongs to an alphabet A for a classification purpose and it is usually coded as a *one-hot vector*, i.e. a binary vector of the size of the alphabet. The framework can be extended to regression problems to $(y_k \in \mathbb{R} \text{ for instance})$.

The most-intuitive way of classifying an image is to apply a \mathcal{MLP} featured by a neuron per pixel in the input layer. The number of neurons could easily become very large: this choice is not parsimonious. Moreover, in real applications, image features extend over patches that can span several pixels and those patch-wise features do not depend on their position in the image (translational invariance). \mathcal{MLP} do not help neither separating scales (characteristics



Figure 15: On the left, the tiff image $\psi(x)$ in RGB scale of geological cross-cut underneath the Kashiwazaki-Kariwa nuclear power plant (see [CGL21]). On the right the digitized discrete image with size $1 \times 3 \times W \times H$ pixels, where the first shape represents number of images (1), the second one the RGB code (3) and the last two the width and height of the image respectively.

features extending over patches of different sizes) nor exploiting possible symmetries and invariant features [Cam20].

A significant advance in this sense was made in 1998 by Y. LeCun and Bengio, whose seminal paper [CBB97] paved the way of designing Convolution Neural Networks \mathcal{CNN} , whose widespread usage in real applications kick-started in 2012, due to the work of Krizhevsky et al. [KSH17] that conceived a deep \mathcal{CNN} for the ImageNet LSVRC-2010 classification contest. The latter consisted in developing a 1000-class classifier (see Section 4.1 for further details) based on a database of 1.2 million high-resolution images. Their AlexNet achieved top-1 and top-5 error rates of 37.5% and 17.0%, respectively, which at that time was considerably better than the previous state-of-the-art. AlexNet is featured by 60 million parameters, arranged in five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers with a final 1000-way softmax. This inspired the switch of paradigm: the concept of neurons is replaced by the concept of *convolutional filters*, as sketched in Figure 16, which capture the a priori knowledge of the problem and that considerably reduce the connections of the first layer, thereby facilitating the optimization of the weights by learning.

As shown in Figure 16, each neuron in the first layer (the green spheres in Figure 16) is only connected to (or responsible for) a $k_W \times k_H$ patch of the initial image typically. Each neuron perform a local linear combination of the pixel values which it is connected to and a non-linear activation is applied. In \mathcal{CNN} , the weights of each neurons are the same, as if the image was chopped into patches, consecutively fed to the same neuron with $k_W \times k_H \times N_c$ weights.



Figure 16: Sketch of convolution layer applied on a the digitized image X of the geological cross-cut underneath the Kashiwazaki-Kariwa nuclear power plant (see [CGL21]). Four feature maps are shown.

This architecture represents a significant reduction in complexity. N_c represents the number of *filters* outputted by the neuron while the visual patch slides along the image. The neuron weight and bias are independent on the position of the patch, providing a translation-equivariant output.

This invariance by translation is obtained by applying a Linear Translation Invariant (LTI) filter to the image, i.e. a convolution, which \mathcal{CNN} are named from:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^{2}} \boldsymbol{H}\left(\boldsymbol{u}\right) \cdot \boldsymbol{\psi}\left(\boldsymbol{x}-\boldsymbol{u}\right) d\boldsymbol{u} + \boldsymbol{b} = \boldsymbol{H} \star \boldsymbol{\psi}\left(\boldsymbol{x}\right) + \boldsymbol{b}$$
(20)

with $\boldsymbol{z}(\mathbb{X}) = \sum_{c=1}^{N_c} z_c(\mathbb{X}) \boldsymbol{e}_c$ being called *feature maps* and \boldsymbol{b} the filter bias. Each feature map $z_c(\mathbb{X}) = \langle \boldsymbol{H} \star \boldsymbol{\psi}, \boldsymbol{e}_c \rangle + b_c$ corresponds to a LTI filter on the image, for a total of N_c filters. The LTI filter in Equation (20) is generically performed over \mathbb{R}^2 though the image is defined over the compact support $[0, 1]^2$, discretized on the $W \times H$ finite grid. Feature maps are therefore the result of a discrete convolution:

$$\boldsymbol{z}[j,i] = \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \boldsymbol{H}[u,v] \cdot \boldsymbol{\psi}[i-u,j-v] + \boldsymbol{b}, \quad (i,j) \in [\![0,W]\!] \times [\![0,H]\!]$$
(21)

with $f[u, v] = f\left(\frac{u}{W}, \frac{v}{H}\right)$ and z being stored in C order, contiguous along the rows. In practice, the discrete convolution requires the use of (p_W, p_H) zero-padding in order to make the filter slide along the whole image, as shown in Figure 17.

The filter impulse response \boldsymbol{H} is causal and its discrete counterpart $\boldsymbol{H}[u, v]$ is defined over the support $[\![0, k_W - 1]\!] \times [\![0, k_H - 1]\!]$, with $0 < k_W \leq W$ and



Figure 17: Sketch of the discrete convolution over an image. Reprinted from [DV18] (see https://github.com/vdumoulin/conv_arithmetic). Zero-padding $p_W = p_H = 1$ is added around the image, in order to make the 3 × 3 kernel slide along the whole image, defined over the $W \times H$ grid.

 $0 < k_H \leq H$ (see Equation (203)). If present, the non-linear activation function acts channel-wise, producing feature maps as $\boldsymbol{z}(\mathbb{X}) = \sum_{c=1}^{N_c} g(z_c(\mathbb{X}))\boldsymbol{e}_c$.

The convolution can be strided and/or dilated. In its continuous form, the strided convolution reads:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^{2}} \boldsymbol{H}\left(\boldsymbol{u}\right) \cdot \boldsymbol{\psi}\left(\operatorname{diag}\left(\boldsymbol{s}_{W}, \boldsymbol{s}_{H}\right) \boldsymbol{x} - \boldsymbol{u}\right) d\boldsymbol{u} + \boldsymbol{b}$$
(22)

with strides $s_W \ge 1$ and $s_H \ge 1$. In its discrete counterpart, each strided convolutional filter skips s_W and s_H pixels respectively while sliding along the two directions and it can be written as:

$$\boldsymbol{z}[j,i] = \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \boldsymbol{H}[u,v] \cdot \boldsymbol{\psi}[s_W \cdot i - u, s_H \cdot j - v] + \boldsymbol{b}, \quad (i,j) \in [\![0,W]\!] \times [\![0,H]\!]$$
(23)

The use of stride allows to reduce the spatial dimension of the feature maps by a factor $\frac{1}{s_W}$ and $\frac{1}{s_H}$ respectively, as shown in Figure 18.

Dilating the convolutional filters corresponds to perform an *atrous convolution*, the French "convolution à trous", and it was implemented in CNN for the first time in [Che+14; YK15]. In other words, dilated convolutions expands the receptive field at a fixed kernel size, by skipping $d_W \geq 1$ and $d_H \geq 1$ pixels between the kernel elements, corresponding to holes in the kernel (see Figure 19).

In its continuous form, dilated convolution is defined by the following expression:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^{2}} \boldsymbol{H}\left(\operatorname{diag}\left(d_{W}, d_{H}\right)\boldsymbol{u}\right) \cdot \boldsymbol{\psi}\left(\boldsymbol{x} - \operatorname{diag}\left(d_{W}, d_{H}\right)\boldsymbol{u}\right) d\boldsymbol{u} + \boldsymbol{b} \quad (24)$$



Figure 18: Sketch of the strided discrete convolution over an image. Reprinted from [DV18] (see https://github.com/vdumoulin/conv_arithmetic). Zero-padding $p_W = p_H = 1$ is added around the image, in order to make the 3 × 3 kernel slide along the whole image, defined over the $W \times H$ grid. The strides $s_W = s_H = 2$ reduce the size of the feature maps by approximately $\frac{1}{s_W}$ and $\frac{1}{s_H}$ respectively (depending on the values of p_W and p_H).



Figure 19: Sketch of the strided discrete convolution over an image. Reprinted from [DV18] (see https://github.com/vdumoulin/conv_arithmetic). The dilation rate $d_W = d_H = 2$ fictitiously widen the kernel size, adding holes in the middle and keeping the same kernel size $k_W = k_H = 3$.

In its discrete form, dilated convolutional filter reads instead:

$$\boldsymbol{z}[j,i] = \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \boldsymbol{H} [d_W \cdot u, d_H \cdot v] \cdot \boldsymbol{\psi} [i - d_W \cdot u, j - d_H \cdot v],$$
(25)
(i,j) $\in [\![0,W]\!] \times [\![0,H]\!]$

Equation (25) shows that the dilated convolutions are nothing more than standard convolutions with effective kernel sizes $\hat{k}_W = k_W + (k_W - 1) \cdot (d_W - 1)$ and $\hat{k}_H = k_H + (k_H - 1) \cdot (d_H - 1)$.

Example 4. Convolutional filters in PyTorch

In PyTorch, a convolutional layer is defined by the class Conv2D as in the following code snippet. Different kernel, stride, padding and dilation rates are adopted, to assess their effect on the image in Figure 15. A hyperbolic tangent activation function is applied to each feature map, in order to compare feature

maps with different kernels, stride, dilation on the same color scale [-1, 1]. However, in deep \mathcal{CNN} architectures (see Section 2.6), convolutional filters are first followed by a pixel-wise non-linear activation function and then by a pooling operation.

```
import torch
 1
       # assure reproductibility
 2
      torch.manual_seed(0)
 3
 4
       # Create convolutional feature maps
 \mathbf{5}
 6
      Nc = 4 # number of output channels
      pW = 1 \# zero-pad along W
 7
      pH = 1 # zero-pad along H
kW = 3 # kernel-size along k
 8
9
      kH = 3 # kernel-size along H
10
      sW = 1 # stride along W
11
      sH = 1 # stride along H
12
      dW = 1 # dilation along W
dH = 1 # dilation along W
13
14
      p = (pW, pH)
k = (kW, kH)
15
16
      s = (sW, sH)
17
      d = (dW, dH)
18
19
      cnn = torch.nn.Sequential(
20
              torch.nn.Conv2d(in_channels=3, out_channels=Nc,
21
                                kernel size=k.
                                stride=s,
22
23
                                padding=p),
              torch.nn.Tanh()
^{24}
^{25}
             )
26
      # compute feature maps
      z = cnn(X).detach().cpu().numpy() # feature maps in numpy format
27
```

Discrete convolutions are performed on images defined on a grid of $W \times H$ pixels indexed by (i, j). The following algebraic relationship between kernel size, stride, padding, dilation rates (along each dimension) holds [DV18]:

$$o_i = \lfloor \frac{i + 2p_W - k_W - (k_W - 1) \cdot (d_W - 1)}{s_W} \rfloor + 1$$
(26)

$$o_{j} = \lfloor \frac{j + 2p_{H} - k_{W} - (k_{H} - 1) \cdot (d_{H} - 1)}{s_{H}} \rfloor + 1$$
(27)

Equation (27) is quite useful to design a convolutional layer: o_i (o_j respectively) represents the output index corresponding to the input index *i* (*j* respectively). The following code snippets show how to plot the feature maps:

```
import matplotlib
1
     import matplotlib.pyplot as plt
2
3
     from matplotlib.pyplot import cm
      # Define color map
4
5
     rainbow_cmap = matplotlib.colormaps['viridis'].resampled(1000)
6
      # loop over the output channels (feature maps)
7
     for c in range(Nc):
8
       zc = z[0,c,:,:] # c-th feature map
9
10
       fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(9,4))
11
```

```
12
13
          # plot image
14
         ax.imshow(zc, vmin=-1.0, vmax=1.0, cmap= rainbow_cmap)
15
16
          # set axes
         ax.set_xticks(range(0,image.size[0]),
17
                           minor=False)
18
19
          ax.set_yticks(range(0,image.size[1]),
20
                           minor=False)
          ax.set_xticklabels([r"0"]+
21
               ['' for _ in range(1,image.size[0]-1)]+
[r"W"], fontsize=18)
22
^{23}
          ax.set_yticklabels([r"0"]+
^{24}
         ax.set_yterrate(1, 0, 1, mage.size[1]-1)]+
[r"H"], fontsize=18)
ax.set_xlabel(r"$j$", fontsize=18)
ax.set_ylabel(r"$i$", fontsize=18)
^{25}
26
27
^{28}
         ax.set_xlim(0,image.size[0])
29
30
          ax.set_ylim(image.size[1],0)
         ax.set_title(r"$z_{{{:>d}}}$".format(c), fontsize=18)
31
32
33
          # save figure
          fig.savefig("tsuda_geology_z{}_k{}{}.png".format(c,
34
35
                                                                               kW.
36
                                                                               kH.
37
                                                                               sW,
                                                                               sH).
38
39
                         dpi=300.
                         bbox inches="tight")
40
```

In Figure 20, the effect of two different zero-padding is shown: for $p_W = p_W = 0$ (Figure 20a-Figure 20d) the finite size kernel cannot slide until the end of the image, leaving a narrow blank band at the two bottom-right edges of the images, that consists in an effective downsampling of the image, since the edges are disregarded [MMD20]. This downsampling vanishes by adopting a zero-padding of size $p_W = p_H = 2$ (Figure 20e-Figure 20h). The effect of different kernel sizes $k_W = k_H = 3$ (Figure 21a-Figure 21d) and $k_W = k_H = 5$ (Figure 21e-Figure 21h) is shown in Figure 21. A wider kernel adds extra parameters to the CNN resulting in a larger receptive field.

Figure 22 shows the effect of changing the stride: $s_W = s_H = 1$ preserves the image size (Figure 22a-Figure 22d), whereas when $s_W = s_H = 2$, the feature maps have half the sizes in both directions (Figure 22e-Figure 22h). Strided convolution reduce the complexity of the initial data, by subsampling it. However, strided convolution do not entail translation-covariant feature maps, with possible loss of the original information contained in the image (see Remark 30). As strided convolutional filters disregard intermediary pixels, translated versions will results in more equivalent outputs if neighbouring pixels are similar to each other across a given patch of the image [MMD20].

Finally, the role of dilation is clear from Figure 23: dilating the convolutional kernel by a rate d = 5 (Figure 23e-Figure 23h) widens the receptive field at fixed kernel size, blurring the feature maps by skipping intermediate pixels and adding the blank border if the padding is not changed accordingly. Moreover,



Figure 20: Effect of the padding for $k_W = k_H = k = 5$, $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ and $p_W = p_H = p = 0$ (a,b,c,d) and $p_W = p_H = p = 2$ for (e,f,g,h).



Figure 21: Effect of the kernel size for $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ and $k_W = k_H = k = 3$ (a,b,c,d) and $k_W = k_H = k = 5$ for (e,f,g,h). In (a,b,c,d), the padding $p_W = p_H = p = 2$, whereas in (e,f,g,h), $p_W = p_H = p = 1$: this difference in padding does not affect the overall result, except the fact that in both cases the kernel properly slides along the whole image, without leaving blank borders.





Figure 22: Effect of the stride for $k_W = k_H = k = 3$, $p_W = p_H = p = 1$, $d_W = d_H = d = 1$ and $s_W = s_H = s = 1$ (a,b,c,d) and $s_W = s_H = s = 2$ for (e,f,g,h).

the image is "shrunk" by the dilation, with partial information loss (the holes in the fictitious kernel).



Figure 23: Effect of the dilation rate for $k_W = k_H = k = 5$, $p_W = p_H = p = 1$, $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ (a,b,c,d) and $d_W = d_H = d = 5$ for (e,f,g,h).

In addition to discrete convolutions, *pooling* operations are often adopted on images. The average- or max-pooling operations also reduce the size of feature maps by outputting either the average or the maximum value of each patch in the image (or in the feature map). Pooling operations are similar to standard convolutional filters, except the fact that the kernel product is replaced by either the average value of the patch or by its maximum value. In particular:

Average pooling (AvgPooling) reads (see Section 3.2 fur further insights):

$$\boldsymbol{z}[j,i] = \frac{1}{k_W \cdot k_H} \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \boldsymbol{\psi} \left[s_W \cdot i + d_W \cdot u, s_H \cdot j + d_H \cdot v \right],$$
(28)
(*i*, *j*) $\in [\![0,W]\!] \times [\![0,H]\!]$

Max-pooling (MaxPooling) reads instead:

$$\boldsymbol{z}[j,i] = \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \max_{u \in [\![0,k_W-1]\!] v \in [\![0,k_H-1]\!]} \max_{\boldsymbol{\psi}} \psi \left[s_W \cdot i + d_W \cdot u, s_H \cdot j + d_H \cdot v \right],$$

(*i*, *j*) $\in [\![0,W]\!] \times [\![0,H]\!]$ (29)

Compared to convolution, pooling operations have no weights nor biases. Moreover, they are channel-wise operations: both Equation (28) and Equation (29) perform average and max operation on each input channel (color of the image or channel of the previous feature map). Therefore, the number of output feature maps is limited to the number of channels fed to the pooling operation.

Example 5. Comparison between AvgPool2d, MaxPool2d and Conv2d in PyTorch

In PyTorch, *AvgPooling* and *MaxPooling* on images are two layers defined by the classes AvgPool2d and MaxPool2d. The code snippet below shows how to set them up, with a hyperbolic tangent activation function.

```
import torch
 1
        # assure reproductibility
 2
       torch.manual_seed(0)
 3
 4
 \mathbf{5}
        # Create convolutional feature maps
       Nc = 4 # number of output channels
 6
       pW = 1 # zero-pad along W
 7
       pH = 1 # zero-pad along H
kW = 3 # kernel-size along W
 8
 9
       kH = 3 # kernel-size along H
10
       sW = 1 # stride along W
sH = 1 # stride along H
11
12
13
       dW = 1 # dilation along W
dH = 1 # dilation along W
14
      p = (pW, pH)
k = (kW, kH)
15
16
       s = (sW, sH)
17
       d = (dW, dH)
18
19
       # average pooling
```

```
avgpool = torch.nn.Sequential(
20
              torch.nn.AvgPool2d(kernel_size=k,
21
                                    stride=s,
22
23
                                    padding=p),
              torch.nn.Tanh()
^{24}
^{25}
              )
      # compute average-pooling feature maps
z_ap = avgpool(X).detach().cpu().numpy() # feature maps in numpy format
26
27
^{28}
29
       # max-pooling
       avgpool = torch.nn.Sequential(
30
31
              torch.nn.MaxPool2d(in_channels=3,
32
                                    kernel_size=k
33
                                    stride=s.
                                    padding=p),
34
              torch.nn.Tanh()
35
36
       # compute average-pooling feature maps
37
       z_mp = avgpool(X).detach().cpu().numpy() # feature maps in numpy format
38
```

The visual effect of of AvgPooling and MaxPooling compared to the standard convolution is depicted in Figure 24. The use of hyperbolic tangent as non-linear activation function allows to compare the output of each layer on the same color scale [-1, 1]

2.6 Deep convolutional architectures

Convolutional layers described in Section 2.5 are generally stacked upon each other, in order to construct deep CNN such as the one depicted in Figure 25.

Deeper layers act on previous feature maps and the spatial localization in the original image is progressively lost. At the end of a stack of convolution/pooling layers, deep \mathcal{CNN} are featured by some \mathcal{MLP} as output layers, to estimate y. [HS15] estimated that the complexity of a \mathcal{CNN} with N_{ℓ} convolutional layer, each one with kernel $k_{W}^{(\ell)} = k_{H}^{(\ell)} = k^{(\ell)}$ and with no \mathcal{MLP} at the end, is of the order of:

$$\mathcal{O}\left(\sum_{\ell}^{N_{\ell}} N_c^{(\ell-1)} \cdot N_c^{(\ell)} \cdot k^{(\ell)2} \cdot m^{(\ell)2}\right)$$
(30)

with $N_c^{(\ell)}$ being the output channels of each convolutional layer and $m^{(\ell)2}$ the size of the output feature maps at each layer ℓ . The complexity is therefore dominated by $m^{(\ell)2}$, which justify the interest in subsampling, via strided convolution or strided pooling. Subsampling operations drastically reduce the amount of neurons in the last output \mathcal{MLP} , leading to even less cumbersome training process. Subsampling has a major downturn though. As a matter of fact, convolutional filters with unitary stride (no subsampling) are translation covariant (see definition in Equation (199)), i.e., the feature maps of a translate images are the translation of the original feature maps. If the original image is subsampled by adopting strided convolution, the similarity between an image and its translated version is lost, because some pixels are skipped as observed in



Figure 24: Comparison between convolution with $k_W = k_H = k = 3$, $p_W = p_H = p = 2$, $s_W = s_H = s = 1$, $d_W = d_H = d = 1$ (a,b,c), average pooling (d,e,f) and max-pooling (g,h,i).



Figure 25: Example of deep \mathcal{CNN} (source: https://github.com/gwding/draw_convnet).

Remark 31 [MMD20]. The remedy consists into use non-strided convolutions, followed by strided average or max pooling operations, that preserve the local homogeneity of the image, drastically decreasing the size of the output feature and therefore the complexity of the \mathcal{CNN} training scheme. Alternatively, the number of feature maps (*channels*) is generally increased accordingly (stride of 2, double the channels), when strided convolutions are adopted. On the contrary, Average and max pooling grant translation invariance (see Section 3) but they preserve the amount of input channels. The image and successive feature maps can be also shrunk by adopting dilated convolution, or even downsampled by avoiding zero-padding (see Figure 20.

Example 6 shows a simple example of CNN adopted to classify CIFAR10 database⁴, containing 60000 32×32 color images in 10 different classes.

Example 6. Example of deep convolutional classifier with PyTorch The following code snippet⁵ shows how to easily construct a deep CNN for CIFAR10 classification.

```
# Source: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
 2
      import torch.nn as nn
      import torch.nn.functional as F
 3
 4
      import torch.optim as optim
       # assure reproductibility
 \mathbf{5}
       torch.manual_seed(0)
 6
       # define CNN class as subclass of the master nn.Module
 7
 8
       # Once initialized, the forward function must be customized
 9
       class CNN(nn.Module):
10
           def __init__(self):
                super().__init__()
self.conv1 = nn.Conv2d(in_channels = 3,
11
12
                                           out_channels = 6,
13
14
                                           kernel_size = (5,5),
15
                                           stride = (1,1)
16
                                           )
17
                self.pool = nn.MaxPool2d(kernel_size = 2,
18
                                             stride = 2)
19
20
                self.conv2 = nn.Conv2d(in_channels = 6,
^{21}
22
                                           out_channels = 16,
                                           kernel_size = (5,5),
23
                                           stride = (1,1)
^{24}
25
                self.a1 = nn.Linear(in_features = 16 * 5 * 5,
26
                                       out_features = 120)
27
                self.a2 = nn.Linear(in_features = 120,
28
29
                                       out features = 84)
                self.a3 = nn.Linear(in_features = 84,
30
                                       out_features = 10)
31
^{32}
           def forward(self, x):
33
34
                x = self.pool(F.relu(self.conv1(x))) # conv+relu+pool (1)
                x = self.pool(F.relu(self.conv2(x))) # conv+relu+pool (2)
x = torch.flatten(x, 1) # flatten all dimensions except batch
x = F.relu(self.a1(x)) # linear+relu (1)
35
36
37
```

⁴https://en.wikipedia.org/wiki/CIFAR-10#cite_note-1

⁵ from PyTorch tutorial, see https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

```
x = F.relu(self.a2(x)) # linear+relu (2)
38
               x = self.a3(x) # final output layer
39
               return x
40
^{41}
^{42}
      h_theta = CNN()
43
^{44}
      # Define loss function
45
      l = nn.CrossEntropyLoss()
46
\mathbf{47}
      # Define SGD optimizer
^{48}
49
      optimizer = optim.SGD(h_theta.parameters(), lr=0.001, momentum=0.9)
```

The CIFAR10 dataset can be easily accessed via PyTorch with the following few lines of code:

```
# Source: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
 1
2
      import torch
3
      import torchvision
      import torchvision.transforms as transforms
4
      import matplotlib.pyplot as plt
 5
      import numpy as np
# assure reproductibility
 6
 7
      torch.manual_seed(0)
 8
      # Define pre-processing operations
transform = transforms.Compose(
9
10
          [transforms.ToTensor(),
11
            transforms.Normalize((0.5, 0.5, 0.5),
12
             (0.5, 0.5, 0.5))])
^{13}
14
15
      batch_size = 4
16
      # Define train dataset (pre-processing)
17
      Dxy_train = torchvision.datasets.CIFAR10(root='./data',
18
19
                                                     train=True,
                                                     download=True,
^{20}
21
                                                     transform=transform)
       # Define train data loader
^{22}
      Dxy_train = torch.utils.data.DataLoader(Dxy_train,
23
                                                    batch_size=batch_size,
^{24}
                                                    shuffle=True,
25
^{26}
                                                    num_workers=2)
27
      # Define test dataset (pre-processing)
^{28}
      Dxy_test = torchvision.datasets.CIFAR10(root='./data',
^{29}
                                                    train=False.
30
                                                    download=True,
31
                                                    transform=transform)
^{32}
33
      #_Define_test_data_loader
^{34}
      Dxy_test = torch.utils.data.DataLoader(Dxy_test,
35
                                                   batch_size=batch_size,
                                                   shuffle=False,
36
                                                   num_workers=2)
37
38
      classes = ('plane', 'car', 'bird', 'cat',
    'deer', 'dog', 'frog', 'horse',
    'ship', 'truck')
39
40
41
42
       # functions to plot an image sample
^{43}
      def imshow(img):
    img = img / 2 + 0.5 # unnormalize
^{44}
45
           npimg = img.numpy() # transform to numpy format
46
47
           plt.imshow(np.transpose(npimg, (1, 2, 0)))
```

```
plt.savefig("CIFAR10_sample.png", dpi=300, bbox_inches="tight")
48
49
50
51
      # get some random training images
      dataiter = iter(Dxy_train)
52
53
     Xi, yi = next(dataiter)
54
      # show images
55
      imshow(torchvision.utils.make_grid(Xi))
56
57
      # print labels
     print('_'.join(f'{classes[yi[j]]:5s}' for j in range(batch_size)))
58
```

```
1
     # Source: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
 ^{2}
      n_e = 100
 3
      for epoch in range(n_e): # loop over the dataset multiple times
 4
           running_loss = 0.0
 \mathbf{5}
           for i, batch in enumerate(Dxy_train, 0):
 6
 7
                # get the inputs; data is a list of [inputs, labels]
               Xi, yi = batch
 8
 9
10
               # zero the parameter gradients
               optimizer.zero_grad()
11
12
               # forward + backward + optimize
13
               h_theta_Xi = h_theta(Xi)
^{14}
15
               loss = l(h_theta_Xi, yi)
16
               loss.backward()
17
               optimizer.step()
18
               # print statistics
19
               running_loss += loss.item()
20
               if i % 2000 == 1999: # print every 2000 mini-batches
print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
^{21}
^{22}
23
                   running_loss = 0.0
      # save NN for future use (inference, fine tuning, ...)
^{24}
      PATH = './cifar_net.pth'
25
      torch.save(net.state_dict(), PATH)
26
27
      # Test on test data
^{28}
29
      dataiter = iter(Dxv test)
30
      Xi, yi = next(dataiter)
^{31}
32
33
       # print images
^{34}
      imshow(torchvision.utils.make_grid(Xi))
      print('GroundTruth: ', ', ', join(f'{classes[yi[j]]:5s}' for j in range(4)))
35
```

Compared to deep \mathcal{MLP} , deep \mathcal{CNN} such as that in Example 6 considerably reduced the initial complexity of the problem because neurons in the convolutional filter maps are specialized and translation covariant and pooling grants the invariance by translation (same class despite its spatial position), overall minimizing the number of weights in each layer [Cam19]. However, deep \mathcal{CNN} can easily reach several hundreds of millions of parameters. The number of parameters is much larger (in general) than the dimensionality of the original data. However, even if the network is trained with thousands or even millions of samples (CIFAR10 is featured by 60000 images), \mathcal{CNN} considerably reduce

the dimension of the problem, ranging around 10^6 . Typically, the CNN output layer has $\approx 10^2$ output weights. The complex intermediate representations $\boldsymbol{z}(\mathbb{X})$ generated by convolutional filters are progressively more and more insensitive to small roto-translations or other symmetries of the image and they linearize the decision boundaries between class (in classification problems).

2.6.1 Transposed convolutions

Transposed convolutions (or "fractionally strided" convolutions) [Zei+10] are the adjoint (transposed) version of the classical convolution defined in Equation (20): in other words, deconvolution. The sketch of the transposed convolution is depicted in Figure 26 In particular, while convolutions are used to en-



Figure 26: Sketch of the discrete transposed convolution over an image. Reprinted from [DV18] (see https://github.com/vdumoulin/conv_arithmetic). Zero-padding $p_W = p_H = 1$ is added around the image, in order to make the 3 × 3 kernel slide along the whole image, defined over the $W \times H$ grid.

code the data, by projecting it onto a reduced-order latent manifold, transposed convolutions are adopted to decode the latent variables into the data space (see Chapter 3 - Unsupervised Learning: Basic Concepts and Application to Particle Dynamics and Chapter 6 - Non-Euclidean machine learning for geomechanics). The transposed convolution operation can be thought of as the gradient of some convolution with respect to its input (see Section 5.1), which is usually how transposed convolutions are implemented in practice [DV18]. In PyTorch, transposed convolution are designed with the class ConvTranspose2d⁶.

2.7 Time-forward prediction

 $\mathcal{N}\mathcal{N}$ have been successfully adopted to ordered flow of information, such as handwriting recognition, speech recognition and in more general to reproduce time-varying physical phenomena. In this sense, the dataset is composed by time series (of signals, of images, of words etc.). In their standard form, feedforward $\mathcal{N}\mathcal{N}$ are fed with fixed-length input data $\boldsymbol{x} \in \mathbb{R}^{d_X}$ and each instance

⁶https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html

 \boldsymbol{x}_i is considered i.i.d. However, the quantities of interest of physical phenomena displaying dynamic behaviour are naturally correlated in time. This aspect can hardly be captured by feed-forwards \mathcal{MLP} due to their internal structure. In order to feed a time series to a \mathcal{MLP} , for regression purposes for instance, the input data stream must be chopped into i.i.d. windows of d_X samples each. However, this represents a quite strong assumption, since long range time correlations could be overlooked. Moreover, trying to feed the \mathcal{MLP} with the whole time-series would entail a major drawback: the number of hidden neurons required to achieve a satisfactory error grows exponentially with the size of the input dimension, as stated in Equation (97):

$$N_K \approx \varepsilon^{\frac{1-d_X}{k}} \tag{31}$$

Finally, the last but not the least reason why the \mathcal{MLP} is not fit for time-series problems is that the time-series itself represent rather irregular functions, i.e., their highest order of (weak) differentiability k is quite low, which implies an even larger number of hidden neurons.

When dealing with discrete time-histories, the data $\mathcal{D}_{XY} = \{(X_i, y_i)\}_{i=1}^N$ is composed of i.i.d. samples of discrete input signals $X = \{\boldsymbol{x} [t]\}_{t=1}^{N_t}$ and discrete output signals $y = \{\boldsymbol{y} [t]\}_{t=1}^{N_t}$, with $\boldsymbol{x} [t] \in \mathbb{R}^{d_X}$ and $\boldsymbol{y} [t] \in \mathbb{R}^{d_Y}$, both of length N_t .

2.8 Recurrent Neural Networks \mathcal{RNN}

Recurrent Neural Networks (\mathcal{RNN}) [Wer90] are inspired by the \mathcal{MLP} but they are conceived for time-series, since their underlying graph of operation is reflexive. In other words, \mathcal{RNN} is a \mathcal{NN} with one or more feedback loops over time steps [Hay98]. \mathcal{RNN} have infinite impulse response, compared to CNN that have a finite one.

At any time step t, the \mathcal{RNN} produces an output $\boldsymbol{y}[t]$, based on the input time step $\boldsymbol{x}[t]$. The standard \mathcal{RNN} $\boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x}[t])$ is defined as follows [GBC16; Sal+18]:

$$a_t(h, x) = W^{(h)}h_{t-1} + W^{(x)}x[t] + b^{(h)}$$
 (32)

$$\boldsymbol{h}_{t}\left(\boldsymbol{a}\right) = \boldsymbol{g}^{\left(h\right)}\left(\boldsymbol{a}_{t}\right) \tag{33}$$

$$\boldsymbol{a}_t^{(o)} = \boldsymbol{W}^{(o)} \boldsymbol{h}_t + \boldsymbol{b}^{(o)}$$
(34)

$$\boldsymbol{z}_t \left(\boldsymbol{a}^{(o)} \right) = \boldsymbol{g}^{(o)} \left(\boldsymbol{a}^{(o)}_t \right) \tag{35}$$

$$\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}\left[t\right]\right) = \boldsymbol{z}_{t} \circ \boldsymbol{h}_{t} \circ \boldsymbol{a}_{t}\left(\cdot, \boldsymbol{h}_{t-1}\right) \circ \boldsymbol{x}\left[t\right]$$
(36)

with $\boldsymbol{h}[t] \in \mathbb{R}^{d_h}$ representing the hidden state states, a sort of "memory" variable that keeps track of past states of the network over many time steps, similarly to what it is usually done in standard auto-regressive models such as

ARMA, ARIMA, NARMAX among others[Whi51; Xue+10]. $\mathbf{W}^{(h)}$ and $\mathbf{b}^{(h)}$ represent the transition weights and biases, associated to the computation of the hidden state. The value of \mathbf{h}_{t-1} in the pre-activation at time $t \ \mathbf{a}_t \ (\mathbf{h}, \mathbf{x}) = \mathbf{W}^{(h)} \mathbf{h}_{t-1} + \mathbf{W}^{(x)} \mathbf{x} \ [t] + \mathbf{b}^{(h)}$ act as feed-back term. $\mathbf{W}^{(o)}$ and $\mathbf{b}^{(o)}$ represents weights and biases of the output layer. The biases can be seen as learnable exogenous variables conditioning the time forecast. $g^{(h)}$ and $g^{(o)}$ are the non-linear activation function for hidden layer and output layer. $\mathbf{W}^{(h)}, \mathbf{W}^{(x)}$ are dense weight matrices for hidden and input layers and $\mathbf{b}^{(h)}, \mathbf{b}^{(o)}$ the bias vectors for hidden and output layers respectively. A scheme of the \mathcal{RNN} function is depicted in Figure 27.



Figure 27: Sketch of \mathcal{RNN} scheme according to Equation (36)

In order to train \mathcal{RNN} , the empirical loss is computed as a sum of the empirical losses computed at each time step:

$$L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}\right) = \frac{1}{N} \sum_{t=1}^{N_{t}} \sum_{i=1}^{N} l\left(\boldsymbol{y}_{i}\left[t\right], \boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\left[t\right]\right)\right)$$
(37)

It is well known that \mathcal{RNN} are difficult to train [Sal+18] because of their inherent functioning: the long time-span of the dynamics of the hidden state variables, which is certainly necessary to forecast future time steps without the inconvenient of finite impulse response functions (such as in \mathcal{CNN}), represents a major cause of training instability. The main reason is the fact the *forward* computational graph is assembled on previous time-steps. For instance, the derivative of $l\left(\boldsymbol{y}_{i}\left[t\right],\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\left[t\right]\right)\right)$ with the respect to $W_{ij}^{(x)}$ reads:

$$\frac{\partial l}{\partial W_{ij}^{(x)}} \left(\boldsymbol{y}_{i}\left[t\right], \boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\left[t\right]\right) \right) = \sum_{k,l,m,n} \frac{\partial l}{\partial z_{t,k}} \frac{\partial g^{(o)}}{\partial h_{t,l}} \frac{\partial g^{(h)}}{\partial a_{t,m}} \left(\frac{\partial a_{t,m}}{\partial h_{t-1,n}} \frac{\partial h_{t-1,n}}{\partial W_{ij}^{(x)}} + \frac{\partial a_{t,m}}{\partial W_{ij}^{(x)}} \right)$$
(38)

with the following terms that explicitly read:

$$\frac{\partial g^{(o)}}{\partial h_{t,l}} = \sum_{p} \frac{\partial g^{(o)}}{\partial a_{t,p}^{(o)}} W_{lp}^{(o)}$$
(39)

$$\frac{\partial a_{t,m}}{\partial h_{t-1,n}} = W_{mn}^{(h)} \tag{40}$$

$$\frac{\partial a_{t,m}}{\partial W_{ij}^{(x)}} = \delta_{mi} x_j [t-1] \tag{41}$$

$$\frac{\partial h_{t-1,n}}{\partial W_{ij}^{(x)}} = \sum_{q,r} \frac{\partial g^{(h)}}{\partial a_{t,q}} W_{q,r}^{(h)} \frac{\partial h_{t-2,r}}{\partial W_{ij}^{(x)}} + \frac{\partial g^{(h)}}{\partial a_{t,i}} x_j[t-1]$$
(42)

System 42 proves the fact that the backward propagation unfolds the \mathcal{RNN} in time, i.e., the term $x_j[t-1]$ appears and a recursion on the hidden state variables h_t . The gradient of the loss function is the result of chain rule operations that goes back to h_0 and it is therefore called Back-Propagation Through Time (BPTT) [Sal+18] that can be generally rewritten, in a more compact way, as follows:

$$\nabla_{\boldsymbol{W}^{(x)}} l\left(\boldsymbol{y}_{i}\left[t\right], \boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\left[t\right]\right)\right) = \nabla_{\boldsymbol{h}_{t}} \ell \cdot \left(\sum_{\tau=1}^{t} \nabla_{\boldsymbol{h}_{\tau}} \boldsymbol{h}_{t} \otimes \nabla_{\boldsymbol{W}^{(x)}} \boldsymbol{h}_{\tau}\right)$$
(43)

$$\begin{cases} \boldsymbol{\nabla}_{\boldsymbol{h}_{\tau}} \boldsymbol{h}_{t} = \prod_{s=\tau+1} \frac{\partial \boldsymbol{h}_{s}}{\partial \boldsymbol{h}_{s-1}} \end{cases}$$
(44)

$$\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}} = \boldsymbol{W}^{(h)T} \left(\sum_k \frac{\partial g^{(o)}}{\partial \boldsymbol{h}_{s-1,k}} \boldsymbol{e}_k \otimes \boldsymbol{e}_k \right)$$
(45)

Equation (45) outlines three orders of time dependencies: an "immediate" contribution (i.e., $\nabla_{W^{(x)}} h_{\tau}$), a "short-term" one and "long-term" one, the latter referring to $\tau \ll t$ [PMB13]. Long-term memory is the main cause of highly instable training processes, since the gradient tends to vanish if the activation functions squash the input (such as σ , tanh, etc), as extensively explained in Section 4.2, and the loss at time t + 1 to increase [Sal+18]. The sufficient condition is that the largest singular value of the recurrent weight matrix $W^{(h)}$ σ_1 must satisfy the condition $\sigma_1 < \frac{1}{\delta}$. In particular, if the diagonal matrix $\frac{\partial g^{(o)}}{\partial h_{s-1,k}} e_k \otimes e_k$ is bounded by above by a term $\gamma \in \mathbb{R}^+$ such as:

$$\left\|\sum_{k} \frac{\partial g^{(o)}}{\partial h_{s-1,k}} \boldsymbol{e}_{k} \otimes \boldsymbol{e}_{k}\right\| \leq \gamma \tag{46}$$

then the Jacobian $\frac{\partial \pmb{h}_s}{\partial \pmb{h}_{s-1}}$ is bounded as follows:

$$\left\|\frac{\partial \boldsymbol{h}_s}{\partial \boldsymbol{h}_{s-1}}\right\| \le \left\|\boldsymbol{W}^{(h)}\right\| \cdot \left\|\sum_k \frac{\partial g^{(o)}}{\partial h_{s-1,k}} \boldsymbol{e}_k \otimes \boldsymbol{e}_k\right\| \le 1$$
(47)

and $\exists \delta \in \mathbb{R}^+$ such that $\left\| \frac{\partial h_s}{\partial h_{s-1}} \right\| \leq \| \leq \delta < 1$. In this case, the gradient in Equation (45) is bounded as follows:

$$\left\|\boldsymbol{\nabla}_{\boldsymbol{W}^{(x)}} l\left(\boldsymbol{y}_{i}\left[t\right], \boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\left[t\right]\right)\right)\right\| = \left\|\boldsymbol{\nabla}_{\boldsymbol{h}_{t}} \ell \left(\sum_{\tau=1}^{t} \boldsymbol{\nabla}_{\boldsymbol{h}_{\tau}} \boldsymbol{h}_{t} \otimes \boldsymbol{\nabla}_{\boldsymbol{W}^{(x)}} \boldsymbol{h}_{\tau}\right)\right\| \leq \delta^{t-\tau} \left\|\boldsymbol{\nabla}_{\boldsymbol{h}_{t}} \ell\right|$$

$$(48)$$

The upper bound in Equation (48) shows that BPTT for long time histories tend to make the gradient with the respect to $\tau \ll t$ to vanish whenever $\sigma_1 < \frac{1}{\delta}$, in 5-10 gradient descent epochs (see Section 3 for a detailed description of backpropagation) [PMB13]. Because of its proneness to vanishing gradient, \mathcal{RNN} have major difficulties in capturing dependencies as the duration of dependencies increases [BSF94]. The BPTT can even explode along some direction, if $\sigma_1 > \frac{1}{\delta}$. Vanishing gradient can be counteracted by employing *ReLU*, *ELU*, SELU and other activation function with non-zero derivative with the respect to their argument, as described and proved in Section 4.2. Exploding gradients are instead counteracted, large batches are considered when using Adam or SGD optimizers, as described in Section 3 and in the example Example 16 that compares different gradient descent algorithms. Alternatively, according to several authors [Wil92; Pér+03; Hay04], \mathcal{RNN} training can highly benefit from Kalman filter application (extended versions), especially for poor datasets, although this strategy is particularly cumbersome. Second order optimization schemes, as described in Section 4.2, such as Newton's methods, are definitely more robust, but still cumbersome, due to the need of computing the Hessian matrix, which in case of a \mathcal{RNN} is prohibitive, due to the time-recursive nature of its formulation. Hessian-free methods, applied to large mini-batches, are rather appealing alternatives to train \mathcal{RNN} , especially for non-convex loss function, avoiding the computation of prohibitive Hessian matrices encompassing several time steps, but approximating them to leverage local quadratic approximation in high dimensionality. Another widely used approach to train \mathcal{RNN} is the family of Genetic Algorithms. For deeper insights, a very complete summary of \mathcal{RNN} training techniques can be found in [Sal+18].

Surprisingly, intertwining \mathcal{MLP} and \mathcal{RNN} in deep architectures make the time-forward \mathcal{NN} prediction way more efficient and it reduces the complexity of the training scheme. The cumbersome internal computation that encompasses long-term dependencies can effectively be replaced by shallow feed-forward computations that increase the abstraction level of the input \boldsymbol{x} [t] time history, with intermediate latent representations $\boldsymbol{\phi}(\boldsymbol{x})$, as outlined in Equation (12). In this way, temporal long- and short-time dependencies are better disentangled and the manifolds near which the data concentrate flattened accordingly. \mathcal{RNN} can be combined with \mathcal{RNN} and \mathcal{MLP} according to several combinations, briefly stated in the following paragraphs [Sal+18]:

1. Deep input-to-hidden

This configuration corresponds to the \mathcal{RNN} standard formulation in Equation (36), that is *de facto* the combination of two $\mathcal{MLPs} \mathbf{h}_{FF}^{(h)}$ and $\mathbf{h}_{FF}^{(o)}$ respectively are fed to the hidden and output states. $\boldsymbol{x}[t]$ is fed to the hidden \mathcal{MLP} that performs the well known following operation:

$$oldsymbol{h}_{FF}^{(h)}\left(oldsymbol{x}\left[t
ight],oldsymbol{h}_{t-1}
ight) = = oldsymbol{g}^{(h)}\left(oldsymbol{a}_{t}
ight) = oldsymbol{g}^{(h)}\left(oldsymbol{W}^{(h)}oldsymbol{h}_{t-1} + oldsymbol{W}^{(x)}oldsymbol{x}\left[t
ight] + oldsymbol{b}^{(h)}
ight)$$

Then, the \mathcal{RNN} output is obtained by feeding the output of a second \mathcal{MLP}

$$\boldsymbol{h}_{FF}^{(o)}\left(\boldsymbol{h}_{t}\right) = \boldsymbol{g}^{(o)}\left(\boldsymbol{a}_{t}^{(o)}\left(\boldsymbol{h}_{FF}^{(h)}\left(\boldsymbol{x}\left[t\right]\right)\right)\right) =$$
$$= \boldsymbol{g}^{(o)}\left(\boldsymbol{W}^{(o)}\boldsymbol{h}_{FF}^{(h)}\left(\boldsymbol{x}\left[t\right],\boldsymbol{h}_{t-1}\right) + \boldsymbol{b}^{(o)}\right)$$

To better capture long-term dependencies, the input can be fed to the latter operation too as follows:

$$\boldsymbol{h}_{FF}^{(o)}\left(\boldsymbol{h}_{t},\boldsymbol{x}\left[t\right]\right) = \boldsymbol{g}^{(o)}\left(\boldsymbol{W}^{(o)}\boldsymbol{h}_{FF}^{(h)}\left(\boldsymbol{x}\left[t\right],\boldsymbol{h}_{t-1}\right) + \boldsymbol{b}^{(o)} + \boldsymbol{W}^{(ox)}\boldsymbol{x}\left[t\right] + \boldsymbol{W}^{(oh)}\boldsymbol{h}_{t-1}\right)$$

$$(49)$$

and as shown schematically in Figure 28.



Figure 28: Sketch of the \mathcal{RNN} version with input-to-hidden connection. $h_t^{(1)}$ and $h_t^{(2)}$ represent the two stacked hidden state variables evolving in time and connected to each other. x_t indicates the input time-series, h_t the hidden state and y_t the predicted out time-series.

2. Deep hidden-to-hidden hidden-to-output

 \mathcal{MLP} can enhance the level of abstraction of the data time-history $\boldsymbol{x}[t]$. leveraging its regularity, with \mathcal{RNN} quickly adjusting to short-time fast changes (for non-stationary signals especially) keeping a good memory

of past events. Further \mathcal{MLP} can be added between hidden and hidden variables, as well as between hidden and output. In the latter case, the \mathcal{MLP} raise the abstraction of the hidden state h_t , facilitating the linearization of the decision boundary in its abstract representation and predicting the target in a more robust way. Figure 29 show an example of this architecture.



Figure 29: Sketch of the \mathcal{RNN} version with hidden-to-hidden connection. x_t indicates the input time-series, h_t the hidden state and y_t the predicted out time-series.

As byproduct, the hidden state are better summarizing the contribution of the previous inputs [Sal+18].

3. Stack of hidden states

Deep \mathcal{RNN} are easily constructed by stacking recurrent layers, with the effect that each layer is steered to act on different time scales, since each layer receives as input an abstract representation of the original time-history $\boldsymbol{x}[t]$. However, in practice, the transitions between consecutive hidden states is generally limited to a few layers, because deep \mathcal{RNN} tend to be even more prone to vanishing and/or exploding gradient occurrence (as described above), failing into capturing long-term time dependency. Moreover, the overall computational burden can easily become prohibitive because the gradient descent must unravel intricate time-dependency across several layers, with major difficulty in splitting the computation over a parallel architecture (in *model parallel* mode).

2.8.1 Bidirectional \mathcal{RNN}

l

In some context, future time-steps $\boldsymbol{x} [t + \tau]$ need to be accounted for, in a acausal framework. Standard \mathcal{RNN} formulated in Equation (36) could leverage future time-steps by delaying the output of a few steps, but this strategy revealed itself poorly effective. Therefore, *bidirectional* \mathcal{RNN} (or \mathcal{BRNN}) were introduced by [SP97], that processes the whole time history back and forth in order to predict the final output. In doing so, two standard \mathcal{RNN} are adopted: one that process $\boldsymbol{x} [t]$ (forward) and the other that processes $\boldsymbol{x} [N_t - t]$ (backward), without reciprocal input/output connection, and hidden states that read [Sal+18]:

$$\left(\overrightarrow{\boldsymbol{h}}_{t} = \boldsymbol{g}^{(h)} \left(\overrightarrow{\boldsymbol{W}}^{(h)} \overrightarrow{\boldsymbol{h}}_{t-1} + \overrightarrow{\boldsymbol{W}}^{(x)} \boldsymbol{x} \left[t \right] + \overrightarrow{\boldsymbol{b}}^{(h)} \right)$$
(50)

$$\begin{cases} \overrightarrow{\mathbf{h}}_{t} = \mathbf{g}^{(h)} \left(\overleftarrow{\mathbf{W}}^{(h)} \overleftarrow{\mathbf{h}}_{t} + \overleftarrow{\mathbf{W}}^{(x)} \mathbf{x} \left[N_{t} - t \right] + \overleftarrow{\mathbf{b}}^{(h)} \right) \tag{51}$$

The final output is defined as a linear combination of the two:

$$\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}\left[t\right]\right) = \boldsymbol{g}^{(o)}\left(\overrightarrow{\boldsymbol{W}}^{(o)}\overrightarrow{\boldsymbol{h}}_{t} + \overleftarrow{\boldsymbol{W}}^{(o)}\overleftarrow{\boldsymbol{h}}_{t} + \boldsymbol{b}^{(o)}\right)$$
(53)

 \mathcal{BRNN} (whose architecture is sketched in Figure 30) is slightly more complicated because the update of state and output neurons can no longer be conducted one at a time [SP97]. Another shortcoming is the fact that the time-history must be known from start to end.

Example 7. Design a \mathcal{RNN} with PyTorch

PyTorch has a proper class definition for standard \mathcal{RNN} , offering the possibility of adding bi-directionality and stacked layers, as the following code snippet shows.

```
import torch
 1
         # assure reproductibility
 2
         torch.manual_seed(0)
 3
        design a 2 layer RNN with 50 hidden states (the size of h_t)

d_X = 1 \text{ # dimension of the input time history <math>x[t] (in R^1)

N_1 = 2 \text{ # number of hidden layers
 4
 5
 6
        d_h = 50 \# dimension of h_t (in R^50)
gativation = 'tanh' # activation function
 7
 8
        bidirectional =False # switch to 'True' for designing a BRNN
bias = True # switch to False if no bias is foreseen
 9
10
        h_theta = torch.nn.RNN(input_size = d_X,
11
                                            hidden_size = d_h,
12
                                             num_layers = N_1,
13
                                             nonlinearity=gactivation,
14
15
                                             bias=bias,
16
                                             bidirectional=bidirectional)
```



Figure 30: Sketch of the \mathcal{BRNN} version with hidden-to-hidden connections. The left-to-right arrow indicates the weights, biases and operations of the forward \mathcal{RNN} , whereas the right-to-left arrow indicates the operations and weights for the backward \mathcal{RNN} . x_t indicates the input time-series, h_t (forward and backward respectively) the hidden state and y_t the predicted out time-series.

2.8.2 Combining \mathcal{RNN} and \mathcal{CNN}

Recurrent Convolutional Neural Network \mathcal{RCNN} [LH15] leverage the \mathcal{RNN} infinite impulse response, capable of leveraging short- and long-time range memory in the data, with the translational covariance (and equivariance to roto-translation/distortion) of \mathcal{CNN} layers described in Section 2.5 and Section 2.6, capable of tracking time-propagating phenomena regardless their time occurrence. \mathcal{RNN} infinite kernel allows to account for non-local context in the signal, facilitating the classification or regression task to connected \mathcal{CNN} , whose aim is to identify the labeling function regardless the time-translation (or spatial translation in an image). \mathcal{RNN} recurrent connections can be intertwined to each convolutional layer in a deep \mathcal{CNN} , to enable it with a memory of past information. In this sense, \mathcal{RNN} help stabilizing the classification performance of a deep \mathcal{CNN} . Alternatively, a \mathcal{RNN} layer can be stacked at the top of a \mathcal{CNN} , whose role is to extract common low-level features from the spatio-temporal phenomenon. The \mathcal{RNN} is fed with the \mathcal{CNN} feature maps, whose high level regularity of abstract representation simplify the \mathcal{RNN} prediction, in a deep input-to-hidden configuration (but replacing the \mathcal{MLP} with a deep \mathcal{CNN}).

2.9 Long-Short Term Memory

Hidden variables are cross and delight of standard \mathcal{RNN} , since they are irremediably necessary in order to capture long range time dependencies and multiscale phenomena. However, the long-range dynamics of those hidden features make \mathcal{RNN} very hard to train, since the gradient of the loss function flows backward to the initial time step. Surprisingly, those hidden variables trigger vanishing and exploding gradient events, which make but from a computational point of view, the required memory becomes easily too large to handle. Moreover, despite the possibility of intertwining \mathcal{MLP} and \mathcal{CNN} with recurrent connections in order to simplify the prediction phase thanks to intermediate abstract representation of the input sequence $\boldsymbol{x}[t]$, these hybrid $\mathcal{N}\mathcal{N}$ are still hard to define and to tune in order to process short and long-term memory features separately. \mathcal{RNN} do not include any explicit notion of time window, which would be somehow useful for time series prediction/classification. The most popular alternative to \mathcal{RNN} are Long Short-Term Memory recurrent neural network, generically known under the acronym \mathcal{LSTM} [HS97]. The objective of their invention was to introduce a scheme that could improve learning long-range dependencies, by learning to forget [GSC00]. As a matter of fact, the main \mathcal{LSTM} difference is the fact that the \mathcal{LSTM} output is stored in extra "memory cell", with input and output connections controlled by gates g that steer the flow of information from previous time steps (and the gradient back-propagation), as sketched in Figure 31. In particular, the forget gate $g_t^{(f)}$ learns weights that control the rate at which the value stored in the memory cell decays [GSC00].

A standard \mathcal{LSTM} is made of 3 gates: input $\boldsymbol{g}_{t}^{(i)}$, forget $\boldsymbol{g}_{t}^{(f)}$, output $\boldsymbol{g}_{t}^{(o)}$ and a memory cell \boldsymbol{c}_{t} . Gates control the update of the memory cell, preventing its modification for multiple time-steps. This allows \mathcal{LSTM} to train for longer iterations than standard \mathcal{RNN} as well as to capture complex time-scale interdependencies. The equations that rule the standard \mathcal{LSTM} components can be summarized in the following system:

$$\boldsymbol{z}_{t}^{(i)} = \boldsymbol{g}^{(i)} \left(\boldsymbol{W}^{(ii)} \boldsymbol{x} \left[t \right] + \boldsymbol{W}^{(ih)} \boldsymbol{h}_{t-1} + \boldsymbol{W}^{(ic)} \boldsymbol{g}_{t}^{(c)} + \boldsymbol{b}^{(i)} \right)$$
(54)

$$\boldsymbol{z}_{t}^{(f)} = \boldsymbol{g}^{(f)} \left(\boldsymbol{W}^{(fi)} \boldsymbol{x} \left[t \right] + \boldsymbol{W}^{(fh)} \boldsymbol{h}_{t-1} + \boldsymbol{W}^{(fc)} \boldsymbol{g}_{t}^{(c)} + \boldsymbol{b}^{(f)} \right)$$
(55)

$$\boldsymbol{c}_{t} = \sum_{j} z_{t,j}(i) \cdot tanh\left(\left\langle \boldsymbol{w}_{j}^{(ci)}, \boldsymbol{x}\left[t\right]\right\rangle + \left\langle \boldsymbol{w}_{j}^{(ch)}, \boldsymbol{h}_{t-1}\right\rangle + b_{j}^{(c)}\right)\boldsymbol{e}_{j} +$$
(56)

$$+\sum_{j} z_{t,j}^{(f)} \cdot z_{t-1,j}(c) \boldsymbol{e}_{j}$$
(57)

$$\boldsymbol{z}_{t}^{(o)} = \boldsymbol{g}_{t}^{(o)} \left(\boldsymbol{W}^{(oi)} \boldsymbol{x} \left[t \right] + \boldsymbol{W}^{(oh)} \boldsymbol{h}_{t-1} + \boldsymbol{W}^{(oc)} \boldsymbol{g}_{t}^{(c)} + \boldsymbol{b}^{(o)} \right)$$
(58)

$$\boldsymbol{h}_{t} = \sum_{j} \boldsymbol{z}_{t,j}^{(o)} \cdot tanh\left(\boldsymbol{z}_{t,j}^{(c)}\right) \boldsymbol{e}_{j}$$
(59)

with:

- $\boldsymbol{W}^{(ii)},~\boldsymbol{W}^{(ih)},~\boldsymbol{W}^{(ic)}$ and $\boldsymbol{b}^{(i)}$ being the weight matrices and the bias



Figure 31: Sketch if the standard \mathcal{LSTM} architecture.

vectors respectively for the input gate, that is represented by a sigmoid activation function applied to each component of the pre-activation, $\boldsymbol{g}_{t}^{(i)}(\boldsymbol{a}) = \sum_{j} \sigma(\boldsymbol{a}_{j}) \boldsymbol{e}_{j};$

- $W^{(fi)}$, $W^{(fh)}$, $W^{(fc)}$ and $b^{(f)}$ being the weight matrices and the bias vectors respectively for the forget gate, that is represented by a sigmoid activation function applied to each component of the pre-activation, $g_t^{(f)}(a) = \sum_j \sigma(a_j) e_j$ as per the input gate;
- $\boldsymbol{w}_{j}^{(ci)}$, $\boldsymbol{w}_{j}^{(ch)}$, and $b_{j}^{(f)}$ being the weight matrix j^{th} row and bias vector component respectively for the update of the memory cell \boldsymbol{c}_{t} , that is based on the values of the previously computed input and forget gated activations $\boldsymbol{z}_{t}^{(i)}$ and $\boldsymbol{z}_{t}^{(f)}$ respectively;
- $W^{(oi)}$, $W^{(oh)}$, $W^{(oc)}$ and $b^{(o)}$ being the weight matrices and the bias vectors respectively for the output gate, that is represented by a sigmoid activation function applied to each component of the pre-activation,

$$\boldsymbol{g}_t^{(o)}(\boldsymbol{a}) = \sum_j \sigma(\boldsymbol{a}_j) \boldsymbol{e}_j$$
 as per the input and forget gate;

According to [LJH15], Equation (59) explains how the memory cell works. For time-steps t for which input and output gates are deactivated by the sigmoid (negative argument) and for which the forget gate does not apply a higher decay (large positive values of each $z_{t,j}^{(f)}$, that make the sigmoid saturate to 1) the memory cell c_t holds its value $z_{t-1}^{(c)}$, providing a non-vanishing gradient term, which remains constant over those periods. This architecture allows the network to potentially remember information for longer periods, depending on how the forget gates activates. The latter being close to 0 (negative argument of the forget sigmoid), makes the term $\sum_j z_{t,j}^{(f)} \cdot z_{t-1,j}(c) e_j \approx 0$, which means that the network "forgets" the previously store memory cell value. The saturation of the sigmoid activations at input, forget and output gates limit the unbounded growth of the internal variables but this aspect can lead to information loss because there is no active selection of which long-term dependency is relevant or not. Moreover, \mathcal{LSTM} has generally four times the number of parameters of a standard \mathcal{RNN} [Sal+18].

Deep $\mathcal{N}\mathcal{N}$ can be constructed by stacking \mathcal{LSTM} hidden layers upon each other, each layer activation defined as:

$$\int \boldsymbol{h}_{t}^{(\ell)} = \boldsymbol{g}^{(\ell)} \left(\boldsymbol{W}^{(ih)} \boldsymbol{h}_{t}^{(\ell-1)} + \boldsymbol{W}^{(hh)} \boldsymbol{h}_{t-1}^{(\ell)} + \boldsymbol{b}_{h}^{(\ell)} \right), \quad \forall 1 \le \ell \le N_{\ell}$$
(60)

$$\boldsymbol{h}_{\boldsymbol{\theta}} = \boldsymbol{g}^{(o)} \left(\boldsymbol{W}^{(o)} \boldsymbol{h}_{t}^{(N_{\ell})} + \boldsymbol{b}^{(o)} \right)$$
(61)

Analogously to bidirection \mathcal{RNN} , bidirectional \mathcal{LSTM} have been conceived too for language understanding [GMH13; WJ16]. For a complete summary of all possible \mathcal{LSTM} variants developed in the literature, refer to [Sal+18].

Example 8. Design a \mathcal{LSTM} with PyTorch

PyTorch has a proper class definition for standard \mathcal{LSTM} , offering the possibility of adding bi-directionality and stacked layers, as the following code snippet shows.

```
import torch
 1
 2
         assure reproductibility
       torch.manual_seed(0)
 3
       # design a 2 layer LSTM with 50 hidden states (the size of h_t)
 4
      d\_X = 1 # dimension of the input time history x[t] (in R^1) N\_1 = 2 # number of hidden layers
 5
 6
       \mathbf{d}_{\mathbf{h}} = 50 \ \# \ dimension \ of \ h_t \ (in \ R^{50})
 7
       gactivation = 'tanh' # activation function
bidirectional =False # switch to 'True' for designing a BRNN
 8
 9
       bias = True # switch to False if no bias is foreseen
10
11
       h_theta = torch.nn.LSTM(input_size = d_X,
                                     hidden_size = d h.
12
                                     num_layers = N_1,
13
                                     nonlinearity=gactivation,
14
                                      bias=bias.
15
                                     bidirectional=bidirectional)
16
```

Further examples on the use of \mathcal{RNN} and \mathcal{LSTM} will be provided in the dedicated hands-on tutorials.

3 Optimizing a Neural Network

As stated in Section 2, Neural Networks \mathcal{NN} are a special type of statistical models \mathcal{H}_{θ} , made of complex functions \mathbf{h}_{θ} defined as composition of non linear ridge activations applied to affine transformations. The composition pattern follows intricate graphs. The search of the "best" set of weights $\boldsymbol{\theta} \in \Theta$ is performed by attempting at minimizing the empirical loss function $L_{\mathcal{D}_{XY}}(\mathbf{h}_{\theta})$, defined over a set of i.i.d. samples $\mathcal{D}_{XY} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, with $\mathbf{x}_i \in \mathcal{X} \subset \mathbb{R}^{d_X}$ and $\mathbf{y}_i \in \mathcal{Y} \subset \mathbb{R}^{d_Y}$ (see problem (\mathcal{P})). The quest for the best predictor is a matter of optimization over open sets $\Theta \subset \mathbb{R}^{d_{\Theta}}$, where d_m is the amount of parameters (weights and biases) the \mathcal{NN} is featured of.

Definition 1. Gradient Descent for Empirical Loss Minimization (\mathcal{G}) Given an open subset $\Theta \subset \mathbb{R}^{d_m}$, Proposition 76 is a sufficient and necessary conditions for $\hat{\theta}$ to be a minimizer of the Empirical Loss function $L_{\mathcal{D}_{XY}}$ (unless $L_{\mathcal{D}_{XY}}$ is convex, which is not always the case) and it reads:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{arg\,min}} \quad L_{\mathcal{D}_{XY}}\left(\boldsymbol{h}_{\theta}\right) \tag{62}$$

The quest for $\hat{\theta}$ is iteratively conducted, following the direction of $-\nabla_{\theta} L_{\mathcal{D}_{XY}}$, according to the following update rule from iteration *i* to iteration +i+1 (the so called *delta rule*:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta^{(i)} \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right), \quad \eta^{(i)} \in \mathbb{R}^+$$
(63)

with $\eta^{(i)} \in \mathbb{R}^+$ being the *learning rate*, and

$$L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right) \geq L_{\mathcal{D}_{XY}}\left(\boldsymbol{\hat{\theta}}\right)$$

Remark 2. The solution of problem (\mathcal{P}) is obtained by computing the gradient of the empirical loss function (see Equation (1)), which reads:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}\right) = \frac{1}{N} \sum_{\left(\boldsymbol{x}_{i}, \boldsymbol{y}_{i}\right) \in \mathcal{D}_{XY}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ell\left(\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\right), \boldsymbol{y}_{i}\right)$$
(64)

Some further theoretical details are provided in Appendix B. Moreover, Section 4.1 outlines the theoreteical background that justify the strategies to minimize $L_{\mathcal{D}_{XY}}$ presented in the following.

Remark 3. The cost of computing the exact gradient in Equation (64), by averaging across the whole dataset instances and then update the weights (the so

called Batch Gradient Descent) is $O(N \cdot d_m)$. This computational cost is reasonably affordable for rather small dataset and shallow \mathcal{NN} , being tailored only for convex and rather smooth loss functions. However, for very large dataset, the floating-point error induced by computing a sum over a large number of addends introduces spurious errors that can make the gradient descent algorithm to fail.

Remark 4. The standard gradient descent practice employs single-precision floating-point format (i.e., numbers are encoded in a 32-bit-long (4 bytes) binary code) and even half-precision floating-point format (i.e., numbers are encoded in a 16-bit-long (2 bytes) binary code). The choice of employing simpleor even half-precision floating-point format is justified by the fact that gradient descent algorithms are run on Graphic Processing Units (GPU), primarily designed for interactive rendering, not computation. Compared to Central Processing Units (CPUs) GPUs have limited cache memory (where to store data to be repeatedly accessed) but higher sequential memory (where to store data that are naturally structured in a contiguous way, such as in an array). Therefore, a GPU is extremely fast and efficient in computing linear algebra operations on contiguously stored data, that the CPU is in charge of feeding it continuously. However, to reach peak performances, those linear algebra operations (such as the gradient descent one) are performed on single- or even half-precision floating-point format, so to be executed faster. Double-precision floating-point format can hinder the GPU performances, due to its limited memory capacity, which is why it is rarely used. However, as observed in Remark 3, this choice implies possible divergence of the Batch Gradient Descent algorithm on large datasets.

3.1 Stochastic Gradient Descent (SGD)

Inspired by the Fisher's approach for maximizing the log-likelihood function⁷ with unbiased estimator, a suitable minimization strategy would be to pursue the minimum via the delta rule in Equation (63) via successive random gradient updates (see) computed from uniformly sampled database \mathcal{D}_{XY} according to the following algorithm, called Stochastic Gradient Descent (SGD):

In other words, the SGD approximates the gradient in Equation (64) by randomly sampling the all the i.i.d. samples over different epochs (n_e is the epoch's number). $\boldsymbol{\theta}^{(i+1)}$ are random vectors that converge towards $\hat{\boldsymbol{\theta}}$ at a certain speed. SGD effectively reduces the computational burden mentiond in Remark 3, since its complexity is estimated to $O(d_m)$, which makes it an appealing alternative to the classical GD alternative. The choice of the learning rate scheduling is

 $^{^7\}mathrm{or}$ minimizing the negative log-likelihood which belongs to the family of empirical loss functions

Algorithm 1 Stochastic Gradient Descent and Batch Gradient Descent

1: i = 02: Initialize $\boldsymbol{\theta}^{(0)}$ 3: $\eta^{(0)}$ 4: while $i < n_e$ do 5: for $t \sim \mathcal{U}\left(\left\{i\right\}_{i=1}^{N}\right)$ do 6: $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta^{(i)} \nabla_{\boldsymbol{\theta}} \ell\left(\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}^{(i)}\right), \quad \eta^{(i)} \in \mathbb{R}^{+}$ 7: end for 8: end while

usually made in the following way [Pey20]:

$$\eta^{(i)} = \frac{\eta^{(0)}}{1 + \frac{i}{i_0}}, \quad i_0 > 0 \tag{65}$$

or, more generally:

$$\eta^{(i)} = \frac{\eta^{(0)}}{1 + i\eta_d} \tag{66}$$

The learning rate update scheme in Equation (65) ensures that $\eta^{(i)} \leq \eta^{(0)}$ and, after a "warm up" phase, it decreases as the epochs go by. The value of η_d represents the so called learning rate decay. However, other scheduling strategies are available, such as Cyclic Learning Rate Scheduler [Smi17]. The latter suggests to first calibrate the lower and upper bounds η_{\min} and η_{\max} respectively (if not known before hand, such as shown in Section 4.3), and then reiterates several loops such as shown in Section 3.1. [Smi17] seemingly suggests that 1 cycle could be enough.

Within the framework of the SGD algorithm, the delta rule in Equation (63) is usually applied by averaging the randomly sampled gradients on "mini-batches" of N_b instances each, i.e., adopting the following algorithm: Averaging on minibatches is certainly cost-effective compared to standard SGD, that corresponds to N mini-batches of $N_b = 1$ instance each. SGD on mini-batches can be seen as a vectorized version of the SGD.

Example 9. Stochastic Gradient Descent for linear regression

```
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import numpy as np
    from matplotlib import pyplot as plt
    # assure reproductibility
    torch.manual_seed(0)
```

```
8 # Data for regression problem
```

```
9 A = 10.1542550
```



Figure 32: Example of Cyclic Learning Rate Scheduler [Smi17].

```
b = 14.7351129
c = 2.34
10
11
12
        epsilon = 10.154988
        N = 200 # number i.i.d. samples
13
        Nb = 10 # number of samples in the mini-batch
14
        # Data Generation
15
        np.random.seed(42)
16
        y = np.random.rand(N, 1)
y = b + A * np.exp(c*x) + epsilon * np.random.randn(N, 1)
17
^{18}
19
        # Shuffled indices
idx = np.arange(N)
20
21
        np.random.shuffle(idx)
22
23
        Ntrain = int(N*0.8)
# Uses first 80% of the dataset for training
train_idx = idx[:Ntrain]
# Uses the last 20% for validation
val_idx = idx[Ntrain:]
^{24}
25
26
27
28
29
        # Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
30
^{31}
^{32}
33
        # Convert Numpy arrays into PyTorch Tensors
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()
34
35
36
37
38
        # Plot the dataset
        # plt.scatter(x,y,color='black',label=r"$D_{XZ}$")
# plt.scatter(x[train_idx],
#
39
40
                            y[train_idx],
color='red',
        #
41
42
^{43}
                               marker="+",
        #
                              label="train dataset")
^{44}
        #
        # plt.scatter(x[val_idx],
45
                              y[val_idx],
46
        #
```

Algorithm 2 Stochastic Gradient Descent on mini-batches

1: i = 02: Initialize $\boldsymbol{\theta}^{(0)}$ 3: $\eta^{(0)} = \eta_0$ 4: while $i < n_e$ do 5: for $j \sim \mathcal{U}\left(\{i\}_{i=1}^{\frac{N}{N_b}}\right)$ do 6: $\boldsymbol{g}_{\theta} = 0$ 7: for $t \sim \mathcal{U}\left(\{j_i\}_{i=1}^{N_b}\right)$ do 8: $\boldsymbol{g}_{\theta} + = \boldsymbol{\nabla}_{\theta}\ell\left(\boldsymbol{h}_{\theta}\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}^{(i)}\right)$ 9: end for 10: $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \frac{\eta^{(i)}}{N_b}\boldsymbol{g}_{\theta}, \quad \eta^{(i)} \in \mathbb{R}^+$ 11: end for 12: end while

```
color='blue',
47
       #
                          marker="x"
^{48}
       #
                          label="validation dataset")
49
       # plt.xlabel(r"$x$")
50
51
       # plt.ylabel(r"$y$")
52
       # plt.legend(frameon=False)
53
       eta = 1e-1 # constant learning rate
# for warmup: https://github.com/Tony-Y/pytorch_warmup
54
55
       n_e = 1000 # number of epochs
56
57
58
        # Creating a MLP model with 1 layer and 1 neuron
59
       h_theta = nn.Sequential(nn.Linear(1, 1), nn.SiLU())
60
       # Define the SGD optimizer
optimizer = optim.SGD(h_theta.parameters(), lr=eta)
61
62
        L_Dxy = nn.MSELoss()
63
64
65
        # Loop over the training epochs on training dataset
66
       i=0
       while i < n_e: # loop over epochs
67
68
         for j in range(0, int(Ntrain/Nb)): # loop over mini-batches
            t_idx = np.arange((*Nb.(j+1)*Nb.1) # mini-batch linear indexing
np.random.shuffle(t_idx) # shuffle mini-batch
69
70
             x_b = x_train_tensor[t_idx]
71
72
             y_b = y_{train_tensor[t_idx]}
             optimizer.zero_grad() # initialize the optimizer
loss = 0.0 # initialize the loss
73
74
             for t in range(Nb): # loop over the samples in the batch
75
               xt = x_b[t]
76
77
               yt = y_b[t]
            yt= y_bt()
yhat = h_theta(xt) # predict the output
loss += L_Dxy(yhat, yt) # Compute Empirical Loss
loss.backward() # Compute derivatives with AutoGrad
optimizer.step() # weight update!
78
79
80
81
```
3.2 Beyond SGD: the role of "momentum"

The SGD in Algorithm 1 and its mini-batch version in Algorithm 2 are firstorder Hessian-free methods that converges slowly and it is sensitive to noise [Sut+13; Pey20]. Moreover, according to Remark 34, the convergence can be hindered when the Hessian is ill-conditioned. A possible improvement is represented by the so called "momentum", using previous gradient memory. The Classical Momentum technique [Pol64] accelerates the gradient descent, with a momentum coefficient $\gamma \in [0, 1)$] that cumulates a velocity vector \boldsymbol{v} (or gradient memory) in directions of persistent reduction in $L_{\mathcal{D}_{XY}}$ across iterations [Sut+13]. Algorithms 3 and 4 outline the SGD algorithm with Classical Momentum, without and with mini-batch average.

Algorithm 3 Stochastic Gradient Descent with Classical Momentum

1: i = 02: Initialize $\theta^{(0)}$ 3: $\eta^{(0)} = \eta_0$ 4: $v^{(0)} = 0$ 5: while $i < n_e$ do 6: for $t \sim \mathcal{U}\left(\{i\}_{i=1}^N\right)$ do 7: $v^{(i+1)} = \eta^{(i)}\left(1 - \langle \tau^{(i)} - 1 \rangle\right) \nabla_{\theta} \ell\left(h_{\theta}\left(x_i\right), y^{(t)}; \theta^{(i)}\right) + \gamma^{(i)} v^{(i)}, \quad \eta^{(i)} \in \mathbb{R}^+, \gamma^{(i)} \in [0, 1)$ 8: $\theta^{(i+1)} = \theta^{(i)} - v^{(i+1)}$ 9: end for 10: end while

The factor $-\langle \tau^{(i)} - 1 \rangle$ dampens the gradient only if $\tau^{(i)} > 1^8$. When no dampening is considered (i.e. $\tau^{(i)} \leq 1$), the Classical Momentum accelerates by a (usually) constant factor γ (often chosen equal to 0.85 or 0.95) the gradient descent in the directions \boldsymbol{e}_{ξ} is small but persistent. For instance, in the minibatch version in Algorithm 4, if $\langle \boldsymbol{g}_{\theta}, \boldsymbol{e}_{\xi} \rangle > 0$

$$\left\langle \boldsymbol{e}_{\xi}, \boldsymbol{v}^{(i)} \right\rangle = \eta^{(i)} \left\langle \boldsymbol{e}_{\xi}, \boldsymbol{g}_{\theta} \right\rangle + \gamma^{(i)} \left\langle \boldsymbol{e}_{\xi}, \boldsymbol{v}^{(i-1)} \right\rangle \ge \gamma^{(i)} \left\langle \boldsymbol{e}_{\xi}, \boldsymbol{v}^{(i-1)} \right\rangle$$
(67)

The inertia allows to overcome local minima and saddle points but, on the contrary, its effect vanishes along the directions of small or oscillating reduction in the objective loss across iterations are amplified by Algorithm 3. For convex functions, momentum-based methods are known to outperform SGD in the early or transient stages [Sut+13]. [Pol64] proved that the Classical Momentum algorithm accelerates convergence to local minima by reducing by a factor $\sqrt{\kappa \left(\mathbf{H}_{L_{\mathcal{D}_{XY}}}\right)}$ the number of iterations required by greedy algorithms

 $^{^{8}\}langle \cdot \rangle$ represent the Macaulay brackets.

Algorithm 4 Stochastic Gradient Descent on mini-batches with Classical Momentum

1: i = 02: Initialize $\boldsymbol{\theta}^{(0)}$ 3: $\eta^{(0)} = \eta_0$ 4: $v^{(0)} = 0$ 5: while $i < n_e$ do for $j \sim \mathcal{U}\left(\left\{i\right\}_{i=1}^{\frac{N}{N_b}}\right)$ do $\boldsymbol{g}_{\theta} = 0$ for $t \sim \mathcal{U}\left(\left\{j_i\right\}_{i=1}^{N_b}\right)$ do 6: 7: 8: $egin{aligned} egin{aligned} egin{aligne} egin{aligned} egin{aligned} egin{aligned} egin$ 9: end for 10: $\begin{array}{l} \boldsymbol{v}^{(i+1)} = \boldsymbol{\eta}^{(i)} \left(1 - \langle \boldsymbol{\tau}^{(i)} - 1 \rangle \right) \boldsymbol{g}_{\boldsymbol{\theta}} + \boldsymbol{\gamma}^{(i)} \boldsymbol{v}^{(i)}, \quad \boldsymbol{\eta}^{(i)} \in \mathbb{R}^+, \boldsymbol{\gamma}^{(i)} \in [0, 1) \\ \boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \frac{\boldsymbol{\eta}^{(i)}}{N_b} \boldsymbol{v}^{(i+1)}, \quad \boldsymbol{\eta}^{(i)} \in \mathbb{R}^+ \end{array}$ 11: 12:end for 13:14: end while

such as the one in Equation (152). In this case, γ must be chosen equal to $\sqrt{\kappa(\mathbf{H}_{L-1})} = 1$

 $\frac{\sqrt{\kappa(\mathbf{H}_{L_{\mathcal{D}_{XY}}})}^{-1}}{\sqrt{\kappa(\mathbf{H}_{L_{\mathcal{D}_{XY}}})}^{+1}}, \text{ similar to the expression in Equation (172). Second-order}$

methods, as seen in Section 4.2, amplify those steps in low-curvature directions , but instead of accumulating changes they reweight the update along each eigen-direction of the curvature matrix by the inverse of the associated curvature [Sut+13]. On the contrary, Classical Momentum algorithms can miss the minimizer, due to the consistent inertia of the gradient descent.

In ${\tt PyTorch},$ the Classical Momentum is an option of the standard ${\tt SGD}$ optimizer.

Example 10. Stochastic Gradient Descent for linear regression with Classical Momentum

This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
2 # Define the SGD optimizer
3 gamma = 0.5/eta # Classical Momentum algorithm
4 # PyTorch adopts a Classical Momentum coefficient eta*mu
5 # which corresponds to gamma in the above mentioned formulas
6 tau = 0.0 # Dampening coefficient
7 optimizer = optim.SGD(h_theta.parameters(), lr=eta,
8 momentum=gamma, dampening=tau)
9 ...
```

3.3 Beyond Classical Momentum SGD: the Nesterov algorithm

A better convergence rate than the Classical Momentum algorithm is provided by the Nesterov algorithm, especially for general smooth (non-strongly) convex functions and a deterministic gradient descent [Sut+13]. The difference with the Classical Momentum method resides in the fact that the gradient of the loss function is computed at the position updated with $\boldsymbol{v}^{(i)}$. The Nesterov Accelerated Gradient algorithm reads: Usually, $\gamma^{(i)}=0.9$. In PyTorch, the Nesterov

Algorithm 5 Nesterov Accelerated Gradient Descent

1: i = 02: Initialize $\theta^{(0)}$ 3: $\eta^{(0)} = \eta_0$ 4: $v^{(0)} = 0$ 5: while $i < n_e$ do 6: for $t \sim \mathcal{U}\left(\{i\}_{i=1}^N\right)$ do 7: $v^{(i+1)} = \eta^{(i)}\left(1 - \langle \tau^{(i)} - 1 \rangle\right) \nabla_{\theta} \ell\left(h_{\theta}\left(x_i\right), y^{(t)}; \theta^{(i)} - \gamma^{(i)}v_i\right) + \gamma^{(i)}v^{(i)}, \eta^{(i)} \in \mathbb{R}^+, \gamma^{(i)} \in [0, 1)$ 8: $\theta^{(i+1)} = \theta^{(i+1)} - v^{(i)}$ 9: end for 10: end while

Algorithm 6 Nesterov Accelerated Gradient Descent on mini-batches

1: i = 02: Initialize $\boldsymbol{\theta}^{(0)}$ 3: $\eta^{(0)} = \eta_0$ 4: $v^{(0)} = 0$ $\begin{aligned} & \text{for } i < n_e \text{ do} \\ & \text{for } j \sim \mathcal{U}\left(\left\{i\right\}_{i=1}^{\frac{N}{N_b}}\right) \text{ do} \\ & \boldsymbol{g}_{\theta} = 0 \\ & \text{for } t \sim \mathcal{U}\left(\left\{j_i\right\}_{i=1}^{N_b}\right) \text{ do} \\ & \boldsymbol{g}_{\theta} + = \boldsymbol{\nabla}_{\theta} \ell\left(\boldsymbol{h}_{\theta}\left(\boldsymbol{x}_i\right), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}_i - \gamma^{(i)} \boldsymbol{v}^{(i)}\right) \\ & \text{ end for } \\ & \boldsymbol{v}^{(i+1)} - \gamma^{(i)} \mathcal{U} = \mathcal{U}_{\theta}^{(i)} \end{aligned}$ 5: while $i < n_e$ do 6: 7: 8: 9: 10:
$$\begin{split} & \boldsymbol{v}^{(i+1)} = \boldsymbol{\eta}^{(i)} \left(1 - \langle \boldsymbol{\tau}^{(i)} - 1 \rangle \right) \boldsymbol{g}_{\boldsymbol{\theta}} + \boldsymbol{\gamma}^{(i)} \boldsymbol{v}^{(i)}, \quad \boldsymbol{\eta}^{(i)} \in \mathbb{R}^+, \boldsymbol{\gamma}^{(i)} \in [0, 1) \\ & \boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \frac{\boldsymbol{\eta}^{(i)}}{N_b} \boldsymbol{g}_{\boldsymbol{\theta}}, \quad \boldsymbol{\eta}^{(i)} \in \mathbb{R}^+ \end{split}$$
11:12:end for 13:14: end while

Accelerated Gradient algorithm is an option of the standard SGD optimizer.

Example 11. Nesterov Accelerated Gradient Descent for linear regression

This example is equivalent to Example 9, with the exception of the definition of the optimizer.

1

2

3

 4

 $\mathbf{5}$

6 7

3.4 Optimizing with adaptive learning rates

Several strategies have been developed to improve the convergence rate of firstorder Hessian-free gradient descent algorithms. Those algorithms are based on adaptive schemes of the learning rate. In the following, the most popular ones are presented. For the sake of simplicity, we consider a deterministic Gradient Descent algorithm

3.4.1 Adaptive Gradient: AdaGrad

[DHS11] proposed an adaptive update rule called *AdaGrad*. The latter is based on the definition of $\mathbb{G}^{(i)}$, defined as:

$$\mathbb{G}^{(i)} = \sum_{k=0}^{i-1} \nabla_{\theta} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(k)} \right) \otimes \nabla_{\theta} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(k)} \right)$$
(68)

The AdaGrad update rule reads:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \left(\operatorname{diag} \mathbb{G}^{(i)} + \epsilon \mathbb{I} \right)^{-\frac{1}{2}} \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right)$$
(69)

with usual values of $\eta = 10^{-2}$ and a numeric tolerance $\epsilon = 10^{-8}$. The diagonal matrix diag(\mathbb{G}) represents the cumulated square derivatives along θ_j , $G_j^{(i)} = \sum_{k=0}^{i-1} \left(\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_j} \left(\boldsymbol{\theta}^{(k)}\right)\right)^2$ from epoch 0 to epoch *i*. Moreover, $\sqrt{\mathrm{Tr}\left(\mathbb{G}^{(i)}\right)}$ represents the ℓ_2 norm of $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_j}$, but with each gradient component associated to one iteration. For $i \gg$, $\frac{G_j^{(i)}}{i-1}$ is an unbiased estimator of the variance $\mathbb{E}\left[\left(\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_j} \left(\boldsymbol{\theta}_i\right)\right)^2\right]$ and it is monotonically increasing. AdaGrad requires to save and update diag $\mathbb{G}^{(i)}$ at each iteration. The effective learning rate of each gradient component is expressed as $\eta'_j = \frac{\eta}{G_j^{(i)} + \epsilon}$. η'_j can become very small, due to the monotonic increase of $G_j^{(i)}$, until the update stops. This aggressive

learning rate decay represents one major shortcoming of the AdaGrad. In order to mitigate such effect, AdaGrad update can be combined with the adaptive learning rate decay in Equation (66).

Example 12. AdaGrad for linear regression

2

5

8

2

4

 $\mathbf{5}$

6

7 8 This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
# Define the AdaGrad optimizer
etad = 0.0 # learning rate decay
3
     epsilon = 1e-8 # tolerance
4
     GO = 0.0
     optimizer = optim.Adagrad(h_theta.parameters(), lr=eta,
6
                                  lr_decay=etad, eps=epsilon, initial_accumulator_value=G0)
```

Root Mean Square Propagation: RMSprop 3.4.2

Another optimization strategy, capable of mitigating the disadvantage of aggressive learning rate decay, proper to AdaGrad, was proposed by [TH12] and called *RMSprop*. The idea is to compute an average value of $\mathbb{G}^{(i)}$, based on a shorter memory window, i.e.:

$$\operatorname{diag}\tilde{\mathbb{G}}^{(i)} = \alpha_r \operatorname{diag}\mathbb{G}^{(i-1)} + (1 - \alpha_r) \left(\operatorname{diag}\mathbb{G}^{(i)} - \operatorname{diag}\mathbb{G}^{(i-1)}\right)$$
(70)

The update rule is the same as Equation (69), but with diag $\tilde{\mathbb{G}}^{(i)}$ to replace diag $\mathbb{G}^{(i)}$ and usually the smoothing constant $\alpha_r=0.9$. The main difference is represented by the fact that $\operatorname{diag} \tilde{\mathbb{G}}^{(i)}$ is not monotonically increasing, therefore the learning rate decay is slower.

Example 13. RMSprop for linear regression

This example is equivalent to Example 9, with the exception of the definition of the optimizer.

```
# Define the RMSprop optimizer
alphar = 0.9 # smoothing constant (called alpha in PyTorch)
3
      epsilon = 1e-8 # tolerance
      GO = 0.0
      optimizer = optim.RMSprop(h_theta.parameters(), lr=eta,
                                  alpha=alphar, eps=epsilon)
```

3.4.3 **RMSprop with momentum: ADADELTA**

The RMSprop with momentum (see Section 3.2) is called ADADELTA [Zei12]. In particular, after having computed $\tilde{\mathbb{G}}^{(i)}$ defined in Equation (70), ADADELTA

computes the vector $U^{(i+1)}$ with momentum, i.e., as follows:

$$\boldsymbol{U}^{(i+1)} = \alpha_u \boldsymbol{U}^{(i)} + (1 - \alpha_u) \boldsymbol{u}^{(i)}$$
(71)

with $\boldsymbol{u}^{(i)}$ that slightly modifies the gradient with a term approximating the inverse Hessian matrix (diagonal), but based on previous gradient update, according to the expression:

$$\boldsymbol{u}^{(i+1)} = \operatorname{diag}^{\frac{1}{2}} \left(\mathbb{U}^{(i)} + \epsilon \mathbb{I} \right) \operatorname{diag}^{-\frac{1}{2}} \left(\tilde{\mathbb{G}}^{(i)} + \epsilon \mathbb{I} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right)$$
(72)

where $\mathbb{U}^{(i)} = \left(\boldsymbol{U}^{(i)} \otimes \boldsymbol{U}^{(i)} \right)^{\frac{1}{2}}$. All the vectors $\boldsymbol{u}^{(i+1)}$ has the same dimensions of $\boldsymbol{\theta}$. The term diag $\frac{1}{2} \left(\mathbb{U}^{(i)} + \epsilon \mathbb{I} \right)$ represents the inverse Hessian approximation (see Equation (155)), with the contribution of the RMSprop update, whereas the term diag $\frac{-1}{2} \left(\tilde{\mathbb{G}}^{(i)} + \epsilon \mathbb{I} \right)$ the contribution of the RMSprop gradient. Finally, $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \boldsymbol{u}^{(i)}$. The advantage of the ADADELTA is that it is parameter-free, since no learning rate *stricto sensu* is required. This prevents the possibility of a learning rate scheduler. Another ADADELTA's shortcoming is represented by the biased momentum estimate in Equation (71) and Equation (72), strictly depending on the weights' initialization.

Example 14. Adadelta for linear regression

This example is equivalent to Example 9, with the exception of the definition of the optimizer.

2

3

4

5 6

7

3.4.4 Adaptive Moment Estimation: Adam

The Adaptive Moment Estimation, Adam, invented by [KB15], is a rather famous optimizer in deep learning. Adam retrieves the idea of momentum, by estimating the first (mean) and second order (variance) moments. Adam endorse the idea of momentum by keeping a moving average of previous gradients (first moment) which will be more or less important depending on the value of a coefficient β_1 . Adam is quite similar to ADADELTA (see Section 3.4.3), but it adds another value (second moment) which will be the sliding average of the previous squared gradients whose square root corresponds to the sliding uncentered variance of the previous gradients. Adam adopts the following steps:

- Gatti 217
- It estimates the diagonal of Hessian as done in ADADELTA by the term $\operatorname{diag}^{\frac{1}{2}}(\mathbb{U}^{(i)} + \epsilon \mathbb{I}) \operatorname{diag}^{-\frac{1}{2}}(\tilde{\mathbb{G}}^{(i)} + \epsilon \mathbb{I})$ but with local accumulation:

$$\operatorname{diag} \mathbb{V}^{(i)} = \beta_2 \mathbb{V}^{(i-1)} + (1 - \beta_2) \operatorname{diag} \mathbb{G}^{(i)}$$

$$(73)$$

• It estimates the first-order gradient moment, similarly to $\boldsymbol{u}^{(i+1)}$ in Equation (71):

$$\boldsymbol{m}^{(i)} = \beta_1 \boldsymbol{m}^{(i-1)} + (1 - \beta_1) \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)}\right)$$
(74)

It normalizes both diag^{V(i)} and *m*⁽ⁱ⁾ to avoid to produce biased estimators:

$$\hat{\boldsymbol{m}}^{(i)} = \frac{\boldsymbol{m}^{(i)}}{1 - \beta_1}, \qquad \hat{\mathbb{V}}^{(i)} = \frac{\mathbb{V}^{(i)}}{1 - \beta_2}$$
(75)

This additional operation reduces the biases of the first iterations where the moving averages are made on a small number of values.

• It updates the weights as:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \cdot \operatorname{diag}^{-\frac{1}{2}} \left(\hat{\mathbb{V}}^{(i)} + \epsilon \mathbb{I} \right) \boldsymbol{\hat{m}}^{(i)}$$
(76)

Adam has several advantages: the per-dimension update inherited from ADADELTA, which is very important to promote and to tackle sparsity. As RMSprop and ADADELTA, it adopts the use of second order momentum on the denominator. This aspect is tightly linked to the information of the statistical model represented by \mathcal{NN} parametrized by its weights and biases $\boldsymbol{\theta}$. As a matter of fact, the Cramér-Rao bound (CRB) presented in Equation (31) provides an insightful view since it states that the Fisher information $\mathbb{I}_F(\boldsymbol{\theta})$ is larger the the inverse variance of the the weights (represented by $\mathrm{diag}^{-\frac{1}{2}\hat{\mathbb{V}}^{(i)}}$). In other words, the optimizer tries to increase the overall informative power of the \mathcal{NN} , to make it more flexible to complex regression or classification tasks. Moreover, Adam has the advantage of integrating momentum, but also another parameter that will prevent the momentum from taking too much importance in updating the weights. These two tools will allow it is up to Adam to dampen the effects of oscillation in the gradient descent.

Example 15. Adam for linear regression

This example is equivalent to Example 9, with the exception of the definition of the optimizer.

3.4.5 Weight decay

The weight decay allows to keep $\|\boldsymbol{\theta}\|$ small. Heuristically, this is a preferable condition to achieve convergence in the optimization process. Compared to explicitly penalize the L^2 norm in the loss function, by adding something of the sort $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|^2$, the weight decays makes the weights tend to zero quite faster. The weight decay consists in simply adding an extra term $\lambda \omega^{(i)}$ either to the gradient $\nabla_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)}\right)$, either directly to the weight update. In this latter case, the weight decay is decoupled from first and second order moments. For instance, with the Adam optimizer, the decoupled weight decay reads:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta \cdot \operatorname{diag}^{-\frac{1}{2}} \left(\hat{\mathbb{V}}^{(i)} + \epsilon \mathbb{I} \right) \hat{\boldsymbol{m}}^{(i)} + \lambda \boldsymbol{\omega}^{(i+1)}$$
(77)

The PyTorch optimizer class implements a weight_decay option. This extra parameter can directly added to the SGD optimizer input list, since the SGD does not estimate neither the first nor the second gradient moments. On the contrary, RMSprop, Adagrad, AdaDelta, Adam do. However, adding the weight_decay parameter to those standard optimizer classes, couples the weight decay with the estimation of the first and second moments, which is not the most effective way to perform weight decay, since it can add spurious biases. Instead, in order to apply a decoupled weight decay as in Equation (77), one should use the SGDW or AdamW optimizers⁹.

Example 16. Compare optimizers in PyTorch

In the following example [Via22], the SGD optimizer, with and without (Nesterov) momentum is compared to Adam, Adadelta, RMSProp and AdamW, in the framework of an optimization problem with $\theta \in \mathbb{R}^2$ and with the following loss function:

$$L(\theta_1, \theta_2) = 8 \cdot \sin\left(\frac{1.5 \cdot \theta_1}{10} - 3\right) \cdot \sin\left(\frac{1.5 \cdot \theta_1}{10} - 3\right) + \\ + 8 \cdot \cos\left(\frac{\theta_2}{10} - 2\right) \cdot \cos\left(\frac{\theta_2}{10} - 2\right) + \\ 0.3 \cdot \left(\left(\frac{\theta_1}{5} - 12\right) \cdot \left(\frac{\theta_1}{5} - 12\right) + \left(\frac{\theta_2}{10} - 8\right) \cdot \left(\frac{\theta_2}{10} - 8\right)\right) + 10$$

$$(78)$$

 $L(\boldsymbol{\theta})$ is depicted in Figure 33.

The comparison is performed according to following criteria:

- 1. Parameter initialization $(\theta_1^{(0)}, \theta_2^{(0)})$ (see Section 4.2.1 for further insights).
- 2. Use of the (Nesterov) momentum and dampening (for SGD) and RMSprop.
- 3. Weight decay for Adam and AdamW.

⁹https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html



Figure 33: $L(\theta) : \mathbb{R}^2 \to \mathbb{R}$ defined in Equation (78). (a) $L(\theta)$ in 3 dimensions. (b) 2-dimensional contour plot.

All tested optimizers share the same (constant) learning rate η =20 and 10000 epochs.

First and foremost, the effect of two different initializations $\boldsymbol{\theta}^{(0-1)} = (135, 135)$ and $\boldsymbol{\theta}^{(0-2)} = (45, 35)$ is tested. The choice of $\boldsymbol{\theta}^{(0-1)}$ implies that the optimizer algorithm starts from a point closer to the minimum $\hat{\boldsymbol{\theta}}$ than $\boldsymbol{\theta}^{(0-2)}$ (see Figure 33). Figure 34 shows the SGD's high sensitivity to the initialization: the algorithm diverges from $\hat{\boldsymbol{\theta}}$ (within the 10000 epochs) when starting from $\boldsymbol{\theta}^{(0-2)}$ (see Figure 34b), which is closer to the minimum itself. This is mostly due to the fact that the function is not convex close to the minimum $\hat{\boldsymbol{\theta}}$.

Adadelta (with $\alpha_u \in [0, 1]$, called rho in PyTorch) fails for both initialization choices, as depicted in Figure 35. This is due to its high dependency on the weight initialization with biased momentum estimate.

Adam (with standard choice of $\beta_1 = 0.9$ and $\beta_2 = 0.999$) is definitely more robust than SGD and more effective than Adadelta, as shown in Figure 36, although the trend is opposite: Adam converges to $\hat{\theta}$ for $\theta^{(0-2)}$ (as shown in Figure 36b) and it gets somehow closer to it than SGD, for $\theta^{(0-1)}$, as depicted in Example 16, compared to Figure 34a. Compared to standard SGD, Adam leverages the first and second order momenta and adapts their contribution to avoid divergence of the gradient descent algorithm. As a matter of fact, SGD needs (first order) momentum too, in order to converge to the minium in a reasonable amount of epochs (10000), as depicted in Figure 34a. This is evident if one compares the latter to the trajectory that SGD with zero momentum follows, depicted in Figure 37a (with $\tau=0.5$ in order to avoid divergence). The lack of momentum



Figure 34: SGD minimization trajectory for $\theta^{(0-1)}$ (a) and $\theta^{(0-2)}$ (b). SGD is featured by momentum $\gamma=0.7$ and dampening $\tau=0.6$. The yellow dot represents the starting point $\theta^{(0)}$, whereas the blue dot represents the sought minimum.



Figure 35: Adadelta minimization trajectory for $\theta^{(0-1)}$ (a) and $\theta^{(0-2)}$ (b). Adadelta is featured by α_u =0.9, but similar results are obtained regardless its value $\alpha_u \in [0, 1]$. The yellow dot represents the starting point $\theta^{(0)}$, whereas the blue dot represents the sought minimum.

leaves the SGD algorithm iterating around local minima. However, a too high momentum γ can lead the SGD algorithm to miss $\hat{\theta}$ because of its inertia, as shown in Figure 37b for $\gamma=0.9$. For practical purposes, this aspect represents



Figure 36: Adam minimization trajectory for $\theta^{(0-1)}$ (a) and $\theta^{(0-2)}$ (b). The yellow dot represents the starting point $\theta^{(0)}$, whereas the blue dot represents the sought minimum.



the major disadvantage of using SGD, compared to $\tt Adam$ adaptivity and faster convergence rate.

Figure 37: SGD minimization trajectory for $\gamma=0$ and $\tau=0.5$ (a) and for $\gamma=0.9$ and $\tau=0.5$. The yellow dot represents the starting point $\theta^{(0-1)}$, whereas the blue dot represents the sought minimum.

Nesterov momentum enhance the momentum effect (no damping is allowed in this algorithm), possibly leading SGD to reach the minimum $\hat{\theta}$ faster (in less

epochs). However, as shown in Figure 38 for different initializations $\boldsymbol{\theta}^{(0-1)}$ (in Figure 38a) and $\boldsymbol{\theta}^{(0-2)}$ (in Figure 38b), Nesterov-SGD algorithm tends to range around the minimum $\hat{\boldsymbol{\theta}}$, but missing it because of the inertia effect (each final step represents a too large increment, that iteratively misses the minimum). In this case, a proper schedule of the learning rate must be adopted. RMSprop is also conceived to counteract aggressive weight updates, based on



Figure 38: SGD minimization trajectory for $\theta^{(0-1)}$ (a) and $\theta^{(0-2)}$ (b). SGD is featured by Nesterov momentum $\gamma=0.3$ and no dampening. The yellow dot represents the starting point $\theta^{(0)}$, whereas the blue dot represents the sought minimum.

the estimated variance of the gradients computed in past iterations. Despite being more stable and effective that SGD with (Nesterov) momentum, for both weight initializations, in this example RMSprop achieve a "stable" gradient descent trajectory only when the momentum γ is compensated by the coefficient α_r and both are equal to 0.5 (see Figure 39b). However, [Mni+16] showed that RMSprop is quite useful for very non-stationary problems, as confirmed by [ACB17], provided that no momentum is added ($\gamma=0$), $\alpha_r=0.9$ and the learning rate is very small. Reducing the learning rate to 12, the minimum is reached with a clear improvement in the stability of the gradient descent trajectories, for both initializations, as shown in Figure 40. However, the instability close to the minimum $\hat{\theta}$ still persists. From those examples, it looks that calibrating SGD, Adadelta, RMSprop in order to find a reasonable trade-off between convergence rate and stability is hard: either a small learning rate is assumed, with zero momentum and a very large number of epochs demanded to reach the minimum, either the dampened momentum is leverage, with high risk of divergence. Standard Adam, on the contrary, seems definitely more robust and flexible to complex optimization tasks. As far as weight decay is concerned, Adam (no weight decay) and AdamW (with weight decay) are compared in Figure 41. The





Figure 39: RMSprop minimization trajectory for $\gamma=0.1$ and $\alpha_r=0.9$ (a) and for $\gamma=\alpha_r=0.5$ (b). The yellow dot represents the starting point $\theta^{(0-1)}$, whereas the blue dot represents the sought minimum.



Figure 40: RMSprop minimization trajectory for for $\theta^{(0-1)}$ (a) and $\theta^{(0-2)}$ (b). RMSprop is featured by $\gamma=0.0$ and $\alpha_r=0.9$ for both cases. The yellow dot represents the starting point $\theta^{(0)}$, whereas the blue dot represents the sought minimum.

use of AdamW does not provide an improvement, compared to standard Adam, despite the extremely low weight decay penalty equal to 0.00001. Adam still represent, for this example, the most robust and efficient optimizer, provided that no tuning is necessary, compared to other optimization techniques.





Figure 41: Minimization trajectories for Adam (a) and AdamW (with weight decay penalty of 0.00001) (b). The yellow dot represents the starting point $\theta^{(0-2)}$, whereas the blue dot represents the sought minimum.

For further insight, see Chapter 2 - Introduction to regression methods.

4 Automatic Differentiation

As stated in Section 2, Neural Networks \mathcal{NN} are a special type of statistical models \mathcal{H}_{θ} , made of complex functions h_{θ} defined as composition of non linear ridge activations applied to affine transformations. The problem they aim at solving is stated in problem (\mathcal{P}). The search of the "best" set of weights $\theta \in \Theta \subset \mathbb{R}^{d_m}$ is performed by attempting at minimizing the empirical loss

function $L_{\mathcal{D}_{XY}}(\mathbf{h}_{\theta})$, defined over a set of i.i.d. samples $\mathcal{D}_{XY} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, with $\mathbf{x}_i \in \mathcal{X} \subset \mathbb{R}^{d_X}$ and $\mathbf{y}_i \in \mathcal{Y} \subset \mathbb{R}^{d_Y}$. The quest for the best predictor is a matter of optimization (see Section 3, and the solution (\mathcal{G}) of problem (\mathcal{P})). Essentially, weights and biases of each neuronal affine transformation are the weights $\boldsymbol{\theta}$ we are looking for. The composition pattern of such weights and biases, within a $\mathcal{N}\mathcal{N}$, follows intricate graphs. Therefore, weights and biases are progressively updated via the *backward propagation* algorithm, based on the derivative chain rule.

4.1 Updating weights with the chain rule

If one adopt the standard Stochastic Gradient Descent framework, with minibaches (see Algorithm 2), the *backward propagation* algorithm consists in computing the derivative of $\frac{\partial L_{D_{XY}}}{\partial \theta_i}$ and the iterative update each weight θ_i according to the following scheme:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \frac{\eta}{N_b} \sum_{k=(j-1)\frac{N_b}{N}}^{j\frac{N_b}{N}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} l\left(\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_k\right), \boldsymbol{y}_k\right) \\ -\eta \cdot \lambda \cdot \sum_{k=(j-1)\frac{N_b}{N}}^{j\frac{N_b}{N}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \Omega(\boldsymbol{w}), \quad \eta \in \mathbb{R}^+$$

$$(79)$$

with $l(\boldsymbol{h}(\boldsymbol{x}), \boldsymbol{y}) : \boldsymbol{\mathcal{Y}} \times \boldsymbol{\mathcal{Y}} \to \mathbb{R}^+$ being a measure of the distance between real label and prediction, such that $l(\boldsymbol{y}, \boldsymbol{y}) = 0$. $\Omega(\boldsymbol{w})$ represents a penalty on the norm of the weights, with a penalty coefficient λ . The reasons behind weight penalty are outlined in Section 2.1. η is the *learning rate*. The back-propagation is usually performed on mini-batches of N_b instances. Equation (79) represents the so called *batch gradient descent*. The partial derivatives $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_i}$ are computed according to the classical *chain rule*, that reads:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} = \frac{1}{N} \sum_{k=1}^{N} \boldsymbol{\nabla}_{\boldsymbol{\theta}} l\left(\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{k}\right), \boldsymbol{y}_{k}\right) + \lambda \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} \Omega\left(\boldsymbol{\theta}\right)$$
with $\boldsymbol{\nabla}_{\boldsymbol{\theta}} l\left(\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right)\right) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right)^{T} \cdot \boldsymbol{\nabla}_{h_{\boldsymbol{\theta}}} l\left(\boldsymbol{h}_{\boldsymbol{\theta}}\right)$
(80)

The Jacobian $\mathbb{J}_{h}^{(k)} = \nabla_{\theta} h_{\theta}^{(k)}(x)$ of the output units at k^{th} layer, with the respect to the weights and biases $\boldsymbol{\theta}^{(k)}$ (of size $u^{(k+1)}$, number of output units). reads:

$$\mathbb{J}_{h}^{(k)} = \left[\begin{array}{cc} \frac{\partial \boldsymbol{h}_{\theta}^{(k)}}{\partial \theta_{1}^{(k)}} & \frac{\partial \boldsymbol{h}_{\theta}^{(k)}}{\partial \theta_{2}^{(k)}} & \cdots & \frac{\partial \boldsymbol{h}_{\theta}^{(k)}}{\partial \theta_{u}^{(k)+1}} \end{array}\right]$$
(81)

The $\nabla_{h_{\theta}} l(h_{\theta})$ represents the gradient of the loss function with the respect to the activation of the output units. Therefore, the chain rule with the respect

to $\boldsymbol{\theta}^{(k)}$ can be explicitly stated as follows:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} l\left(\boldsymbol{h}_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right)\right) = \begin{bmatrix} \frac{\partial l}{\partial \boldsymbol{\theta}_{1}^{(k)}} \\ \frac{\partial l}{\partial \boldsymbol{\theta}_{2}^{(k)}} \\ \frac{\partial l}{\partial \boldsymbol{\theta}_{2}^{(k)}} \\ \frac{\partial l}{\partial \boldsymbol{\theta}_{2}^{(k)}} \\ \frac{\partial l}{\partial \boldsymbol{\theta}_{2}^{(k)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial h_{1}^{(k)}}{\partial \boldsymbol{\theta}_{1}^{(k)}} & \frac{\partial h_{2}^{(k)}}{\partial \boldsymbol{\theta}_{1}^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} \\ \frac{\partial h_{1}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} & \frac{\partial h_{2}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} \\ \frac{\partial h_{1}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} & \frac{\partial h_{2}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} \\ \frac{\partial h_{1}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} & \frac{\partial h_{2}^{(k)}}{\partial \boldsymbol{\theta}_{2}^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \boldsymbol{\theta}_{u^{(k+1)}}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_{1}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{u^{(k+1)}}^{(k)}} \\ \frac{\partial l}{\partial \boldsymbol{\theta}_{u^{(k+1)}}^{(k)}} & \frac{\partial h_{2}^{(k)}}{\partial \boldsymbol{\theta}_{u^{(k+1)}}^{(k)}} & \cdots & \frac{\partial h_{u^{(k+1)}}^{(k)}}{\partial \boldsymbol{\theta}_{u^{(k+1)}}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_{1}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^{(k)}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial l}{\partial h_{2}^{(k)}} \\ \frac{\partial l}{\partial h_{2}^$$

The chain "rules" the automatic differentiation In order to better understand the back propagation and automatic differentiation, the following paragraph outlines all the necessary operations to compute the back-propagation for a classification problem with N_y classes, a dataset $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ with $y_i \in \{1, ..., N_y\}$ (see Chapter 4 - Classification Techniques in Machine Learning). The \mathcal{NN} classifier is represented by the following \mathcal{MLP} h_{θ} , defined as:

$$h_{\theta}\left(\boldsymbol{x}\right) = \sigma_{y}\left(\sum_{c=1}^{N_{y}} h_{c}^{\left(N_{\ell}\right)}\left(\boldsymbol{x}\right) \boldsymbol{e}_{c}\right)$$
(83)

where the output activation function $g^{(o)}$ is equal to the softmax_y function, that reads:

$$h^{(o)} = g^{(o)}(\boldsymbol{a}^{(o)}) = \sigma_y\left(\boldsymbol{a}^{(o)}\right) = \text{softmax}_y\left(\boldsymbol{a}^{(o)}\right) = \frac{e^{a_y^{(o)}(\boldsymbol{x})}}{\sum\limits_{c=1}^{N_y} e^{a_c^{(o)}(\boldsymbol{x})}}$$
(84)

In particular, the pre-activation function of the output layer reads as N_y entries that read:

$$a_{c}^{(o)}(\boldsymbol{x}) = h_{c}^{(N_{\ell})} = g_{c}^{(N_{\ell})}\left(a_{c}^{(N_{\ell})}(\boldsymbol{x})\right), \quad 1 \le c \le N_{y}$$
 (85)

Finally, the classifier is trained to minimize the Negative Log-Likelihood loss that reads:

$$l(h_{\theta}(\boldsymbol{x}), y) = -\sum_{c=1}^{N_{y}} \ln\left(\sigma_{c}\left(\boldsymbol{a}^{(o)}(\boldsymbol{x})\right)\right) \chi_{(y=c)}$$
(86)

The following steps outline the different steps of the back-propagation, applying the chain rule to Equation $(86)^{10}$.

¹⁰The complete tutorial can be found at https://www.youtube.com/watch?v=p5tL2JqCRDo& list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH&index=12 (Hugo Larochelle video-lecture series "Neural networks").

1. Derivative at output activation:

$$\frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),\boldsymbol{y}\right)}{\partial\sigma_{c}} = -\frac{\chi_{\left(\boldsymbol{y}=c\right)}}{\sigma_{c}\left(\boldsymbol{a}^{\left(o\right)}\left(\boldsymbol{x}\right)\right)}$$
(87)

2. Gradient @ output activation (see softmax_y definition in Equation (91))

$$\nabla_{\sigma} l\left(h_{\theta}\left(\boldsymbol{x}\right), \boldsymbol{y}\right) = \sum_{c=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right), \boldsymbol{y}\right)}{\partial \sigma_{c}} \boldsymbol{e}_{c} = \\ = -\sum_{c=1}^{N_{y}} \frac{\chi_{(y=c)}}{\sigma_{c}\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)} \boldsymbol{e}_{c} = -\frac{\boldsymbol{e}_{y}}{\sigma_{y}\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)}$$
(88)

3. Derivative at activation (output layer $N_\ell,$ see Equation (85))

$$\frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial h_{c}^{\left(N_{\ell}\right)}} = \sum_{c'=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial \sigma_{c'}} \cdot \frac{\partial \sigma_{c'}}{\partial h_{c}^{\left(N_{\ell}\right)}}\Big|_{\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)}$$
(89)

with:

$$\frac{\partial \sigma_{c'}}{\partial h_c^{(N_\ell)}}\Big|_{\boldsymbol{a}^{(o)}(\boldsymbol{x})} = \frac{\partial \sigma_{c'}}{\partial a_c^{(o)}} = \sigma_{c'}\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right) \cdot \left(\chi_{(c'=c)} - \sigma_c\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)\right)$$
(90)

4. Gradient at activation (output layer N_{ℓ} , see Equation (85)) Considering that the softmax output activation function can be rewritten, with the use of the indicator function $\chi_{y=c}$, as:

$$h^{(o)}\left(\boldsymbol{a}^{(o)}\right) = \sigma_{y}\left(\boldsymbol{a}^{(o)}\right) = \sum_{c=1}^{N_{y}} \sigma_{c}\left(\boldsymbol{a}^{(o)}\right) \chi_{y=c} = \left\langle \sum_{c=1}^{N_{y}} \chi_{c=y} \boldsymbol{e}_{c}, \boldsymbol{\sigma}\left(\boldsymbol{a}^{(o)}\right) \right\rangle$$
(91)

$$\nabla_{h^{(N_{\ell})}} l\left(h_{\theta}\left(\boldsymbol{x}\right), \boldsymbol{y}\right) = \sum_{c=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right), \boldsymbol{y}\right)}{\partial h_{c}^{(N_{\ell})}} \boldsymbol{e}_{c} = \\ = \sum_{c=1}^{N_{y}} \sum_{c'=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right), \boldsymbol{y}\right)}{\partial \sigma_{c'}} \cdot \frac{\partial \sigma_{c'}}{\partial h_{c}^{(N_{\ell})}} \Big|_{\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)} = \\ = -\sum_{c=1}^{N_{y}} \left(\chi_{(y=c)} - \sigma_{c}\right) \boldsymbol{e}_{c} = -\left(\boldsymbol{e}_{y} - \boldsymbol{\sigma}\left(\boldsymbol{a}^{(o)}\right)\left(\boldsymbol{x}\right)\right)$$
(92)

5. Derivative at pre-activation (output layer N_{ℓ})

$$\frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),\boldsymbol{y}\right)}{\partial a_{c}^{\left(N_{\ell}\right)}} = \sum_{c'=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),\boldsymbol{y}\right)}{\partial h_{c'}^{\left(N_{\ell}\right)}} \cdot \frac{\partial h_{c'}^{\left(N_{\ell}\right)}}{\partial a_{c}^{\left(N_{\ell}\right)}}$$
(93)

with:

$$\frac{\partial h_{c'}^{(N_{\ell})}\left(\boldsymbol{x}\right)}{\partial a_{c}^{(N_{\ell})}} = \frac{\partial g_{c'}^{(N_{\ell})}}{\partial a_{c}^{(N_{\ell})}} \left(a_{c'}^{(N_{\ell})}\left(\boldsymbol{x}\right)\right) \chi_{(c'=c)}$$
(94)

and

$$\frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),\boldsymbol{y}\right)}{\partial a_{c}} = -\sum_{c'=1}^{N_{y}} \left(\chi_{(y=c')} - \sigma_{c'}\right) \frac{\partial g_{c'}}{\partial a_{c}^{(N_{\ell})}} \left(a_{c'}^{(N_{\ell})}\left(\boldsymbol{x}\right)\right) \chi_{(c'=c)} = \\ = -\left(\chi_{(y=c)} - \sigma_{c}\right) \frac{\partial g_{c}^{(N_{\ell})}}{\partial a_{c}^{(N_{\ell})}} \left(a_{c}^{(N_{\ell})}\left(\boldsymbol{x}\right)\right)$$

$$\tag{95}$$

6. Gradient at pre-activation (provided that $g_c^{(N_\ell)} = g \quad \forall c$)

$$\nabla_{a^{(N_{\ell})}} l\left(h_{\theta}\left(\boldsymbol{x}\right), y\right) = \sum_{c=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right), y\right)}{\partial a_{c}^{(N_{\ell})}} \boldsymbol{e}_{c} =$$

$$= \nabla_{h} l\left(h_{\theta}\left(\boldsymbol{x}\right), y\right) \odot \nabla_{a} g =$$

$$= -\left(\boldsymbol{e}_{y} - \boldsymbol{\sigma}\left(\boldsymbol{a}^{(o)}\left(\boldsymbol{x}\right)\right)\right) \odot \nabla_{a} g \qquad (96)$$

7. Derivative at weight (output layer N_{ℓ}) Considering that

$$a_{c}^{(N_{\ell})} = \sum_{c'=1}^{N_{y}} W_{cc'}^{(N_{\ell})} h_{c'}^{(N_{\ell-1})} + b_{c}^{(N_{\ell})}$$
(97)

$$\frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial W_{cc'}^{(N_{\ell})}} = \sum_{n=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial a_{n}^{(N_{\ell})}} \frac{\partial a_{n}^{(N_{\ell})}}{\partial W_{cc'}^{(N_{\ell})}} = \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial a_{c}^{(N_{\ell})}} h_{c'}^{(N_{\ell}-1)}$$

$$(98)$$

8. Derivative at bias (output layer N_{ℓ})

$$\frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial b_{c}^{\left(N_{\ell}\right)}} = \sum_{n=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial a_{n}^{\left(N_{\ell}\right)}} \frac{\partial a_{n}^{\left(N_{\ell}\right)}}{\partial b_{c}^{\left(N_{\ell}\right)}} = \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)}{\partial a_{c}^{\left(N_{\ell}\right)}} \cdot 1 \qquad(99)$$

9. Gradient at weight $W_{cc'}^{(N_{\ell})}$ (output layer N_{ℓ}) Considering Equation (97) once again:

$$\boldsymbol{\nabla}_{W^{(N_{\ell})}}l(h_{\theta}(\boldsymbol{x}), y) = \boldsymbol{\nabla}_{a}l(h_{\theta}(\boldsymbol{x}), y) \otimes \boldsymbol{h}^{(N_{\ell}-1)}$$
(100)

10. Gradient at bias (output layer N_{ℓ})

$$\boldsymbol{\nabla}_{b^{(N_{\ell})}}l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right) = \boldsymbol{\nabla}_{a}l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)$$
(101)

11. Derivative at activation (hidden layer $N_{\ell-1}$) Considering Equation (97) once again:

$$\frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),\boldsymbol{y}\right)}{\partial h_{c}^{(N_{\ell-1})}} = \sum_{n=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),\boldsymbol{y}\right)}{\partial a_{n}^{(N_{\ell})}} \frac{\partial a_{n}^{(N_{\ell})}}{\partial h_{c}^{(N_{\ell-1})}} = \sum_{n=1}^{N_{y}} \frac{\partial l\left(h_{\theta}\left(\boldsymbol{x}\right),\boldsymbol{y}\right)}{\partial a_{n}^{(N_{\ell})}} W_{nc}^{(N_{\ell})} \tag{102}$$

12. Gradient at activation (hidden layer $N_{\ell-1}$)

$$\boldsymbol{\nabla}_{h^{(N_{\ell}-1)}}l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right) = \boldsymbol{W}^{(N_{\ell})T}.\boldsymbol{\nabla}_{a}l\left(h_{\theta}\left(\boldsymbol{x}\right),y\right)$$
(103)

13. Reiterate points 5 to 13 for $\ell = 1 \dots N_{\ell} - 1$, considering that, for all layers $1 \leq k \leq N_{\ell}$ the following expression holds:

$$h_c^{(k)} = g^{(k)} \left(\sum_{c'=1}^{u^{(k)}} W_{cc'}^{(k)} h^{(k-1)} + b^{(k)} \right)$$
(104)

with $u^{(k)}$ being the number of hidden neurons (or units) in layer k

In PyTorch, the automatic differentiation is performed by the autograd library. autograd organizes the different operations in a directed acyclic graph (DAG, see Figure 42). Moreover, it keeps track of each variable (a tensor) along with the executed operations on it (including the gradient of such operations with the respect to the input, stored in the object attribute grad and the object function grad_fn) and their resulting new tensors. Input tensors are the leaves of the DAG, while the output tensors are the roots of the DAG. Once the loss is computed in the forward function (a basic method of the class nn.Module, that assembles the computational graph), the back-propagation is performed by calling the method loss.backward(), that is automatically performed by exploiting the computational graph assembled in the forward pass. This is possible because PyTorch performs the chain rule operations on the DAG from every operation that involves a gradient-computing tensor or its dependencies, as shown in Figure 43. In order to explicitly update the tensors, one needs to use the context torch.no_grad(). The next example clarifies this point.



Figure 42: The PyTorchViz package allows to visualize the computational graph associated to a PyTorch tensor, by adopting the make_dot function.



Figure 43: Automatic Differentiation scheme for a function f(x) = g(a(x)), with $a = w \cdot x + b$ and g a ridge function. Reprinted and modified from [Fey18]

Example 17. Automatic Differentiation with PyTorch

```
import numpy as np
1
2
       import torch
       # assure reproductibility
3
^{4}
      torch.manual_seed(0)
      # Data for regression problem
A = 10.1542550
5
6
      b = 14.7351129
7
      epsilon = 0.1
8
9
      N = 200 \# number i.i.d. samples
10
       # Data Generation
11
      np.random.seed(42)
12
      y = np.random.rand(N, 1)
y = b + A * np.exp(x) + epsilon * np.random.randn(N, 1)
13
14
15
16
       # Shuffles the indices
      idx = np.arange(N)
np.random.shuffle(idx)
17
18
19
```

```
# Uses first 80 random indices for train
20
       train_idx = idx[:int(N*0.8)]
# Uses the remaining indices for validation
^{21}
22
       val_idx = idx[int(N*0.8):]
23
^{24}
       \ensuremath{\textit{\#}} Generates train and validation sets
25
       x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
26
27
28
       # Convert Numpy arrays into PyTorch Tensors
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()
29
30
31
32
       alpha = 1e-1 # learning rate
33
       n_epochs = 1000 # number of epochs
^{34}
35
       \ensuremath{\texttt{\#}} initialize weights and bias
36
       w = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
37
       b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
38
39
       for epoch in range(n_epochs):
40
           yhat = b + w * x_train_tensor
^{41}
42
             error = y_train_tensor - yhat
43
            loss = (error ** 2).mean()
44
             # Perform autodiff!
45
            loss.backward()
46
47
            print(w.grad)
48
49
             print(b.grad)
50
             # We need to use NO_GRAD to manually update the gradients
51
            with torch.no_grad():
52
                 w -= alpha * w.grad
b -= alpha * b.grad
53
54
55
            # Drop the old gradients
56
57
            w.grad.zero_()
            b.grad.zero_()
58
59
60
       print(A, b)
```

In practice, the tensor update is performed automatically too, by using the **PyTorch optimizer** library, presented in Section 3.

Example 18. Back-propagation through a multi-class classifier with PyTorch AutoGrad library

In this example, a $\mathcal{N}\mathcal{N}$ classifier is trained over a dummy dataset (iris dataset), with $N_y = 3$ and with two hidden layers of 4 and 8 neurons respectively. The PyTorchViz package, one again, allows to visualize the computational graph associated to a PyTorch tensor, by adopting the make_dot function. In PyTorch, compared to Equation (86), the softmax_y vector of probabilities in combination with the Negative Log-Likelihood loss for multi-class classification, can be replaced by creating a model with linear output activation function $g^{(o)}(a_c^{(o)}) = a_c^{(o)}$ and using the CrossEntropyLoss class to compute the Negative Log-Likelihood from the so-called logits $a_c^{(o)}$, i.e., the score obtained

by the classifier h_{θ} for each class. The cross entropy $\mathbb{H}(p||p_{\theta})$, between two probability distributions: $p(y_1, y_2, \dots, y_{N_y})$ (the "real" probability distribution of the classes) and the model probability $p_{\theta}(y|\boldsymbol{x})$:

$$\mathbb{H}(p||p_{\theta}) = -\sum_{c=1}^{N_{y}} p(y_{i} = 1, y_{j \neq i} = 0) \cdot \ln p_{\theta}(y_{i}|\boldsymbol{x})$$
(105)

which corresponds to Equation (86).

```
# Source:
 1
       \label{eq:linear} \ensuremath{\texttt{\#}}\ https://machinelearningmastery.com/building-a-multiclass-classification-model-in-pytorch/delta-formula}
 2
      import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
 3
 4
 \mathbf{5}
       import torch
 6
       import torch.nn as nn
 7
 8
       import torch.optim as optim
 9
       import tqdm
       from sklearn.model_selection import train_test_split
10
       from sklearn.preprocessing import OneHotEncoder
11
      import copy
# uncomment the next line to install torchviz if necessary
12
13
       # ! pip install torchviz
14
15
      from torchviz import make_dot
16
       # To assure reproductibility, fix the random seed
torch.manual_seed(0) # to assure reproductibility
np.random.seed(0) # to assure reproductibility
17
18
19
20
      # Download and parse the IRIS dataset (from the UCI Machine Learning repository)
# This dataset was conceived by Sir Ronald Fisher and it is among the best-known
^{21}
22
      # dataset for pattern recognition
data_url= "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
23
^{24}
       data = pd.read_csv(data_url, header=None)
25
      X = data.iloc[:, 0:4]
y = data.iloc[:, 4:]
^{26}
27
28
      # The IRIS dataset is composed of the three class labels:
29
      # 1. Iris-setosa
30
      # 2. Iris-versicolor
^{31}
      #3. Iris-virginica
^{32}
      # apply one-hot encoding
33
      ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False).fit(y)
^{34}
      y = ohe.transform(y)
35
36
       # convert pandas DataFrame (X) and numpy array (y) into PyTorch tensors
37
      X = torch.tensor(X.values, dtype=torch.float32)
38
39
      y = torch.tensor(y, dtype=torch.float32)
40
       # split into train and test sets
41
       X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)
42
43
       # define the MLP sequentially with torch.nn
44
       h_theta = nn.Sequential()
^{45}
46
        # first_hidden_layer_pre-activation_a^(1)
47
       h_theta.add_module('a1', nn.Linear(4, 8))
        # first hidden layer activation function g^(1)
48
      h_theta.add_module('g1', nn.ReLU())
49
50
        # second hidden layer pre-activation a^2
       h_theta.add_module('a2', nn.Linear(8, 3))
51
52
       # define the loss function
53
```

```
loss_fn = nn.CrossEntropyLoss() # loss function
54
       optimizer = optim.Adam(h_theta.parameters(), lr=0.001) # optimizer
55
56
       # prepare model and training parameters
 57
       n_epochs = 10 # number of training epochs
batch_size = 5 # size of the mini batch
 58
59
       batches_per_epoch = len(X_train) // batch_size
60
61
       best_acc = - np.inf
                               # init to negative infinity
62
       best_weights = None
63
        # track loss history
 ^{64}
       train_loss_hist = []
train_acc_hist = []
 65
66
       test_loss_hist = []
67
       test_acc_hist = []
68
69
       # training loop over the epochs
 70
 71
       for epoch in range(n_epochs):
            epoch_loss = []
epoch_acc = []
 72
73
            # set model in training mode
 74
 75
            h_theta.train()
 76
            with tqdm.trange(batches_per_epoch, unit="batch", mininterval=0) as bar:
                bar.set_description(f"Epoch {epoch}")
 77
 78
                for i in bar:
                    # take a batch
start = i * batch_size
79
80
                    X_batch = X_train[start:start+batch_size]
y_batch = y_train[start:start+batch_size]
 81
^{82}
 83
                     # infer (forward)
 84
                    y_pred = h_theta(X_batch)
 85
                     # compute the loss
                    loss = loss_fn(y_pred, y_batch)
86
                     # reset previously saved gradients and empty the optimizer memory
87
                     optimizer.zero_grad()
 88
                       run backward propagation
 89
90
                     loss.backward()
91
                     # update weights
92
                     optimizer.step()
                     # compute and store metrics
93
                     acc = (torch.argmax(y_pred, 1) == torch.argmax(y_batch, 1)).float().mean()
^{94}
                     epoch_loss.append(float(loss))
 95
                     epoch_acc.append(float(acc))
96
                    bar.set_postfix(
    loss=float(loss),
97
98
                         acc=float(acc)
99
                     )
100
            # set model in evaluation mode to infer the class in the test set
101
            # without storing gradients for brackprop
102
103
           h_theta.eval()
            # infer the class over the test set
y_pred = h_theta(X_test)
104
105
106
            ce = loss_fn(y_pred, y_test)
107
108
109
            acc = (torch.argmax(y_pred, 1) == torch.argmax(y_test, 1)).float().mean()
110
            ce = float(ce)
            acc = float(acc)
111
            train_loss_hist.append(np.mean(epoch_loss))
112
            train_acc_hist append(np mean(epoch_acc))
113
            test_loss_hist.append(ce)
114
115
            test_acc_hist.append(acc)
116
            if acc > best_acc:
                best_acc = acc
117
                best_weights = copy.deepcopy(h_theta.state_dict())
118
            print(f"Epoch {epoch} validation: Cross-entropy={ce:.2f}, Accuracy={acc*100:.1f}%")
119
```

```
120
121
        # plot computational graph with torchviz
       h_theta.eval()
y_pred = h_theta(X_test)
122
123
       ce = loss_fn(y_pred, y_test)
124
       make_dot(y_pred, params=dict(h_theta.named_parameters()),
125
         show_attrs=True, show_saved=True).render("classifier_graph", format="png")
126
127
       make_dot(ce, params=dict(h_theta.named_parameters()),
128
         show_attrs=True, show_saved=True).render("loss_graph", format="png")
129
130
131
        # Restore best model
132
       h_theta.load_state_dict(best_weights)
133
134
       # Plot the loss and accuracy for train and test sets
plt.plot(train_loss_hist, label="train")
plt.plot(test_loss_hist, label="test")
135
136
137
138
       plt.xlabel("epochs")
       plt.ylabel("cross entropy")
139
140
       plt.legend()
       plt.show()
141
142
       plt.plot(train_acc_hist, label="train")
143
       plt.plot(test_acc_hist, label="test")
144
145
       plt.xlabel("epochs")
146
       plt.ylabel("accuracy")
147
       plt.legend()
148
       plt.show()
```

4.2 Countermeasures to vanishing gradients

Automatic Differentiation is based on the classical *chain rule* outlined in Equation (80). During backpropagation, gradients that "flow" into a neuron are proportional to the activation of the input neurons they are connected with. When the activation values are small, gradients vanish and the neurons do not learn anything. If one considers a feed-forward Multi-Layer Perceptron on the dataset $\mathcal{D}_{XY} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, with $\boldsymbol{x}_i \in \mathbb{R}^{d_X}, y \in \mathbb{R}$, made of N_ℓ hidden layers reads:

$$h_{\theta}(\boldsymbol{x}) = g^{(o)} \circ a^{(o)} \circ g^{(N_{\ell})} \circ a^{(N_{\ell})} \circ \dots \circ g^{(\ell)} \circ a^{(\ell)} \circ \dots g^{(1)} \circ a^{(1)}(\boldsymbol{x})$$

$$a^{(\ell)}(u) = w^{(\ell)} \cdot u + b^{(\ell)}, \quad 1 < \ell \le N_{\ell}$$

$$h^{(\ell)}(u) = g^{(\ell)}\left(a^{(\ell)}(u)\right), \quad 1 \le \ell \le N_{\ell}, \quad h^{(o)}(u) = g^{(o)}(u)$$

$$a^{(1)}(\boldsymbol{x}) = \left\langle \boldsymbol{w}^{(1)}, \boldsymbol{x} \right\rangle + b^{(1)},$$
(106)

If one computes the derivative of the loss function $L_{\mathcal{D}_{XY}}$ is based on the chain rule. For instance, in order to update $w_c^{(1)}$ the following derivative must be computed:

$$\frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h^{(N_\ell)}} \left(\prod_{\substack{\ell=1\\N_\ell>1}}^{N_\ell - 1} \frac{\partial g^{(N_\ell + 1-\ell)}}{\partial a^{(N_\ell + 1-\ell)}} \cdot \frac{\partial a^{(N_\ell + 1-\ell)}}{\partial h^{(N_\ell - \ell)}} \right) \frac{\partial g^{(1)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial w_c^{(1)}}$$
(107)

with the following update rule:

$$w_c^{(1)(i+1)} = w_c^{(1)(i)} - \eta \frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}}$$
(108)

A vanishing gradient make the iterative back-propagation scheme to fail, since the weights tends to remain constant due to the fact that $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}} \approx 0$. The choice of the activation functions g plays a major role in making the gradient vanish. In particular, the activation functions g with limited *output range*, i.e., whose derivative $\frac{\partial g}{\partial a}$ has a compact support (the so called *region of significance*), can yield derivatives close to 0 for certain values of the pre-activation a (see Figure 44).



Figure 44: Examples of activations functions g(a) and their derivative with the respect to the pre-activation function a.

The gradient $\nabla_{w^{(1)}} L_{\mathcal{D}_{XY}}$ tends to vanish when the derivative

$$\frac{\partial g^{(N_\ell+1-\ell)}}{\partial a^{(N_\ell+1-\ell)}} \approx 0 \tag{109}$$

This effect is even emphasized for deeper networks, because the product

$$\prod_{\ell=1}^{N_{\ell}-1} \frac{\partial g^{(N_{\ell}+1-\ell)}}{\partial a^{(N_{\ell}+1-\ell)}} \cdot \frac{\partial a^{(N_{\ell}+1-\ell)}}{\partial h^{(N_{\ell}-\ell)}} \approx 0$$
(110)

even faster if the same activation function g is chosen for all layers.

The most commonly adopted activation function has been g = ReLU so far [GBB11]. As a matter of fact, deep \mathcal{NN} featured by ReLU with hard zero threshold, in combination with ℓ^1 -norms on the weights (to bound the positive activation values a > 0), promote sparsity and ease the gradient backpropagation on active units [GBB11]. After being initialized with uniform sampling, almost half of the hidden neurons output value is zero, with this fraction increasing whenever the ℓ^1 regularization is adopted. In this sense, active neurons alleviates the vanishing gradient since $\frac{\partial g}{\partial a}|_{a>0} = 1$. The \mathcal{MLP} becomes a piece-wise linear function, with neurons that are either deactivated a < 0 or operating in linear regime a > 0. The basis of those sparse piece-wise linear ridge functions helps representing the labelling of regression function $f : \mathbf{x} \mapsto y$ over a basis of ridge functions (see Section 2.1 for further details about the Universal Approximation Theorem).

Example 19. Vanishing gradient effect for deep MLP classifiers

In this example, a 10-layer deep \mathcal{MLP} , is conceived for the multi-class classification problem outlined in Section 4.1, with $N_y=3$ classes and with $fan_{in} = fan_{out} = 5$ for all hidden layers (see Example 18 for further details). The depth of the \mathcal{MLP} designed in the present PyTorch example makes it prone to vanishing gradients, at the deepest layers especially. In particular, the example compares the evolution of the mean (over the database samples) gradient norm with the training epochs, for two different activation functions, namely tanhand ReLU. The results are depicted in Figure 45. From Figure 45, it is clear that :

- When the activation function saturates and the \mathcal{MLP} has too many layers, the gradient of the loss function with the respect to the weights and biases of the initial layers vanishes, preventing the loss to converge to 0 (see Figure 45a and Figure 45c).
- Replacing the activation function by a non-saturating one, such as $g^{(\ell)} = ReLU$, prevents the gradient to vanish at all layers which in turns make the loss function converge to 0.

The PyTorch code is reported below.

1

2

3

4

 $\mathbf{5}$

6

```
# Source:
# https://machinelearningmastery.com/building-a-multiclass-classification-model-in-pytorch/
# https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
```



Figure 45: Loss function evolution with epoch of training (a-b) and mean gradient norm of each layer of the \mathcal{MLP} (c-d). (a) and (c) refer to the case with the activation function $g^{(\ell)} = tanh$, whereas (b) and (d) refer to the case with activation function $g^{(\ell)} = ReLU$.

```
import torch.nn as nn
 8
9
      import torch.optim as optim
      import tqdm
10
      from sklearn.model_selection import train_test_split
11
      from sklearn.preprocessing import OneHotEncoder
^{12}
13
      import copy
      # install torchviz (optional)
14
      # ! pip install torchviz
15
      # from torchviz import make_dot
16
17
18
      # To assure reproductibility, fix the random seed
      torch.manual_seed(0) # to assure reproductibility
np.random.seed(0) # to assure reproductibility
19
20
21
      # Download and parse the IRIS dataset (from the UCI Machine Learning repository)
22
23
      # This dataset was conceived by Sir Ronald Fisher and it is among the best-known
      # dataset for pattern recognition
data_url= "https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
^{24}
25
      data = pd.read_csv(data_url, header=None)
26
      X = data.iloc[:, 0:4]
27
^{28}
      y = data.iloc[:, 4:]
29
30
      # The IRIS dataset is composed of the three class labels:
^{31}
      # 1. Iris-setosa
      # 2. Iris-versicolor
32
     # 3. Iris-virginica
33
      # apply one-hot encoding
34
      ohe = OneHotEncoder(handle_unknown='ignore', sparse_output=False).fit(y)
35
      y = ohe.transform(y)
36
37
      # convert pandas DataFrame (X) and numpy array (y) into PyTorch tensors X = torch.tensor(X.values, dtype=torch.float32)
38
39
      y = torch.tensor(y, dtype=torch.float32)
40
      d_X = X.shape[1]
41
      Ny = y.shape[-1]
42
       # split into train and test sets
^{43}
      X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=True)
44
^{45}
       # Function to track grad norm
46
^{47}
      def track_grad_norm(model, norm_type='fro'):
        # Collect parameters (weights and biases from all layers)
48
49
        parameters = [p for p in model.parameters()
50
        if p.grad is not None and p.requires_grad]
# Defin the total norm and initialize to 0
51
        total_norm = {k: 0.0 for k,_ in model.named_parameters()}
52
        if len(parameters) > 0:
53
             # get weight device (initial weight)
54
             device = parameters[0].grad.device
55
             # Assign values of the grad to each layer (in the dictionary)
56
57
             for n,p in model.named_parameters():
               total_norm[n]=torch.norm(p.grad.detach(), norm_type).to(device).item()
58
        return total_norm
59
60
      loss_fn = nn.CrossEntropyLoss() # loss function
61
62
       # Design the MLP
63
      h_theta = nn.Sequential()
64
      nl = 10 # number of hidden layers
fan_in = 5 # number of hidden units per neuron
65
66
      # Define the dictionary of different activation functions
67
      activations={'tanh': nn.Tanh(), 'ReLU': nn.ReLU(), 'Sigmoid': nn.Sigmoid() }
68
69
      # chosen activation function
act = 'tanh' # switch to ReLU or Sigmoid
70
71
      activation = activations[act]
72
73
      # Initialize latex dictionary for plotting
```

```
latex_dict={}
74
       # Iterate over layer number
75
      for l in range(nl):
 76
 77
        if 1==0:
          h_theta.add_module('a{:>d}'.format(l), nn.Linear(d_X, fan_in))
 78
79
         else:
           h_theta.add_module('a{:>d}'.format(l), nn.Linear(fan_in, fan_in))
80
81
        h_theta.add_module('g{:>d}'.format(l), activation)
 ^{82}
         # Update the latex dictionary for plotting
83
         latex_dict["a{:>d}.weight".format(1)]=r"$\habla_{{ \ M}} } a^{{(:>d)}}".format(1)
 84
 85
        latex_dict["a{:>d}.bias".format(1)]=r"$\frac{{\partial a^{{({:>d})}}}}{{\partial b}}*".format(1)
 86
       # define output layer
      h_theta.add_module('aout', nn.Linear(fan_in, Ny))
87
       # Update the latex dictionary for plotting
88
       latex_dict['aout.weight'.format(1)]=r"$\nabla_{{ \mathbf{{W}} }} a^{{(o)}}$"
 89
       latex_dict['aout.bias'.format(1)]=r"$\frac{{\partial a^{{(o)}}}}{{\partial b}}$"
 90
^{91}
      # Initialize weights
# h_theta.apply(kaiming_normal_init)
init_weight=False
^{92}
93
^{94}
95
96
       optimizer = optim.Adam(h_theta.parameters(), lr=0.001) # optimizer
97
98
      # prepare model and training parameters
99
      n_epochs = 300 # number of training epochs
batch_size = 5 # size of the mini batch
100
101
      batches_per_epoch = len(X_train) // batch_size
102
103
104
      best_acc = - np.inf  # init to negative infinity
105
      best_weights = None
       # track loss history
106
      train_loss_hist = []
107
       train_acc_hist = []
108
       test_loss_hist = []
109
       test_acc_hist = []
110
111
      norm_grad={n:[] for n,_ in h_theta.named_parameters()}
112
113
       # # Initialize weights
114
      # h_theta.apply(xavier_normal_init)
115
       # training loop over the epochs
116
      for epoch in range(n_epochs):
117
118
           epoch_loss = []
           epoch_acc = []
119
           epoch_norm_grad = {n:[] for n,_ in h_theta.named_parameters()}
120
            # set model in training mode
121
           h_theta.train()
122
123
           with tqdm.trange(batches_per_epoch, unit="batch", mininterval=0) as bar:
124
               bar.set_description(f"Epoch {epoch}")
125
               for i in bar:
126
127
                    # take a batch
                    start = i * batch_size
128
129
                    X_batch = X_train[start:start+batch_size]
                    y_batch = y_train[start:start+batch_size]
# infer (forward)
130
131
                    y_pred = h_theta(X_batch)
132
                      compute the loss
133
                    loss = loss_fn(y_pred, y_batch)
134
135
                    \ensuremath{\textit{\#}}\xspace reset previously saved gradients and empty the optimizer memory
136
                    optimizer.zero_grad()
137
                    \# run backward propagation
                    loss.backward()
138
                    # update weights
139
```

```
140
                    optimizer.step()
                    # compute and store metrics
141
                    acc = (torch.argmax(y_pred, 1) == torch.argmax(y_batch, 1)).float().mean()
142
                    epoch_loss.append(float(loss))
143
                    epoch_acc.append(float(acc))
144
145
                    bar.set_postfix(
                        loss=float(loss).
146
                        acc=float(acc)
147
                    )
148
                    # Update the norm of the grad per sample
update_grad_norm_batch = track_grad_norm(h_theta)
149
150
151
                    for p,ng in update_grad_norm_batch.items():
152
                      epoch_norm_grad[p] append(ng)
           # Compute the mean norm of the grad
153
           for p,ng in epoch_norm_grad.items():
154
             norm_grad[p].append(np.mean(ng))
155
           # set model in evaluation mode to infer the class in the test set
156
157
           # without storing gradients for brackprop
158
           h_theta.eval()
           # infer the class over the test set
y_pred = h_theta(X_test)
159
160
161
           ce = loss_fn(y_pred, y_test)
162
           acc = (torch.argmax(y_pred, 1) == torch.argmax(y_test, 1)).float().mean()
163
164
           ce = float(ce)
           acc = float(acc)
165
           train_loss_hist.append(np.mean(epoch_loss))
166
           train_acc_hist.append(np.mean(epoch_acc))
167
           test_loss_hist.append(ce)
168
169
           test_acc_hist.append(acc)
170
           # norm_grad.append(np.mean(epoch_norm_grad))
171
           # if acc > best_acc:
                  best_acc = acc
           #
172
           # best_weights = copy.deepcopy(h_theta.state_dict())
print(f"Epoch {epoch} validation: Cross-entropy={ce:.2f}, Accuracy={acc*100:.1f}%")
173
174
175
176
       # Plot the loss for train and test sets
      fig, ar = plt.subplots(figsize=(5,5))
ax.plot(train_loss_hist, label="train",color='steelblue',linewidth=3)
ax.plot(test_loss_hist, label="test",color='lightskyblue',linewidth=3,linestyle='--')
ax.set_xlabel(r"epochs")
177
178
179
180
       ax.set_ylabel(r"$\mathit{L}_{\mathcal{D}_{XY}}")
181
       ax.set_xlim(0,300)
182
183
       ax.set_ylim(0.0,1.2)
184
       ax.legend()
185
       if init_weight:
186
         fig.savefig("loss_classifier_init_{:>d}_{:>d}_{:>s}.png".format(nl, fan_in, act),
187
                      dpi=300, bbox_inches="tight")
188
189
       else:
         190
191
192
193
       # Plot the mean gradient norm
       colors = [plt.cm.jet(c) for c in np.linspace(0,1,len(list(norm_grad.keys()))) ]
194
195
196
       fig, ax = plt.subplots(figsize=(5,5))
       for i, (n, ng) in enumerate(norm_grad.items()):
197
       ax.semilogy(ng, label=latex_dict[n].color=colors[i],linewidth=3)
# ax.plot(test_loss_hist, label="test")
198
199
       ax.set_xlabel("epochs")
200
201
       202
       ax.set_xlim(0,300)
       ax.set_ylim(1e-5,1e2)
203
       ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.4),ncol=4,)
204
       if init_weight:
205
```

```
206 fig.savefig("norm_grad_classifier_init_{:>d}_{:>d}_{:>s}.png".format(nl, fan_in, act),
207 dpi=300, bbox_inches="tight")
208 else:
209 fig.savefig("norm_grad_classifier_{:>d}_{:>d}_{:>s}.png".format(nl, fan_in, act),
210 dpi=300, bbox_inches="tight")
```

4.2.1 Weight initialization

The weight initialization can play a major role case of vanishing gradient, since, depending on the activation function at stake, for deeper neural networks, it can occur that $\nabla_{w^{(1)}} L_{\mathcal{D}_{XY}} \approx 0$ in Equation (107), whenever the following condition occurs:

$$\frac{\partial a^{(N_\ell-\ell)}}{\partial h^{(N_\ell-\ell-1)}} = w^{(N_\ell-\ell)} \approx 0 \tag{111}$$

Weight initialization was firstly introduced in the seminal work of Bengio et al. [Ben+06]. The authors proposed to optimize the weights of a \mathcal{NN} by greedy unsupervised steps, i.e., by training one layer at time and freezing all the previous ones. This approach, by late 2000s, was the leading strategy to avoid trivial initializations, such as zero or constant initialization. The latter does not necessarily imply that the gradient vanishes, provided that the network is shallow though, as for instance in case of a \mathcal{MLP} $N_{\ell} = 1$. For the latter, Equation (107) simplifies in the following expression:

$$\frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_c^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h^{(1)}} \frac{\partial g^{(1)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial w_c^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial a^{(1)}} \cdot x_c \tag{112}$$

If two neurons are added to the \mathcal{MLP} with one hidden layer, Equation (112) becomes:

$$\int \frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_{1c}^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h_1^{(1)}} \frac{\partial h_1^{(1)}}{\partial a_1^{(1)}} \cdot x_c$$
(113)

$$\left|\frac{\partial L_{\mathcal{D}_{XY}}}{\partial w_{2c}^{(1)}} = \frac{\partial g^{(o)}}{\partial a^{(o)}} \cdot \frac{\partial a^{(o)}}{\partial h_2^{(1)}} \frac{\partial h_2^{(1)}}{\partial a_2^{(1)}} \cdot x_c$$
(114)

with $\mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$, and $h_i^{(1)} = g(a_i^{(1)})$ (see Figure 1). In this case, the trivial initialization $\mathbf{W} = cst$. and $\mathbf{b}^{(1)} = cst$. does not make the gradient vanish (if and only if $N_{\ell} = 1$) but since $a_1^{(1)} = a_2^{(1)} = 0$, then $h_1^{(1)} = g(a_1^{(1)}) = h_1^{(1)} = g(a_1^{(1)})$, so that $w_{1c}^{(1)(i)} = w_{2c}^{(1)(i)}$ until convergence. This symmetric evolution of the weights is rather restrictive though, since it cannot be prevented after the first gradient descent iteration.

A smarter weight initialization strategy is to sample them from the standard normal or uniform distribution. However, this strategy can incur into either vanishing gradients, because the activation function can saturate for large random samples (see Equation (109)) or because the weight value is too small (see Equation (111)). When randomly sampled, the weight value is dominated by

the choice of their variance. In other words, the optimum sampling solution would be an unbiased random sampling with the "optimum" variance that avoid vanishing gradients. The zero-mean avoids introducing spurious bias shifts (see Section 5.2 and [CUH16]) and asymmetric weight configurations. The "optimum" variance should be chosen in order to decorrelate weights, biases and input in the pre-activation output $a_i^{(k)}$ at each layer k (see Equation (97)), so that its variance reads:

$$\mathbb{V}(a_i^{(k)}) = \mathbb{V}\left(\sum_{j=1}^{u^{(k)}} W_{ij}^{(k)} h_j^{(k-1)}\right) + \mathbb{V}\left(b_i^{(k)}\right)$$
(115)

with

$$\mathbb{V}\left(\sum_{j=1}^{u^{(k)}} W_{ij}^{(k)} h_j^{(k-1)}\right) = \sum_{j=1}^{u^{(k)}} \mathbb{V}\left(W_{ij}^{(k)}\right) \cdot \left(\mathbb{E}\left[h_j^{(k-1)}\right]\right)^2 + \\ + \sum_{j=1}^{u^{(k)}} \left(\mathbb{E}\left[W_{ij}^{(k)}\right]\right)^2 \cdot \mathbb{V}\left(h_j^{(k-1)}\right) + \\ + \sum_{j=1}^{u^{(k)}} \mathbb{V}\left(W_{ij}^{(k)}\right) \cdot \mathbb{V}\left(h_j^{(k-1)}\right)$$
(116)

If one assumes that unbiased weights, biases and inputs, with constant variances $\mathbb{V}(W^{(k)})$, $\mathbb{V}(b^{(k)})$ and $\mathbb{V}(h^{(k-1)})$ respectively, Equation (116) can be rewritten as:

$$\mathbb{V}\left(a_{i}^{(k)}\right) = \left(u^{(k)} \cdot \mathbb{V}(W^{(k)})\right)^{i} \cdot \mathbb{V}(h^{(k-1)}) + \mathbb{V}\left(b_{i}^{(k)}\right)$$
(117)

In order to assure that weights, biases and inputs are decorrelated, Equation (117) must hold, by avoiding the quantity $(u^{(k)} \cdot \mathbb{V}(W^{(k)}))^i$ to vanish or explode. In order to do so, the most effective strategy consists into initializing the biases to 0 and the weight variance as follows:

$$\mathbb{V}(W^{(k)}) = \frac{1}{u^{(k)}} \Rightarrow \mathbb{V}\left(a_i^{(k)}\right) = \mathbb{V}(h^{(k-1)}) + \mathbb{V}\left(b_i^{(k)}\right)$$
(118)

or, in order to account for both forward and backward propagation, the Xavier (or Glorot) initialization proposed by [GB10] is preferred:

$$\mathbb{V}(W^{(k)}) = \frac{2}{u^{(k)} + u^{(k+1)}} \tag{119}$$

Xavier (or Glorot) initialization in Equation (119) avoids the pre-activation variance to explode or to vanish both in forward and in backward propagation, with opposite flow of information. This initialization grants a good stability of the training scheme. If the weights are sampled with standard normal distribution, Equation (119) provides the standard deviation. If a uniform distribution

 $\mathcal{U}(-c,c)$ is chosen, its variance being $\frac{c^2}{3}$, then the end points can be selected as $\pm \sqrt{\frac{6}{u^{(k)}+u^{(k+1)}}}$.

However, [Kum17] proved that Xavier (or Glorot) initialization seems to highly impact the backward propagation of \mathcal{NN} featured by ReLU activation functions, due to the asymmetric nature of this activation function that is identically zero for negative argument. Therefore, [He+15] proposed to initialize the weight variance as follows:

$$\mathbb{V}(W^{(k)}) = \frac{2}{u^{(k)}} \tag{120}$$

with end point for uniform distribution $\pm \sqrt{\frac{6}{u^{(k)}}}$. This strategy goes under the name of He (or Kaiming) initialization. For both Xavier (or Glorot) and He (or Kaiming) initialization, a gain can be multiplied to the variance initialization.

Example 20. Different weight initialization techniques for a multi-class classifier with PyTorch.

In the literature, the number of input units $u^{(k)}$ and the number of output units $u^{(k+1)}$ are referred as to fan_{in} and fan_{out} respectively. PyTorch class torch.nn.init¹¹, adopted to perform weight initialization, follows the same notation. The following example shows the weight distribution for different initializations, for a 1-layer \mathcal{MLP} with $fan_{in} = 100$ and $fan_{out} = 100$.

```
import matplotlib.pyplot as plt
 1
      import seaborn as sns
 2
 3
      import numpy as np
 ^{4}
       # import pandas as pd
 \mathbf{5}
      import torch
 6
      import torch.nn as nn
 7
       # To assure reproductibility, fix the random seed
 8
      torch.manual_seed(0) # to assure reproductibility
 9
10
       # define the 1-layer MLP
11
      h_theta = nn.Sequential()
12
      h_theta.add_module('a1', nn.Linear(100, 100))
h_theta.add_module('g1', nn.ReLU())
13
14
15
       # define initialization functions
16
17
      def constant_init(m):
18
          if type(m) == nn.Linear:
               torch.nn.init.constant_(m.weight, 2)
19
20
^{21}
      def uniform init(m):
22
           if type(m) == nn.Linear:
^{23}
               torch.nn.init.uniform_(m.weight,a=-0.5, b=0.5)
^{24}
25
      def normal_init(m):
           if type(m) == nn.Linear:
26
27
               torch.nn.init.normal_(m.weight)
28
^{29}
      def xavier_uniform_init(m):
30
           if type(m) == nn.Linear:
31
               torch.nn.init.xavier_uniform_(m.weight)
32
```

¹¹https://pytorch.org/docs/stable/nn.init.html

```
def xavier_normal_init(m):
    if type(m) == nn.Linear:
33
34
                 torch.nn.init.xavier_normal_(m.weight)
35
36
       def kaiming_uniform_init(m):
37
38
            if type(m) == nn.Linear:
                 torch.nn.init.kaiming_uniform_(m.weight)
39
40
      def kaiming_normal_init(m):
^{41}
           if type(m) == nn.Linear:
^{42}
                 torch.nn.init.kaiming_normal_(m.weight)
^{43}
^{44}
45
      # Applying different initializations to model
46
47
       # constant
^{48}
      h_theta.apply(constant_init)
49
       constant_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
50
       constant_weights = torch.concatenate(constant_weights).detach().cpu().numpy()
51
52
53
       # uniform
54
      h_theta apply(uniform_init)
      uniform_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
uniform_weights = torch.concatenate(uniform_weights).detach().cpu().numpy()
55
56
57
58
       # normal.
      h_theta.apply(normal_init)
59
      normal_weights = torch.concatenate(normal_weights).detach().cpu().numpy()
60
61
62
       # xavier uniform
63
64
      h_theta.apply(xavier_uniform_init)
      m_theta.appry(xavid_unifor_unifor_unifor_xavie_unifor_weights = fm.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
xavier_uniform_weights = torch.concatenate(xavier_uniform_weights).detach().cpu().numpy()
65
66
67
68
       # xavier normal
69
      h_theta.apply(xavier_normal_init)
      xavier_normal_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
xavier_normal_weights = torch.concatenate(xavier_normal_weights).detach().cpu().numpy()
70
71
72
73
       # kaiming uniform
^{74}
       h_theta.apply(kaiming_uniform_init)
      kaiming_uniform_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
kaiming_uniform_weights = torch.concatenate(kaiming_uniform_weights).detach().cpu().numpy()
75
76
77
78
       # kaiming normal
79
       h_theta.apply(kaiming_normal_init)
       kaiming_normal_weights = [m.weight.flatten() for m in h_theta.modules() if type(m)==nn.Linear]
80
       kaiming_normal_weights = torch.concatenate(kaiming_normal_weights).detach().cpu().numpy()
81
82
       weights = {"uniform": uniform_weights,
83
                     "normal": normal_weights,
84
                    "Xavier uniform": xavier_uniform_weights,
85
                    "Xavier normal": xavier_normal_weights,
86
                    "Kaiming uniform": kaiming_uniform_weights,
87
88
                    "Kaiming normal": kaiming_normal_weights}
89
       # plot histograms
90
      fig, ax = plt.subplots(nrows=2, ncols=3, sharex=True, sharey=True)
91
92
93
       ax=ax.flatten()
^{94}
95
       for a, (k,v) in enumerate(weights.items()):
96
         # compute histogram
         counts, bins = np.histogram(v, bins=v.size//10)
97
         ax[a].hist(bins[:-1], bins, weights=counts,
98
```

```
Gatti 245
```

```
align='left', color='k', density=True,
 99
100
             log=True)
101
           ax[a].set_title(k)
102
           if a>2:
             ax[a].set_xlabel(r"$W_{ij}$")
103
          if a==0 or a==3:
    ax[a].set_ylabel("#")
    ax[a].set_xlim(-3.0,3.0)
104
105
106
107
          ax[a].set_ylim(0,5.0)
108
        fig.savefig("weight_initialization.png",dpi=300,bbox_inches='tight')
```





As shown in Figure 46, the weight distribution for Kaiming is highly affected by the value of fan_{in} , whereas the other distributions remain stable, since they either do depend on both fan_{in}

References

- [ACB17] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks". In: International conference on machine learning. PMLR, 2017, pp. 214–223.
- [Bac17] Bach, Francis. "Breaking the curse of dimensionality with convex neural networks". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 629–681.
- [Bar93] Barron, Andrew R. "Universal approximation bounds for superpositions of a sigmoidal function". In: *IEEE Transactions on Information theory* 39.3 (1993), pp. 930–945.
- [Ben+06] **Bengio, Yoshua et al.** "Greedy layer-wise training of deep networks". In: Advances in neural information processing systems 19 (2006).
- [BSF94] Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.
- [Cam19] Campagne, J.E. L'apprentissage par réseaux de neurones profonds. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2019). 2019. DOI: https://doi.org/10. 4000/annuaire-cdf.16767. URL: https://www.di.ens.fr/ ~mallat/College/Cours-2019-Mallat-Jean-Eric-Campage. pdf.
- [Cam20] Campagne, J.E. Modè les multi-échelles et réseaux de neurones convolutifs. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2020). 2020. URL: https://www. di.ens.fr/~mallat/College/Cours2020-Mallat-Jean-Eric-Campagne.pdf.
- [CBB97] Cun, Yann Le, L. Bottou, and Y. Bengio. "Reading checks with multilayer graph transformer networks". In: 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing. Vol. 1. 1997, 151–154 vol.1. DOI: 10.1109/ICASSP.1997.599580.
- [CGL21] Castro-Cruz, David, Filippo Gatti, and Fernando Lopez-Caballero. "Assessing the impact of regional geology on the ground motion model variability at the Kashiwazaki-Kariwa Nuclear Power Plant (Japan) via physics-based numerical simulation". en. In: Soil Dynamics and Earthquake Engineering 150 (Nov. 2021), p. 106947. ISSN: 02677261. DOI: 10.1016/j.soildyn.2021.106947. URL: https://linkinghub.elsevier.com/retrieve/pii/S0267726121003699.
- [Che+14] Chen, Liang-Chieh et al. "Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs". en. In: arXiv:1412.7062 (June 2014). arXiv:1412.7062 [cs]. URL: http://arxiv.org/abs/ 1412.7062.
- [CUH16] Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponen-
tial Linear Units (ELUs)". en. In: arXiv:1511.07289 (Feb. 2016). arXiv:1511.07289 [cs]. URL: http://arxiv.org/abs/1511.07289.

- [DHS11] Duchi, John, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: J. Mach. Learn. Res. 12.null (July 2011), pp. 2121–2159. ISSN: 1532-4435.
- [DV18] Dumoulin, Vincent and Francesco Visin. "A guide to convolution arithmetic for deep learning". en. In: arXiv:1603.07285 (Jan. 2018). arXiv:1603.07285 [cs, stat]. URL: http://arxiv.org/abs/1603.07285.
- [ES16] Eldan, Ronen and Ohad Shamir. "The power of depth for feedforward neural networks". In: Workshop and Conference Proceedings. Vol. 49. PMLR, 2016, pp. 1–34.
- [Fey18] Feydy, J. Automatic differentiation for applied mathematicians. Tech. rep. ENS Paris et Paris Saclay, 2018. URL: https://www. ljll.math.upmc.fr/gtt/beamers/GTT_13-03-2018_Feydy.pdf.
- [Fra+22] Fragemann, Jana et al. Review of Disentanglement Approaches for Medical Applications: Towards Solving the Gordian Knot of Generative Models in Healthcare. en. Mar. 2022. DOI: 10.36227/ techrxiv.19364897.v1. URL: https://www.techrxiv.org/ articles/preprint/Review_of_Disentanglement_Approaches_ for_Medical_Applications_Towards_Solving_the_Gordian_ Knot_of_Generative_Models_in_Healthcare/19364897/1.
- [GB10] Glorot, Xavier and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.
- [GBB11] Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323. URL: https: //proceedings.mlr.press/v15/glorot11a.html.
- [GBC16] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.
- [GMH13] Graves, Alex, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks".
 en. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing. Vancouver, BC, Canada: IEEE, May 2013, pp. 6645–6649. ISBN: 978-1-4799-0356-6. DOI: 10.1109/ICASSP.

2013.6638947. URL: http://ieeexplore.ieee.org/document/6638947/.

- [GSC00] Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins.
 "Learning to Forget: Continual Prediction with LSTM". en. In: Neural Computation 12.10 (Oct. 2000), pp. 2451–2471. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/089976600300015015. URL: https: //direct.mit.edu/neco/article/12/10/2451-2471/6415.
- [Hay04] Haykin, Simon. Kalman filtering and neural networks. John Wiley & Sons, 2004.
- [Hay98] Haykin, Simon. Neural networks: a comprehensive foundation. Prentice Hall PTR, 1998.
- [He+15] He, Kaiming et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". en. In: arXiv:1502.01852 (Feb. 2015). arXiv:1502.01852 [cs]. URL: http://arxiv.org/abs/ 1502.01852.
- [HS15] **He, Kaiming and Jian Sun**. "Convolutional Neural Networks at Constrained Time Cost". In: *Proceedings of the IEEE Conference* on Computer Vision and Pattern Recognition (CVPR). June 2015.
- [HS97] Hochreiter, Sepp and Jürgen Schmidhuber. "Long Short-Term Memory". en. In: Neural Computation 9.8 (Nov. 1997), pp. 1735– 1780. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1997.9.8.
 1735. URL: https://direct.mit.edu/neco/article/9/8/1735-1780/6109.
- [KB15] Kingma, Diederik P. and Jimmy Ba. "Adam: A Method for Stochastic Optimization". en. In: 3rd International Conference for Learning Representations arXiv:1412.6980 (Jan. 2015). arXiv:1412.6980 [cs]. URL: http://arxiv.org/abs/1412.6980.
- [Kla+17] Klambauer, Günter et al. "Self-Normalizing Neural Networks".
 en. In: arXiv:1706.02515 (Sept. 2017). arXiv:1706.02515 [cs, stat].
 URL: http://arxiv.org/abs/1706.02515.
- [KM18] Kim, Hyunjik and Andriy Mnih. "Disentangling by factorising". In: International Conference on Machine Learning. PMLR, 2018, pp. 2649–2658.
- [KSH17] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton.
 "ImageNet classification with deep convolutional neural networks".
 en. In: Communications of the ACM 60.6 (May 2017), pp. 84–90.
 ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386. URL: https: //dl.acm.org/doi/10.1145/3065386.
- [Kum17] Kumar, Siddharth Krishna. "On weight initialization in deep neural networks". en. In: arXiv:1704.08863 (May 2017). arXiv:1704.08863 [cs]. URL: http://arxiv.org/abs/1704.08863.
- [LeC+98] LeCun, Yann et al. "Efficient BackProp". In: Neural Networks: Tricks of the Trade. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Vol. 1524. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. ISBN: 978-3-

540-65311-0. DOI: 10.1007/3-540-49430-8_2. URL: http://link.springer.com/10.1007/3-540-49430-8_2.

- [LH15] Liang, Ming and Xiaolin Hu. "Recurrent convolutional neural network for object recognition". In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2015, pp. 3367– 3375.
- [LJH15] Le, Quoc V, Navdeep Jaitly, and Geoffrey E Hinton. "A simple way to initialize recurrent networks of rectified linear units". In: arXiv preprint arXiv:1504.00941 (2015).
- [Mai99] Maiorov, V.E. "On Best Approximation by Ridge Functions". en. In: Journal of Approximation Theory 99.1 (July 1999), pp. 68-94. ISSN: 00219045. DOI: 10.1006/jath.1998.3304. URL: https:// linkinghub.elsevier.com/retrieve/pii/S0021904598933044.
- [MMD20] Mouton, Coenraad, Johannes C. Myburgh, and Marelie H. Davel. "Stride and Translation Invariance in CNNs". en. In: vol. 1342. arXiv:2103.10097 [cs]. 2020, pp. 267–281. DOI: 10.1007/ 978-3-030-66151-9_17. URL: http://arxiv.org/abs/2103. 10097.
- [Mni+16] Mnih, Volodymyr et al. "Asynchronous methods for deep reinforcement learning". In: International conference on machine learning. PMLR, 2016, pp. 1928–1937.
- [MP43] McCulloch, Warren S. and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". en. In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 0007-4985, 1522-9602. DOI: 10.1007/BF02478259. URL: http://link. springer.com/10.1007/BF02478259.
- [Pér+03] Pérez-Ortiz, Juan Antonio et al. "Kalman filters improve LSTM network performance in problems unsolvable by traditional recurrent nets". In: *Neural Networks* 16.2 (2003), pp. 241–250.
- [Pey20] Peyré, Gabriel. Course notes on Optimization for Machine Learning. Notes de cours de l'École Normale Supérieure. 2020. URL: https://mathematical-tours.github.io.
- [PMB13] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: International conference on machine learning. Pmlr, 2013, pp. 1310–1318.
- [Pol64] Polyak, B.T. "Some methods of speeding up the convergence of iteration methods". en. In: USSR Computational Mathematics and Mathematical Physics 4.5 (Jan. 1964), pp. 1–17. ISSN: 00415553. DOI: 10.1016/0041-5553(64)90137-5. URL: https://linkinghub.elsevier.com/retrieve/pii/0041555364901375.
- [ROS57] ROSENBLATT, MURRAY. "Some Purely Deterministic Processes". In: Journal of Mathematics and Mechanics 6.6 (1957), pp. 801-810. ISSN: 00959057, 19435274. URL: http://www.jstor. org/stable/24900623.

- [Rou72] Rousseau, Jean-Jacques. Discours sur l'origine et les fondemens de l'inégalité parmi les hommes. Vol. 1. chez Marc Michel Rey, 1772.
- [Sal+18] Salehinejad, Hojjat et al. "Recent Advances in Recurrent Neural Networks". en. In: arXiv:1801.01078 (Feb. 2018). arXiv:1801.01078
 [cs]. URL: http://arxiv.org/abs/1801.01078.
- [Smi17] Smith, Leslie N. "Cyclical Learning Rates for Training Neural Networks". en. In: 2017 IEEE Winter Conference on Applications of Computer Vision (WACV). Santa Rosa, CA, USA: IEEE, Mar. 2017, pp. 464–472. ISBN: 978-1-5090-4822-9. DOI: 10.1109/WACV.2017.58. URL: http://ieeexplore.ieee.org/document/7926641/.
- [SP97] Schuster, Mike and Kuldip K Paliwal. "Bidirectional recurrent neural networks". In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.
- [Sut+13] Sutskever, Ilya et al. "On the importance of initialization and momentum in deep learning". In: Proceedings of the 30th International Conference on Machine Learning. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1139– 1147. URL: https://proceedings.mlr.press/v28/sutskever13. html.
- [TH12] Tieleman, Tijmen and Geoffrey Hinton. "Lecture 6.5-rmsprop, coursera: Neural networks for machine learning". In: University of Toronto, Technical Report 6 (2012).
- [Via22] Vialle, Stéphane. Séquence Thématique ST7-76 Similation à haute performance (2021-2022). "Ingénieur" Curriculum, Centrale-Supélec. 2022.
- [Wer90] Werbos, P.J. "Backpropagation through time: what it does and how to do it". en. In: Proceedings of the IEEE 78.10 (Oct. 1990), pp. 1550–1560. ISSN: 00189219. DOI: 10.1109/5.58337. URL: http: //ieeexplore.ieee.org/document/58337/.
- [Whi51] Whittle, Peter. Hypothesis testing in time series analysis. Vol. 4. Almqvist and Wiksells boktr., 1951.
- [Wil92] Williams, Ronald J. "Training recurrent networks using the extended Kalman filter". In: International joint conference on neural networks. Vol. 4. Citeseer, 1992, pp. 241–246.
- [WJ16] Wang, Shuohang and Jing Jiang. "Learning Natural Language Inference with LSTM". en. In: arXiv:1512.08849 (Nov. 2016). arXiv:1512.08849 [cs]. URL: http://arxiv.org/abs/1512.08849.
- [Xue+10] Xuemei, Li et al. "Hybrid support vector machine and ARIMA model in building cooling prediction". In: 2010 International Symposium on Computer, Communication, Control and Automation (3CA). Vol. 1. 2010, pp. 533–536. DOI: 10.1109/3CA.2010. 5533864.

- [YK15] Yu, Fisher and Vladlen Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions". en. In: arXiv:1511.07122 (Apr. 2015). arXiv:1511.07122 [cs]. URL: http://arxiv.org/abs/1511.07122.
- [Zei+10] Zeiler, Matthew D. et al. "Deconvolutional networks". In: 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2010, pp. 2528–2535. DOI: 10.1109/CVPR.2010. 5539957.
- [Zei12] Zeiler, Matthew D. "ADADELTA: An Adaptive Learning Rate Method". en. In: arXiv:1212.5701 (Dec. 2012). arXiv:1212.5701 [cs]. URL: http://arxiv.org/abs/1212.5701.

Artificial Neural Networks: advanced topics

Filippo Gatti

Université Paris-Saclay CentraleSupélec, ENS Paris-Saclay, CNRS LMPS Laboratoire de Mécanique Paris Saclay UMR 9026

This chapter is a follow-up of the basic introduction provided in the chapter Artificial Neural Networks: layer architectures, optimizers and automatic differentiation, which prepares the readership to the following theoretical insights. In particular, the chapter rephrases the machine learning problem according to an information theory paradigm, that highlights the deep entanglement between the data science perspectives: the probabilistic and the deterministic one. Moreover, the chapter describes the fundamental theoretical result that paved the way to modern machine learning: the universal approximation theorem for a 1-hidden-layer perceptron. This section is followed by a continuum mechanics interpretation of convolutional neural networks, proving why convolutional layers are fundamental in image classification. Finally, further insights on the optimization of a neural network are provided, focusing on the convergence of first-order gradient descent methods. Finally, the automatic differentiation is explained in analogy with tensor algebra, along with some advanced strategies to avoid vanishing gradients in back-propagation algorithms. Some subsections are tagged as [RECAP], since they are meant to refresh the readership's basics on optimization and signal processing fundamentals. The chapter is largely inspired, among others, by Stéphane Mallat's Data Science lecture notes at Collège de France, as well as by different lecture notes of CentraleSupélec's engineering curriculum.

1 Information theory

The theory of information represents a fundamental chapter of the long way to modern machine learning. The inception of such a theory is due to Ronald Fisher and to Claude Shannon, who laid the theoretical framework in their famous works On the mathematical foundations of theoretical statistics [Fis22]

and A Mathematical Theory of Communication [Sha48]. In the following, the two perspectives will be briefly introduced.

1.1 Basics of Measure Theory [RECAP]

This section is a summary of the *Statistics and Learning* class of Centrale-Supélec [Cou20].

In the following, consider a multi-variate random variable defined over the probability space $\boldsymbol{X} : (\Omega, \mathcal{E}, \mathbb{P}) \to (\mathcal{X}, \Xi)$, with a σ -algebra $\mathcal{E} \in \mathcal{B}(\mathbb{R})$ on Ω ($\mathcal{B}(\mathbb{R})$) are the Borel's sets) and an unknown probability law \mathbb{P} such that $\forall A \in \mathcal{E}_{\mathcal{X}}$, $P_X(A) = \mathbb{P}(\boldsymbol{X}^{-1}(A)) = \mathbb{P}(\boldsymbol{X} \in A)$. If the random variable is continuous, $\mathcal{X} \subset \mathbb{R}^{d_X}$ and $\Xi = \mathcal{B}(\mathcal{X})$; if instead \mathcal{X} is of finite cardinality or countable, $\Xi = \mathcal{P}(\mathcal{X})$, the power set of \mathcal{X} .

Moreover, we assume that P_X is σ -finite measure dominated by a σ -finite measure μ , i.e. if $\forall \varepsilon > 0, \exists \delta(\varepsilon)$ such that $P_X(A) < \varepsilon, \forall A \in \Xi$ such that $\mu(A) < \delta(\varepsilon)$. In other words, the negligible sets for μ are negligible for P_X too. In this framework, P_X is absolutely continuous with the respect to μ and famous the Radon-Nikodym theorem holds (see also [Bil95; Cou20]):

Theorem 1. Radon-Nikodym theorem

For two σ -finite measures on a sigma algebra Ξ , namely μ and P_X such that P_X is absolutely continuous with the respect to μ , $\exists p_X \in L^1(\mu)$ such that

$$P_X(A) = \int_A p_X(\boldsymbol{x}) \cdot \mu(d\boldsymbol{x}), \qquad \forall A \in \Xi$$

 p_x is called probability density of P_X and it corresponds to the Radon-Nikodym derivative $p_X = \frac{dP_X}{d\mu}$.

In the following, we will consider only Lebesgue reference measures μ for continuous random variables, i.e.:

$$P_X(A) = \int_A p_X(\boldsymbol{x}) \cdot \mu(d\boldsymbol{x}), \qquad \forall A \in \Xi = \mathcal{B}(\mathbb{R}^{d_X})$$
(1)

and countable reference measures for discrete random variables. In this latter case, the probability distribution is discrete and it corresponds to the point mass function corresponding to a counting measure on a subset the power set $\Xi = \mathcal{P}(\mathcal{X})$, such that:

$$P_X(A) = \int_A p_X(\boldsymbol{x}) \cdot \mu(d\boldsymbol{x}) = \int_A p(\boldsymbol{x}) \cdot \sum_{\boldsymbol{x}_i \in A} \delta_{\boldsymbol{x}_i}(d\boldsymbol{x}) =$$
$$= \sum_{\boldsymbol{x}_i \in A} P_X(\boldsymbol{X} = \boldsymbol{x}_i), \forall A \in \Xi = \mathcal{P}(\mathcal{X})$$
(2)

with $\sum_{\boldsymbol{x}_i \in \mathcal{X}} P_X(\boldsymbol{X} = \boldsymbol{x}_i) = 1.$

1.2Reminders of basic statistics [RECAP]

Theorem 2. Strong law of large numbers (Kolmogorov) Given a set of random variables $\mathbf{X} = (X_i)_{i=1}^N$ independent identically dis-tributed (i.i.d.) and Lebesgue integrable, with expected value $\mu_X < +\infty$, and a sample average $\bar{X}_N = \frac{1}{N} \sum_{i=1}^N X_i$, then :

$$\mathbb{P}\left[\lim_{N \to +\infty} \bar{X}_n - \mu_X\right] = 1$$

which means that the sample average \bar{X}_N converges almost surely to the expected value μ .

Theorem 3. Weak law of large numbers (Khintchine)

Given a set of random variables $\mathbf{X} = (X_i)_{i=1}^{N}$ independent identically distributed (i.i.d.) and Lebesgue integrable, with expected value $\mu_X < +\infty$, and a sample average $\bar{X}_N = \frac{1}{N} \sum_{i=1}^{N} X_i$, then :

$$\forall \varepsilon > 0, \quad \lim_{N \to +\infty} \mathbb{P}\left[|\bar{X}_n - \mu_X| \le \varepsilon \right] = 1$$

which means that the sample average \bar{X}_N converges in probability to μ

Proof. The theorem proof is based on the Chebyshev's inequality, that states that for a wide class of probability distributions, only a certain fraction of values at a certain distance from the expected value a probability to occur larger than the distance itself:

$$\mathbb{P}\left[|X-\mu| \ge \varepsilon\right] \le \frac{\sigma_X^2}{\varepsilon^2} \tag{3}$$

Applying Equation (3) to the estimator \bar{X}_N (which is a linear combination of random variables and has a variance $\sigma_{\bar{X}_N}^2 = \frac{\sigma_X^2}{N}$, one obtains:

$$\begin{aligned} \forall \varepsilon > 0 \quad 1 \ge \lim_{N \to +\infty} \mathbb{P}\left[|\bar{X}_n - \mu_X| \le \varepsilon \right] = \\ = 1 - \lim_{N \to +\infty} \mathbb{P}\left[|\bar{X}_n - \mu_X| > \varepsilon \right] \ge 1 - \lim_{N \to +\infty} \frac{\sigma_X^2}{N\varepsilon^2} \end{aligned}$$

which proves the convergence in probability. The estimator \bar{X}_N of μ_X is consistent.

Definition 4. Consistent estimator

An estimator $\boldsymbol{\theta}_N = \boldsymbol{\theta}(X_1, \dots, X_N)$ is said to be *consistent* if it converges in probability to its limit $\boldsymbol{\theta}$.

Theorem 5. Given a series of random variables $(X_i)_{i=1}^N$ that converges in probability to X, then any continuous function $f : \mathcal{X} \to \mathbb{R}$ converges in probability too.

Proof. The continuity of f is expressed by the following expression:

$$\forall \varepsilon > 0, \exists \delta(\varepsilon) > 0 | \quad |f(X_N) - f(X)| \le \varepsilon, \forall X_N \in [X - \delta; X + \delta]$$

Since X_N converges to X in probability, the following inequality holds:

$$1 \ge \mathbb{P}\left[|f(X_N) - f(X)| \le \varepsilon\right] \ge \mathbb{P}\left[|X_N) - X| \le \delta\right]$$

Taking the limit $N \to +\infty$ of the last inequality:

$$1 \ge \lim_{N \to +\infty} \mathbb{P}\left[|f(X_N) - f(X)| \le \varepsilon \right] \ge 1$$

which proves the statement.

Remark 6. Since \bar{X}_N (defined in Theorem 3) is a consistent estimator of μ_X , then thanks to Theorem 5, \bar{X}_N^2 converges in probability to μ_X and the estimator $\overline{Y}_N = \frac{1}{N} \sum_{i=1}^N X_i^2$ converges in probability to $\mathbb{E}[X_i^2]$. Therefore, the variance estimator of each $X_i s_N^2$ that reads:

$$s_N^2 = \frac{1}{N} \sum_{i=1}^{N} \left(X_i - \bar{X}_N \right)^2 \tag{4}$$

converges in probability to the variance of X_i :

$$\lim_{N \to +\infty} \mathbb{P}\left(|s_N^2 - \sigma_{X_i}^2| \le \varepsilon\right) =$$

$$= \lim_{N \to +\infty} \mathbb{P}\left[\left|\frac{1}{N}\sum_{i=1}^N \left(X_i^2 + \bar{X}_N^2 - 2X_i\bar{X}_N\right) - \sigma_{X_i}^2\right| \le \varepsilon\right] =$$

$$= \lim_{N \to +\infty} \mathbb{P}\left[|\bar{Y}_N - X_N^2 - \sigma_{X_i}^2| \le \varepsilon\right] =$$

$$= \lim_{N \to +\infty} \mathbb{P}\left[|\bar{Y}_N - X_N^2 - \mathbb{E}\left[X_i^2\right] + \mu_X| \le \varepsilon\right] = 1$$
(5)

Definition 7. Convergence in distribution A series of random variables $(X_i)_{i=1}^N$ sampled from a probability distribution $p_N\left((X_i)_{i=1}^N\right)$ converges to the probability distribution p(X) for $N \to +\infty$ if $\forall a | P_X(A = \{X \le a\})$ is continuous and $\lim_{N \to +\infty} P_N(A = \{X \le a\}) = P_X(a)$ with $P_N(A = \{X \le a\}) = \int_A p_N(x) \mu(dx)$

Theorem 8. Central limit theorem

Given a set of i.i.d. random variables $\mathcal{D}_X = \{X_i\}_{i=1}^N$ with expected value $\mathbb{E}_{X_i} = \mu$ and variance $\mathbb{V}[X_i] = \sigma^2 < +\infty$ and a random variable $\bar{X}_N = \frac{1}{N} \sum_{i=1}^N X_i$, the random variable

$$Z_N = \sqrt{N} \frac{X_N - \mu}{\sigma}$$

has zero expected value, unit variance and its probability distribution $p(Z_N)$ converges in distribution to a standard normal distribution $\mathcal{N}(0,1)$

Proof. From Theorem 3, the weak law of large numbers grants that \bar{X}_N converges in probability to μ and its variance $\mathbb{V}\left[\bar{X}_N\right] = \frac{\sigma^2}{N}$. The *characteristic function* of the function $Y_j = \frac{X_j - \mu}{\sigma}$ has a Taylor expansion around its zero expected value that reads:

$$\varphi_{Y_j}(t) = \mathbb{E}\left[e^{itY_j}\right] = 1 - \frac{t^2}{2} + o(t^2)$$

Therefore, the characteristic function of Z_N reads:

$$\varphi_{Z_N}(t) = \mathbb{E}\left[e^{itZ_N}\right] = \prod_{j=1}^N \mathbb{E}\left[e^{it\frac{Y_j}{\sqrt{N}}}\right] = \left(\varphi_{Y_1}\left(\frac{t}{\sqrt{N}}\right)\right)^N = \left(1 - \frac{t^2}{2N} + o\left(\frac{t^2}{N}\right)\right)^N$$

For $N \to +\infty$ the following results holds:

$$\lim_{N \to +\infty} \varphi_{Z_N}(t) = \lim_{N \to +\infty} \left(1 - \frac{t^2}{2N} + o(\frac{t^2}{N}) \right)^N =$$
$$= e^{-\frac{t^2}{2}} = \mathbb{E}_{\xi \sim \mathcal{N}(0,1)} \left[e^{it\xi} \right] = \varphi_{\xi}(t), \quad t \in \mathbb{R}$$

In other words, the characteristic function of Z_N converges to the one of a random variable with standard normal probability distribution. Thanks to Lévy theorem, the probability distribution $p(Z_N)$ converges in distribution to $\mathcal{N}(0, 1)$.

1.3 The Fisher's approach

Fisher [Fis22] firstly stated the concept of *information* for the *inference* problem stated in the following. Provided a family of parametric probability law, called statistical model $\mathcal{H}_{\boldsymbol{\theta}} := \{P_{\boldsymbol{\theta}}, \boldsymbol{\theta} \in \Theta\}$, if $P \in \mathcal{H}_{\boldsymbol{\theta}}$, there exist $\boldsymbol{\theta}^*$ such that $P_{\boldsymbol{\theta}^*} = P$. $\boldsymbol{\theta}^*$ is unique if and only if $\mathcal{H}_{\boldsymbol{\theta}}$ is identifiable, i.e., if the map $\boldsymbol{\theta} \mapsto P_{\boldsymbol{\theta}}$ is injective. In this sense, the value of $\boldsymbol{\theta}$ is an index to identify any probability distribution in $\mathcal{H}_{\boldsymbol{\theta}}$. Fisher proposed to discover P by finding the corresponding value of the parameter set $\hat{\boldsymbol{\theta}}$ that make $P_{\hat{\boldsymbol{\theta}}}$ the closest possible to $P = P_{\boldsymbol{\theta}^*}$. Fisher proposed to infer the "best" estimator $\hat{\boldsymbol{\theta}}$ from N realizations of observed

variable at stake X, collected in the data set $\mathcal{D}_X = \{x_i \in \mathcal{X}\}_{i=1}^N$. Therefore, the $\hat{\theta}$ depends on \mathcal{D}_X . In the remainder of the chapter, \mathcal{X} is assumed to be isomorphic to \mathbb{R}^{d_X} , Θ a Borel set of \mathbb{R}^p and for the sake of simplicity we assume \mathcal{H}_{θ} as dominated by a Lebesgue or discrete σ -finite measure μ and represented by the probability density functions p_{θ} . What is the *best* estimator $\hat{\theta}(\mathcal{D}_X)$ from which one can infer $p(\mathcal{E}_{\mathcal{X}})$? In other words, what is the most parsimonious choice of θ in order to span the whole $\mathcal{E}_{\mathcal{X}}$? What is the information about $\mathcal{E}_{\mathcal{X}}$ contained in \mathcal{D}_X and how we can extract it?

First, some regularity conditions for the statistical model \mathcal{H}_{θ} must be defined [Bil95; Cou20]:

- C1 Θ is an open set and $p_{\theta}(\boldsymbol{x}) > 0 \iff p_{\theta'} > 0, \forall \boldsymbol{x} \in \mathcal{X} \text{ and } \forall (\boldsymbol{\theta}, \boldsymbol{\theta}') \in \Theta^2$. This implies that all $p_{\theta} \in \mathcal{H}_{\theta}$ have the same support denoted \Box .
- C2 $\forall \boldsymbol{\theta} \in \Theta, p_{\boldsymbol{\theta}}$ can be differentiated under the integral¹:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \int_{\boldsymbol{\Box}} p_{\boldsymbol{\theta}} \cdot \mu(d\boldsymbol{x}) = \int_{\boldsymbol{\Box}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}} \cdot \mu(d\boldsymbol{x}) \tag{7}$$

Moreover, according to Fisher, the estimator $\hat{\theta}$ must be [Billingsley'1995Campagne'2022; Fis22]:

- unbiased (see Equation (30))
- consistent (i.e. it must converge in probability to θ^* , the parameter corresponding to the data probability distribution P)
- provided that \mathcal{D}_X is *exhaustive*, i.e., sufficient to characterize the real yet unknown probability distribution $P = P_{\theta^*}$

For Fisher, the way to find a consistent estimator $\hat{\boldsymbol{\theta}}(\mathcal{D}_X)$ is to maximize the *likelihood* $p_{\boldsymbol{\theta}}(\mathcal{D}_X) = \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_X)$, in order to *infer* any sample from the "true" probability distribution $\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}$, from the chosen probability family $\mathcal{H}_{\boldsymbol{\theta}}$:

$$\hat{\boldsymbol{\theta}}(\mathcal{D}_X) = \operatorname*{arg\,max}_{\boldsymbol{\theta}\in\Theta} p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{x}\in\mathcal{D}_X)$$
(8)

When \mathcal{D}_X represents a dataset of independent identically distributed (i.i.d.) variables, the likelihood can be factorized as

$$p_{\boldsymbol{\theta}}(\mathcal{D}_X) = \prod_{i=1}^N p_{\boldsymbol{\theta}}(\boldsymbol{x}_i) \tag{9}$$

$$\left| \frac{\partial p_{\boldsymbol{\theta}}}{\partial \theta_k} \right| \le g \tag{6}$$

¹According to [Bil95], C2 holds if the gradient is locally dominated by an integrable function g, i.e., it exist a neighborhood \Box of θ and g such that $\int_{\Box} g(x)\mu(dx) < +\infty$ such that almost everywhere on a neighborhood V of x:

Equation (8) is justified by the following theorem [Cam22]:

Theorem 9. Given a set of i.i.d. random variables $\mathcal{D}_X = \{x_i \in \mathcal{X}\}_{i=1}^N$, then:

$$\forall \boldsymbol{\theta} \neq \boldsymbol{\theta}^*, \forall \varepsilon > 0 \quad \lim_{N \to +\infty} \mathbb{P} \left[p_{\boldsymbol{\theta}} \left(\mathcal{D}_X \right) - p_{\boldsymbol{\theta}^*} \left(\mathcal{D}_X \right) < \varepsilon \right] = 1$$

Proof. In order to prove the convergence in probability of the event $p_{\theta^*} > p_{\theta}$, one can prove the rephrase the convergence in probability as:

$$\forall \boldsymbol{\theta} \neq \boldsymbol{\theta}^*, \forall \varepsilon > 0 \quad \lim_{N \to +\infty} \mathbb{P}\left[\frac{1}{N} \ln\left(\frac{p_{\boldsymbol{\theta}}\left(\mathcal{D}_X\right)}{p_{\boldsymbol{\theta}^*}\left(\mathcal{D}_X\right)}\right) < \varepsilon\right] = 1 \tag{10}$$

Provided that $\mathcal{D}_X = \{ \boldsymbol{x}_i \in \mathcal{X} \}_{i=1}^N$ is a set of i.i.d. variables, the likelihood is defined by Equation (9). It holds that:

$$\frac{1}{N}\ln\left(\frac{p_{\boldsymbol{\theta}}\left(\mathcal{D}_{X}\right)}{p_{\boldsymbol{\theta}^{*}}\left(\mathcal{D}_{X}\right)}\right) = \frac{1}{N}\sum_{i=1}^{N}\ln\left(\frac{p_{\boldsymbol{\theta}}\left(\boldsymbol{x}_{i}\right)}{p_{\boldsymbol{\theta}^{*}}\left(\boldsymbol{x}_{i}\right)}\right)$$
(11)

According to the weak law of large numbers in Theorem 3 (because the factors $\ln\left(\frac{p_{\theta}(\boldsymbol{x}_i)}{p_{\theta^*}(\boldsymbol{x}_i)}\right)$ are independent identically distributed, the empirical average $\frac{1}{N}\sum_{i=1}^{N}\ln\left(\frac{p_{\theta}(\boldsymbol{x}_i)}{p_{\theta^*}(\boldsymbol{x}_i)}\right)$ is a consistent estimator of the expected value $\mathbb{E}_{\boldsymbol{x}\sim p_{\theta^*}}\left[\ln\left(\frac{p_{\theta}(\boldsymbol{X})}{p_{\theta^*}(\boldsymbol{X})}\right)\right]$, i.e. (see Theorem 5):

$$\forall \varepsilon > 0 \lim_{N \to +\infty} \mathbb{P}\left[\left| \frac{1}{N} \sum_{i=1}^{N} \ln\left(\frac{p_{\theta}\left(\boldsymbol{x}_{i}\right)}{p_{\theta^{*}}\left(\boldsymbol{x}_{i}\right)} \right) - \mathbb{E}_{\boldsymbol{x} \sim p_{\theta^{*}}}\left[\ln\left(\frac{p_{\theta}\left(\boldsymbol{X}\right)}{p_{\theta^{*}}\left(\boldsymbol{X}\right)} \right) \right] \right| < \varepsilon \right] = 1$$
(12)

The Jensen inequality states that [Cam22]:

$$\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\ln \left(\frac{p_{\boldsymbol{\theta}} \left(\boldsymbol{X} \right)}{p_{\boldsymbol{\theta}^*} \left(\boldsymbol{X} \right)} \right) \right] < 0$$
(13)

since ln function is a concave function and therefore²:

$$\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}}\left[\ln\left(\frac{p_{\boldsymbol{\theta}}\left(\boldsymbol{X}\right)}{p_{\boldsymbol{\theta}^*}\left(\boldsymbol{X}\right)}\right)\right] \leq \ln\left(\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}}\left[\frac{p_{\boldsymbol{\theta}}\left(\boldsymbol{X}\right)}{p_{\boldsymbol{\theta}^*}\left(\boldsymbol{X}\right)}\right]\right) = 0$$

Provided the consistency of the estimator, expressed in Equation (12), the latter converges in probability to a negative "true" expected value, as proved

²If the random variable is discrete, according to Proposition 59, the linear combination of concave (convex) functions with positive coefficients (the point-mass probability distribution evaluate at each value) is concave. In the continuous case, the same argument holds, because $\int \ln p(x) \cdot p(x)$ is a linear combination with positive coefficients p(x).

by the Jensen inequality in Equation (13). The objective of the theorem to prove the convergence in probability expressed in Equation (10) is granted by choosing $\varepsilon = \frac{|\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}}\left[\ln\left(\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x})}{p_{\boldsymbol{\theta}^*}(\boldsymbol{x})}\right)\right]|}{2}$ and substituting it into Equation (12), in order to prove that

$$\lim_{N \to +\infty} \mathbb{P}\left[\left| \frac{1}{N} \sum_{i=1}^{N} \ln\left(\frac{p_{\theta}\left(\boldsymbol{x}_{i}\right)}{p_{\theta^{*}}\left(\boldsymbol{x}_{i}\right)} \right) - \mathbb{E}_{\boldsymbol{x} \sim p_{\theta^{*}}}\left[\ln\left(\frac{p_{\theta}\left(\boldsymbol{X}\right)}{p_{\theta^{*}}\left(\boldsymbol{X}\right)} \right) \right] \right| < \frac{\left| \mathbb{E}_{\boldsymbol{x} \sim p_{\theta^{*}}}\left[\ln\left(\frac{p_{\theta}\left(\boldsymbol{X}\right)}{p_{\theta^{*}}\left(\boldsymbol{X}\right)} \right) \right] \right|}{2} \right] = 1$$

and therefore that:

$$\forall \boldsymbol{\theta} \neq \boldsymbol{\theta}^*, \forall \varepsilon > 0 \quad \lim_{N \to +\infty} \mathbb{P}\left[\frac{1}{N} \ln\left(\frac{p_{\boldsymbol{\theta}}\left(\mathcal{D}_X\right)}{p_{\boldsymbol{\theta}^*}\left(\mathcal{D}_X\right)}\right) < \varepsilon\right] = 1$$

which proves the statement.

Theorem 9 proves that the "true" probability distribution of the data, corresponding to θ^* (if $p_{\theta^*} \in \mathcal{H}_{\theta}$), corresponds to the maximum on \mathcal{H}_{θ} . However, the assumption $p_{\theta^*} \in \mathcal{H}_{\theta}$ is not always satisfied: the "true" probability distribution that generated the dataset \mathcal{D}_X is not known beforehand, which is why Fisher proposed to approximate the it by maximizing the likelihood over a chosen parameter space Θ , that generates \mathcal{H}_{θ} , leading to the "best" parameter estimator $\hat{\theta}$. The "best" parameter $\hat{\theta}$ does not necessarily exist, nor it is unique. For the sake of simplicity, Equation (8) is replaced by the following Maximum Log-Likelihood Estimation:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{arg\,max}} \ln \mathcal{L}(\boldsymbol{\theta}; \boldsymbol{X}) = \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{arg\,max}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{X})$$
(14)

Since \mathcal{D}_X represents a limited dataset of independent identically distributed (i.i.d.) realizations, the maximization problem in Equation (14) is approximated by finding the estimator of $\hat{\theta}$, noted as $\hat{\theta}(\mathcal{D}_X)$:

$$\hat{\boldsymbol{\theta}}(\mathcal{D}_X) = \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{arg\,max}} \ \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_X) = \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{arg\,max}} \ \sum_{i=1}^N \ln p_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$
(15)

 $\hat{\theta}$ is a deterministic yet unknown parameter depending on the choice of \mathcal{D}_X . The value of $\hat{\theta}$ relies on the optimization problem in Equation (14), with a finite dataset \mathcal{D}_X at stake.

Substituting the likelihood with the log-likelihood is possible because of the following theorem:

Theorem 10. Maximum Likelihood of $g(\theta)$. [Bil95; Cou20] Given a set of i.i.d. observations $\mathcal{D}_X = \{x_i \in \mathcal{X}\}_{i=1}^N$ and θ the of parameters that maximizes the likelihood $\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_X)$. Then $g\left(\hat{\boldsymbol{\theta}}\right)$ is the maximum likelihood estimator of $q(\boldsymbol{\theta})$.

Proof. The proof resides on the definition of the following Legendre transform, widely used in the thermodynamic theory of thermoelasticity:

$$\mathcal{L}^{*}(\boldsymbol{\eta}; \mathcal{D}_{X}) = \sup_{\boldsymbol{\theta}: g(\boldsymbol{\theta}) = \eta} \mathcal{L}\left(g^{-1}(\boldsymbol{\eta}); \mathcal{D}_{X}\right)$$
(16)

If $g: \Theta \mapsto g(\Theta)$ is a bijection, Equation (16) reduces to

$$\mathcal{L}^{*}(\boldsymbol{\eta}; \mathcal{D}_{X}) = \prod_{i=1}^{N} p(\boldsymbol{x}_{i} | g^{-1}(\boldsymbol{\eta})) = \mathcal{L}\left(g^{-1}(\boldsymbol{\eta}); \mathcal{D}_{X}\right)$$

since $\boldsymbol{\theta} = g^{-1}\left(\boldsymbol{\eta}\right)$ is unique. Therefore

$$\sup_{\boldsymbol{\eta}} \mathcal{L}\left(\boldsymbol{\eta}; \mathcal{D}_X\right) = \sup_{\boldsymbol{\theta}} \mathcal{L}\left(\boldsymbol{\theta}; \mathcal{D}_X\right)$$

However, if g is not bijective, $\exists \theta_1 \neq \theta_2$ such that $\eta = g(\theta_1) = g(\theta_2)$ so that $\mathcal{L}^*(\eta; \mathcal{D}_X) = \prod_{i=1}^N p(\boldsymbol{x}_i | g^{-1}(\eta))$ is not uniquely defined. In this general case, the Legendre transform in Equation (16) must be adopted.

Remark 11. Theorem 10 and Theorem 9 imply that, if the "true" probability distribution $\boldsymbol{\theta}^*$ that generates the dataset \mathcal{D}_X belongs to the open set Θ , i.e., if $p_{\boldsymbol{\theta}^*} \in \mathcal{H}_{\boldsymbol{\theta}}$, enlarging progressively the size of the realization set N, $\hat{\boldsymbol{\theta}}(\mathcal{D}_X) = \hat{\boldsymbol{\theta}}_N$ represents a unique sequence converging towards $\boldsymbol{\theta}^*$. As a matter of fact, due to Theorem 9, $\forall \boldsymbol{\theta} \in O_{\varepsilon}(\boldsymbol{\theta}^*)$, with $O_{\varepsilon}(\boldsymbol{\theta}^*)$ being an open neighborhood of $\boldsymbol{\theta}^*$, the probability associated to the set of observations S_N defined as:

$$S_{N} := \left\{ \boldsymbol{x} | \ln p_{\boldsymbol{\theta}^{*}} \left(\boldsymbol{x} \right) > \sup_{\boldsymbol{\theta} \in \partial O_{\varepsilon}(\boldsymbol{\theta}^{*})} \ln p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) \right\}$$

converges to 1:

$$\forall \varepsilon > 0 \quad \lim_{N \to +\infty} \mathbb{P}\left[|\mathbb{P}\left(S_N\right) - 1| < \varepsilon \right] = 1$$

which means that all the samples, when the data set of realization is sufficiently large, most probably belong to S_N , with the maximum log-likelihood over the closure $\bar{O}_{\varepsilon}(\boldsymbol{\theta}^*)$ corresponding to a point $\boldsymbol{\theta}^* \in \text{Int}(\Theta)$. Because of the regularity condition C2 on $p_{\boldsymbol{\theta}}$, the latter is continuous and differentiable over $O_{\varepsilon}(\boldsymbol{\theta}^*)$. Thanks to Rolle's theorem, $\exists \boldsymbol{\theta}_N \in \bar{O}_{\varepsilon}(\boldsymbol{\theta}^*)$ such that $\nabla_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}_N; \boldsymbol{x}) = \boldsymbol{0}$. This implies that $S_N \subset \tilde{S}_N$, defined as:

$$\tilde{S}_N := \left\{ \boldsymbol{x} | \exists \boldsymbol{\theta}_N \in \bar{O}_{\varepsilon}(\boldsymbol{\theta}^*) \text{ such that } \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}\left(\boldsymbol{\theta}_N; \boldsymbol{x}\right) = \boldsymbol{0} \right\}$$

since a priori $\ln p_{\theta^*}(\boldsymbol{x}) > \ln p_{\theta_N}(\boldsymbol{x})$ is not granted. Therefore, $\mathbb{P}(S_N) \leq \mathbb{P}(\tilde{S}_N) \leq 1$. Due to the convergence in probability of $\mathbb{P}(S_N)$ to 1, then $\mathbb{P}(\tilde{S}_N)$ converges in probability to 1, which means that for $\forall \varepsilon > 0, \exists$ an open neighborhood $O_{\varepsilon}(\boldsymbol{\theta}^*)$ to which $\boldsymbol{\theta}_N$ most probably belongs, converging most probably towards $\boldsymbol{\theta}^*$, by definition.

Remark 12. The converging sequence $\boldsymbol{\theta}_N$ in Remark 11 maximizes the log-likelihood in the sense that

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}} \left(\boldsymbol{\theta}_N; \mathcal{D}_X \right) = \boldsymbol{0}$$

This expression has multiple solutions $\boldsymbol{\theta}_N$ but thanks to what exposed in the Remark 11 proves that $\boldsymbol{\theta}_N$ converges in probability towards $\boldsymbol{\theta}^*$ and it this sense one should interpret the quest for the MLE.

The following theorem proves that the sequence θ_N converges to θ^* in standard normal probability distribution (see Theorem 17). Before proving this aspect, other regularity conditions of the statistical models must be introduced.

If the regularity condition C1 holds and $\mathcal{L}(\boldsymbol{\theta}; \boldsymbol{X})$ (and $\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_X)$) is differentiable almost everywhere in \Box , one can define the *score* as:

$$\boldsymbol{s}\left(\boldsymbol{\theta};\boldsymbol{X}\right) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta};\boldsymbol{X}) \tag{17}$$

and its estimator over a limited dataset $s(\theta; \mathcal{D}_X)$. If conditions C1 and C2 hold, from Equation (14) one can prove that the score is a centered random vector in θ^* :

$$\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*; \boldsymbol{X}) = \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} s(\boldsymbol{\theta}^*; \boldsymbol{X}) = \boldsymbol{0}$$
(18)

For a finite set of i.i.d. samples \mathcal{D}_X^3 , Equation (18) is straightforward:

$$\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*; \mathcal{D}_X) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \ln \left(p_{\boldsymbol{\theta}^*}(\boldsymbol{\theta}^*; \mathcal{D}_X) \right) = \\ = \sum_{i=1}^N \frac{p_{\boldsymbol{\theta}^*}(\boldsymbol{x}_i)}{p_{\boldsymbol{\theta}^*}(\boldsymbol{x}_i)} \boldsymbol{\nabla}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}^*}\left(\boldsymbol{x}_i\right) = \frac{1}{N} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \sum_{i=1}^N p_{\boldsymbol{\theta}^*}(\boldsymbol{x}_i) = 0$$
(19)

The score in $\hat{\boldsymbol{\theta}}$ is nihil $\boldsymbol{s}\left(\hat{\boldsymbol{\theta}}; \mathcal{D}_X\right) = \boldsymbol{0}$ by Remarks 11 and 12.

Example 1. Likelihood maximization with PyTorch

In the following example, the maximum likelihood criterion is implemented in **PyTorch**, in order to find the parameters of the statistical model

$$\mathcal{H}_{\boldsymbol{\theta}} := \left\{ \frac{1}{\left(2\pi\theta_{\sigma}^{2}\right)^{\frac{1}{2}}} e^{-\frac{\left(x-\theta_{\mu}\right)^{2}}{2\cdot\theta_{\sigma}^{2}}}; \frac{1}{2\theta_{\sigma}} e^{-\frac{\left|x-\theta_{\mu}\right|}{\theta_{\sigma}}} \right\}$$

of probability distributions $p_{\theta} : \mathbb{R} \to \mathbb{R}, p_{\theta} : x \mapsto p_{\theta}(x)$ and with

$$\Theta := \left\{ heta_\mu, heta_\sigma
ight\}, \quad heta_\mu \in \mathbb{R}, heta_\sigma \in \mathbb{R}^+$$

³The proof holds for any p_{θ} satisfying the C1 and C2 conditions, both for discrete and continuous random variables. For the latter, an integral over \mathcal{X} replaces the sum and by applying condition C2 in order to derive under the integral and viceversa.

The two probability distributions in \mathcal{H}_{θ} correspond to Gaussian and Laplace distributions respectively. The "true" probability distribution belongs to \mathcal{H}_{θ} and it is set by selecting two parameters

$$\bar{\theta}_{\mu} = \mu_X, \quad \bar{\theta}_{\sigma} = \sigma_X$$

Therefore, $p_{\bar{\theta}_{\mu},\bar{\theta}_{\sigma}} = p_{\mu_X,\sigma_X} \in \mathcal{H}_{\theta}$, which implies that $\hat{\theta} \approx \theta^*$. The random variable X generates the data set of i.i.d. realizations $\mathcal{D}_X = \{x_i\}_{i=1}^N$, with N = 10000. For the sake of simplicity, instead of maximizing the empirical average log-likelihood defined in Equation (15), the code minimizes the empirical average *Negative Log-Likelihood* \mathcal{NLL} , defined as:

$$\mathcal{NLL}(\boldsymbol{\theta}; \mathcal{D}_X) = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{arg\,min}} \sum_{i=1}^N \ln \frac{1}{p_{\boldsymbol{\theta}}(\boldsymbol{x}_i)}$$
(20)

which still satisfies the convergence in probability expressed in Theorem 9 that reads:

$$\forall \boldsymbol{\theta} \neq \boldsymbol{\theta}^*, \forall \varepsilon > 0 \quad \lim_{N \to +\infty} \mathbb{P}\left[\frac{1}{N} \ln p_{\boldsymbol{\theta}}\left(\mathcal{D}_X\right) < \varepsilon + \frac{1}{N} p_{\boldsymbol{\theta}^*}\left(\mathcal{D}_X\right)\right] = 1 \quad (21)$$

In particular, given the statistical model in this example, the \mathcal{NLL} function for a Gaussian distribution reads:

$$\mathcal{NLL}(\boldsymbol{\theta}; \mathcal{D}_X) = \frac{N}{2} \ln (2\pi) + N \ln \theta_{\sigma} + \frac{1}{2\theta_{\sigma}^2} \sum_{i=1}^{N} (x_i - \theta_{\mu})^2$$
(22)

and for a Laplace distribution:

$$\mathcal{NLL}(\boldsymbol{\theta}; \mathcal{D}_X) = N\left(\ln \theta_{\sigma} + \ln 2\right) + \frac{1}{\theta_{\sigma}} \sum_{i=1}^{N} |x_i - \theta_{\mu}|$$
(23)

The minimization is achieved by using the AdamW optimizing algorithm (see Section 3.4.5 and https://pytorch.org/docs/stable/generated/torch.optim. AdamW.html), over n_e =100000 epochs.

The code below minimizes the \mathcal{NLL} to find the "best" estimators $\hat{\theta}_{\mu} \approx \bar{\theta}_{\mu} = \mu_X$ and $\hat{\theta}_{\sigma} \approx \bar{\theta}_{\sigma} = \sigma_X$. The result of the iterative minimization is depicted in Figure 1

```
1 # Source: https://stackoverflow.com/questions/72469496/
2 #how-to-use-pytorch-for-maximum-likelihood-estimation-with-restrict
3 #-optimization
4 import numpy as np
5 import torch
6 from matplotlib import pyplot as plt
7
```

```
8
9
```

Gaussian probability distribution



Figure 1: Iterative minimization (with AdamW algorithm) of $\mathcal{NLL}(\theta_{\mu}, \theta_{\sigma}; \mathcal{D}_X)$. The convergence of the "best" parameters $(\hat{\theta}_{\mu}, \hat{\theta}_{\sigma})$ towards the "true" parameters $(\bar{\theta}_{\mu}, \bar{\theta}_{\sigma})$ is shown, with the respect to the iteration number (epoch). The "true" probability distribution corresponds to $\mathcal{N}(\mu_X, \sigma_X)$, with $\mu_X = 5$ and $\sigma_X = 2$. The figure was generated with code below.

```
111
10
      # Fix the pseudo-random generator see to grant reproductibility
^{11}
12
      torch.manual_seed(0)
13
      # Create a dataset of nX samples from normal distribution
     N = 10000
14
15
      # real mean
     mu_X = torch.tensor(np.array([5.0]),
16
                           dtype=torch.float64,
17
18
                           requires_grad=False).tile((N,))
19
      # real standard deviation
20
     sigma_X = torch.tensor(np.array([2.0]),
                          dtype=torch.float64
21
                          requires_grad=False)
22
^{23}
^{24}
      # generate random samples from p=N(mu_X, sigma_X)
25
     D_X = torch.normal(mean=mu_X, std=sigma_X)
26
     D_X.requires_grad = False
27
      # Initialize the values of the estimators
^{28}
29
      theta_mu = torch.tensor(np.array([0.5]),
                               dtype=torch.float64,
30
^{31}
                               requires_grad=True)
      theta_sigma = torch.tensor(np.array([10.0]),
32
                                  dtype=torch.float64,
33
                                  requires_grad=True)
34
35
      # Define the optimizer
36
37
     learning_rate = 0.0001
38
      optimizer = torch.optim.AdamW([theta_mu, theta_sigma], lr = learning_rate)
39
     n = 100000
40
41
      # Minimize the Negative Log-Likelihood iteratively
42
43
      track_nll=[]
^{44}
      track_theta_mu = []
45
      track_theta_sigma = []
```

```
for epoch in range(n_e):
46
        # Compute negative log-likelihood
47
        nll = N*(0.5*np.log(2.0*np.pi)+theta_sigma.log())
^{48}
        nll+= ((((D_X-theta_mu)/theta_sigma).pow(2))/2.0).sum()
49
        optimizer.zero_grad()
50
51
        nll.backward()
52
53
        if epoch % 100 == 0:
54
         print("NLL: {}; theta_mu: {}; theta_sigma: {}".format(nll.data.numpy(),
55
                                                                      theta_mu.data.numpy(),
56
57
                                                                     theta_sigma.data.numpy()))
58
        optimizer.step()
        track_nll.append(float(nll))
59
        track_theta_mu.append(float(theta_mu))
60
        track_theta_sigma.append(float(theta_sigma))
61
62
63
      # plot convergence curves
^{64}
      fig, ax = plt.subplots(nrows=1, ncols=3, sharex=True, figsize=(12,4))
      65
66
                  linewidth=3,
67
68
                  label=r"$\mathcal{NLL}$")
69
      ax[1].plot(track_theta_mu,
70
                  color='k'
                  linewidth=3.
71
                  label=r"$\theta \mu$")
72
      ax[2].plot(track_theta_sigma,
73
                  color='k',
74
75
                  linewidth=3,
76
                  label=r"$\theta_\sigma$")
      ax[0].set_xlim(0,n_e)
77
      ax[0].set_xlabel("epochs")
ax[1].set_xlabel("epochs")
78
79
      ax[2].set_xlabel("epochs")
80
      ax[0].set_title(r"$\mathcal{NLL}\left(\theta_\mu,\theta_\sigma;\mathcal{D}_x\right)$", fontsize=15)
ax[1].set_title(r"$\theta_\mu\left(\mathcal{D}_x\right)$",
81
^{82}
83
                       fontsize=15)
      ax[2].set_title(r"$\theta_\sigma\left(\mathcal{D}_x\right)$",
fontsize=15)
84
85
      fig.savefig("NLL_gauss.png", dpi=300, bbox_inches="tight")
86
87
```

```
# Source: https://stackoverflow.com/questions/72469496/
 1
2
     {\it \#how-to-use-pytorch-for-maximum-likelihood-estimation-with-restrict}
      #-optimization
 3
 4
     import numpy as np
      import torch
 \mathbf{5}
     from matplotlib import pyplot as plt
 6
 7
      ...
 8
9
     Laplace\ probability\ distribution
10
     from torch.distributions.laplace import Laplace
11
12
      # Fix the pseudo-random generator see to grant reproductibility
     torch.manual_seed(0)
13
14
      # Create a dataset of nX samples from normal distribution
15
     N = 10000
     # real mean
16
     mu_X = torch.tensor(np.array([5.0]),
17
                          dtype=torch.float64,
18
                           requires_grad=False).tile((N,))
^{19}
20
      # real standard deviation
^{21}
```

sigma_X = torch.tensor(np.array([2.0]),



Figure 2: Iterative minimization (with AdamW algorithm) of $\mathcal{NLL}(\theta_{\mu}, \theta_{\sigma}; \mathcal{D}_X)$. The convergence of the "best" parameters $(\hat{\theta}_{\mu}, \hat{\theta}_{\sigma})$ towards the "true" parameters $(\bar{\theta}_{\mu}, \bar{\theta}_{\sigma})$ is shown, with the respect to the iteration number (epoch). The "true" probability distribution corresponds to Laplace (μ_X, σ_X) , with μ_X =5 and σ_X =2. The figure was generated with code below.

```
dtype=torch.float64,
22
                           requires_grad=False)
23
24
25
      # generate random samples from p=N(mu_X, sigma_X)
      generator = Laplace(mu_X, sigma_X)
26
27
      D_X = generator.sample()
^{28}
      D_X.requires_grad = False
29
      # Initialize the values of the estimators
30
      theta_mu = torch.tensor(np.array([0.5]),
31
                                dtype=torch.float64,
32
33
                                requires_grad=True)
34
      theta_sigma = torch.tensor(np.array([10.0]),
                                   dtype=torch.float64,
35
36
                                   requires_grad=True)
37
      # Define the optimizer
38
39
      learning_rate = 0.0001
40
      optimizer = torch.optim.AdamW([theta_mu, theta_sigma], lr = learning_rate)
41
      n_e = 100000
42
43
44
      # Minimize the Negative Log-Likelihood iteratively
45
      track_nll=[]
46
      track_theta_mu = []
      track_theta_sigma = []
47
      for epoch in range(n_e):
    # Compute negative log-likelihood
48
49
        nll = N*(np.log(2.0)+theta_sigma.log())
50
        nll+= (D_X-theta_mu).abs().sum()/theta_sigma.abs()
51
52
        optimizer.zero_grad()
53
        nll.backward()
54
55
        if epoch % 100 == 0:
56
          print("NLL: {}; theta_mu: {}; theta_sigma: {}".format(nll.data.numpy(),
57
                                                                    theta_mu.data.numpy(),
58
```

```
theta_sigma.data.numpy()))
59
        optimizer.step()
60
        track_nll.append(float(nll))
61
        track_theta_mu.append(float(theta_mu))
62
        track_theta_sigma.append(float(theta_sigma))
63
64
65
      # plot convergence curves
      fig, ax = plt.subplots(nrows=1, ncols=3, sharex=True, figsize=(12,4))
66
      ax[0].plot(track_nll,
67
                  color='k',
68
                  linewidth=3,
69
70
                  label=r"$\mathcal{NLL}$")
      ax[1].plot(track_theta_mu,
71
72
                  color='k'
                  linewidth=3,
73
74
                  label=r"$\theta_\mu$")
75
      ax[2].plot(track_theta_sigma,
76
                  color='k'
77
                  linewidth=3,
                  label=r"$\theta_\sigma$")
78
      ax[0].set_xlim(0,n_e)
79
      ax[0].set_xlabel("epochs")
80
81
      ax[1].set_xlabel("epochs")
      ax[2].set_xlabel("epochs")
ax[0].set_title(r"$\mathcal{NLL}\left(\theta_\mu, \theta_\sigma; \mathcal{D}_x\right)$",
82
83
                       fontsize=15)
84
      ax[1].set_title(r"$\theta_\mu\left(\mathcal{D}_x\right)$",
85
                       fontsize=15)
86
      ax[2].set_title(r"$\theta_\sigma\left(\mathcal{D}_x\right)$",
87
88
                       fontsize=15)
      fig.savefig("NLL_laplace.png", dpi=300, bbox_inches="tight")
89
```

Based on Equation (19) and assuming an unbiased estimator (see Equation (30)), i.e., $\mathbb{E}_{\mathcal{D}_X \subset \mathcal{X}}\left(\hat{\boldsymbol{\theta}}\right) = 0$, in his seminal work, Fisher stated the concept of *information* associated to the parameters $\mathbb{I}(\boldsymbol{\theta}^*)$ as the variance of $\nabla_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*; \boldsymbol{X})$:

$$\mathbb{I}_{F}(\boldsymbol{\theta}^{*};\boldsymbol{X}) = \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^{*}}}\left[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*};\boldsymbol{X}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\hat{\theta}};\boldsymbol{X})\right] \ge 0 \qquad (24)$$

 $\mathbb{I}_F(\boldsymbol{\theta}^*; \boldsymbol{X})$ in Equation (24) is called Fisher Information Matrix (FIM), and it is a $N \times N$ positive semidefinite matrix. The FIM corresponds to the variance of the score $\mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}}[\boldsymbol{s}(\boldsymbol{\theta}^*; \boldsymbol{X})]$ If \boldsymbol{X} is composed by i.i.d. variables, the FIM reads:

$$\mathbb{I}_{F}(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}) = \sum_{i=1}^{N} \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \right] - \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{j}) \right] = N \mathbb{I}_{F}(\boldsymbol{\theta}^{*}; X_{1})$$
(25)

with the term $\sum_{i=1}^{N} \sum_{j=1, j\neq i}^{N} \mathbb{E}_{\mathbf{x}\sim p_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*; x_i) \otimes \nabla_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^*; x_j) \right] = \mathbf{0}$ because of the fact that the score function is centered in $\boldsymbol{\theta}^*$ [Bil95; Cou20; Cam22]. Equation (25) is another way to express the fact that the variance of

a sum of i.i.d. variable is the sum of the variances. If $\mathbb{I}_F(\boldsymbol{\theta}^*)$ is positive definite, then it defines a Riemann metric on the N-dimensional parameter space. A set of i.i.d. random variables, with high Fisher information has a score with large variance, i.e., it is *informative*. The "narrower" (in average on \mathcal{D}_X) is minimum of the log-likelihood, the more informative is θ^* and therefore the higher is the variance of the scire (the largest the span of data around the minimum) [Cam22; Cou20]. Under the two following extra regularity conditions:

- C3 For almost every $\boldsymbol{x} \in \Box$, it exists a continuous mapping $h: \Theta \to p_{\boldsymbol{\theta}}$ such that $h: \theta \times \Box \mapsto h_{\theta}(x)$ with $h \in C^2(\Theta)$
- C4 $\forall \theta \in \Theta, \forall i, j, 1 \leq i, j \leq p$ the transport theorem can be applied as follows⁴:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \int_{\Box} \boldsymbol{\nabla}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\boldsymbol{\theta}; \boldsymbol{X}) \ \mu(d\boldsymbol{x}) = \int_{\Box} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\boldsymbol{\theta}; \boldsymbol{X}) \ \mu(d\boldsymbol{x})$$
(26)

Under C1, C2, C3 and C4 regularity conditions and for $\nabla_{\theta} p_{\theta}(\theta^*; X) \in C^2(\Theta)$, the Fisher's information can be also expressed, following Equation (25), as follows:

$$\mathbb{I}_{F}(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}) = -\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\mathbf{H}_{\ell}(\boldsymbol{\theta}^{*}, \mathcal{D}_{X}) \right]$$
(27)

with \mathbf{H}_{ℓ} being the Hessian matrix of $\ell(\boldsymbol{\theta}; \mathcal{D}_X) = \ln p_{\boldsymbol{\theta}}(\mathcal{D}_X)$. For i.i.d. samples in \mathcal{D}_X , this is proven by considering Equation (24) and conditions C1, C2, C3 and C4:

$$\mathbb{I}_{F}(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}) = \sum_{i=1}^{N} \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \right] = \\ = \sum_{i=1}^{N} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) = \\ = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \left(\sum_{i=1}^{N} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}} \cdot p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \right) - \\ - \sum_{i=1}^{N} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \left(\ln p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \cdot p_{\boldsymbol{\theta}}(\boldsymbol{\theta}^{*}; x_{i}) \right) = -\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\mathbf{H}_{\ell}(\boldsymbol{\theta}^{*}, \mathcal{D}_{X}) \right]$$
(28)

In order to determine the accuracy of the iterative gradient descent methods employed in machine learning to approximate θ^* with $\hat{\theta}$ (see Section 4.2), the Cramér-Rao bound (CRB) can be helpful since it represents an accuracy limit to θ^* .

Theorem 13. Cramér-Rao bound Given a set of i.i.d. variables $\mathcal{D}_X = \{\mathbf{x}_i\}_{i=1}^N$, sampled from a probability dis-tribution $p_{\mathbf{\theta}^*} \in \mathcal{H}_{\mathbf{\theta}}$, with $p_{\mathbf{\theta}^*}(\mathbf{x}) = \prod_{i=1}^N p_{\mathbf{\theta}^*}(\mathbf{x}_i)$, and a statistical model (or

 $^{^{4}}$ Under the same conditions that led to Equation (6)

algorithm) to estimate θ^* , that reads $\theta = A(x)$ then the mean estimator reads:

$$\bar{\boldsymbol{\theta}} = \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[A\left(\boldsymbol{X} \right) \right]$$

and the variance of the estimator is bounded by below as follows:

$$\mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left(\mathbf{A} \left(\boldsymbol{X} \right) \right) \ge \frac{\|\boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \bar{\boldsymbol{\theta}}\|^2}{N \cdot \mathbb{I}_F(\boldsymbol{\theta}^*; X_1)}$$
(29)

A is unbiased if:

$$\mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left(\mathbf{A} \left(\boldsymbol{X} \right) \right) \geq \frac{1}{N \cdot \mathbb{I}_F(\boldsymbol{\theta}^*; X_1)}$$
(30)

Proof. The gradient $\nabla_{\theta^*} A$ can be computed by differentiating under the integral, as:

$$\begin{aligned} \boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \mathbf{A} &= \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\mathbf{A} \left(\boldsymbol{X} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \ln p_{\boldsymbol{\theta}^*} \left(\boldsymbol{X} \right) \right] = \\ &= \mathbb{C}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\mathbf{A} \left(\boldsymbol{X} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \ln p_{\boldsymbol{\theta}^*} \left(\boldsymbol{X} \right) \right] + \\ &+ \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\mathbf{A} \left(\boldsymbol{X} \right) \right] \cdot \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \ln p_{\boldsymbol{\theta}^*} \left(\boldsymbol{X} \right) \right] \end{aligned}$$

Since θ^* generates the i.i.d. dataset the score is centered (see Equation (18)), which implies that the following expression holds

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}^{*}} \mathbf{A} = \mathbb{C}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^{*}}} \left[\mathbf{A} \left(\boldsymbol{X} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}^{*}} \ln p_{\boldsymbol{\theta}^{*}} \left(\boldsymbol{X} \right) \right]$$

and the square norm is bounded by above as follows (Cauchy-Schwartz's in-equality):

$$\begin{aligned} \|\boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \mathbf{A}\|^2 &= \left\| \mathbb{C}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\mathbf{A} \left(\boldsymbol{X} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}^*} \left(\boldsymbol{X} \right) \right] \right\|^2 \leq \\ &\leq \mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\mathbf{A} \left(\boldsymbol{X} \right) \right] + \mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \ln p_{\boldsymbol{\theta}^*} \left(\boldsymbol{X} \right) \right] \end{aligned}$$

The term $\mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^*} \ln p_{\boldsymbol{\theta}^*} (\boldsymbol{X}) \right]$ corresponds to $\mathbb{I}_F (\boldsymbol{\theta}^*; \boldsymbol{X}) = N \cdot \mathbb{I}_F (\boldsymbol{\theta}^*; X_1)$ which proves the CRB in the general case. A is unbiased if $\boldsymbol{\bar{\theta}} = \boldsymbol{\theta}^*$ which implies that:

$$\mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left(\mathbf{A} \left(\boldsymbol{X} \right) \right) = \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left[\left(\boldsymbol{\bar{\theta}} - \boldsymbol{\theta}^* \right)^2 \right] \ge \frac{1}{N \cdot \mathbb{I}_F(\boldsymbol{\theta}^*; X_1)}$$
(31)

Remark 14. Choosing an unbiased Equation (31) CRB allow to assess the accuracy of the estimator $\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}\in\Theta}{\arg \max \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}_X)}$, obtained, for instance, via

gradient descent algorithm $A(\mathbf{X})$.

The CRB states that the higher is the information of an estimator $\hat{\theta}$, the higher is the variance of the score and, because of the CRB, the lower is its variance (the narrower is the log-likelihood maximum). The FIM and the CRB steer the convergence rate of any algorithm that attempts at maximize the log-likelihood of any statistical model adopted to infer from the available dataset. However, from a practical standpoint, the FIM matrix (dense) can easily reach a unbearable computational cost, which is why, in practice, it is rarely explicitly adopted. First order gradient descent methods are adopted instead (see Equation (63)).

Remark 15. Provided the CRB for an unbiased estimator, its "efficiency" can be measure as the ratio between the variance lower bound expressed by the CRB and stated in Equation (31) and the variance of the estimator:

$$\eta_{\boldsymbol{\theta}} = \frac{1}{N \cdot \mathbb{I}_{F}(\boldsymbol{\theta}^{*}; X_{1}) \cdot \mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^{*}}} \left(\mathbf{A} \left(\boldsymbol{X} \right) \right)} \leq 1$$
(32)

Remark 16. One can notice that, maximizing the log-likelihood leads to minimize the so called Kullback-Leibler distance $\mathbb{D}_{KL}\left(p\|p_{\theta}\right) = \mathbb{E}_{x \sim p}\left[\ln \frac{p}{p_{\theta}}\right] < +\infty^{5}$ between p_{θ} and the true probability distribution p(x) (unknown). As a matter of fact:

$$\max_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{\boldsymbol{x}\sim p} \left[\ln p_{\boldsymbol{\theta}} \right] = \max_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{\boldsymbol{x}\sim p} \left[\ln \frac{p_{\boldsymbol{\theta}}}{p} \right] + \max_{\boldsymbol{\theta}\in\Theta} \underbrace{\mathbb{E}_{\boldsymbol{x}\sim p} \left[\ln p \right]}_{\leq 0} \leq \min_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{\boldsymbol{x}\sim p} \left[\ln \frac{p}{p_{\boldsymbol{\theta}}} \right]$$
(33)

 $\mathbb{D}_{KL}(p||p_{\theta})$ measures a *distance* between the real probability distribution and parametric probability density p_{θ} . Note that $\mathbb{D}_{KL}(p||p_{\theta}) \neq \mathbb{D}_{KL}(p_{\theta}||p)$. It is worth noticing that, in practice, $\mathbb{D}_{KL}(p||p_{\theta}) > \mathbb{D}_{KL}(p||\mathcal{H}_{\theta}) = \inf_{\theta \in \Theta} \mathbb{D}_{KL}(p||\mathcal{H}_{\theta}) \neq$ 0 if $p \notin \mathcal{H}_{\theta}$. This means that our estimator will have poor chances to discover P [Cou20; Cam22].

Example 2. Compute $\mathbb{D}_{KL}(p_{\theta} \| p)$ with PyTorch⁶

To avoid underflow issues when computing this quantity, this loss expects the argument input in the log-space. As all the other losses in PyTorch, this function expects the first argument, input, to be the output of the model (e.g. the neural network) and the second, target, to be the observations in the dataset.

⁵As a convention $0 \cdot \ln 0 = 0 \cdot \ln \frac{0}{0} = 0$

 $^{^{6}} https://pytorch.org/docs/stable'/generated/torch.nn.KLDivLoss.html$

```
from torch import nn
1
^{2}
      import torch.nn.functional as F
       # loss = loss_pointwise.sum() / input.size(0)
3
      kl_loss = nn.KLDivLoss(reduction="batchmean")
4
       # input should be a distribution in the log space
\mathbf{5}
6
      ptheta = F.log_softmax(torch.randn(3, 5, requires_grad=True), dim=1)
       # Sample a batch of distributions. Usually this would come from the dataset
7
      p = F.softmax(torch.rand(3, 5), dim=1)
8
9
      output = kl_loss(ptheta, p)
10
       # alternatively
11
      kl_loss = nn.KLDivLoss(reduction="batchmean", log_target=True)
log_target = F.log_softmax(torch.rand(3, 5), dim=1)
output = kl_loss(ptheta, log_target)
12
13
14
```

The following theorem proves that, provided an extra regularity condition of the statistical model and thanks to the CRB, the MLE estimator converges in standard normal probability to θ^* .

Theorem 17. *MLE convergence in standard normal probability* [Cou20; Cam22]. Provided a statistical model $\mathcal{H}_{\boldsymbol{\theta}} := \{P_{\boldsymbol{\theta}}, \boldsymbol{\theta} \in \Theta \subset \mathbb{R}^{d_{\Theta}}\}$ with regularity conditions C1, C2, C3, C4, an i.i.d. dataset $\mathcal{D}_X = \{\boldsymbol{x}_i\}_{0 < i \leq N}$ generated by a parameter $\boldsymbol{\theta}^* \in Int(\Theta)$, a Maximum Log-likelihood Estimator $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}\in\Theta} \ell(\boldsymbol{\theta}; \mathcal{D}_X)$ and a further regularity condition C5 of $\mathcal{H}_{\boldsymbol{\theta}}$ that reads:

$$\|\boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}}\| < M(\boldsymbol{X}) \quad \forall \boldsymbol{\theta} \in \Theta, \mathbb{E}_{\boldsymbol{x} \in p_{\boldsymbol{\theta}}}[M(\boldsymbol{x})] < +\infty$$
(34)

all MLE sequences $\hat{\theta}_N$ converge in standard normal probability to θ^* :

$$\lim_{N \to +\infty} \mathbb{P}\left[\left| p_{\sqrt{N}\left(\hat{\boldsymbol{\theta}}_{N} - \boldsymbol{\theta}^{*}\right)} - \mathcal{N}\left(\boldsymbol{0}, \mathbb{I}_{F}^{-1}\left(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}\right)\right) \right| \right] = 1$$
(35)

Therefore, because of the CRB, the MLE is an asymptotically "optimum" estimator, i.e. its efficiency $\eta_{\hat{\theta}_N}$ (defined in Equation (32)) converges in probability to 1:

$$\lim_{N \to \infty} \mathbb{P}\left[\left| \frac{1}{N \cdot \mathbb{I}_F(\boldsymbol{\theta}^*; X_1) \cdot \mathbb{V}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^*}} \left(\mathbf{A} \left(\boldsymbol{X} \right) \right)} - 1 \right| \right] = 1$$
(36)

Proof. According to Remark 11, the MLE sequence $\hat{\boldsymbol{\theta}}_N$ exists and it is unique. Moreover, the estimator maximizes the log-likelihood, so its score is nihil in $\hat{\boldsymbol{\theta}}_N$ but not necessarily in $\boldsymbol{\theta}^*$. Therefore, a Taylor expansion around $\hat{\boldsymbol{\theta}}_N$ (see the regularity conditions C1 to C5) reads:

$$\boldsymbol{s}\left(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}\right) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}^{*}}\left(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}\right) = \mathbf{H}_{\ell}\left(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}\right)\left(\boldsymbol{\theta}^{*} - \boldsymbol{\hat{\theta}}_{N}\right) + o\left(\left\|\boldsymbol{\theta}^{*} - \boldsymbol{\hat{\theta}}_{N}\right\|\right)$$

The score $s(\theta^*; \mathcal{D}_X)$ (a gradient, therefore linear) is the sum of N i.i.d. scores with zero mean (see Equation (18)) and variance equal to the FIM. In this case, the central limit Theorem 8 states that the probability distribution of

the variable $\sqrt{\frac{N}{\mathbb{I}_{F}(\boldsymbol{\theta}^{*};\mathcal{D}_{X})}} \boldsymbol{s}\left(\boldsymbol{\theta}^{*};\mathcal{D}_{X}\right)$ converges in distribution (see Definition 7) to $\mathcal{N}(\mathbf{0},\mathbf{I})$, or alternatively, the probability distribution of the variable $\sqrt{N}\boldsymbol{s}\left(\boldsymbol{\theta}^{*};\mathcal{D}_{X}\right)$ converges in distribution to $\mathcal{N}(\mathbf{0},\mathbb{I}_{F}\left(\boldsymbol{\theta}^{*};\mathcal{D}_{X}\right))$. Now, the Taylor expansion of the Hessian around $\boldsymbol{\theta}^{*}$ reads:

$$\begin{split} \mathbf{H}_{\ell}\left(\boldsymbol{\hat{\theta}}_{N}; \mathcal{D}_{X}\right) &= \mathbf{H}_{\ell}\left(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}\right) + \left(\boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \boldsymbol{\nabla}_{\boldsymbol{\theta}} \otimes \boldsymbol{s}\left(\boldsymbol{\theta}^{*}; \mathcal{D}_{X}\right)\right) \cdot \left(\boldsymbol{\hat{\theta}}_{N} - \boldsymbol{\theta}^{*}\right) + \\ &+ o\left(\|\boldsymbol{\hat{\theta}}_{N} - \boldsymbol{\theta}^{*}\|\right) \end{split}$$

with $\nabla_{\boldsymbol{\theta}} \otimes \nabla_{\boldsymbol{\theta}} \otimes \nabla_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}} (\boldsymbol{\theta}; \mathcal{D}_X) = \nabla_{\boldsymbol{\theta}} \otimes \nabla_{\boldsymbol{\theta}} \otimes \boldsymbol{s} (\boldsymbol{\theta}; \mathcal{D}_X)$. Provided C3 and C4, the FIM $N\mathbb{I}_F(\boldsymbol{\theta}^*; X_1)$ is equal to $\mathbf{H}_\ell(\boldsymbol{\theta}^*; \mathcal{D}_X)$ (see Equation (27)). Provided C5, the latter term $\nabla_{\boldsymbol{\theta}} \otimes \nabla_{\boldsymbol{\theta}} \otimes \boldsymbol{s} (\hat{\boldsymbol{\theta}}; \mathcal{D}_X)$ is bounded by $M(\mathcal{D}_X)$ and therefore it tens to **0** for $N \to +\infty$ which implies the limit :

$$\lim_{N \to +\infty} \sqrt{N} \left(\boldsymbol{\hat{\theta}}_N - \boldsymbol{\theta}^* \right) = \lim_{N \to +\infty} \frac{\sqrt{N} \boldsymbol{s} \left(\boldsymbol{\theta}^*; \boldsymbol{\mathcal{D}}_X \right)}{N \mathbb{I}_F \left(\boldsymbol{\theta}^*; X_1 \right)}$$

converges in distribution to $\mathcal{N}(\mathbf{0}, \mathbb{I}_F^{-1}(\boldsymbol{\theta}^*; \mathcal{D}_X))$, which proves the statement.

Remark 18. Theorem 17 establishes the confidence intervals of the MLE algorithm, providing the possibility of assessing the probability (standard normal) that the MLE approaches the "true" probability distribution $p_{\theta^*}(x)$ [Cam22].

Remark 19. For \mathcal{NN} , the Fisher approach can fail because $\boldsymbol{\theta}^*$ is not unique, since the loss function $L_{\mathcal{D}_{XY}}$ defined in (\mathcal{P}) in Section 2 is not convex [Cam22]. Moreover, in practice, the dimension of the probability space $m \gg d_X$, which implies that the problem is over-parametrized, which makes it difficult to find unbiased estimators [Cam22].

1.4 The Shannon's approach

The concept of *information* was better defined by Shannon, in the framework of the so called *source coding problem*, i.e. the problem of a source emitting a message and a receiver capturing it. The message is *meaningful* only if the receiver had no *a priori* knowledge of the message itself. Part of the information attached to the message is lost if the receiver has already have a clue of it [Clo22]. Deterministic messages bare zero information. Essentially, the whole *theory of information* relies on the basic concept of *entropy* that - in analogy with thermodynamics - provides an average level of *information* or *uncertainty* of a random variable's value. In its discrete form, the *entropy* of a discrete random variable X that can assume any value in $\mathcal{D}_X = \{x_i \in \mathcal{X}\}_{i=1}^N$ with the discrete probability distributions $p(\mathcal{D}_X) = (p_1, \ldots, p_N)$ reads:

$$\mathbb{H}(p(\mathcal{D}_X)) = -\sum_{i=0}^{N} p(X = \boldsymbol{x}_i) \cdot \ln(p(X = \boldsymbol{x}_i)) = -\mathbb{E}_{x \sim p}\left[\ln p(x)\right]$$
(37)

Some examples:

• The entropy of the discrete uniform distribution ${\mathcal U}$ reads:

$$\mathbb{H}(\mathcal{U}(\mathcal{D}_X)) = -\sum_{i=1}^N \frac{1}{N} \cdot \ln\left(\frac{1}{N}\right) = \ln N = \ln |\mathcal{D}_X|$$
(38)

In this case, each $\boldsymbol{x}_i \in \mathcal{D}_X$ has the same probability $\frac{1}{N}$

- The entropy of a deterministic variable of distribution $p(\pmb{x}_i)\delta(\pmb{x}_i-x_j)$ reads:

$$\mathbb{H}(\delta) = -\sum_{i=1}^{N} \delta(\boldsymbol{x}_i - \boldsymbol{x}_j) \cdot \ln \delta(\boldsymbol{x}_i - \boldsymbol{x}_j) = -1 \cdot \ln 1 = 0$$
(39)

• The discrete uniform distribution has the largest entropy:

$$0 \le \mathbb{H}(p) \le \mathbb{H}(\mathcal{U}(\mathcal{D}_X)) = \ln N \tag{40}$$

As a matter of fact,

$$\max_{\sum_{i=1}^{N} p_i = 1} \mathbb{H}(p_1, \dots, p_N) = \sum_{i=1}^{N} \min_{\sum_{i=1}^{N} p_i = 1} p_i \ln p_i \qquad \underset{p_i \in [0,1]}{\operatorname{arg\,min}} p_i \ln p_i = \frac{1}{e}$$

Unfortunately, choosing $p_i = \frac{1}{e}$ does not necessarily yield $\sum_{i=1}^{N} p_i = 1$, but, recalling that $p(\boldsymbol{x}_i) = \frac{1}{\alpha} p(\alpha \boldsymbol{x}_i)$ and assuming $p(\boldsymbol{x}_i) = p(x_j)$, $\forall i, j \in N$, the uniform probability distribution $p = p_i = p(\alpha \boldsymbol{x}_i)$ can be normalized by $\alpha = \frac{e}{N}$, i.e. $p_i = \frac{1}{N}$ with $\boldsymbol{x}_i \sim \mathcal{U}(\mathcal{D}_X)$, whose entropy is $\ln N$ (see Equation (38)) [Cam22].

A practical explanation of what the Shannon's entropy means is provided in [Rio18] and adapted to mechanics in [Clo22]. Consider the Darcy's law for a fixed total discharge Q across a set of N independent and laterally isolated parallel channels of length L and area A, with 1D laminar flow and random hydraulic conductivity. The total discharge

$$Q = \left(\sum_{i=1}^{N} K_i\right) \frac{\Delta h \cdot A}{L}$$

with an overall hydraulic head Δh . The random hydraulic conductivity of each channel K_i is considered as a random variable, sampled from a set of values



Figure 3: Example of the Darcy's law in a set of N parallel channels with random hydraulic conductivity K_i .

 $\mathcal{D}_K = \{K_1, \ldots, K_b\}$ with cardinality $|\mathcal{D}_K| = b$ and $p(K) = (p_1, \ldots, p_b)$. What is then the average number of channels with random hydraulic conductivity $K_i \in \mathcal{D}_K$ that delivers the total discharge Q? If one considers a uniform discrete distribution of hydraulic conductivity $p_i = \frac{1}{b}$, $K \sim \mathcal{U}(\mathcal{D}_K)$, given the fact that one could pick N times any value of K_i in \mathcal{D}_K , the probability of a set of N independent channels is defined as $p\left(\{K_i\}_{i=1}^N\right) = b^{-N}$ and the average number of channels reads:

$$\mathbb{E}_{n\sim p}\left[N\right] = -\mathbb{E}_{n\sim p}\left[\log_{b} p\left(\left\{K_{i}\right\}_{i=1}^{N}\right)\right] = H(p_{1},\ldots,p_{N}) - \frac{1}{\ln b} \qquad (41)$$

For uniform distribution, $H(p_1, \ldots, p_N)$ is a positive monotonic function of N. $-\frac{1}{\ln b}$ plays the role of normalization constant. Instead, for a generic distribution of the K_i , the likelihood of N reads:

$$p\left(\{K_i\}_{i=1}^N\right) = \prod_{i=1}^N p_i^{b_i} = b^N\left(\sum_{i=1}^N f_i \log_b p_i\right) = b^{-N \cdot (\mathbb{D}_{KL}((f_1, \dots, f_N) || (p_1, \dots, p_N)) + \mathbb{H}(f_1, \dots, f_N))}$$
(42)

with b_i being the number of times that K_i is picked and $f_i = \frac{b_i}{N}$ its empirical frequency $(\sum_{i=1}^{b} f_i = 1)$. For the law of large numbers (by Bernoulli), $f_i \xrightarrow[N \to \infty]{} p_i$, so in this case, when the number of channels is very large, the likelihood of N reads:

$$p\left(\{K_i\}_{i=1}^N\right) = b^{-N \cdot \mathbb{H}(p_1,\dots,p_N)}$$

$$\tag{43}$$

The expected number of channels is the Shannon's entropy of the set of pipes. A large entropy $H(p_1, \ldots, p_N) \to 0$ leads to a low likelihood of N, since the total discharge value remains highly uncertain, yet less likely to occur, whereas a decrease in $H(p_1, \ldots, p_N)$ translates into a more likely-to-occur configuration of N channels. In a deterministic case, only one possible configuration is possible, with probability $p\left(\{K_i\}_{i=1}^N\right) = 1$.

Moreover, any b_i follows the Binomial distribution $b_i \sim \mathcal{B}(N, p_i) = {N \choose b_i} p_i^{b_i} (1 - b_i)^{b_i} p_i^{b_i} p_i^{b_i} (1 - b_i)^{b_i} p_i^{b_i} (1 - b_i)^{b_i} p_i^{b_i} (1 - b_i)^{b_i} p_i^{b_i} p_i^{b_i} p_i^{b_i} (1 - b_i)^{b_i} p_i^{b_i} p_i^{$

 $(p_i)^{N-b_i}$ and $\mathbb{E}_{f_i}(f_i) = \frac{1}{N} \mathbb{E}_{b_i \sim \mathcal{B}(N, p_i)}[b_i] = p_i^7$. Therefore, Equation (43) corresponds to the likelihood for an average value of b_i , i.e., of having $p\left(\{K_i\}_{i=1}^N\right) \approx$ $\prod_{i=1}^{N} p_i^{\mathbb{E}[b_i]} = \prod_{i=1}^{N} p_i^{Np_i}$. Random configurations of N channels with large entropy deliver a highly unknown total discharge, but they occur with low probability. Instead, "typical" configurations, with average number of channels with hydraulic conductivity K_i have low entropy and high probability to occur. The number of possible combinations (with repetitions) of values of hydraulic conductivity is $N_b = \frac{(N+b-1)!}{b! \cdot (N-1)!}$ and it correspond to the number of possible total discharge values $Q = \left(\sum_{i=1}^{N} f_i K_i\right) \frac{\Delta h \cdot A}{L}$. Each value Q is attained $N_Q = \frac{N!}{\prod_{i=1}^{N} b_i!}$ unique times with a probability that is the sum over N_Q the disjoints configurations $p_Q = N_Q \cdot p\left(\{K_i\}_{i=1}^N\right)$. If one considers the "typical" configurations, corresponding to N very very large, $p_Q = N_Q \cdot p\left(\{K_i\}_{i=1}^N\right) \approx 1 \ [\text{Clo22}]^8$. This implies $\frac{1}{N} \frac{\ln N_Q}{\ln b} \approx \mathbb{H}(p\left(\{K_i\}_{i=1}^N\right))$, which means that the entropy approximates - in logarithmic scale - the average number of i.i.d. configurations (or *states*) that delivers the total discharge Q, normalized with the respect to the base b and of the number of channels. In a non-"typical" configura-tion, $N_Q \leq b^{-N \cdot \mathbb{H}((f_1, \dots, f_N) || (p_1, \dots, p_N))}$ with $\mathbb{H}((f_1, \dots, f_N) || (p_1, \dots, p_N))$ representing the cross-entropy between the empirical frequencies approximating the probability distribution of K_i . The cross-entropy is defined as:

$$\mathbb{H}((p_1, \dots, p_N) \| (q_1, \dots, q_N)) = -\sum_{i=1}^N p_i \cdot \ln q_i$$
(44)

1.4.1 Why the logarithm?

Shannon's observed that, being the *uncertainty* or *surprise* associated to a random variable X inversely proportional to its probability of occurrence $P(\omega)$, deterministic events provide poor information and two events measured separately provide a total amount of information equal to the sum of the two single contributions. Based on this evidence, Shannon defined the *self-information* of an event $\omega \in (\Omega, \mathcal{E}, P)$ as a strictly decreasing monotonic function of the probability P(X):

$$\mathbb{I}(\omega) = f(P(\omega)) \ge 0 \tag{45}$$

⁸When N is very large, the Stirling's approximation applies: $N! \approx (2\pi N)^{\frac{1}{2}} \cdot N^N \cdot e^{-N}$ and $\ln b_i! = b_i \cdot \ln b_i - b_i = Np_i \cdot \ln(Nb_i) - Np_i$. With this approximation, $N_Q \approx (2\pi N)^{\frac{1}{2}} b^{N \cdot \log_b(e)} \cdot (\mathbb{H}(f_1, ..., f_N)) + \mathbb{D}_{KL}((f_1, ..., f_N)) \| (p_1, ..., p_N)) = (2\pi N)^{\frac{1}{2}} p^{-\log_b(e)}(N) \cdot e^{N \mathbb{D}_{KL}((f_1, ..., f_N)) \| (p_1, ..., p_N))}$ [Clo22]. More at https://michael-franke.github.io/intro-data-analysis/the-maximum-entropy-principle.html

with $f : [0,1] \mapsto [0,\infty)$ such that $\mathbb{I}(\omega) = 0$ if $P(\omega) = 1$ and $\mathbb{I}(\omega) = \infty$ if $P(\omega) = 0$. f has to be additive for two independent events $(\omega_1, \omega_2) \in \Omega^2$, i.e. $I(\omega_1 \cap \omega_2) = \mathbb{I}(\omega_1) + \mathbb{I}(\omega_2)$ with $P(\omega_1 \cap \omega_2) = P(\omega_1) \cdot P(\omega_2)$. The additive property implies that:

$$\mathbb{I}(\omega_1 \cap Y) = f(P(\omega_1 \cap \omega_2)) = f(P(\omega_1) \cdot P(\omega_2)) = \mathbb{I}(\omega_1) + \mathbb{I}(\omega_2) = f(P(\omega_1)) + f(P(\omega_2))$$

$$(46)$$

Equation (46) happens to be a Cauchy's logarithmic functional equation, whose only monotone solution is in the form $f(x) = -\log_b(x) = -\frac{\ln x}{\ln b}$, with b > 1 (since $f: [0, 1] \mapsto \infty$) and

$$\mathbb{I}(\boldsymbol{\omega}) = f(P(\boldsymbol{\omega})) = -\log_b P(\boldsymbol{\omega}) = \frac{-\ln P(\boldsymbol{\omega})}{\ln b}, \quad b > 1$$
(47)

Therefore, the Shannon's entropy is the expected self-information of a random variable, quantifying how surprising the random variable is *on average*:

$$\mathbb{H}(p(\omega_1),\ldots,p(\omega_N)) = \sum_{\omega_i \in \omega}^n p(\omega_i) \cdot I(\omega_i) = \mathbb{E}_{\omega_i \sim p_i} \left[I(\boldsymbol{\omega}) \right]$$
(48)

1.4.2 Shannon's entropy: some fundamental property

The Shannon's entropy of random variable X can be defined as the limit (in probability sense) of the self-information of a uniform random variable. This is proven by the following theorem:

Theorem 20 (Mallat and Campagne [Cam22]). If X_i are *i.i.d.*, with probability law P_X , with a set of realization $x_i \in \mathcal{D}_X$, with probability distribution $p(x_i)$:

$$\forall \epsilon > 0, \lim_{N \to \infty} \mathbb{P}\left[\left| -\frac{1}{N} \ln P(X_1, \dots, X_N) - \mathbb{H}(P(X)) \right| \le \epsilon \right] = 1$$
(49)

This result is rather fundamental. Mallat provides an insightful interpretation of Theorem 20: the self-information $-\frac{1}{N} \ln P_X(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N)$ is concentrated on a surfaces defined by the equation $-\frac{1}{N} \ln P_X(\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N) \approx \mathbb{H}(X)$, provided an arbitrary small thickness ϵ . In machine learning practice, algorithms are trained by assuming datasets of i.i.d. samples, trying to approximate the underlying - yet unknown - entropy $\mathbb{H}(P(\boldsymbol{X}))$, associated with the true data probability distribution. However, in order to achieve a good approximation of the real entropy, several i.i.d. examples are required, so to refine the thickness ϵ .

• The joint entropy of two discrete random variables X and Y reads:

$$\mathbb{H}(p(\boldsymbol{X},\boldsymbol{Y})) = -\sum_{i,j=1}^{N} p(\boldsymbol{X} = \boldsymbol{x}_i, \boldsymbol{Y} = \boldsymbol{y}_j) \ln p(\boldsymbol{X} = \boldsymbol{x}_i, \boldsymbol{Y} = \boldsymbol{y}_j) =$$
(50)
= $-\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p} \left[\ln(p(\boldsymbol{X},\boldsymbol{Y}))\right]$

• The conditional entropy of two discrete random variables X and Y reads:

$$\mathbb{H}(p(\boldsymbol{Y}|\boldsymbol{X} = \boldsymbol{x}_i)) = -\sum_{j=1}^{N} p(\boldsymbol{Y} = \boldsymbol{y}_j | \boldsymbol{X} = \boldsymbol{x}_i) \ln p(\boldsymbol{Y} = y_j | \boldsymbol{X} = \boldsymbol{x}_i) =$$
$$= -\mathbb{E}_{\boldsymbol{y} \sim p(\boldsymbol{y}|\boldsymbol{x})} \left[\ln(p(\boldsymbol{Y}|\boldsymbol{X} = \boldsymbol{x}_i)) \right]$$
(51)

and

٠

$$\mathbb{H}(p(\boldsymbol{Y}|\boldsymbol{X})) = -\mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x})} \left[\mathbb{H}(\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}=\boldsymbol{x})) \right] =$$

$$= -\sum_{i,j=1}^{N} p(\boldsymbol{Y}=\boldsymbol{y}_{j}|\boldsymbol{X}=\boldsymbol{x}_{i}) p(\boldsymbol{X}=\boldsymbol{x}_{i}) \ln \boldsymbol{p}(\boldsymbol{Y}=\boldsymbol{y}_{j}|\boldsymbol{X}=\boldsymbol{x}_{i}) =$$

$$= -\sum_{i,j=1}^{N} p(\boldsymbol{X}=\boldsymbol{x}_{i},\boldsymbol{Y}=y_{j}) \ln p(\boldsymbol{Y}=\boldsymbol{y}_{j}|\boldsymbol{X}=\boldsymbol{x}_{i}) = -\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p} \left[\ln p(\boldsymbol{Y}|\boldsymbol{X}) \right]$$

$$(52)$$

$$\mathbb{H}(p(\boldsymbol{X},\boldsymbol{Y})) = \mathbb{H}(p(\boldsymbol{Y}|\boldsymbol{X})) + \mathbb{H}(p(\boldsymbol{X})) = \mathbb{H}(p(\boldsymbol{X}|\boldsymbol{Y})) + \mathbb{H}(p(\boldsymbol{Y})) \quad (53)$$

- The Kullback-Leibler distance $\mathbb{D}_{KL}(p||q)$ has the following properties:
 - $\mathbb{D}_{KL}(p||q) \ge 0, \text{ since, due to the concavity of } \ln(x), \ln(y) \ln(x) \le (\ln x)' \cdot (y x) \text{ and setting } x = p \text{ and } y = q \text{ the following expression holds:}$

$$\mathbb{E}_{p}\left[\ln\frac{q}{p}\right] = -\mathbb{D}_{KL}\left(p\|q\right) \le \mathbb{E}_{p}\left[\frac{q-p}{p}\right] = 0$$

 $-\mathbb{D}_{KL}(p||q) = 0$ implies that $p \equiv q$ if and only if

 $\operatorname{supp}(p) \cap \operatorname{supp}(q) \neq \emptyset$

Otherwise, when p and q have two disjoint supports, it often occurs that $\mathbb{D}_{KL}(p||q) = 0$ or $\mathbb{D}_{KL}(p||q) = 0 \to \infty$. This pathological situation occurs quite often in practice, especially for genereative algorithms such as GAN (see [Goo+14]).

- The Kullback-Leibler distance between p(X, Y) and $p(X) \cdot p(Y)$ is called *mutual information* $\mathbb{I}(X, Y)$ [Che+16; Cam22]:

$$\mathbb{I}(p(\boldsymbol{X},\boldsymbol{Y})) = \mathbb{H}(p(\boldsymbol{Y})) - \mathbb{H}(p(\boldsymbol{Y}|\boldsymbol{X})) =$$

= $\mathbb{H}(p(\boldsymbol{X})) - \mathbb{H}(p(\boldsymbol{X}|\boldsymbol{Y})) = \mathbb{D}_{KL}(p(\boldsymbol{X},\boldsymbol{Y}) || p(\boldsymbol{X}) \cdot p(\boldsymbol{Y}))$ (54)

The proof is straightforward, since $\mathbb{H}(p(\mathbf{X}, \mathbf{Y})) = \mathbb{H}(p(\mathbf{Y}|\mathbf{X})) + \mathbb{H}(p(\mathbf{X})) = \mathbb{H}(p(\mathbf{X}|\mathbf{Y})) + \mathbb{H}(p(\mathbf{Y})) \ge 0$ and

$$\mathbb{I}(p(\boldsymbol{X}, \boldsymbol{Y})) = -\mathbb{H}(p(\boldsymbol{X}, \boldsymbol{Y})) + \mathbb{H}(p(\boldsymbol{X})) + \mathbb{H}(p(\boldsymbol{Y})) =$$
$$= \mathbb{E}_{(\boldsymbol{x}, \boldsymbol{y}) \sim p} \left[\frac{p(\boldsymbol{X}, \boldsymbol{Y})}{p(\boldsymbol{X}) \cdot p(\boldsymbol{Y})} \right] = \mathbb{D}_{KL} \left(p(\boldsymbol{X}, \boldsymbol{Y}) \| p(\boldsymbol{X}) \cdot p(\boldsymbol{Y}) \right) \ge 0$$

- $\mathbb{I}(p(\boldsymbol{X}, \boldsymbol{Y}))$ is strictly connected to the notion of *self-information* presented in Section 1.4.1. $\mathbb{I}(p(\boldsymbol{X}, \boldsymbol{Y}))$ if $\boldsymbol{X}, \boldsymbol{Y}$ are two independent events.
- The notion of mutual information is widely used in practice. Maximizing $\mathbb{I}(p(\boldsymbol{X}, \boldsymbol{Y}))$ at constant $\mathbb{H}(p(\boldsymbol{X}))$ requires $\mathbb{H}(p(\boldsymbol{Y}|\boldsymbol{X})) \to 0$, i.e., forcing the *disentanglement* between \boldsymbol{X} and \boldsymbol{Y} [Che+16].
- The mutual information enters in one of the alternative definitions of the ELBO, in Equation (88):

$$ELBO = \mathbb{E}_{\boldsymbol{x} \sim p} \left[\mathbb{E}_{\boldsymbol{z} \sim q_{\phi}(\boldsymbol{Z} \mid \boldsymbol{x})} \left[\ln p_{\boldsymbol{\theta}}(\boldsymbol{Z} \mid \boldsymbol{X}) \right] \right] - \\ - \mathbb{D}_{KL} \left(p_{\boldsymbol{\theta}}(\boldsymbol{Z}) \| q_{\phi}(\boldsymbol{Z}) \right) - \mathbb{I}(\boldsymbol{Z}, \boldsymbol{X})$$
(55)

Maximize the ELBO demands to minimize $\mathbb{I}(\boldsymbol{Z}, \boldsymbol{X}) = \mathbb{H}(q_{\phi}(\boldsymbol{X})) - \mathbb{H}(q_{\phi}(\boldsymbol{X}|\boldsymbol{Z}))$ [Mak19], i.e. find a *joint data distribution* $q_{\phi}(\boldsymbol{X}, \boldsymbol{Z})$ "highly generative", i.e. with a large conditional entropy $\mathbb{H}(q_{\phi}(\boldsymbol{X}|\boldsymbol{Z}))$. In this sens, the statistical model associate to q_{ϕ} can widely span the dimensionality of whole dataset \mathcal{X} .

1.4.3 From discrete to continuous

The definition of entropy in Equation (37) can be easily extended to continuous random variables with \mathcal{X} isomorphic to \mathbb{R}^{d_X} . In this case, it is called *differential entropy* [Cam22]:

$$\mathbb{H}_d(p(\boldsymbol{X})) = -\int_{\mathcal{X}} p(\boldsymbol{x}) \cdot \ln(p(\boldsymbol{x})) \cdot \mu(d\boldsymbol{x})$$
(56)

Compared to $\mathbb{H}(p(\mathbf{X}))$, $\mathbb{H}_d(p(\mathbf{X}))$ is not always positive. For instance, $\mathbb{H}_d[\mathcal{U}([0,b])] = \log b < 0$ if b < 1. The differential entropy of a multivariate normal distribution reads:

$$\mathbb{H}_{d}(\mathcal{N}_{(\boldsymbol{\mu},\boldsymbol{\Sigma})}(\boldsymbol{X})) = -\ln\left((2\pi)^{-\frac{n}{2}}\det(\boldsymbol{\Sigma})^{-\frac{1}{2}}\right)\mathbb{E}_{\boldsymbol{x}\sim\mathcal{N}_{(\boldsymbol{\mu},\boldsymbol{\Sigma})}}\left[(\boldsymbol{x}-\boldsymbol{\mu})\otimes\boldsymbol{\Sigma}^{-1}\left(\boldsymbol{x}-\boldsymbol{\mu}\right)\right] = \frac{d_{X}}{2}\ln 2\pi + \frac{1}{2}\mathrm{Tr}\left(\boldsymbol{\Sigma}\right)$$
(57)



Figure 4: Differential entropy $\mathbb{H}_d(p_{\epsilon})$ of a normal distribution $p_{\epsilon} = \mathcal{N}(0, 1)$.

The differential entropy of a normal distribution increases with the dimension of the data space d_X .

Moreover, $\operatorname{Tr}(\Sigma)$ is dominated by the spectral radius $\rho(\Sigma) = \sup_{\lambda \in \sigma(\Sigma)} |\lambda|$. In other words, the surprise related to a random multivariate normal variables increase with the dimension of the space in which the variable lives and with the correlation between its components Σ . The factor $\frac{1}{2}\operatorname{Tr}(\Sigma)$ represents a factor of "scale", since $\mathbb{H}_d(bX) = \mathbb{H}_d(X) + \ln b$, in analogy with what presented for its discrete counterpart in Section 1.4.

1.4.4 Typical set

In the discrete case, the approximation evoked in the Section 1.4.2 for large number of observations holds because of the well-know law of large numbers (the weak form in Theorem 3), which implies that the estimated mean of a set of i.i.d. variables with the same average converges in probability to the true mean. In other words, Equation (43) is formally expressed as (see [Cam22]):

$$\lim_{N \to +\infty} \mathbb{P}\left[\left| \frac{1}{N} \log_b p\left(\{K_i\}_{i=1}^N \right) + \mathbb{H}(p_1, \dots, p_N) \right| \le \varepsilon \right] = 1, \quad \forall \varepsilon > 0$$
 (58)

Equation (58) defines the "typical" set as:

$$A_{\varepsilon}^{(N)} = \left\{ (K_1, K_2, \dots, K_N) \in \mathcal{D}_K^N | \\ \mathbb{P}\left[\left| \frac{1}{N} \log_b p\left(\{K_i\}_{i=1}^N \right) + \mathbb{H}(p_1, \dots, p_N) \right| > 1 - \varepsilon \right] \right\}$$
(59)

Equation (58) represents the extension of the typical set to a large number of observations. In this case, almost all the observations fall belong to the typical set, $\forall \varepsilon > 0$. Therefore, the concept of typical set allows defining the notion on Shannon's entropy for a discrete variable as well as to provide the lower and upper bounds of its probability distribution that read:

$$b^{-N(\mathbb{H}(p)+\varepsilon)} \le p\left(\{K_i\}_{i=1}^N\right) \le b^{-N(\mathbb{H}(p)-\varepsilon)} \tag{60}$$

Equation (60) provided that for the typical set, the additivity of the entropy function allows to write $N\mathbb{H}(p) = \mathbb{H}(p_1, p_2, \ldots, p_N)$, which does not depend on the realization itself. Moreover, the observations reach the *asymptotic equipartion* of states, by randomly occupying the typical set. Moreover, the inequality in Equation (60) allows to compute the cardinality of the typical set, since:

$$1 = \sum_{\{K_i\}_{i=1}^{N} \in \mathcal{D}_K^N} p\left(\{K_i\}_{i=1}^N\right) \ge \sum_{\{K_i\}_{i=1}^{N} \in A_{\varepsilon}^{(N)}} p\left(\{K_i\}_{i=1}^N\right) \ge$$

$$\ge \sum_{\{K_i\}_{i=1}^{N} \in A_{\varepsilon}^{(N)}} b^{-N(\mathbb{H}(p)+\varepsilon)} = \operatorname{card}\left(A_{\varepsilon}^{(N)}\right) b^{-N(\mathbb{H}(p)+\varepsilon)}$$

$$1 - \varepsilon \le \sum_{\{K_i\}_{i=1}^{N} \in A_{\varepsilon}^{(N)}} p\left(\{K_i\}_{i=1}^N\right) \le$$

$$\le \sum_{\{K_i\}_{i=1}^{N} \in A_{\varepsilon}^{(N)}} p\left(\{K_i\}_{i=1}^N\right) b^{-N(\mathbb{H}(p)-\varepsilon)} = \operatorname{card}\left(A_{\varepsilon}^{(N)}\right) b^{-N(\mathbb{H}(p)-\varepsilon)}$$
(61)

In the continuous case, the equivalent of Equation (58) reads:

$$\lim_{N \to +\infty} \mathbb{P}\left[\left| \frac{1}{N} \ln p\left(\{K_i\}_{i=1}^N \right) + \mathbb{H}_d\left(p\left(\boldsymbol{X} \right) \right) \right| \le \varepsilon \right] = 1, \quad \forall \varepsilon > 0$$
 (62)

since:

$$\lim_{N \to +\infty} \frac{1}{N} \ln p\left(\left\{K_i\right\}_{i=1}^N\right) = \mathbb{E}_{\boldsymbol{k} \sim p}\left[\ln p\left(\boldsymbol{X}\right)\right]$$
(63)

However, the number of elements in the typical set is infinite, in the continuous case. Therefore, one should define the typical volume $\Omega\left(A_{\varepsilon}^{(N)}\right) = \int_{A_{\varepsilon}^{(N)}} d\boldsymbol{x}$, in the Lebesgue measure, bounded as follows (see Equation (61), with b = 2):

$$(1-\varepsilon) \cdot 2^{N(\mathbb{H}_d(K)-\varepsilon)} \le \Omega\left(A_{\varepsilon}^{(N)}\right) \le 2^{N(\mathbb{H}_d(K)+\varepsilon)}$$
(64)

since

$$1 = \int_{\Omega_{K}} p(k) dk \ge \int_{A_{\varepsilon}^{(N)}} p(k) dk \ge 2^{-N(\mathbb{H}_{d}(p)+\varepsilon)}$$

$$1 - \varepsilon \le \int_{A_{\varepsilon}^{(N)}} p(k) dk \le 2^{-N(\mathbb{H}_{d}(p)-\varepsilon)} \cdot \Omega\left(A_{\varepsilon}^{(N)}\right)$$
(65)

Therefore, thanks to Equation (65), one assumes that the typical volume has a size $\approx \frac{1}{p(K)}$, the probability being constant regardless the realization and close to the inverse of the size of the volume set (uniform), and that the differential entropy $\mathbb{H}_d(p)$ can be seen as the logarithm to the base 2 of the characteristic *N*-dimensional length of the typical volume tessellation.

1.5 The principle of maximum entropy (MaxEnt)

Any statistical model conceived to infer a physical problem must be compatible with the available observations (\boldsymbol{x}_k, y_k) . The latter serve as verification baseline. In particular, the easiest strategy to calibrate a statistical model is to match the average observations $\mu_{y_k}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{x}\sim p}[y_k(\boldsymbol{x})]$, with $p(\boldsymbol{x})$ being the true (yet unknown) data probability distribution. Any statistical model $\boldsymbol{h}_{\boldsymbol{\theta}}$ formulated to infer new samples of the quantity of interest y, is associate to a parametric probability distribution $p_{\boldsymbol{\theta}}$, whose differential entropy must be compatible with the constraints imposed by the observations. However, these constraints on the observations are not sufficient to approximate the "true" probability distribution p, based on the statistical model $p_{\boldsymbol{\theta}}$. According to [Jay57], $p_{\boldsymbol{\theta}}$ must maximize its differential entropy, so that the typical volume is the largest as possible and the probability distribution on it is the most uniform possible. This framework is resolved by the following theorem [Cam22]:

Theorem 21. Gibbs-Boltzmann theorem

Provided a family of probability distributions $q \in \mathcal{H}$, the solution to the following constrained optimization problem:

$$\begin{cases} \max_{q \in \mathcal{H}} \mathbb{H}_d(q) = \max_q \left(-\int_{\Omega_X} \ln q\left(\boldsymbol{x} \right) \cdot q\left(\boldsymbol{x} \right) d\boldsymbol{x} \right) \\ c_k\left(q\right) = \mu_{y_k} - \int_{\Omega_X} y_k\left(\boldsymbol{x} \right) \cdot q\left(\boldsymbol{x} \right) d\boldsymbol{x} = 0, \quad c_k : \mathbb{R}^n \to \mathbb{R}, 1 \le k \le K \quad (67) \end{cases}$$

if it exists it is defined as
$$p_{\theta} \in \mathcal{H}$$
, which reads:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \operatorname*{arg\,max}_{q \in \mathcal{H}, \mathbf{c}: \boldsymbol{c} = \boldsymbol{0}} \mathbb{H}_{d}(q) = \frac{e^{-\sum_{k=1}^{K} y_{k}(\boldsymbol{x}) \cdot \boldsymbol{\theta}_{k}}}{Z}$$

with Z a normalization constant. Moreover, $\mathbb{H}_d(p_{\theta}) \geq \mathbb{H}_d(p)$, $\forall \theta \in \mathcal{H}_{\theta} \subset \mathcal{H}$ and if $p_{\theta} = p$, then $\mathbb{H}_d(p_{\theta}) = \mathbb{H}_d(p)$

Proof. The proof of the Gibbs-Boltzmann theorem is proven thanks to the Kuhn-Tucker theorem. The set

$$\mathcal{K} = \left\{ q | c(q) = 0, c_k \in \mathcal{C}^1 \left(\mathbb{R}^n \right), \forall k = 0, \dots, K \right\}$$

represents the set of continuous and differentiable constraint functions. If the derivatives $\frac{dc_k}{dq}$ are linearly independent, and if $\exists w \in \mathbb{R}$ such that its Gâteaux derivative⁹ along w satisfies the expression:

$$D_w c_k(q) = 0, k = 0, \dots, K$$

the constraints are "qualified" (or active). In this context, the Kuhn-Tucker theorem grants the existence of a probability distribution $p_{\theta} \in \mathcal{H}_{\theta} \cap \mathcal{K}$ and of a set of coefficients $\lambda_1, \ldots, \lambda_K$, with $\lambda_i \in \mathbb{R}$, for which:

$$D_{q}\mathbb{H}_{d}(p_{\theta}) + \sum_{k=0}^{K} D_{q}c_{k}(p_{\theta}) = 0, \quad \forall q \in \mathcal{K}$$
(68)

The result of Equation (68) can be recast into the condition for which the Lagrangian function defined as:

$$L(q, \boldsymbol{\lambda}) = \mathbb{H}_{d}(q) + \sum_{k=1}^{K} \lambda_{k} \cdot c(q) + \lambda_{0} \left(\int_{\Omega_{X}} q(\boldsymbol{x}) \cdot \mu(d\boldsymbol{x}) - 1 \right)$$

 p_{θ} represents a stationary point of the Lagrangian functional, since the Kuhn-Tucker theorem in Equation (68) can be reformulated as:

$$D_{q}L(p_{\theta}, \boldsymbol{\lambda}) = \frac{\partial L}{\partial t}(p_{\theta} + tq, \boldsymbol{\lambda})\Big|_{t=0} =$$

$$= -\int_{\Omega_{X}} q(\boldsymbol{x}) \cdot (\ln p_{\theta}(\boldsymbol{x}) + 1) \cdot \mu(d\boldsymbol{x}) -$$

$$-\sum_{k=1}^{K} \lambda_{k} \cdot \int_{\Omega_{X}} y_{k}(\boldsymbol{x}) \cdot q(\boldsymbol{x}) \cdot \mu(d\boldsymbol{x}) + \lambda_{0} = 0, \quad \forall q \in \mathcal{K}$$
(69)

which leads to:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = p_{\boldsymbol{\lambda}}(\boldsymbol{x}) = e^{\lambda_0 - 1} \cdot e^{-\sum_{k=1}^{K} \lambda_k \cdot y_k(\boldsymbol{x})}$$
(70)

Equation (70) implies that the penalty coefficients λ_i are the parameters that structure the space $\mathcal{H}_{\boldsymbol{\theta}}$. However, the definition of $p_{\boldsymbol{\theta}}$ in Equation (70) needs to be normalized, in order to assure that $\int_{\Omega_X} p_{\boldsymbol{\theta}}(\boldsymbol{x}) \, \mu(d\boldsymbol{x}) = 1$. This is achieved if:

$$\int_{\Omega_{X}} e^{\lambda_{0}-1} \cdot e^{-\sum_{k=1}^{K} \lambda_{k} \cdot y_{k}(\boldsymbol{x})} \cdot \mu(d\boldsymbol{x}) = 1 \iff$$

$$\iff Z = e^{1-\lambda_{0}} = \int_{\Omega_{X}} e^{-\sum_{k=1}^{K} \lambda_{k} \cdot y_{k}(\boldsymbol{x})} \cdot \mu(d\boldsymbol{x})$$

$$\xrightarrow{9} D_{w}f(q) = \frac{df(q+tw)}{dt}\Big|_{t=0}$$
(71)
The stationarity of the Lagrangian function also implies that p_{θ} must respect the constraint and force the expected value to converge to the "true" one $(\boldsymbol{\mu})$:

$$\nabla_{\lambda} \mathcal{L} \left(p_{\boldsymbol{\theta}}, \boldsymbol{\lambda} \right) = \mathbf{0} \Rightarrow \int_{\Omega_{X}} y_{k} \cdot p \left(\boldsymbol{x} \right) \mu(d\boldsymbol{x}) = \mu_{y_{k}} \left(\boldsymbol{x} \right) = \int_{\Omega_{X}} y_{k} \cdot p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) \mu(d\boldsymbol{x}) \quad (72)$$

The family of exponential probability distributions Theorem 21 states that the probability distribution that maximizes the entropy belongs to the exponential family. Moreover, the probability distribution that maximizes the entropy allows to describe the log-likelihood function as:

$$\ln p_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right) = -\ln Z - \sum_{k=1}^{K} y_{k}\left(\boldsymbol{x}\right) \cdot \theta_{k} = -\ln Z - \left\langle \boldsymbol{y}\left(\boldsymbol{x}\right), \boldsymbol{\theta} \right\rangle$$
(73)

The gradient of the term $-\ln Z$ reads:

$$-\boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln Z = \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \right]$$
(74)

which implies the gradient of log-likelihood function reads:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) = \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \right] - \boldsymbol{y} \left(\boldsymbol{x} \right)$$
(75)

and the following property:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) = p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) = \left(\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \right] - \boldsymbol{y} \left(\boldsymbol{x} \right) \right) \cdot p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right)$$
(76)

Moreover, the Fisher's information computed as the mean Hessian in Equation (28) corresponds to the covariance of the observations $y_k(\boldsymbol{x})$ if $p_{\boldsymbol{\theta}}$ maximizes the entropy [Cam22]:

$$\nabla_{\boldsymbol{\theta}} \otimes \nabla_{\boldsymbol{\theta}} \left(\ln p_{\boldsymbol{\theta}} \right) = \int_{\Omega_{X}} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) \otimes \boldsymbol{y} \left(\boldsymbol{x} \right) \mu \left(d\boldsymbol{x} \right) =$$

$$= \int_{\Omega_{X}} \left(\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \right] - \boldsymbol{y} \left(\boldsymbol{x} \right) \right) \otimes \boldsymbol{y} \left(\boldsymbol{x} \right) \cdot p_{\boldsymbol{\theta}} \left(\boldsymbol{x} \right) \cdot \mu \left(d\boldsymbol{x} \right) =$$

$$= \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \right] \otimes \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \right] - \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \otimes \boldsymbol{y} \left(\boldsymbol{X} \right) \right] =$$

$$= -\mathbb{C}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left(\boldsymbol{y} \left(\boldsymbol{X} \right) \right)$$

$$(77)$$

Remark 22. The interesting aspect of Equation (77) is that it depends only on the information provided by the observations $y_k(\boldsymbol{x})$. In the case of neural networks, y_k depends on the weights themselves though, since $y_k = h_{\boldsymbol{\theta}}(\boldsymbol{x}_k)$ [Cam22]. The maximum entropy principle has a strong analogy with the Fisher information theory, since the value of the "best" estimator $\hat{\boldsymbol{\theta}}$ that maximizes the

log-likelihood is unbiased (see Equation (30) for the definition). Equation (75) adds the fact that $\hat{\theta}$ makes the observations $\boldsymbol{y}(\boldsymbol{X})$ converge towards their average values, that maximize the log-likelihood. Moreover, for a sufficiently large dataset $N \to +\infty$, the MLE $\hat{\theta}_N$ tends to be normally distributed, i.e., its probability distribution tends to belong to the exponential family. In other words, the MLE not only maximizes the (log-)likelihood but also the entropy of the underlying dataset. In other words, the maximum entropy of a statistical model is reached by maximizing the log-likelihood of an exponential probability distribution [Cam22].

Remark 23. The quest for the best estimator can be seen as a minimax optimization problem. The optimization targets the approximation of the unknown differential entropy lower bound $\mathbb{H}_d(p)$ represented by the "true" data probability distribution $(\mathbb{H}_d(p_{\theta}) \geq \mathbb{H}_d(p))$. In order to achieve this lower bound, a statistical model \mathcal{H}_{θ} is chosen (the choice of the architecture of the \mathcal{NN}) among all the possible statistical models \mathcal{H} . By adjusting the parameters θ , the parametric probability distribution $p_{\theta} \in \mathcal{H}_{\theta}$ induced by the statistical model must realize the maximum of its differential entropy and comply with the observations (see Theorem 21). To summarize, the optimization task that neural networks try to accomplish is expressed by the following minimax problem:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{arg\,min}} \max_{\boldsymbol{p}_{\boldsymbol{\theta}} \in \mathcal{H}_{\boldsymbol{\theta}}} \mathbb{H}_{d} \left(p_{\boldsymbol{\theta}} \left(Y | \boldsymbol{X} \right) \right)$$
(78)

1.6 Learning with variational inference and reconstruction

In supervised learning, the dataset is labeled, i.e.

$$\mathcal{D}_{XY} = \{(oldsymbol{x}_i, oldsymbol{y}_i) \in \mathcal{X} imes \mathcal{Y}\}_{i=1}^N$$

The learning task is therefore to approximate the conditional probability distribution $p(\boldsymbol{y}|\boldsymbol{x})$ with a statistical model $\mathcal{H}_{\boldsymbol{\theta}}$. In this case the "best" approximation can be found by minimizing the conditional entropy as follows:

$$\hat{\boldsymbol{\theta}}(\mathcal{D}_{XY}) = \underset{\boldsymbol{\theta}\in\Theta}{\operatorname{arg\,min}} \mathbb{E}_{(x,y)\in\mathcal{D}_{XY}} \ln \frac{1}{p_{\boldsymbol{\theta}}(\boldsymbol{Y}|\boldsymbol{X})}$$
(79)

The first order gradient method adopted to minimize the loss function reads (see Equation (63)):

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta^{(i)} \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right), \quad \eta^{(i)} \in \mathbb{R}^+$$

The analogy with the concept of MaxEnt and MLE exposed in Remark 22, in order to minimize the negative log-likelihood \mathcal{NLL} (equivalent to maximize

the (log-)likelihood, as expressed by Equation (20)) the MLE can be found according to the following recursive formula [Cam22]:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta^{(i)} \left(\boldsymbol{\mu}_{\boldsymbol{y}} - \mathbb{E}_{\boldsymbol{x} \sim \boldsymbol{p}_{\boldsymbol{\theta}^{(i)}}} \left[\boldsymbol{y} \left(\boldsymbol{X} \right) \right] \right), \quad \eta^{(i)} \in \mathbb{R}^+$$
(80)

By identification,

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right) = \boldsymbol{\mu}_{\boldsymbol{y}} - \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^{(i)}}}\left[\boldsymbol{y}\left(\boldsymbol{X}\right)\right]$$

The recursive algorithm in Equation (80) ends when the entropy of the parametric probability distribution is close to its maximum value. However, in machine learning applications, the dataset at stake is sampled from an unknown probability distribution $p(\boldsymbol{x})$. There is no certainty that $p \in \mathcal{H}_{\boldsymbol{\theta}}$. This implies that computing $\boldsymbol{\mu}_{\boldsymbol{y}} = \mathbb{E}_{\boldsymbol{x} \sim p} [\boldsymbol{y}(\boldsymbol{X})]$ is rather intricate since $\boldsymbol{\theta}^* \in \Theta$ such that $\boldsymbol{x} \sim p = p_{\boldsymbol{\theta}^*}$ may not exist. $\boldsymbol{\mu}_{\boldsymbol{y}}$ could be estimated via its sample average, provided that enough observations are available (curse of dimensionality). As far as the computation of $\mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}^{(i)}}} [\boldsymbol{y}(\boldsymbol{X})]$ goes, the parametric distribution induced by the set of weights $p_{\boldsymbol{\theta}^{(i)}}$ is known, which pave the way to the use of the family of cumbersome, yet effective, Monte Carlo methods (Importance Sampling, Metropolis-Hastings, Gibbs Sampling, Markov chains,...). Neural networks follow another strategy: a \mathcal{NN} is conceived to estimate $\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{h}_{\boldsymbol{\theta}}(\boldsymbol{x})$ and its statistics, depending on the weights themselves.

In unsupervised learning, in order to maximize the likelihood of the statistical model, the common approach is to consider a *latent manifold* $(\mathcal{Z}, \mathcal{E}_Z, P_Z)$ with a arbitrary yet unknown probability distribution, that represents an encoded version of the data at stake [Mak19]. The joint probability distribution p(X, Z) is unknown but it can be approximated by choosing an arbitrary distribution p(Z) (often Gaussian or Uniform, for the sake of simplicity) and identify a *joint model distribution* that reads

$$p_{\theta}(\boldsymbol{X}, \boldsymbol{Z}) = p_{\theta}\left(\boldsymbol{X} | \boldsymbol{Z}\right) \cdot p(\boldsymbol{Z})$$
(81)

The aggregate prior distribution:

$$p_{\boldsymbol{\theta}}(\boldsymbol{X}) = \int_{\mathcal{Z}} p_{\boldsymbol{\theta}}(\boldsymbol{X}|\boldsymbol{z}) \cdot \mu(d\boldsymbol{z})$$
(82)

corresponds to the likelihood in Equation (8), that is maximized to find the best $\boldsymbol{\theta} \in \Theta$ provided an arbitrary generative distribution $p(\boldsymbol{Z})$. However, one can also identify a *joint data distribution* as follows:

$$q_{\phi}(\boldsymbol{X}, \boldsymbol{Z}) = q_{\phi}(\boldsymbol{Z} | \boldsymbol{X}) \cdot p(\boldsymbol{X})$$
(83)

The aggregate posterior distribution is identified by:

$$q_{\phi}(\boldsymbol{Z}) = \int_{\mathcal{X}} q_{\phi}(\boldsymbol{Z}|\boldsymbol{x}) \cdot p(\boldsymbol{x})\mu(d\boldsymbol{x})$$
(84)

The maximum likelihood matching consists into discover p(X) by matching it with $p_{\theta}(X)$. Learning the real data distribution with variational inference means to match $p_{\theta}(X, Z)$ and $q_{\phi}(X, Z)$.

Finally, one can define the *joint reconstruction distribution* as:

$$r_{(\phi;\theta)}(\boldsymbol{X}, \boldsymbol{Z}) = p(\boldsymbol{X} | \boldsymbol{Z}) \cdot q(\boldsymbol{Z}) \neq q_{\phi}(\boldsymbol{X}, \boldsymbol{Z}) \neq p_{\theta}(\boldsymbol{X}, \boldsymbol{Z})$$
(85)

and its aggregate reconstruction distribution [Mak19] as:

$$r_{(\phi;\theta)}(\boldsymbol{X}) = \int_{\mathcal{X}} r(\boldsymbol{X}, \boldsymbol{z}) \cdot \mu(d\boldsymbol{z})$$
(86)

There are different ways to maximize the log-likelihood. The most intuitive one is the maximize its *variational lower bound* or *Evidence Lower BOund* (ELBO), as proposed by Kingma and Welling [KW22]. The authors stated the following inequality¹⁰:

$$\mathbb{E}_{\boldsymbol{x} \sim p} \left[\ln p_{\boldsymbol{\theta}}(\boldsymbol{X}) \right] \geq \\ \geq \mathbb{E}_{\boldsymbol{x} \sim p} \left[\mathbb{E}_{\boldsymbol{z} \sim q_{\phi}(\boldsymbol{Z} \mid \boldsymbol{x})} \left[\ln p_{\boldsymbol{\theta}}(\boldsymbol{Z} \mid \boldsymbol{X}) \right] \right] - \mathbb{E}_{\boldsymbol{x} \sim p} \left[\mathbb{D}_{KL} \left(q_{\phi}(\boldsymbol{Z} \mid \boldsymbol{X}) \| p_{\boldsymbol{\theta}}(\boldsymbol{Z}) \right) \right] = \text{ELBO}$$
(87)

According to Equation (87), maximizing the ELBO maximizes the log-likelihood, towards $p_{\theta}(\mathbf{X})$ matching the real data distribution $p(\mathbf{X})$. In order to maximize the ELBO, one needs to:

- Make the posterior $q_{\phi}(\boldsymbol{Z}|\boldsymbol{X})$ match the arbitrary probability distribution on the latent manifold $p_{\theta}(\boldsymbol{Z})$. In this way, the negative term in ELBO represented by the Kullback-Leibler distance between the two above mentioned distributions $\mathbb{E}_{\boldsymbol{x}\sim p}\left[\mathbb{E}_{\boldsymbol{z}\sim q_{\phi}(\boldsymbol{Z}|\boldsymbol{x})}\left[-\ln p_{\theta}(\boldsymbol{Z}|\boldsymbol{X})\right]\right]$ goes to zero;
- Maximize the log-likelihood of the conditional distribution $p_{\theta}(X|Z)$

For both tasks, the learning algorithm has to perform the maximization "in average" over the available dataset. However, the ELBO maximization can be reformulated in other ways [Mak19], such as :

$$ELBO = \mathbb{E}_{\boldsymbol{x} \sim p} \left[\mathbb{E}_{\boldsymbol{x} \sim q_{\phi}(\boldsymbol{Z}|\boldsymbol{x})} \left[\ln p_{\boldsymbol{\theta}}(\boldsymbol{Z}|\boldsymbol{X}) \right] \right] - \mathbb{D}_{KL} \left(p_{\boldsymbol{\theta}}(\boldsymbol{Z}) \| q_{\phi}(\boldsymbol{Z}) \right) - \mathbb{I}(\boldsymbol{Z}, \boldsymbol{X}) = \\ = \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}(\boldsymbol{Z})} \left[\mathbb{D}_{KL} \left(q_{\phi}(\boldsymbol{X}|\boldsymbol{Z}) \| p_{\boldsymbol{\theta}}(\boldsymbol{X}|\boldsymbol{Z}) \right) \right] - \mathbb{D}_{KL} \left(q_{\phi}(\boldsymbol{Z}) \| p_{\boldsymbol{\theta}}(\boldsymbol{Z}) \right) - \mathbb{H}(p(\boldsymbol{X})) = \\ = -\mathbb{D}_{KL} \left(q_{\phi}(\boldsymbol{X}, \boldsymbol{Z}) \| r_{\phi; \boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{Z}) \right) - \mathbb{D}_{KL} \left(q_{\phi}(\boldsymbol{Z}) \| p_{\boldsymbol{\theta}}(\boldsymbol{Z}) \right) - \mathbb{H}(p(\boldsymbol{X}))$$

$$\tag{88}$$

 $^{10}\mathrm{With}$ some mathematical intricacy, one can prove that

$$\begin{split} \mathbb{E}_{\boldsymbol{x} \sim p} \left[\ln p_{\boldsymbol{\theta}}(X) \right] &= \mathbb{E}_{\boldsymbol{x} \sim p} \left[\mathbb{D}_{KL} \left(q_{\phi}(Z|X) \| p_{\boldsymbol{\theta}}(Z|X) \right) \right] - \mathbb{E}_{\boldsymbol{x} \sim p} \left[\mathbb{E}_{z \sim q_{\phi}(Z|x)} \left[-\ln p_{\boldsymbol{\theta}}(Z|X) \right] \right] - \\ &- \mathbb{E}_{\boldsymbol{x} \sim p} \left[\mathbb{D}_{KL} \left(q_{\phi}(Z|X) \| p_{\boldsymbol{\theta}}(Z) \right) \right] \end{split}$$

Equation (88) implies the notion of Shannon's entropy $\mathbb{H}(p(\mathbf{X}))$ of the real data distribution and mutual information $\mathbb{I}(\mathbf{Z}, \mathbf{X})$. Those two concepts will be introduced in the following subsections.

2 Theoretical aspects of Multi-Layer Perceptrons

2.1 Why the \mathcal{MLP} ?

Historically, the \mathcal{MLP} has been largely studied (see for instance [HDD; Cyb89; Hec92; Les+93; Pin99]) because of the following fundamental result: the universal approximation theorem.

Theorem 24. Universal approximation theorem for a 1-hidden-layered perceptron [Cam19]

Given a function $g \in \mathcal{C}(\mathbb{R})$ and a class of function

$$\mathcal{H}_g := span\left\{g(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b), \boldsymbol{w} \in \mathbb{R}^{d_X}, b \in \mathbb{R}\right\}$$

, then \mathcal{H}_g is dense for the uniform convergence on compact space, iff g is not polynomial.

This theorem implies that a \mathcal{MLP} $h_{\theta} \in \mathcal{H}_g$, with one hidden layer, can approximate any function $f : \mathbb{R}^{d_X} \mapsto \mathbb{R}$ if g is not polynomial. In this case, Theorem 24 states that \mathcal{H}_g is dense in $\mathcal{C}(\mathbb{R}^{d_X})$ [Cam19], i.e. that

Theorem 25. $\forall \varepsilon > 0, \exists h_{\theta} \in \mathcal{H}_g \text{ such that } \forall \mathcal{X}_{\Box} \subset \mathbb{R}^{d_X} \text{ compact set, } \forall \boldsymbol{x} \in \mathbb{R}^{d_X} |f(\boldsymbol{x}) - h_{\theta}(\boldsymbol{x})| \leq \varepsilon$

For the full proof of this fundamental theorem, we refer to [Cam19]. However, the proof's outline unveils some interesting aspects related to the \mathcal{MLP} architecture. In particular, the proof relies on the following fundamental lemma:

Lemma 26. Approximation with Fourier basis [Cam19]

Any function $f \in \mathcal{C}(\mathbb{R}^{d_X})$ can be approximated on any compact set of \mathbb{R}^{d_X} , with an arbitrary precision ε , with a Fourier series. In other words:

The proof is provided by Equation (210). The Fourier theory assures that Equation (89) is verified whenever $f \in C^{\infty}(\mathbb{R}^{\infty})$. But this is a rather strong restriction for practical purposes. As outlined in the following, based on Lemma 26,

the Theorem 25 adopts a non-polynomial approximation of the harmonic functions $\cos(\langle \boldsymbol{w}_n, \boldsymbol{x} \rangle)$ and $\sin(\langle \boldsymbol{w}_n, \boldsymbol{x} \rangle)$, in order to prove the universal approximation property. In particular, the harmonic functions are approximated - in analogy with Equations (13) and (14) - by a series of K ridge functions

$$g\left(\sum_{k=1}^{K} w_k^{(o)} \left< \boldsymbol{w}_n, \boldsymbol{x} \right> + b^{(o)}\right)$$

which concludes the proof of Theorem 24.

In particular, Mallat [Cam19] shows that the bounded compact support \mathcal{X}_{\Box} can be embedded in a hyperrectangle $[-T,T]^{d_X}$, on which a hyperrectangular regular grid of points is defined. This grids serves as support for the harmonic orthonormal basis $\mathcal{B}_{d_X} := \bigotimes_{m=1}^{d_X} \mathcal{B}_m$ of $L^2\left([-T,T]^{d_X}\right)$, obtained by tensorization of the basis $\mathcal{B}_m := \left\{e^{\frac{i\pi n_m t}{T}}\right\}_{n_m \in \mathbb{Z}}$ of $L^2\left([-T,T]\right)^{11}$. Thus, f can be decomposed on its basis (see Equation (210)):

$$f(\boldsymbol{x}) = \frac{1}{(2T)^{d_X}} \sum_{(n_1,\dots,n_{d_X})\in\mathbb{Z}^{d_X}} \hat{f}(\boldsymbol{w}_{(n_1,\dots,n_{d_X})}) \prod_{m=1}^{d_X} e^{\frac{i\pi n_m x_m}{T}} = \frac{1}{(2T)^{d_X}} \sum_{n\in\mathbb{Z}} \hat{f}(\boldsymbol{w}_n) \cdot e^{i\langle \boldsymbol{w}_n, \boldsymbol{x} \rangle}$$

$$(90)$$

For practical purposes, a truncation of the infinite series in Equation (90) is required. This corresponds to filter the high frequencies by forcing $\{\boldsymbol{w}_n\}_{n>N_0} = \mathbf{0}$. The truncation error e, in $L^2\left(\left[-T,T\right]^{d_X}\right)$ norm is defined as¹²:

$$\langle \boldsymbol{f}, \boldsymbol{g} \rangle_{L^{2}([-T,T])} = \int_{-T}^{T} \langle \boldsymbol{f}^{*}(t), \boldsymbol{g}(t) \rangle dt$$

For any two values of the basis \mathcal{B}_m , it holds that:

$$\left\langle e^{\frac{i\pi n_j t}{T}}, e^{\frac{i\pi n_k t}{T}} \right\rangle_{L^2([-T,T])} = \int_{-T}^T e^{\frac{-i\pi n_j t}{T}} \cdot e^{\frac{i\pi n_k t}{T}} dt = \delta_{jk}$$

Which proves their orthonormality.

¹²According to [Cam18], the proof of the Lemma is based on a uniform convergence norm, but $\|\cdot\|_{L^2}$ and $|\cdot|$ are not equivalent in infinite dimension. However, if $f \in C^{\infty}$, the uniform convergence is proved, by Theorem 40. Unfortunately, this is not true if $f \in C^0$, but the Stone-Weierstrass theorem allows to approximate (via uniform convergence) the function with a series of polynomial functions p(x) that can be differentiated infinite times, whose Fourier coefficients $\hat{p}(k)$ decay fast and therefore the approximation is still valid.

¹¹The standard scalar product in $L^2([-T,T])$ reads

Gatti 289

$$e^{2} = \frac{1}{(2T)^{d_{X}}} \int_{[-T,T]^{d_{X}}} \left| f\left(\boldsymbol{x}\right) - \frac{1}{(2T)^{d_{X}}} \sum_{n=1}^{N_{0}} \hat{f}(\boldsymbol{w}_{n}) \cdot e^{i\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle} \right|^{2} dt \leq \\ \leq \frac{1}{(2T)^{d_{X}}} \sum_{\|\boldsymbol{w}_{n}\|_{2} \geq C(\varepsilon)} |\hat{f}(\boldsymbol{w}_{n})|^{2}$$

$$(91)$$

The truncation threshold $C(\varepsilon)$ depends on the accuracy of the approximation expressed by $e \leq \varepsilon$ and it implies to limit the norm of the weights:

$$\|\boldsymbol{w}_n\|_2 \le C(\varepsilon) \qquad \forall n > N_0$$

In practice, this truncation is imposed by adding a penalty to the empirical loss in Equation (92), corresponding to the L^p -norm of the weights (often L^2 or L^1):

$$L_{\mathcal{D}_{XY}}\left(\boldsymbol{h}_{\theta}\right) = \frac{1}{N} \sum_{\left(\boldsymbol{x}_{k}, \boldsymbol{y}_{k}\right) \in \mathcal{D}_{XY}} \ell\left(\boldsymbol{h}_{\theta}\left(\boldsymbol{x}_{k}\right), \boldsymbol{y}_{k}\right) + \lambda \cdot \|\boldsymbol{w}\|_{p}$$
(92)

Another interesting aspect of Theorem 24 and inherently of Lemma 26 is related to the behaviour of the Fourier coefficients $\hat{f}(\boldsymbol{w}_n)$. The latter are the interpolation weights for reconstructing the signal with the Inverse Fourier Transform (see Equation (210). Therefore, their values are intimately related to the function regularity. Theorem 40 implies that, provided an arbitrary precision ε , the higher the regularity of f, the faster the decay of its Fourier coefficients with $\|\boldsymbol{w}\|_2$. This means that a lower number of wavelengths is demanded to approximate f. This result allows to estimate the number of weights needed by a 1-hidden layer \mathcal{MLP} to approximate any regular function $f \in C^p$, based on the Theorem 24. In particular, the number of weights $N \propto M^{d_X}$ increases with the dimension of the input data d_X . Another reason why the number of weights in the hidden layer can easily become very very large is the irregularity of the function f to approximate: which implies to employ a larger number of high-frequency Fourier coefficient to reach the desired precision $e \leq \varepsilon$ [Cam19]. As a matter of fact, irregular functions have a broader-band Fourier spectrum (see for instance the Dirac delta $\delta(x-x_0)$). In this case, M can become very large because of the *Heisenberg uncertainty principle* defined by the Theorem 43 that explicitly states the underlying trade-off between time and frequency localization [Mal09].

Controlling the error via a L^2 norm as done in Equation (91) does not necessarily imply the uniform convergence stated in Theorem 25, unless $f \in C^{\infty}(\mathbb{R}^{d_x})$. However, even for $f \in C^0(\mathbb{R}^{d_x})$, the Stone-Weierstrass theorem that states the possibility of approximating f with a series of functions $p_i(\boldsymbol{x}) = \int_{\mathcal{X}_{\square}} N_i(\boldsymbol{x} - \boldsymbol{y}) \cdot f(\boldsymbol{y}) d\boldsymbol{y}$, with N_i polynomials of order q, provided that $\int_{\mathcal{X}_{\square}} N_i(\boldsymbol{x}) d\boldsymbol{x} = 1$.

In this case,

$$\forall \varepsilon > 0, \forall \boldsymbol{x} \in \mathcal{X}_{\Box}, \exists R(\varepsilon) \text{ such that }, \forall \boldsymbol{x} \in [-R, R]^{d_{X}} \quad |f(\boldsymbol{x}) - \sum_{i=1}^{N} \alpha_{i} \cdot p_{i}(\boldsymbol{x})| \leq \varepsilon$$

One can easily recognize here the basics of the Finite Element Method. Moreover, the Fourier coefficients $\hat{N}_i(k)$ decay as fast as $\|\boldsymbol{w}\|^{-q}$, but, compared with Equation (89), the series of polynomes N_i does not represent the Fourier transform of f [Cam19]. Plus, Theorem 25 demands non-polynomial functions in order to assure the uniform convergence.

Finally, the Theorem 24 adopts ridge functions such as those presented in Section 2.2 to assure the universal approximation power of a 1-hidden-layer \mathcal{MLP} . First, based on the Hölder's inequality, each argument in the approximating the Fourier series can be bounded as follows on \mathcal{X}_{\Box} :

$$\langle \boldsymbol{w}_n, \boldsymbol{x} \rangle = \sum_{i=1}^{d_X} w_{n,i} x_i \le \left(\sum_{i=1}^{d_X} w_{n,i}^2\right)^{\frac{1}{2}} \cdot \left(\sum_{i=1}^{d_X} x_i^2\right)^{\frac{1}{2}} = \|\boldsymbol{w}_n\|_2 \cdot \|\boldsymbol{x}\|_2 \le C(\varepsilon) \cdot T \cdot d_X$$
(93)

Recalling the expression of the truncation error $e \leq \varepsilon$ in Equation (91), the approximation based on ridge functions employed to approximate the harmonic functions can be expressed as¹³:

$$|\hat{f}(\boldsymbol{w}_{n})| \cdot |\cos\left(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle\right) - \sum_{k=1}^{K} w_{k}^{(o)} \cdot g(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle + b_{n})| \leq \\ \leq \frac{M \cdot |\cos\left(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle + b_{n}\right) - \sum_{k=1}^{K} w_{k}^{(o)} \cdot g(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle + b_{n})|}{1 + \|\boldsymbol{w}_{n}\|^{p+1+\varepsilon}}$$

$$\text{with } \langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle \leq C(\varepsilon) \cdot T \cdot d_{X}$$

$$(94)$$

The approximation in Equation (94) represents the outcome of a 1-hiddenlayer \mathcal{MLP} with linear output activation function $g^{(o)}(a) = a$ and $b^{(o)} = 0$. For instance, if one selects a combinations of translated (via the biases b_n) ReLU functions for the hidden layer activation function g, the approximation in Equation (94) represents a piece-wise linear interpolation of f (see Remark 39 [Cam19]). In order to keep the error

$$\begin{split} \boldsymbol{e}_{n}^{'} &= |\cos\left(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle\right) - \sum_{k=1}^{K} w_{k}^{(o)} \cdot g(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle + b_{n})| + \\ &+ |\sin\left(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle\right) - \sum_{k=1}^{K} w_{k}^{(o)} \cdot g(\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle + b_{n})| \end{split}$$

 $^{^{13}}$ For the sinus function, the same considerations apply.

small enough, i.e., $e'_n \leq \frac{\varepsilon}{N}$, with N being the number of weights for which $\|\boldsymbol{w}_n\|_2$, one needs at least $K = \frac{2T \cdot N \cdot C(\varepsilon) \cdot d_X}{\varepsilon}$ per each Fourier coefficient. The total amount of ridge functions g to approximate $f(\boldsymbol{x})$ is then $K \cdot N = \frac{2T \cdot N^2 \cdot C(\varepsilon) \cdot d_X}{\varepsilon}$. In particular, if we fix $C(\varepsilon)$, $N \approx C(\varepsilon)^{d_X}$ in order to keep $e = \mathcal{O}(\frac{C(\varepsilon)}{2T})$. Therefore, the total number of ridge functions required by the \mathcal{MLP} is

$$N_K = K \cdot N \approx \frac{2T \cdot C(\varepsilon)^{2d_X + 1} \cdot d_X}{\varepsilon}$$

(*N* sine et cosine functions approximated by *K* ridge functions each). Whenever the Fourier coefficients decrease slowly with the frequency, because of the irregularity of the function, $C(\varepsilon)$ increases accordingly and therefore one requires a larger number of hidden neurons *N* to keep the a lower approximation error. The increasing dimension d_X of the dataset imply an even larger number of parameters.

Considering that $||f||_{L^2(\mathbb{R})} < ||f||_{W^{k,2}(\mathbb{R})} \le ||f||_{L^2(\mathbb{R})} + ||f^{(k)}||_{L^2(\mathbb{R})}$, and according to Equation (91), the L^2 approximation error is bounded as follows [Cam19]:

$$e^{2} \leq \frac{1}{(2T)^{d_{X}}} \sum_{\|\boldsymbol{w}_{n}\|_{2} \geq C(\varepsilon)} |\hat{f}(\boldsymbol{w}_{n})|^{2} < \frac{1}{(2T)^{d_{X}}} \sum_{\|\boldsymbol{w}_{n}\|_{2} \geq C(\varepsilon)} \frac{1 + \|\boldsymbol{w}_{n}\|_{2}^{2k}}{C(\varepsilon)^{2k}} |\hat{f}(\boldsymbol{w}_{n})|^{2}$$

$$(05)$$

If $f \in W^{k,2}$, $\exists 0 < B < +\infty$ such that $||f||_{W^{k,2}(\mathbb{R})} \leq ||f||_{L^2(\mathbb{R})} + ||f^{(k)}||_{L^2(\mathbb{R})} < B$, which implies that the approximation error is bounded, i.e. $e < \frac{B}{C(\varepsilon)^{2k}} = \varepsilon$, which implies that [Cam19]:

$$C(\varepsilon) = \left(\frac{B}{\varepsilon}\right)^{\frac{1}{2k}} \qquad N_K \approx \frac{2T \cdot B^{\frac{2d_X+1}{2k}} \cdot d_X}{\varepsilon^{\frac{2d_X+1}{2k}}} \tag{96}$$

Equation (96) implies that, for a fixed approximation accuracy ε , the number of hidden neurons in 1-hidden-layer \mathcal{MLP} increases as $\varepsilon^{\frac{2d_X+1}{2k}}$, i.e., the number of neuron remains constant whether the regularity of the function k increases with the dimension of the data space d_X . Therefore, the 1-hidden-layer \mathcal{MLP} can theoretically approximate poorly regular functions in large dimension, provided that the number of hidden neurons is large enough (very large) [Cam19]. The number of hidden neurons estimated by Mallat in [Cam19] can be refined though. In particular, according to [Mai99], the following theorem holds:

Theorem 27. Lower bounds for approximation by \mathcal{MLP} neural networks [Mai99] Given a function $f : [-T,T]^{d_X} \to \mathbb{R}$, T > 0, with $f \in W^{k,2}$, then it exists a 1-hidden-layer \mathcal{MLP} h_{θ} , with sigmoid activation function and N_K hidden neurons, such that the L^2 -error $e = \|f - h_{\theta}\|_{L^2([-T,T]^{d_X})}$ is lower than a tolerance ε if:

$$N_K \approx \varepsilon^{\frac{1-d_X}{k}} \tag{97}$$

This theorem implies that to reduce the error by an order or magnitude, the number of neurons must be multiplied by $10^{\frac{d_X-1}{k}}$. For instance, an $W^{2,2}$ function defined over \mathbb{R}^3 demands to multiply by a factor 10 to decrease by 10 the approximation error.

2.2 On the sparsity of the \mathcal{MLP}

To go further, [Bar93] proposed a strategy to weaken the assumptions on the function regularity, yet breaking the curse of dimensionality. In particular, [Bar93] proposes to focus on functions with L^1 first order derivative. As a matter of fact, if $f \in W^{1,1}(\mathbb{R}^{d_X})$, then:

$$f \in W^{1,1}\left(\mathbb{R}^{d_X}\right) \iff \|f\left(\boldsymbol{x}\right)\|_{L^1\left(\mathbb{R}^{d_X}\right)} < +\infty, \quad \|\boldsymbol{\nabla}_x f\left(\boldsymbol{x}\right)\|_{L^1\left(\mathbb{R}^{d_X}\right)} < +\infty$$
(98)

According to Remark 41 and following the strategy based on the Fourier analysis, proposed by S.Mallat [Cam19], Equation (98) implies the norm $||f(\boldsymbol{x})||_{W^{1,1}(\mathbb{R}^{d_X})} = \langle +\infty \text{ is equivalent to } ||f(\boldsymbol{x})||_{L^1(\mathbb{R}^{d_X})} + ||\nabla_x f(\boldsymbol{x})||_{L^1(\mathbb{R}^{d_X})}$. Since $\nabla_x f(\boldsymbol{x}) \in L^1(\mathbb{R}^{d_X})$, its Fourier transform exists and it reads $i\boldsymbol{w}\hat{f}(\boldsymbol{w})$. Recalling the conditions of Equation (89), and bounding the compact support \mathcal{X}_{\Box} where the data live, with a regular grid discretizing the volume $[-T,T]^{d_X} \supseteq \mathcal{X}_{\Box}$, we can locally approximate $\nabla_x f(\boldsymbol{x})$ with its Fourier series on $[-T,T]^{d_X}$:

$$\boldsymbol{\nabla}_{x} f\left(\boldsymbol{x}\right) = \sum_{\boldsymbol{w}_{n} \in \mathbb{Z}^{d_{X}}} i\boldsymbol{w}_{n} \hat{f}\left(\boldsymbol{w}_{n}\right) e^{-i\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle}, \quad \forall \boldsymbol{x} \in [-T, T]^{d_{X}}$$
(99)

On the compact support $\left[-T,T\right]^{d_{X}}$, the L^{1} -norm of $\nabla_{x}f(x)$ is bounded as follows:

$$\begin{aligned} \|\boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}\right)\|_{L^{1}\left(\mathbb{R}^{d_{X}}\right)} &= \int_{\left[-T,T\right]^{d_{X}}} \|\boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}\right)\|_{1}d\boldsymbol{x} = \\ &\int_{\left[-T,T\right]^{d_{X}}} \|\sum_{\boldsymbol{w}_{n}\in\mathbb{Z}^{d_{X}}} i\boldsymbol{w}_{n}\hat{f}\left(\boldsymbol{w}_{n}\right)e^{-i\langle\boldsymbol{w}_{n},\boldsymbol{x}\rangle}\|_{1}d\boldsymbol{x} \leq \\ &\leq \int_{\left[-T,T\right]^{d_{X}}} \sum_{\boldsymbol{w}_{n}\in\mathbb{Z}^{d_{X}}} \|\boldsymbol{w}_{n}\|_{1} \cdot |\hat{f}\left(\boldsymbol{w}_{n}\right)|d\boldsymbol{x} < +\infty \end{aligned}$$
(100)

In the revisited version of the work of Barron [Bar93], by S. Mallat [Cam19], the regularity condition in Equation (100) is adopted show the accuracy of approximating f with the function in Equation (18).

Theorem 28. Approximation bounds of a neural network [Bar93; Cam19] Given a function $f : \mathbb{R}^n \to \mathbb{R}$, with $f \in W^{1,1}(\mathbb{R}^{d_X})$ and provided a constant $C_f = \sum_{\boldsymbol{w}_n \in \mathbb{Z}^{d_X}} |\hat{f}(\boldsymbol{w}_n)| < +\infty$, then $\exists \{\boldsymbol{w}_n\}_{n \leq N_K}$, with $K = \frac{C_f^2}{\varepsilon^2}$ such that $\|f(\boldsymbol{x}) - \sum_{n=1}^{N_K} \hat{f}(\boldsymbol{w}_n) e^{-i\langle \boldsymbol{w}_n, \boldsymbol{x} \rangle}\|_{L^2(\mathbb{R}^{d_X})} \leq \frac{C_f^2}{N_K}$

Proof. In order to prove the revisited version of [Bar93] work, S.Mallat in [Cam19] proposes to reindexing the Fourier coefficients $\hat{f}(\boldsymbol{w}_n)$ in descending order, i.e., according to an indexing $\mathcal{I} : \mathbb{Z} \to \mathbb{N}$ such that that

$$|\hat{f}\left(\boldsymbol{w}_{n}\right)| > |\hat{f}\left(\boldsymbol{w}_{m}\right)| \Longrightarrow \quad \mathcal{I}(n) < \mathcal{I}(m), \forall m, n \in \mathbb{Z}$$

 C_f can therefore be rewritten as

$$C_{f} = \sum_{n \in \mathbb{N}} |\hat{f}(\boldsymbol{w}_{n})| = \sum_{n=1}^{N_{K}} |\hat{f}(\boldsymbol{w}_{n})| + \sum_{n > N_{K}} |\hat{f}(\boldsymbol{w}_{n})|$$

Since the Fourier coefficients have been reordered in descending order:

$$C_{f} \geq \sum_{n=1}^{N_{K}} |\hat{f}(\boldsymbol{w}_{n})| \geq N_{K} |\hat{f}(\boldsymbol{w}_{N_{K}})|$$

which implies that the N_K^{th} smaller Fourier coefficient is lower than $\frac{C_f}{N_K}$. To conclude the proof, the approximation error can be rewritten as:

$$\|f(\boldsymbol{x}) - \sum_{n=1}^{N_{K}} \hat{f}(\boldsymbol{w}_{n}) e^{-i\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle} \|_{L^{2}(\mathbb{R}^{d_{X}})}^{2} = \|\sum_{n > N_{K}} \hat{f}(\boldsymbol{w}_{n}) e^{-i\langle \boldsymbol{w}_{n}, \boldsymbol{x} \rangle} \|_{L^{2}(\mathbb{R}^{d_{X}})}^{2} \leq \\ \leq \sum_{n > N_{K}} \frac{C_{f}^{2}}{k^{2}} \leq C_{f}^{2} \int_{N_{K}}^{+\infty} \frac{1}{x^{2}} dx = \frac{C_{f}^{2}}{N_{K}}$$

Remark 29. The original version of Theorem 28 by [Bar93] proves that considering $W^{1,1}(\mathbb{R}^{d_X})$ functions, the L^2 -error of approximation is bounded by above by the constant $\frac{\int_{\mathbb{R}^{d_X}} \|\boldsymbol{w}\|_1 \cdot |\hat{f}(\boldsymbol{w})| d\boldsymbol{w}}{\sqrt{N_K}}$.

The importance of Theorem 28 resides in the fact that, penalizing the learning algorithm with a ℓ^1 norm on the weights θ allows to promote sparsity and achieve parsimony. In other words, the 1-hidden-layer \mathcal{MLP} will learn how to approximate the labelling function with the least amount of hidden features (the famous above mentioned dictionary) and with an approximation error that is not dependent on the dimension of the input space (or alternatively, for a fixed accuracy, the design of the \mathcal{NN} will require an amount of hidden neurons that is independent of the dimension of the data space.

3 Advanced topics in \mathcal{CNN}

In the following, a collection of labeled images $\mathcal{D}_{XY} = \{\mathbb{X}_k, y_k\}_{k=1}^N$ of $W \times H$ pixels each is considered. Any image \mathbb{X}_k of the database is represented by a set

of pixels, each one associated to a color expressed by a 3-dimensional vector on the Red Green Blue (RGB) scale $\boldsymbol{\psi} : \mathbb{R}^3 \to \mathbb{R}^3$ (see Section 2.5). Therefore, the image $\mathbb{X}_k \in \mathbb{R}^{W \times H \times 3}$ can be alternatively indicated by the color vector $\boldsymbol{\psi}$ as:

$$\mathbb{X} (x_1, x_2, :) = \psi (x_1, x_2, 0) = = \psi_R (x_1, x_2, 0, :) e_1 + \psi_G (x_1, x_2, 0, :) e_2 + \psi_B (x_1, x_2, 0, :) e_3, (x_1, x_2) \in [0, 1]^2$$
(101)

For the sake of simplicity, the image is spanned over a $[0,1]^2$ domain, by a regular grid of $W \times H$ pixels with a 2D index $\boldsymbol{x} = x_1 \boldsymbol{e}_1 + x_2 \boldsymbol{e}_2, (x_1, x_2) \in [0,1]^2$. In the following, in order to simplify the mathematical formulation, it is assumed that the image $\boldsymbol{\psi} : \mathbb{R}^3 \to \mathbb{R}^3$ with $x_3 = 0$ for all 2D image and $\boldsymbol{\psi} : (x_1, x_2, 0) \mapsto \boldsymbol{\psi} (x_1, x_2, 0)$. The 0 is omitted to avoid cumbersome notation. The label y_k belongs to an alphabet A for a classification purposes, and $\boldsymbol{y}_k \in \mathbb{R}^{d_Y}$ in regression problems. The alphabet is usually coded as a *one-hot vector*, i.e. a binary vector of the size of the alphabet.

3.1 A group-theory vision of classification

If it exists a labeling function $f : \mathbb{X} \mapsto y$, we can study its regularity and invariance with respect to any transformation applied to the image, that preserves all the labeling function level sets

$$L_{c}(f) = \begin{bmatrix} \mathbb{X} \in \mathcal{X} \\ f(\mathbb{X}) = c \end{bmatrix}, \quad \forall c \in \operatorname{Im}(f)$$

. This invariance is expressed by a symmetry group [Mal16; Cam20]¹⁴:

$$G := [g | \forall \mathbb{X} \in L(f), \quad f(g.\mathbb{X}) = f(\mathbb{X})]$$
(102)

Some standard transformation:

• Translation:

$$g\mathbb{X} = g\boldsymbol{\psi}\left(\boldsymbol{x}\right) = \boldsymbol{\psi}\left(\boldsymbol{x} - \boldsymbol{c}_{g}\right)$$

with $\boldsymbol{c}_{g} \in \mathbb{R}^{2}$ (103)

¹⁴One can prove that G is a groups since, $\forall g_1, g_2, g_3 \in G$ and $\forall \mathbb{X} \in \mathcal{X}$:

- $f((g_1g_2)g_3\mathbb{X}) = f(g_3\mathbb{X}) = f(\mathbb{X}) = f(g_3\mathbb{X}) = f(g_2(g_3\mathbb{X})) = f(g_1(g_2g_3)\mathbb{X})$ (associative)
- $\exists Id$ such that $f(Id.\mathbb{X}) = f(\mathbb{X})$ (neutral element)
- $\exists g^{-1}$ such that $f\left(g^{-1}g.\mathbb{X}\right) = f\left(g^{-1}\mathbb{X}\right) = f\left(\mathbb{X}\right)$ (inverse element)

The group si Abelian if $g_1g_2 = g_2g_1$ (optional).

• Rotation:

$$g\mathbb{X} = g\boldsymbol{\psi}(\boldsymbol{x}) = \boldsymbol{\psi}\left(\boldsymbol{R}_{g}\boldsymbol{x}\right)$$

with $g \in [0, 2\pi]$ and $\boldsymbol{R}_{g} \in SO(2)$ (104)

• Roto-translation:

$$g\mathbb{X} = g\boldsymbol{\psi}\left(\boldsymbol{x}\right) = \boldsymbol{\psi}\left(\boldsymbol{R}_{g}\left(\boldsymbol{x} - \boldsymbol{c}_{g}\right)\right)$$

with $g \in [0, 2\pi], \boldsymbol{c}_{g} \in \mathbb{R}^{2}$ and $\boldsymbol{R}_{g} \in SO(2)$ (105)

• Normalization:

$$g\mathbb{X} = g\boldsymbol{\psi}\left(\boldsymbol{x}\right) = s_g\boldsymbol{\psi}\left(\boldsymbol{x}\right)$$

with $s_g \in \mathbb{R}$ (106)

• More general, the group of diffeomorphisms acts according to the following law:

$$g\left(\mathbb{X}\left(\boldsymbol{x}\right)\right) = \boldsymbol{\psi}\left(g\left(\boldsymbol{x}\right)\right) \tag{107}$$

If we knew all the elements of G, we would know f entirely, since we could apply several deformation to the image preserving all the level sets $f(x) = c, \forall c$. S. Mallat, who contributed to develop this original theory, asserts that it is hard to know before hand all the possible strain fields which the labelling function is invariant to. We can guess some (the classification is insensitive to rototranslations, for instance) but definitely not all of them [Mal16; Cam20]. In other words, the dimension of G is too large. But we can infer it from a subgroup of plausible strain fields H. As a matter of fact, $\forall g \in H, \mathbb{X}$ and $g\mathbb{X}$ belong to the same equivalence class. For instance, the image \mathbb{X} engenders the following equivalence class:

$$[\mathbb{X}] := \left\{ \mathbb{X}^{'} \in L\left(f\right) \mid \quad \mathbb{X}^{'} = h\mathbb{X}, h \in H \right\}$$

Note that, since $f(\mathbb{X}') = f(h\mathbb{X}) = f(\mathbb{X})$. The ensemble quotient associated to $[\mathbb{X}]$ reads:

$$L(f) | H := ([\mathbb{X}] \in \mathcal{P}(\mathcal{X}), \mathbb{X} \in \mathcal{X})$$
(108)

with $\mathcal{P}(\mathcal{X})$ being the power set of \mathcal{X} . For all $\mathbb{X}_0 \in L(f) | H, f(g\mathbb{X}_0) \in [\mathbb{X}_0]$: in other words, for each image \mathbb{X}_0 , classified, for instance, as the image of "cat", despite any roto-translation or other known transformations of the group symmetries $g \in H$, the class of the transformed image $g\mathbb{X}_0$ remains classified as "cat". The advantage of this strategy, based on the group theory, formulated by S. Mallat is that it reduces the dimensionality: as a matter of fact, we can find the subgroup of transformations H, with $\dim(L(f)|H) = \dim(L(f)) - \dim(H)$, that are common to all the images belonging to the the equivalence class, i.e. to all the images classified by f in the same way.

3.1.1 Continuum Mechanics applied to image classification?

Several invertible transformation g_1, g_2, \ldots, g_n can be applied to \mathbb{X} , but the labelling function should not be affected. All groups of G can potentially generate subgroups $\{g^k, k \in \mathbb{Z}\}$. Any set of groups $A = \{g_1, \ldots, g_n\}$ generates a subgroup $\{A\}$, whose elements are defined by permutations of the product $g_1g_2\ldots g_n$. In particular, it is interesting to study the group orbit $O_{\mathbb{X}} = \{g\mathbb{X}\}_{g\in G}$ for which $f(O_{\mathbb{X}}) = f(\mathbb{X})$. We borrow from continuum mechanics (see [Hil79; TN04; KD21] the theoretical framework necessary to describe continuous transformations that can be applied to the image, provided that each infinitesimal transformation preserves the label $f(\mathbb{X})$. This approach provides a more flexible description of the ensemble of possible group actions $g \in H$ on the image, that can be a priori very very large and not limited to finite roto-translations. In particular, we can think of the image pixels as a set of points $\{X_n\}_{n=1}^{W \times H} \in B^{15}$, disposed on a regular grid on the \mathbb{R}^2 space, via the *placement* function $p: B \to (\mathcal{E}, \mathcal{R})$ (the 2D euclidean space, with reference system \mathcal{R} defined by (O, e_1, e_2) , which is smooth and it preserves the orientation (positive Jacobian, i.e., two points cannot occupy the same position). B is defined as a "body" [TN04], an abstract manifold of dimension 2, endowed with a Fréchet derivative [KD21]. The image can be placed in $(\mathcal{E}, \mathcal{R})$ in different configurations t, occupying a volume $\mathbb{R}^3 \supset \Omega_t = \operatorname{Im}(p_t) = p_t(B)$. The space of configurations is defined as the ensemble of placement functions $p_t \in C^{\infty}(\mathbf{B}, \mathcal{E})$ and it called *embedding* $\mathrm{Emb}^{\infty}(\mathbf{B}, \mathcal{E}) \subset C^{\infty}(\mathbf{B}, \mathcal{E})$, which is an infinite-dimensional differentiable manifold. In particular, the placement $p_0(\boldsymbol{p}_n)$ places any pixel of any image in the reference configuration p_0 , in a coordinate of the \mathbb{R}^3 space referred as to

$$\boldsymbol{x}_{0}=p_{0}\left(\boldsymbol{p}
ight),\qquadorall \boldsymbol{p}\in\mathrm{B}$$

and the same pixel in a configuration p_t ,

$$\boldsymbol{x} = p_t \left(\boldsymbol{p} \right), \qquad \forall \boldsymbol{p} \in \mathbf{B}$$

More generically, in analogy with mechanics, the color can be seen as a spatial pixel-wise vector field $\psi(x;t)$, where the "time" t stands for the nondimensional continuum index defining successive transformations $g_1, g_2, \ldots, g_t, \ldots$. As a matter of fact, the image can be distorted from an arbitrary reference configuration p_0 and classified in the same way all along the orbit $O_{\mathbb{X}} = \{g_t \mathbb{X}\}_{g_t \in G, t \in \mathbb{R}}$. Moreover, data augmentation techniques are directly based upon enlarging the dataset at stake by applying arbitrary rototranslations or other kinds of diffeomorphims to the original image dataset (see Figure 5). Now, we should introduce a transformation from two different configurations, in order to deform or roto-translate the image, i.e., a diffeomorphisms between two configurations, called *deformation* $\varphi : \Omega_0 \to \Omega_t$ and defined as:

$$oldsymbol{x} = oldsymbol{arphi}\left(oldsymbol{x}_{0}
ight) \qquad oldsymbol{arphi} = p_{t}\circ p_{0}^{-1}$$

 $^{^{15}}X$ are denoted as material coordinates



https://www.kaggle.com/code/kanncaa1/convolutional-neural-network-cnn-tutorial

Figure 5: Data augmentation on Modified National Institute of Standards and Technology database (MNIST) dataset [LeC98]

The image in the arbitrary reference configurations occupies a volume Ω_0 and, when distorted by a diffeomorphisms, it occupies a volume Ω_t , so that φ : $\Omega_0 \to \Omega_t$. One can notice that the phase flow associated to $\varphi(\boldsymbol{x})$ and denoted as $\varphi(\boldsymbol{x};t)$ has the following properties:

$$\boldsymbol{\varphi}\left(\boldsymbol{x}_{0},0\right) = Id(\boldsymbol{x}_{0}) = \boldsymbol{x}_{0}, \qquad \boldsymbol{\varphi}\left(\boldsymbol{\varphi}\left(\boldsymbol{x}_{0},s\right),t\right) = \boldsymbol{\varphi}\left(\boldsymbol{x}_{0},s+t\right)$$
(109)

If $\varphi \in C^{1}(\Omega_{0})$, its gradient exists and it defined as:

$$d\boldsymbol{x}(t) = \mathbb{F}(\boldsymbol{x}_0, t) \, d\boldsymbol{x}_0 + o\left(\|\boldsymbol{d}\boldsymbol{x}_0\|\right) \qquad \mathbb{F}(\boldsymbol{x}_0, t) = \sum_{n=1}^2 \frac{\partial \boldsymbol{\varphi}\left(\boldsymbol{x}_0, t\right)}{\partial x_n} \otimes \boldsymbol{e}_n \quad (110)$$

The linearization or variation of a spatial tensor field $\psi(\boldsymbol{x})$ is formally equivalent, in mechanics, to the Lie derivative of the spatial field itself. The Lie algebra is based on the well know *pull-back* operation by any $p \in \text{Emb}^{\infty}(B, \mathcal{E})$ [KD21]:

$$\Psi(\boldsymbol{X}) = p^* \boldsymbol{\psi} = J \mathbb{F}^{-1} \left(\boldsymbol{\psi} \circ p \right)$$
(111)

with $J = \det(\mathbb{F})$ and \mathbb{F}^{-1} the inverse gradient. The inverse of the pull-back is the push-forward:

$$\boldsymbol{\psi}\left(\boldsymbol{x}\right) = p_* \boldsymbol{\Psi} = \frac{1}{J} \mathbb{F} \boldsymbol{\Psi} \circ p^{-1}$$
(112)

The pull-back operation allows to refer to the body B itself, where each pixel is identified in an abstract sense. The Lie algebra defines the fixed tangent planes of the differentiable geodetic O_X , generated by the gradient $\mathbb{F}(\boldsymbol{X};t)$, that allow to pass from a transformation at time t to another one infinitesimally close at time t + dt. The *Lie derivative* is an infinitesimal version of the pull-back and it reads, for a contravariant vector $\boldsymbol{\psi}$, along the vector \boldsymbol{X} of material coordinates [KD21]:

$$L_{\boldsymbol{u}}\boldsymbol{\psi} = \frac{\partial}{\partial t}\Big|_{t=0}\boldsymbol{\varphi}\left(\boldsymbol{x};t\right)^{*}\boldsymbol{\psi}$$
(113)

In other words, the Lie derivative consists into compute the pull-back on ψ , compute its Gateaux derivative $\frac{d}{dt}\Big|_{t=0} \Psi(\mathbf{X} + t\mathbf{c})$ and finally push it forward to the spatial configuration. For the sake of clarity, let us define the Lie derivative for a vector field ψ , by firstly defining the following phase flow in \mathbb{R}^3 (Landau notation):

$$\varphi(\boldsymbol{x};-t) = \boldsymbol{x} - t\boldsymbol{u}(\boldsymbol{x}) + o(t)$$
(114)

and the associated Gateaux derivative along $y \in \mathbb{R}^3$:

$$D_{\boldsymbol{y}}\boldsymbol{\varphi}\left(\boldsymbol{x};-t\right) = \boldsymbol{y} - tD_{\boldsymbol{y}}\boldsymbol{u}\left(\boldsymbol{x}\right) + o(t)$$
(115)

From now on, let us assume that $u\left(x
ight)$ is Fréchet differentiable, with the bounded linear operator

$$\mathbb{D}_{\boldsymbol{x}}\boldsymbol{u} = \sum_{n=1}^{3} \frac{\partial \boldsymbol{u}\left(\boldsymbol{x}\right)}{\partial x_{n}} \otimes \boldsymbol{e}_{n}$$

that represents the gradient of the vector \boldsymbol{u} and for which

$$D_{oldsymbol{y}}oldsymbol{u}\left(oldsymbol{x}
ight)=\mathbb{D}_{oldsymbol{x}}oldsymbol{u}\left(oldsymbol{x}
ight).oldsymbol{y}$$

In continuum mechanics, this implies that the displacement field does include discontinuity or fractures. Plugging $\boldsymbol{y} = \boldsymbol{\varphi}(\boldsymbol{x};t)$ in Equation (115), the latter reads:

$$D_{\varphi(\boldsymbol{x};t)}\varphi(\boldsymbol{x};-t) = (\mathbf{I} - t\mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}(\boldsymbol{x}) + o(t))(\boldsymbol{x} + t\boldsymbol{u}(\boldsymbol{x}) + o(t)) =$$

= $\varphi(\boldsymbol{x};t) - tD_{\boldsymbol{x}}\boldsymbol{u}(\boldsymbol{x}) + o(t)$ (116)

Moreover, the Taylor expansion of the continuous map ψ reads:

$$\boldsymbol{\psi}\left(\boldsymbol{\varphi}\left(\boldsymbol{x};t\right)\right) = \boldsymbol{\psi}\left(\boldsymbol{x}\right) + tD_{\boldsymbol{u}}\boldsymbol{\psi}\left(\boldsymbol{x}\right) + o(t) \tag{117}$$

The Lie derivative defined in Equation (113) can now be explicitly written as:

$$L_{\boldsymbol{u}}\boldsymbol{\psi} = \lim_{t \to 0} \frac{D_{\boldsymbol{\varphi}(\boldsymbol{x};t)}\boldsymbol{\varphi}\left(\boldsymbol{x};-t\right)\left(\boldsymbol{\psi}\left(\boldsymbol{\varphi}\left(\boldsymbol{x};t\right)\right)\right) - \boldsymbol{\psi}\left(\boldsymbol{x}\right)}{t} =$$

$$= \lim_{t \to 0} \frac{\left(\mathbf{I} - t\mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\left(\boldsymbol{x}\right) + o(t)\right)\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right) + tD_{\boldsymbol{u}}\boldsymbol{\psi}\left(\boldsymbol{x}\right) + o(t)\right) - \boldsymbol{\psi}\left(\boldsymbol{x}\right)}{t} =$$

$$= \lim_{t \to 0} \frac{\left(\mathbf{I} - t\mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\left(\boldsymbol{x}\right) + o(t)\right)\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right) + t\mathbb{D}_{\boldsymbol{x}}\boldsymbol{\psi}\left(\boldsymbol{x}\right)\boldsymbol{u} + o(t)\right) - \boldsymbol{\psi}\left(\boldsymbol{x}\right)}{t} =$$

$$= \mathbb{D}_{\boldsymbol{x}}\boldsymbol{\psi}\left(\boldsymbol{x}\right)\boldsymbol{u}\left(\boldsymbol{x}\right) - \mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\left(\boldsymbol{x}\right)\boldsymbol{\psi}\left(\boldsymbol{x}\right) =$$

$$= D_{\boldsymbol{u}}\boldsymbol{\psi}\left(\boldsymbol{x}\right) - D_{\boldsymbol{\psi}}\boldsymbol{u}\left(\boldsymbol{x}\right) = [\boldsymbol{u},\boldsymbol{\psi}]$$
(118)

with $[u, \psi]$ being called the *Lie brackets*. Coming back to images \mathbb{X} , considered as planar continuum bodies, for the sake of simplicity, the translation g in

Equation (103), corresponds to a deformation $\varphi(x; -1) = x - c_g$, with $c_g = u(x)$, such that:

$$g\mathbb{X} = \boldsymbol{\psi}\left(\boldsymbol{x} - \boldsymbol{c}_{g}\right) = \boldsymbol{\psi}\left(\boldsymbol{x} - \boldsymbol{u}\left(\boldsymbol{x}\right)\right) = \mathbb{X} + D_{-\boldsymbol{u}\left(\boldsymbol{x}\right)}\boldsymbol{\psi}\left(\boldsymbol{x}\right) + o\left(\|\boldsymbol{u}\|\right) = \boldsymbol{\psi}\left(\boldsymbol{\varphi}\left(\boldsymbol{x};1\right)\right)$$

Locally, the group g of dimension 2 reflects a translation on a image in its actual configuration x, with a hyper tangent plan defined by

$$\mathbb{D}_{\boldsymbol{x}}\boldsymbol{\psi}\left(\boldsymbol{x};t\right) = \sum_{n=1}^{2} \frac{\partial \boldsymbol{\psi}\left(\boldsymbol{x}\right)}{\partial x_{n}} \otimes \boldsymbol{e}_{n}$$

The translation on the hyper tangent plane is provided by u_1 and u_2 that represents the local *displacement* components on the hyper-plane tangent to the group orbit and along which infinitesimally-close translations from the local diffeomorphism can be performed. The group action is resolved by the term $D_{-\boldsymbol{u}(\boldsymbol{x})}\boldsymbol{\psi}(\boldsymbol{x})$. S. Mallat in his works [Mal16; Cam20] proposes to refer to the reference configuration \boldsymbol{x}_0 and to decompose the displacement into two components: a global and a local one, obtained by approximating the displacement vector at the first order Taylor expansion around it:

$$\boldsymbol{u}\left(\boldsymbol{x}\right) = \boldsymbol{u}\left(\boldsymbol{x}_{0}\right) + \mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\left(\boldsymbol{x}_{0}\right)\left(\boldsymbol{x} - \boldsymbol{x}_{0}\right)$$

with an associated deformation (at first order in Equation (114), for t = 1) that reads:

$$\boldsymbol{\varphi}\left(\boldsymbol{x};-1\right) = \left(\mathbf{I} - \mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\left(\boldsymbol{x}_{0}\right)\right)\left(\boldsymbol{x} - \boldsymbol{x}_{0}\right) + \boldsymbol{x}_{0} - \boldsymbol{u}\left(\boldsymbol{x}_{0}\right)$$
(119)

The local action is represented by the local strain $(\mathbf{I} - \mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}(\boldsymbol{x}_0))(\boldsymbol{x} - \boldsymbol{x}_0)$ and the global translation by $\boldsymbol{x}_0 - \boldsymbol{u}(\boldsymbol{x}_0)$. According to Equation (117), the image transformation, based on this local diffeomorphism reads:

$$\boldsymbol{\psi}\left(\boldsymbol{\varphi}\left(\boldsymbol{x};-1\right)\right) = \boldsymbol{\psi}\left(\boldsymbol{x}\right) + \mathbb{D}_{\boldsymbol{x}}\boldsymbol{\psi}\left(\boldsymbol{x}\right)\left(\left(\mathbf{I} - \mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\right)\left(\boldsymbol{x} - \boldsymbol{x}_{0}\right) + \boldsymbol{x}_{0} - \boldsymbol{u}\left(\boldsymbol{x}_{0}\right)\right) \quad (120)$$

In analogy with group theory, the term $\mathbb{D}_{\boldsymbol{x}}\boldsymbol{\psi}\left(\left(\mathbf{I}-\mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\right)(\boldsymbol{x}-\boldsymbol{x}_{0})+\boldsymbol{x}_{0}-\boldsymbol{u}\left(\boldsymbol{x}_{0}\right)\right)$ represent the group action on $\boldsymbol{\psi}$. In analogy with the infinitesimal strain theory in mechanics, the *small displacement* hypothesis is granted by

$$\sup_{\boldsymbol{x}\in\Omega_{t}}\left\|\boldsymbol{u}\left(\boldsymbol{x}\right)\right\|\ll\mathbf{L}$$

and the *small strain* assumption by

$$\sup_{\boldsymbol{x}\in\Omega_{t}}\left\|\mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\left(\boldsymbol{x}\right)\right\|\ll1$$

with L being the characteristic size of the image (in this case, since the coordinates are normalized, L = 1).

3.1.2 Group invariance

The Lie algebra allow us to conceive a classifier h_{θ} insensitive to diffeomorphism (i.e., approximating the labelling function f, if it exists), by directly working on the grid of pixels. The dimension of such group of diffeomorphism is potentially high [Mal16; Cam20]. In the following, we attempt at approximating the labelling function f with a function $h_{\theta} : \mathcal{X} \to \mathbb{R}$ defined as:

$$h_{\theta}\left(\mathbb{X}\right) = \sum_{n=1}^{N} \phi_{n}(\mathbb{X}) w_{n} = \left\langle \boldsymbol{z}\left(\mathbb{X}\right), \boldsymbol{w}\right\rangle$$
(121)

The idea of S.Mallat is to learn f by making the group symmetries of h_{θ} to match the same invariants $g \in G$. In other words, it is necessary that the G is a symmetry group of z, i.e. the ensemble quotient in Equation (108) [Mal16; Cam20]. The problem is that z cannot be constructed out of the symmetry groups of G, since those are potentially infinite and not known before hand. If we consider the group of diffeomorphisms, we hope we can learn the subgroup of diffeormophisms that do not affect the labelling function by adjusting the weights \boldsymbol{w} , since forcing \boldsymbol{z} to learn the invariant would be too cumbersome and this strategy could lead to overfit the data. Moreover, S. Mallat showed a remarkable result in his works [Mal16; Cam20]: the canonical group invariant are not working for approximating the labelling function. In simple words, learning the canonical group invariants means to learn the parameters $s_g, \boldsymbol{c}_q, \boldsymbol{R}_q$ defined in Equations (103), (104) and (106) respectively. In other words, in analogy with mechanics, the idea behind this strategy is to refer all the configuration to the reference one x_0 , provided that this reference configuration is known before hand (a *deformable templates*, see [GM98]). For instance:

• Scale renormalization:

$$g\mathbb{X} = s_{g}\boldsymbol{\psi}(\boldsymbol{x}) = \boldsymbol{\psi}(\boldsymbol{x}_{0}) \,\forall s_{g} \Rightarrow s_{g} = \frac{\sum_{i=1}^{H} \sum_{j=1}^{W} \|\boldsymbol{\psi}(x_{0,1j}, x_{0,2i})\|}{\sum_{i=1}^{H} \sum_{j=1}^{W} \|\boldsymbol{\psi}(x_{1j}, x_{2i})\|}$$
(122)

If the average norm of $\boldsymbol{\psi}(\boldsymbol{x}_0)$ is 1, all possible scale factors s_g of the sort do not change $\boldsymbol{\psi}(\boldsymbol{x}) = g^{-1}\boldsymbol{\psi}(\boldsymbol{x}_0) = \left(\sum_{i=1}^{H}\sum_{j=1}^{W} \|\boldsymbol{\psi}(x_{1j}, x_{2i})\|\right)\boldsymbol{\psi}(\boldsymbol{x}_0).$

• Recentering:

$$g\mathbb{X} = \psi(\mathbf{x} - \mathbf{c}_{g}) = \psi(\mathbf{x}_{0})$$

$$\mathbf{c}_{g} = \frac{\sum_{j=1}^{W} x_{1j} \|\psi(x_{1j}, x_{2i}) \|\mathbf{e}_{1} + \sum_{i=1}^{H} x_{2i} \|\psi(x_{1j}, x_{2i}) \|\mathbf{e}_{2}}{\sum_{i=1}^{H} \sum_{j=1}^{W} \|\psi(x_{1j}, x_{2i})\|}$$

$$\Rightarrow \mathbf{c}_{g} = \mathbf{x} - \mathbf{x}_{0}$$
(123)

with $\boldsymbol{\psi}\left(\boldsymbol{x}\right)=g^{-1}\boldsymbol{\psi}\left(\boldsymbol{x}_{0}\right)=\boldsymbol{\psi}\left(\boldsymbol{x}_{0}+\boldsymbol{c}_{g}\right)$

However, this approach is doomed to fail according to S. Mallat [Mal16; Cam20], since the template image is not accessible nor defined in some cases. S. Mallat proposes an alternative approach, i.e., he proposes to remove the diffeomorphism, by selecting a feature space F orthogonal to the weight space Θ :

$$h_{\theta}\left(\mathbb{X}\right) = \langle \boldsymbol{z}\left(g\mathbb{X}\right), \boldsymbol{w} \rangle = h_{\theta}\left(g\mathbb{X}\right) = \langle \boldsymbol{z}\left(g\mathbb{X}\right), \boldsymbol{w} \rangle$$
$$\iff \langle \boldsymbol{z}\left(\mathbb{X}\right) - \boldsymbol{z}\left(g\mathbb{X}\right), \boldsymbol{w} \rangle \quad \forall \boldsymbol{w} \in \Theta$$
(124)

Retrieving the infinitesimal strain theory in Equation (120) and the Taylor expansion in Equation (117) with t = -1, Equation (124) can be linearized as follows:

$$\langle \boldsymbol{z} \left(\boldsymbol{\mathbb{X}} \right) - \boldsymbol{z} \left(\boldsymbol{g} \boldsymbol{\mathbb{X}} \right), \boldsymbol{w} \rangle = \langle \boldsymbol{z} \left(\boldsymbol{\psi} \left(\boldsymbol{x} \right) \right) - \boldsymbol{z} \left(\boldsymbol{\psi} \left(\boldsymbol{\varphi} \left(\boldsymbol{x} ; -1 \right) \right) \right), \boldsymbol{w} \rangle \approx \\ \approx -\mathbb{D}_{\boldsymbol{\psi}} \boldsymbol{z} \left(\boldsymbol{\psi} \left(\boldsymbol{x} \right) \right) \left(\boldsymbol{\psi} \left(\boldsymbol{\varphi} \left(\boldsymbol{x} ; -1 \right) \right) \right) \quad \forall \boldsymbol{w} \in \Theta$$

$$(125)$$

In other words, Equation (125) highlights the fact that the gradient of the hidden features z engenders the space $\mathcal{Z} = \Theta^{\perp}$, for all images $\mathbb{X} \in L(f)$, identified via the learning process that determines the optimum weights w. A trivial strategy to achieve this result would be to force the invariance of $z(g\mathbb{X}) = z(\mathbb{X})$ with the respect of any group action, so that $z(\mathbb{X}) = z(g\mathbb{X}) = z(\psi(x_0))$. However, this requirement can be rather prohibitive to achieve. A weaker condition is to assure that the feature a Lipschitz functions [Mal16; Cam20], in order to assure they remains "close" for infinitesimal group action, i.e., the following property must hold:

$$\|\boldsymbol{z}\left(\mathbb{X}\right) - \boldsymbol{z}\left(g\mathbb{X}\right)\| \leq C\left(\sup_{\boldsymbol{x}\in\Omega_{t}}\|\boldsymbol{u}\left(\boldsymbol{x}\right)\| + \sup_{\boldsymbol{x}\in\Omega_{t}}\|\mathbb{D}_{\boldsymbol{x}}\boldsymbol{u}\left(\boldsymbol{x}\right)\|\right)\|\boldsymbol{z}\left(\mathbb{X}\right)\|$$
(126)

Therefore, provided that a small strain and/or a small translations is applied to the image, if the hidden features are Lipschitz, their variation are bounded. However, natural images can be rather discontinuous, which hinders the extraction or Lipschitz features that are stable against infinitesimal groups actions. In this case, seemingly small strain and translations can definitely cause a mismatch between the original and deformed image and the distance

$$\|\boldsymbol{z} (\boldsymbol{\mathbb{X}}) - \boldsymbol{z} (\boldsymbol{g} \boldsymbol{\mathbb{X}})\| = \|\boldsymbol{z} (\boldsymbol{\mathbb{X}})\| + \|\boldsymbol{z} (\boldsymbol{g} \boldsymbol{\mathbb{X}})\| - 2\sum_{i=1}^{W} \sum_{j=1}^{H} \langle \boldsymbol{z} (\boldsymbol{\psi} (i,j)), \boldsymbol{z} (\boldsymbol{\psi} (i,j)) \rangle \approx \|\boldsymbol{z} (\boldsymbol{\mathbb{X}})\|$$
(127)

because the projection of a features onto the new one is small because the strain is not "small" enough to assure the feature stability. There is no absolute limit amplitude for diffeomorphism to assure the fact that the features remain Lipschitz.

If one considers a linear diffeomorphisms, with $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{\varepsilon}.\boldsymbol{x}$, with a small symmetric strain tensor $\boldsymbol{\varepsilon} = \varepsilon_{11}\boldsymbol{e}_1 \otimes \boldsymbol{e}_1 + \varepsilon_{22}\boldsymbol{e}_2 \otimes \boldsymbol{e}_2$, with $0 \leq \varepsilon_1 \ll 1, 0 \leq \varepsilon_2 \ll 1$, representing constant bi-directional stretching, Equation (120) reads:

$$\boldsymbol{\psi}\left(\boldsymbol{\varphi}\left(\boldsymbol{x};-1\right)\right) = \boldsymbol{\psi}\left(\boldsymbol{x}\right) + \mathbb{D}_{\boldsymbol{x}}\boldsymbol{\psi}\left(\boldsymbol{x}\right)\left(\left(\mathbf{I}-\boldsymbol{\varepsilon}\right)\boldsymbol{x}\right)$$
(128)

If one considers features $\boldsymbol{z}(\mathbb{X})$ obtained by Fourier transform of the image, defined as follows:

$$\boldsymbol{z}\left(\mathbb{X}\right) = \mathcal{F}\left(\mathbb{X}\right) = \int_{\mathbb{R}^{2}} \boldsymbol{\psi}\left(\boldsymbol{x}\right) e^{-i\langle \boldsymbol{k}, \boldsymbol{x} \rangle} d\boldsymbol{x} = \hat{\boldsymbol{\psi}}\left(\boldsymbol{k}\right)$$
(129)

the features corresponding to the bi-directionally stretched image read:

$$\boldsymbol{z} (g\boldsymbol{\mathbb{X}}) = \mathcal{F} (g\boldsymbol{\mathbb{X}}) = \int_{\mathbb{R}^2} \boldsymbol{\psi} (\boldsymbol{x} - \boldsymbol{\varepsilon}.\boldsymbol{x}) e^{-i\langle \boldsymbol{k}, \boldsymbol{x} \rangle} d\boldsymbol{x} =$$
$$= \det^{-1} (\mathbf{I} - \boldsymbol{\varepsilon}) \int_{\mathbb{R}^2} \boldsymbol{\psi} (\boldsymbol{x}) e^{-i\langle (\mathbf{I} - \boldsymbol{\varepsilon})^{-T} \boldsymbol{k}, \boldsymbol{x} \rangle} d\boldsymbol{x} =$$
$$= \frac{1}{(1 - \varepsilon_1) (1 - \varepsilon_2)} \boldsymbol{\hat{\psi}} \left((\mathbf{I} - \boldsymbol{\varepsilon})^{-T} \boldsymbol{k} \right)$$
(130)

According to Equation (130), a bi-directional stretch smoothens the image, since high-frequencies features are contracted by a factor $(\mathbf{I} - \boldsymbol{\varepsilon})^{-T}$ towards the low frequencies and amplified by a factor $\frac{1}{(1-\varepsilon_1)(1-\varepsilon_2)}$. Therefore, even for small strain, the Fourier features are not necessarily stable by stretching (see Section 3.1.2). The Lipschitz condition is strictly related with features that assure the scale separability, which is obtained by wavelet transform in combination with non-linear activations (see [Mal09]).

3.2 Average Pooling

The most trivial strategy to force invariance by translation and dilatation ($\varphi = x + c + \text{diag}(\alpha_1, \alpha_2, 0) x$), mentioned in Section 3.1.2 is to apply consecutively multiple group actions to each feature map and sum the contribution, such as :

$$\sum_{g \in G} g.\mathbb{X} = \sum_{g \in G} \psi \left(\boldsymbol{x} - \boldsymbol{c}_g \right)$$
(131)

because Equation (131) represents the average image, that can be rewritten as $\sum_{g \in G} \psi\left(\mathbf{x}'_{g}\right)$ by change of variables $\mathbf{x}'_{g}(\mathbf{x}) = \mathbf{x} + \mathbf{c} + \text{diag}\left(\alpha_{1}, \alpha_{2}, 0\right)\mathbf{x} - \mathbf{c}_{g}$. Another interesting interpretation is given by the fact that the Fourier transform of a translated vector $\boldsymbol{\psi}\left(\mathbf{x} - \mathbf{c}_{g}\right)$ in \mathbb{R}^{2} is independent of \mathbf{c}_{g} in the following condition [Cam20]:

$$\mathcal{F}\left(\boldsymbol{\psi}\left(\boldsymbol{x}-\boldsymbol{c}_{g}\right)\right)=e^{i\left\langle\boldsymbol{k},\boldsymbol{c}_{g}\right\rangle}\cdot\mathcal{F}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right)=\mathcal{F}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right)\forall\boldsymbol{c}_{g}\iff\boldsymbol{k}=\boldsymbol{0}$$
(132)

with $\hat{\psi}(\mathbf{0}) = \mathcal{F}(\psi(\mathbf{x}))(\mathbf{0})$ being the average of the $\psi(\mathbf{x})$. Averaging the image cancels all local image features and all the previously applied group actions

along the group orbit (and of their order).

To avoid this loss of information, $\boldsymbol{z}(\mathbb{X})$ can be obtained by performing an averaging as in Equation (131) on a linear combination of N_c feature maps $z_c(\mathbb{X})$, corresponding to different *channels*. In other words, according to S. Mallat [Cam20], the features are obtained by *average pooling*, defined as:

$$\boldsymbol{z}\left(\mathbb{X}\right) = \sum_{c=1}^{N_c} \sum_{g \in G} \boldsymbol{e}_c z_c \left(\boldsymbol{\psi} \left(\boldsymbol{x} - \boldsymbol{c}_g\right)\right) = \\ = \sum_{c=1}^{N_c} \left(\sum_{g \in G} g z_c \left(\mathbb{X}\right) \boldsymbol{e}_c\right) = AvgPooling\left(\sum_{c=1}^{N_c} z_c \left(\mathbb{X}\right) \boldsymbol{e}_c\right)$$
(133)

Average pooling in Equation (133) grants the sought invariance by translation (or other group action) provided that each feature map z_c (each channel) is equivariant by translation:

$$\sum_{c=1}^{N_c} z_c \left(\mathbb{X}\right) \boldsymbol{e}_c = \sum_{c=1}^{N_c} \left(\sum_{g \in G} z_c \left(g\mathbb{X}\right) \boldsymbol{e}_c \right) = \sum_{c=1}^{N_c} \left(\sum_{g \in G} z_c \left(\psi \left(\boldsymbol{x} - \boldsymbol{c}_g\right)\right) \boldsymbol{e}_c \right) =$$
$$= \sum_{c=1}^{N_c} \left(\sum_{g \in G} g z_c \left(\mathbb{X}\right) \boldsymbol{e}_c \right)$$
(134)

The equivariance by translation of each feature map suggests that each channel should be rendered by a covariant operator, such as the linear convolution (see Figure 6). In other words, features z are invariant by group action (translation, rotation, diffeomorphisms) thanks to average pooling performed on feature maps that commute with the same group action, i.e., if the image is deformed, each feature map follows the same deformation, according to the following expression:

$$z_{c}(g\mathbb{X}) = z_{c}\left(\boldsymbol{\psi}\left(\boldsymbol{x} - \boldsymbol{c}_{g}\right)\right) = z_{c}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right) - \boldsymbol{c}_{g}\right) = gz_{c}\left(\boldsymbol{x}\right)$$
(135)

Pooling layers can be applied on the top of a standard \mathcal{MLP} of the type described in Section 2.2, for classification purposes. However, \mathcal{MLP} are not necessarily producing translation-equivariant output. The average pooling output is independent of translations and dilatation, so that the classification is not affected (see Section 2.5 for technical details about the *AvgPooling* implementation). c_g is called *stride* and it can be larger than 1 pixel, which consists into an average operation with subsampling. Moreover, the average operation does not require the previous knowledge of G such as in the approach that learns the canonical invariants (see the recentering in Equation (123)).



Figure 6: Convolutional feature maps extracted from a digit image belonging to the Omniglot dataset (https://www.omniglot.com) and translated by using the elasticdeform library (https://elasticdeform.readthedocs.io/en/latest/license.html).

3.3 Convolutional layer

Assuming equivariant group action on the feature maps, the effect of any further group actions g' on the feature maps vanishes (because of the sum over all the groups):

$$\boldsymbol{z}\left(g'\boldsymbol{\mathbb{X}}\right) = \sum_{c=1}^{N_c} \sum_{g \in G} z_c\left(g\left(g'\boldsymbol{\mathbb{X}}\right)\right) \boldsymbol{e}_c = \sum_{c=1}^{N_c} \sum_{g \in G} g' z_c\left(g\boldsymbol{\mathbb{X}}\right) \boldsymbol{e}_c$$
(136)

In order to force the equivariance to translation (but also with the respect to rotation and other small strain) of each feature maps $\boldsymbol{z}(\mathbb{X}) = \sum_{c=1}^{N_c} z_c(\mathbb{X}) \boldsymbol{e}_c$, the latter must be the result of a *convolution*, as defined in Appendix A and reported in the following expression:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^{2}} \boldsymbol{H}\left(\boldsymbol{u}\right) \cdot \boldsymbol{\psi}\left(\boldsymbol{x}-\boldsymbol{u}\right) d\boldsymbol{u} + \boldsymbol{b}$$
(137)

with H the causal convolution *kernel* (i.e. the response function of the LTI system) and with b the filter bias. H has usually a compact support. Since the image is a discrete signal defined by via the sampling functions s_1 and s_2 as (see Equation (226)):

Gatti 305

$$\boldsymbol{\psi}\left[i,j\right] = s_1\left(\frac{x_{1i}}{W}\right) \cdot s_2\left(\frac{x_{2j}}{H}\right) \mathbf{I} \star \boldsymbol{\psi}\left(\boldsymbol{x}\right)$$
(138)

the feature maps are the result of a discrete convolution that reads:

$$\boldsymbol{z}[j,i] = \sum_{u=0}^{k_W-1} \sum_{v=0}^{k_H-1} \boldsymbol{H}[u,v] \cdot \boldsymbol{\psi}[i-u,j-v] + \boldsymbol{b}, \quad (i,j) \in [\![0,W]\!] \times [\![0,H]\!]$$
(139)

with $f[u, v] = f\left(\frac{u}{W}, \frac{v}{H}\right)$ and \boldsymbol{z} being stored in C order, contiguous along the rows. Each feature map $z_c(\mathbb{X}) = \langle \boldsymbol{H} \star \boldsymbol{\psi}, \boldsymbol{e}_c \rangle + b_c$ corresponds to a LTI filter on the image, for a total of N_c filters. Refer to Section 2.5 for an extensive excursus on practical aspects related to the the implementation of discrete convolution.

Remark 30. Strided convolutions, defined as:

$$\boldsymbol{z}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right)\right) = \int_{\mathbb{R}^{2}} \boldsymbol{H}\left(\boldsymbol{u}\right) \cdot \boldsymbol{\psi}\left(\operatorname{diag}\left(\boldsymbol{s}_{W}, \boldsymbol{s}_{H}, \boldsymbol{0}\right) \boldsymbol{x} - \boldsymbol{u}\right) d\boldsymbol{u}$$
(140)

with strides $s_W \geq 1$ and $s_H \geq 1$, are a way of reducing the dimensionality of the data, but they represent a loss of information (since some pixels are skipped) and they disrupt the translation-equivariance (or covariance, stated in Equation (199)) because they infringe the Nyquist-Shannon theorem [AW19]. In other words, when the convolution is strided, if the underlying image is translated, the resulting feature maps do not match the translated feature maps of the original image. As a matter of fact, discrete signals should be the result of a sampling at at least twice the highest frequency in the input analog signal, in order to grant its correct reconstruction. In the case of strided convolution, the image/feature map is sampled according of the stride values, disregarding some of the information contained in the original signal. In order to counteract this problem, the number of feature maps increases accordingly, but still they are not translation-equivariant [MMD20]. On the other hand, downsampling due to non-strided convolutions or pooling, break the hypothesis (valid for continuous signals) of translation invariance [SG20]. This effect is mitigated by adding zero-padding at the edge of the image.

Remark 31. Despite breaking feature maps' equivariance, subsampling is still largely employed with success. According to [MMD20], this is mostly because of the subsampling "shiftability" property. Shiftability is defined as the translation scaled by the associated subsampling stride, i.e.:

$$z_{c}(g\mathbb{X}) = z_{c}\left(\boldsymbol{\psi}\left(\boldsymbol{x} - \boldsymbol{c}_{g}\right)\right) = z_{c}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right) - \operatorname{diag}\left(\frac{1}{s_{W}}, \frac{1}{s_{H}}, 0\right)\boldsymbol{c}_{g}\right)$$
(141)

with s_W and s_H being the stride along the image width and height respectively. In other words, the subsampled feature maps obtained by strided convolution are equivariant to all group translations by a vector that is a fraction of the stride. Shiftability is a scaled version of the translation-equivariance. For

a stride couple (s_W, s_H) , there exist $s_W \cdot s_H$ equivariant translations of the feature map, i.e. $\forall (\alpha_W, \alpha_H)$, with $1 \leq \alpha_W \leq s_W$ and $1 \leq \alpha_H \leq s_H$ the following expression holds:

$$z_{c}(g\mathbb{X}) = z_{c}\left(\boldsymbol{\psi}\left(\boldsymbol{x} - \operatorname{diag}\left(\alpha_{W}, \alpha_{H}\right)\boldsymbol{c}_{g}\right)\right) = z_{c}\left(\boldsymbol{\psi}\left(\boldsymbol{x}\right) - \operatorname{diag}\left(\frac{\alpha_{W}}{s_{W}}, \frac{\alpha_{H}}{s_{H}}, 0\right)\boldsymbol{c}_{g}\right)$$
(142)

Equation (142) proves that subsampling reduces the so called "image movement" [MMD20], defined as the amount of translation of the feature map from the untranslated position \mathbf{x}_0 : shiftable feature maps are equivariant to $\alpha_W \geq 1$ and $\alpha_H \geq 1$ times larger. However, subsampling reduces the so called "image similarity" due to the information loss of skipping pixels. The reduced similarity prevents the invariance to translation of feature maps. The preserved similarity depends on the "local homogeneity" of the image [MMD20]: the larger the correlation length of $\boldsymbol{\psi}(\mathbf{x})$, the higher the local homogeneity and the larger will be the preserved similarity despite subsampling. However, the input image $\boldsymbol{\psi}(\mathbf{x})$ should not be treated explicitly (e.g., by applying a Gaussian blur). Instead, feature maps remain "similar" by using strided pooling: the stride (subsampling) reduces the "image movement", while, at the same time, pooling increases local homogeneity, by averaging over the kernel [MMD20].

The role of non-linear activation Convolutional filters are applied to the image or to the feature maps of the previous layers, followed by non-linear activation functions γ^{16} that are also equivariant by translation and small strain (see Figure 7):

$$\gamma\left(z_c\left(g\mathbb{X}\right)\right) = g\gamma\left(z_c\left(\mathbb{X}\right)\right) \tag{143}$$

The easiest choice is to adopt pixel-wise non-linear activation functions [Cam20].



Figure 7: Action of the average pooling, convolution and convolution with ReLU activation on a distorted image gX.

Moreover, S. Mallat suggests that non-linear activation functions can be adopted to threshold local pattern (or noise).

 $^{16\}gamma$ is used instead of the standard notation g, in order to avoid confusion with the group action.

4 Optimizing a Neural Network

4.1 Gradient Descent (GD) algorithm: first order approach

If one considers a function $L_{\mathcal{D}_{XY}}: \Theta \to \mathbb{R}$, with $\Theta \subset \mathbb{R}^{d_{\Theta}}$ and $f \in C^1(\Theta)$, the Taylor expansion expressed by the following expression holds:

$$T_{\hat{\boldsymbol{\theta}}}L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}\right) = L_{\mathcal{D}_{XY}}\left(\hat{\boldsymbol{\theta}}\right) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{\theta}}L_{\mathcal{D}_{XY}}\left(\hat{\boldsymbol{\theta}}\right), \boldsymbol{\theta} - \hat{\boldsymbol{\theta}} \right\rangle + o\left(\|\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\|\right) \quad (144)$$

with $o\left(\|\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\|\right)$ being a quantity such that $\lim_{\|\boldsymbol{\theta} - \hat{\boldsymbol{x}}\| \to 0} \frac{o\left(\|\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\|\right)}{\|\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\|} = 0$. Being K convex, the following function is positive (by Theorem 73):

$$\left\langle \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\hat{\theta}}\right), \boldsymbol{\theta} - \boldsymbol{\hat{\theta}} \right\rangle \ge 0, \quad \boldsymbol{\theta} \in K$$
 (145)

This means that, at the first order,

$$T_{\hat{\boldsymbol{\theta}}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right) \ge L_{\mathcal{D}_{XY}} \left(\hat{\boldsymbol{\theta}} \right)$$

and

$$\left\langle \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\hat{\theta}}\right), \boldsymbol{\theta} - \boldsymbol{\hat{\theta}} \right\rangle = \mathcal{O}\left(\left\| \boldsymbol{\theta} - \boldsymbol{\hat{\theta}} \right\| \right)$$

. In this sense, to minimize $L_{\mathcal{D}_{XY}}$ on a convex set K one needs to search in the direction of $-\nabla_{\theta}L_{\mathcal{D}_{XY}}$, i.e., at the first order:

$$\boldsymbol{\theta} = \boldsymbol{\hat{\theta}} + \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\hat{\theta}} \right)$$
(146)

with $\eta \in \mathbb{R}^+$ being the *learning rate*, and replacing it in Equation (145), the Euler's inequality is satisfied and $L_{\mathcal{D}_{XY}}(\boldsymbol{\theta}(\eta)) \geq L_{\mathcal{D}_{XY}}(\boldsymbol{\hat{\theta}})$, $\forall \eta \in \mathbb{R}^+$. In alternative, the following function $\xi : \mathbb{R}^+ \times \Theta \to \mathbb{R}$

$$\boldsymbol{\xi} : (\eta, \boldsymbol{\theta}) \mapsto L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} - \eta \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right) \right)$$
(147)

gives an idea of how $L_{\mathcal{D}_{XY}}$ evolves along the half-line directed along $-\nabla_{\theta}L_{\mathcal{D}_{XY}}(\boldsymbol{\theta})$. The Taylor expansion of ξ with the respect to η reads:

$$T_{0}\xi(\eta;\boldsymbol{\theta}) = L_{\mathcal{D}_{XY}}(\boldsymbol{\theta}) - \eta \|\boldsymbol{\nabla}_{x}L_{\mathcal{D}_{XY}}(\boldsymbol{\theta})\|^{2} + o(\eta)$$
(148)

In this sense, provided $\eta > 0$ small enough, $L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} - \eta \cdot \nabla_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right) \right) \leq L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right)$. The choice of the gradient is even the optimal one, due to the following result (derived from Equation (145):

$$-r \frac{\nabla_{\theta} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right)}{\|\nabla_{\theta} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right)\|} = \operatorname*{arg\,min}_{\|\boldsymbol{\delta}\boldsymbol{\theta}\|=r} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} + \boldsymbol{\delta}\boldsymbol{\theta} \right)$$
(149)

Therefore, the standard Gradient Descent algorithm (GD) iteratively updates the weights according to the following scheme (inspired by Equation (146)):

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \eta^{(i)} \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right), \quad \eta^{(i)} \in \mathbb{R}^+$$
(150)

This choice is particularly attractive from a computational standpoint, since:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} - \eta^{(i)} \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right) \right) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i+1)} \right)$$
(151)

However, $\eta^{(i)}$ must be small enough to grant the Euler's inequality

$$T_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\hat{\boldsymbol{\theta}} \right) \leq L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right)$$

to be satisfied at the first order. η can be updated at each iteration. For instance, the greedy choice of $\eta^{(i)}$ reads:

$$\eta^{(i+1)} = \underset{\eta>0}{\operatorname{arg\,min}} \, \xi\left(\eta; \boldsymbol{\theta}^{(i)}\right) \tag{152}$$

The greedy choice is reached when

$$\xi'\left(\eta^{(i)},\boldsymbol{\theta}^{(i)}\right) = \left\langle \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)} - \eta^{(i)}\boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right)\right), \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i+1)}\right) \right\rangle = 0$$

which is reached when the gradients of the loss function at two consecutive epochs are orthogonal.

4.2 Gradient Descent algorithm: second order approach

If $L_{\mathcal{D}_{XY}}$ is twice differentiable, then its Taylor's expansion at the second order reads:

$$T_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\delta} \boldsymbol{\theta} \right) = L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right), \boldsymbol{\delta} \boldsymbol{\theta} \right\rangle + \frac{1}{2} \left\langle \mathbf{H}_{L_{\mathcal{D}_{XY}}} \left(\boldsymbol{\theta} \right). \boldsymbol{\delta} \boldsymbol{\theta}, \boldsymbol{\delta} \boldsymbol{\theta} \right\rangle + o(\|\boldsymbol{\delta} \boldsymbol{\theta}\|^2)$$
(153)

Analogously, the Hessian $\mathbf{H}_{L_{\mathcal{D}_{XY}}}(\boldsymbol{\theta})$ appears in the Taylor's expansion of the gradient:

$$T_{\boldsymbol{\theta}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\delta} \boldsymbol{\theta} \right) = \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta} \right) + \mathbf{H}_{L_{\mathcal{D}_{XY}}} \left(\boldsymbol{\theta} \right) . \boldsymbol{\delta} \boldsymbol{\theta} + o(\| \boldsymbol{\delta} \boldsymbol{\theta} \|)$$
(154)

The Hessian $\mathbf{H}_{L_{\mathcal{D}_{XY}}}(\boldsymbol{\theta})$ is symmetric. Moreover, if the Hessian is positive semidefinite (i.e., non-negative eigenvalues), then $L_{\mathcal{D}_{XY}}$ is convex. $L_{\mathcal{D}_{XY}}$ is strictly convex is the Hessian's eigenvalues are strictly positive (see Remark 58).

Therefore, the Hessian allows to decide whether θ is a local minimum (positive semi-definite, see Theorem 75) or a global minimum (positive definite, see Theorem 78). Moreover, second order methods are based on the Hessian, i.e.,

the so called *Newton's method*, which are based on the general gradient descent algorithm:

$$\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \mathbf{H}_{L_{\mathcal{D}_{XY}}}^{-1} \left(\boldsymbol{\theta}^{(i)} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right)$$
(155)

Second order methods, if the Hessian is positive definite, converge faster than the first method in Equation (150), but more computationally expensive, since the Hessian must be computed and inverted at each epoch [Pey20]. The first order approximation defined by Equation (150) is a special case of the generalized Gradient Descent in Equation (150), for $\mathbf{H}_{L_{\mathcal{D}_{XY}}}\left(\boldsymbol{\theta}^{(i)}\right) = \eta^{(i)}\mathbf{I}$.

Remark 32. The optimal conditioning is achieved for Hessians that are close to the identity. In the supervised case, if the loss function $L_{\mathcal{D}_{XY}}$ is interpreted as the average log-likelihood of the parametric probability distribution $p_{\theta}(\boldsymbol{y}|\boldsymbol{x}) = p_{\theta}(\boldsymbol{y} = \boldsymbol{h}_{\theta}(\boldsymbol{x}))$, under certain regularity conditions for p_{θ} (see Section 1.3 for further details), the Hessian can be interpreted as the Fisher information associated to \boldsymbol{h}_{θ} . Moreover, if p_{θ} belongs to the family of exponential probability distributions (among which, there it exists the probability distribution that maximizes the conditional entropy $\mathbb{H}(\boldsymbol{Y}|\boldsymbol{X})$, according to the principle of maximum entropy exposed in Section 1.5), of the type

$$p_{\boldsymbol{\theta}} = \frac{e^{-\langle \boldsymbol{\theta}, \boldsymbol{y} \rangle}}{Z}$$

then the Hessian corresponds to the minus covariance of the observations $-Cov_{\boldsymbol{x}\sim p_{\boldsymbol{\theta}}}[\boldsymbol{y}(\boldsymbol{X})]$ [Cam22]. In this case, each diagonal term of the Hessian corresponds to the variance $Var_{\boldsymbol{x}\sim p_{\boldsymbol{\theta}}}(y_k(\boldsymbol{X}))$. Therefore, in order to facilitate the optimal conditioning and force the Hessian to approximate the identity, one can perform the so called *batch normalization* (BatchNorm), which consists into normalizing y_k and replace it by:

$$y_{k}^{'} = \frac{y_{k} - \mathbb{E}_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left[y_{k} \right]}{Var_{\boldsymbol{x} \sim p_{\boldsymbol{\theta}}} \left(y_{k} \left(\boldsymbol{X} \right) \right)}$$
(156)

The BatchNorm operation eases the convergence of the gradient descent algorithm, since it replaces the diagonal terms of the Hessian by 1, without computing the Hessian explicitly. In \mathcal{NN} , BatchNorm is often applied after the pre-activation at each layer, so to force the Hessian diagonal to 1. However, this is not always sufficient, since in complex \mathcal{NN} with non-linear activation functions, the Hessian (or the FIM, see Equation (24)) is not expressed in its orthonormal basis, which is unknown *a priori*. Therefore, there could be extra diagonal terms that prevent the optimal conditioning of the Hessian.

The analogy with the classical pre-conditioning methods in linear algebra is evident. Replacing $\theta^{(i+1)} - \theta^{(i)} = \delta \theta$ in Equation (153), the following second

order Taylor's expansion is obtained:

$$T_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i+1)} - \boldsymbol{\theta}^{(i)} \right) = L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right) - \left\langle \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right), \mathbf{H}_{L_{\mathcal{D}_{XY}}} \left(\boldsymbol{\theta}^{(i)} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right) \right\rangle + o(\|\mathbf{H}_{L_{\mathcal{D}_{XY}}} \left(\boldsymbol{\theta}^{(i)} \right) \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \left(\boldsymbol{\theta}^{(i)} \right) \|)$$

$$(157)$$

Given the expression in Equation (157), if $\mathbf{H}_{L_{\mathcal{D}_{XY}}}\left(\boldsymbol{\theta}^{(i)}\right)$ is positive definite and $\nabla_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right) \neq \mathbf{0}$, then $L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i+1)}\right) < L_{\mathcal{D}_{XY}}\left(\boldsymbol{\theta}^{(i)}\right)$, which means that the Newton's method are effectively attempting to iteratively minimize the Empirical Loss function [Pey20].

4.3 Convergence analysis of first order methods

Theorem 33. Convergence analysis of functions with Lipschitz gradients [Nes83]

A function $f: K \to \mathbb{R}$, defined over a non-empty convex set $K \subset H$, with Hbeing a Hilbert space, f proper and strictly convex on K, with $f \in C^1(K)$ and a gradient $\nabla_x f \in Lip(K)$ with Lipschitz constant β . If $\exists \delta_{\min}, \delta_{\max} \in \mathbb{R}^+$ such that:

$$0 < \frac{1}{\beta} = \delta_{\min} \le \delta_k < \delta_{\max} = \frac{2}{\beta}$$

then $\exists \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \delta_k \nabla_x f(\boldsymbol{x}_k)$ that converges to $\hat{\boldsymbol{x}} = \operatorname*{arg\,min}_{\boldsymbol{x} \in K} f(\boldsymbol{x})$ and there exists a constant A > 0 such that

$$f(\boldsymbol{x}_{k}) - f(\boldsymbol{\hat{x}}) \le \frac{cst.}{k+1}$$
(158)

Furthermore, if f is strongly convex of coefficient α , there exists $0 \leq \rho < 1$ such that:

$$\|\boldsymbol{x}_{k} - \hat{\boldsymbol{x}}\| \le \rho^{k} \|\boldsymbol{x}_{0} - \hat{\boldsymbol{x}}\|$$
(159)

with $0 < \rho \leq 1$.

Proof. Following the proof presented by [Pey20], since f is strongly convex on a convex set K, by Theorem 77 it exists a unique minimizer $\hat{\boldsymbol{x}}$. Moreover, the strict convexity implies Item 2. The fact that $\nabla_x f \in \text{Lip}(K)$ implies that $\forall (\boldsymbol{x}_{k+1}, \boldsymbol{x}_k) \in K^2$:

$$f(\boldsymbol{x}_{k+1}) \leq f(\boldsymbol{x}_{k}) + \langle \boldsymbol{\nabla}_{x} f(\boldsymbol{x}_{k}), \boldsymbol{x}_{k+1} - \boldsymbol{x}_{k} \rangle + \frac{\beta}{2} \|\boldsymbol{x}_{k+1} - \boldsymbol{x}_{k}\|^{2}$$
(160)

as stated by the upper bound in Proposition 81. Replacing the update scheme $\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = -\delta_k \nabla_x f(\boldsymbol{x}_k)$ in Equation (160), one obtains the following expression:

$$f(\boldsymbol{x}_{k+1}) \leq f(\boldsymbol{x}_{k}) + \delta_{k} \left(\frac{\delta_{k} \cdot \beta}{2} - 1\right) \|\boldsymbol{\nabla}_{x} f(\boldsymbol{x}_{k})\|^{2}$$
(161)

A necessary requirement to pursue the minimizer of f is $\delta_k < \frac{2}{\beta} = \delta_{\max}$, so that $f(\boldsymbol{x}_{k+1}) \leq f(\boldsymbol{x}_k)$ and therefore $\tau_k = \delta_k \left(\frac{\delta_k \beta}{2} - 1\right) < 0$. To prove the existence of a lower bound δ_{\min} the convexity property Item 2 must be evoked:

$$f(\hat{\boldsymbol{x}}) \ge f(\boldsymbol{x}_k) + \langle \boldsymbol{\nabla}_x f(\boldsymbol{x}_k), \hat{\boldsymbol{x}} - \boldsymbol{x}_k \rangle$$
(162)

Comparing Equation (161) and Equation (162), replacing the update scheme $\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = -\delta_k \boldsymbol{\nabla}_x f(\boldsymbol{x}_k)$ and considering that $\tau_k < 0$ leads to:

$$f(\mathbf{x}_{k+1}) \leq f(\hat{\mathbf{x}}) + \langle \nabla_{x} f(\mathbf{x}_{k}), \mathbf{x}_{k} - \hat{\mathbf{x}} \rangle + \tau_{k} \|\nabla_{x} f(\mathbf{x}_{k})\|^{2} =$$

$$= f(\hat{\mathbf{x}}) + \frac{\|\mathbf{x}_{k} - \hat{\mathbf{x}} + 2\tau_{k} \nabla_{x} f(\mathbf{x}_{k})\|^{2} - \|\mathbf{x}_{k} - \hat{\mathbf{x}}\|^{2}}{4\tau_{k}} =$$

$$= f(\hat{\mathbf{x}}) + \frac{\|\mathbf{x}_{k} - \hat{\mathbf{x}}\|^{2} - \|\mathbf{x}_{k} - \hat{\mathbf{x}} + 2\tau_{k} \nabla_{x} f(\mathbf{x}_{k})\|^{2}}{4|\tau_{k}|} =$$

$$= f(\hat{\mathbf{x}}) + \frac{\|\mathbf{x}_{k} - \hat{\mathbf{x}}\|^{2} - \|\mathbf{x}_{k+1} - \hat{\mathbf{x}} + \delta_{k} (\delta_{k}\beta - 1) \nabla_{x} f(\mathbf{x}_{k})\|^{2}}{4|\tau_{k}|}$$
(163)

From Equation (163), it must be noted that, in order to assure that $f(\boldsymbol{x}_{k+1}) \ge f(\hat{\boldsymbol{x}})$, one must assure that:

$$\|\boldsymbol{x}_{k} - \boldsymbol{\hat{x}}\|^{2} - \|\boldsymbol{x}_{k+1} - \boldsymbol{\hat{x}} + \delta_{k} \left(\delta_{k}\beta - 1\right) \boldsymbol{\nabla}_{x} f\left(\boldsymbol{x}_{k}\right)\|^{2} \ge 0$$
(164)

Adopting the well known relation $\|\boldsymbol{a}\|^2 - \|\boldsymbol{b}\|^2 \le \|\boldsymbol{a} - \boldsymbol{b}\|^2$, one obtains:

$$\delta_k^2 \left(\delta_k \beta - 2\right)^2 \|\boldsymbol{\nabla}_x f\left(\boldsymbol{x}_k\right)\|^2 \ge \|\boldsymbol{x}_k - \hat{\boldsymbol{x}}\|^2 - \\ -\|\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}} + \delta_k \left(\delta_k \beta - 1\right) \boldsymbol{\nabla}_x f\left(\boldsymbol{x}_k\right)\|^2 \ge 0$$
(165)

and again:

$$\|\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}} + \delta_k \left(\delta_k \beta - 1\right) \boldsymbol{\nabla}_x f\left(\boldsymbol{x}_k\right) \|^2 = \|\boldsymbol{\nabla}_x f\left(\boldsymbol{x}_k\right) \| \cdot \left(\frac{\|\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}\|^2}{\|\boldsymbol{\nabla}_x f\left(\boldsymbol{x}_k\right)\|} + \delta_k^2 \left(\delta_k \beta - 1\right)^2 \|\boldsymbol{\nabla}_x f\left(\boldsymbol{x}_k\right)\| + 2\delta_k \left(\delta_k \beta - 1\right) \|\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}\|\right)$$
(166)

and finally, calling $\varepsilon = \frac{\|\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}\|}{\|\boldsymbol{x}_k - \hat{\boldsymbol{x}}\|}$:

$$g(\varepsilon) = \varepsilon^2 + 2\delta_k \left(\delta_k \beta - 1\right)\varepsilon + \delta_k^2 \beta^2 \left(2\delta_k^2 \beta^2 - 6\delta_k \beta + 5\right) - 1 \ge 0$$
 (167)

In order to satisfy Equation (167) for all $\varepsilon > 0$, $\delta_k > \frac{1}{\beta} = \delta_{\min}$, as shown - for different values of β - in Figure 8 (blue-graded curves). Orange-graded lines in Figure 8 corresponds to values of $\delta_k < \frac{1}{\beta}$: in this case, it is possible to achieve $g(\varepsilon) > 0$ with $\beta \leq 1$, provided that $\varepsilon > \varepsilon_{\min} > 0$. This means that $\|\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}\| > \varepsilon_{\min} \|\boldsymbol{x}_k - \hat{\boldsymbol{x}}\|$ which implies that the GD algorithm may not converge. For $\beta > 1$, the algorithm never converges for $\delta_k < \frac{1}{\beta}$. Summing Equation (163) n_e times, one obtains:

$$n_{e}\left(f\left(\boldsymbol{x}_{n_{e}-1}\right)-f\left(\boldsymbol{\hat{x}}\right)\right) \leq \sum_{k=0}^{n_{e}-1} f\left(\boldsymbol{\hat{x}}_{k+1}\right)-n_{e} \cdot f\left(\boldsymbol{\hat{x}}\right)$$

$$\leq \sum_{k=0}^{n_{e}} \frac{\|\boldsymbol{x}_{k}-\boldsymbol{\hat{x}}\|^{2}-\|\boldsymbol{x}_{k+1}-\boldsymbol{\hat{x}}+\delta_{k}\left(\delta_{k}\beta-1\right)\boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}_{k}\right)\|^{2}}{4|\tau_{k}|}$$
(168)

If for instance $\delta = \delta_{\min} = \frac{1}{\beta}$, then:

$$f(\boldsymbol{x}_{n_{e}-1}) - f(\boldsymbol{\hat{x}}) \leq \frac{\beta}{2n_{e}} \sum_{k=0}^{n_{e}-1} \|\boldsymbol{x}_{k} - \boldsymbol{\hat{x}}\|^{2} - \|\boldsymbol{x}_{k+1} - \boldsymbol{\hat{x}}\|^{2} =$$

$$= \frac{2}{n_{e}\beta} \left(\|\boldsymbol{x}_{0} - \boldsymbol{\hat{x}}\|^{2} - \|\boldsymbol{x}_{n_{e}} - \boldsymbol{\hat{x}}\|^{2} \right) \leq \frac{\beta \|\boldsymbol{x}_{0} - \boldsymbol{\hat{x}}\|^{2}}{2n_{e}}$$
(169)

Equation (169) proves Equation (158) [Pey20].

Finally, following [Pey20], in order to prove the second part of the theorem, expressed in Equation (159), f has to be strongly convex, which implies, by Item 2 and adopting the assumption that $\nabla_x f(\hat{x}) = 0$, that :

$$\frac{\alpha}{2} \|\boldsymbol{x}_{k+1} - \boldsymbol{\hat{x}}\| \leq f(\boldsymbol{x}_{k+1}) - f(\boldsymbol{\hat{x}}) \leq \frac{\|\boldsymbol{x}_{k} - \boldsymbol{\hat{x}}\|^{2} - \|\boldsymbol{x}_{k+1} - \boldsymbol{\hat{x}} + \delta_{k}(\delta_{k}\beta - 1)\boldsymbol{\nabla}_{x}f(\boldsymbol{x}_{k})\|^{2}}{4|\tau_{k}|}$$
(170)

Replacing, for instance, $\delta_k = \frac{1}{\beta}$ in Equation (170) it implies that:

$$\frac{\alpha}{2} \|\boldsymbol{x}_{k+1} - \boldsymbol{\hat{x}}\| \leq \frac{\beta}{2} \left(\|\boldsymbol{x}_k - \boldsymbol{\hat{x}}\|^2 - \|\boldsymbol{x}_{k+1} - \boldsymbol{\hat{x}}\| \right)$$

and the following expression proves the statement in Equation (159):

$$\|\boldsymbol{x}_{n_{e}} - \hat{\boldsymbol{x}}\| \leq \left(\frac{\beta}{\eta + \beta}\right)^{\frac{n_{e}}{2}} \|\boldsymbol{x}_{0} - \hat{\boldsymbol{x}}\|$$
(171)

with $\rho = \sqrt{\frac{\beta}{\alpha + \beta}}$ and $0 < \rho \le 1$ since $\eta \le 0$.

Remark 34. As noted in Remark 82, $\frac{\beta}{\alpha}$ represents the conditioning number of the Hessian matrix $\kappa(\mathbf{H}_f)$. In this sense, Equation (171) can be rewritten as:

$$\|\boldsymbol{x}_{n_{e}} - \hat{\boldsymbol{x}}\| \leq \left(\frac{\kappa \left(\mathbf{H}_{f}\left(\boldsymbol{x}\right)\right)}{1 + \kappa \left(\mathbf{H}_{f}\left(\boldsymbol{x}\right)\right)}\right)^{\frac{n_{e}}{2}} \|\boldsymbol{x}_{0} - \hat{\boldsymbol{x}}\|$$
(172)

For large conditioning number, the convergence of the algorithm is harder and strongly dependent on the initial guess x_0 .

5 On the automatic differentiation

5.1 Backward propagation via adjoint gradient operator

In order to described how the Automatic Differentiation algorithm works, some basics must be recalled.

For any tensor field (i.e. multi-linear application) of order m + n, denoted as $\mathbb{T} : \mathbb{R}^n \times \mathbb{R}^m \mapsto \mathbb{R}$, with $\mathbb{T} \in \mathbb{R}^n \otimes \mathbb{R}^m$ and $\forall \boldsymbol{x} \in \mathbb{R}^m$, the derivative of the tensor field $\mathbb{T}(\cdot, \boldsymbol{x})$ along the vector $\boldsymbol{v} \in \mathbb{R}^m$ reads [For+15]:

$$D_{\boldsymbol{v}}\mathbb{T}(\cdot,\boldsymbol{x}) = \lim_{h \to 0} \frac{\mathbb{T}(\cdot,\boldsymbol{x}+h\boldsymbol{v}) - \mathbb{T}(\cdot,\boldsymbol{x})}{h}, \qquad h \in \mathbb{R}, \forall \boldsymbol{v} \in \mathbb{R}^m$$
(173)

If it exists, $D_{\boldsymbol{v}}\mathbb{T}(\cdot, \boldsymbol{x}) \in \mathbb{R}^n$ and the partial derivative along the i^{th} component \boldsymbol{e}_i of the orthonormal base \mathcal{B}_m reads [For+15]:

$$\frac{\partial \mathbb{T}(\cdot, \boldsymbol{x})}{\partial x_i} = D_{\boldsymbol{e}_i} \mathbb{T}(\cdot, \boldsymbol{x}), \qquad \forall \boldsymbol{x} \in \mathbb{R}^m$$
(174)

The gradient of $\mathbb{T}(\cdot, \mathbf{x})$ is defined as the operator $\nabla_x : \mathbb{R}^m \mapsto \mathbb{R}^n$ [For+15]:

$$\boldsymbol{\nabla}_{\boldsymbol{x}} \mathbb{T}(\cdot, \boldsymbol{x}) : \boldsymbol{v} \mapsto D_{\boldsymbol{v}} \mathbb{T}(\cdot, \boldsymbol{x})$$
(175)

The gradient is a tensor field of order m + n + 1 and a linear operator on v:

$$D_{a\boldsymbol{v}+b\boldsymbol{u}}\mathbb{T}(\cdot,\boldsymbol{x}) = \lim_{h \to 0} \frac{\mathbb{T}(\cdot,\boldsymbol{x}+h\cdot(a\boldsymbol{v}+b\boldsymbol{u})) - \mathbb{T}(\cdot,\boldsymbol{x})}{h} = a \cdot D_{\boldsymbol{v}}\mathbb{T}(\cdot,\boldsymbol{x}) + b \cdot D_{\boldsymbol{u}}\mathbb{T}(\cdot,\boldsymbol{x})$$
(176)

which implies that:

$$D_{\boldsymbol{v}}\mathbb{T}(\cdot,\boldsymbol{x}) = \sum_{i=1}^{m} \langle \boldsymbol{v}, \boldsymbol{e}_i \rangle D_{\boldsymbol{e}_i} \mathbb{T}(\cdot,\boldsymbol{x}) = \sum_{i=1}^{m} \langle \boldsymbol{v}, \boldsymbol{e}_i \rangle \frac{\partial \mathbb{T}(\cdot,\boldsymbol{x})}{\partial x_i} = \boldsymbol{\nabla}_{\boldsymbol{x}} \mathbb{T}.\boldsymbol{v},$$

$$\boldsymbol{\nabla}_{\boldsymbol{x}}\mathbb{T} = \sum_{i=1}^{m} \frac{\partial \mathbb{T}(\cdot,\boldsymbol{x})}{\partial x_i} \otimes \boldsymbol{e}_i$$
(177)

Since the gradient is a tensor $\nabla_x \mathbb{T}(\cdot, x)$ of order m + n + 1, it maps any vector $v \in \mathbb{R}^m$ into its dual space $\mathbb{R}^{m*} = \mathbb{R}^n$ and the following expression holds:

$$D_{\boldsymbol{v}}\mathbb{T}^*(\boldsymbol{u},\boldsymbol{x}) = \langle \boldsymbol{\nabla}_{\boldsymbol{x}}\mathbb{T}.\boldsymbol{u},\boldsymbol{v}\rangle \in \mathbb{R}, \qquad \forall \boldsymbol{v},\boldsymbol{x} \in \mathbb{R}^m, \forall \boldsymbol{u} \in \mathbb{R}^n \qquad (178)$$

In the back-propagation framework, the above mentioned basic notions related to the gradient of tensor fields can be applied to $\mathbb{T}(\cdot, \boldsymbol{x}) = L_{\mathcal{D}_{XY}}(\boldsymbol{\theta}) \in \mathbb{R}$, with $\boldsymbol{x} \equiv \boldsymbol{\theta} \in \mathbb{R}^m$. Moreover, the adjoint correspond to to transposition operator. The quest for a minimum is steered by the following well established Euler's inequality

Theorem 35. (Necessary condition, A convex set) Given any open set $\Omega \subset \mathbb{R}^n$, a convex subset $A \subset \Omega$ and a function $f : \Omega \to \mathbb{R}$, $f : \mathbf{x} \mapsto f(\mathbf{x})$ with $f \in C^1(\Omega)$, the necessary condition for $\bar{\mathbf{x}}$ to be a local minimum of f on Areads:

$$\langle \boldsymbol{\nabla}_x f\left(\bar{\boldsymbol{x}}\right), \boldsymbol{y} - \bar{\boldsymbol{x}} \rangle \ge 0, \qquad \forall \boldsymbol{y} \in A$$
(179)

Theorem 35 implies the iterative expression in Equation (79), provided a convex set Θ and a function $L_{\mathcal{D}_{XY}} \in \mathcal{C}^1(\Theta)$. The gradient of each weight θ_j is updated as in Equation (79):

$$\theta_{j}^{(i+1)} = \theta_{j}^{(i)} - \alpha \frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_{j}} \left(\theta_{j}^{(i)}\right) = \theta_{j}^{(i)} - \alpha \cdot D_{\boldsymbol{e}_{j}} L_{\mathcal{D}_{XY}}(1, \boldsymbol{\theta}^{(i)}) = \theta_{j}^{(i)} - \alpha \cdot \boldsymbol{\nabla}_{\boldsymbol{\theta}} L_{\mathcal{D}_{XY}} \cdot \boldsymbol{e}_{j}$$

$$\tag{180}$$

In particular, the Automatic Differentiation exploits the basics differential algebra presented above for composite non-linear functions of the type $L_{\mathcal{D}_{XY}}(\boldsymbol{\theta}) = L_{\mathcal{D}_{XY}} \circ l \circ g_i \circ a_j(\theta_k)$, adopting the chain rule in Equation (80). As a matter of fact, considering, without loss of generality, a composite function $f = h \circ g$ with $f : \mathbb{R}^m \mapsto \mathbb{R}$, $h : \mathbb{R}^m \mapsto \mathbb{R}$ and $g : \mathbb{R} \mapsto \mathbb{R}$, with $f : \boldsymbol{x} \mapsto f(\boldsymbol{x}), g : y \mapsto z = g(y)$ and $h : \boldsymbol{x} \mapsto y = h(\boldsymbol{x})$, the chain rule in Equation (80) applies to f, g and h according to the following expressions:

$$D_{\boldsymbol{x}}f(\boldsymbol{x}) = D_{\boldsymbol{y}}g(h(\boldsymbol{x})) \cdot D_{\boldsymbol{v}}h(\boldsymbol{x}) = \left(\frac{dg}{dy}\right)^* (h(\boldsymbol{x})) \cdot D_{\boldsymbol{v}}h(\boldsymbol{x})$$
(181)

Moreover, the adjoint derivative reads:

$$D_{\boldsymbol{v}}f^{*}(\boldsymbol{x}) = D_{\boldsymbol{v}}h^{*}(\boldsymbol{x}).D_{\boldsymbol{y}}g^{*}(h(\boldsymbol{x})) = D_{\boldsymbol{v}}h^{*}(\boldsymbol{x})\frac{dg}{dy}(h(\boldsymbol{x}))$$
(182)

Replacing Equation (182) in Equation (178), the gradient of f is immediately computed. It is worth noticing that, once the forward pass $L_{\mathcal{D}_{XY}}(\theta) = L_{\mathcal{D}_{XY}} \circ$ $l \circ g_i \circ a_j(\theta_k)$ is performed, the backward pass in Equation (182) is obtained with no further computation, since $D_{\boldsymbol{v}}h^*(\boldsymbol{x}) = D_{\boldsymbol{v}}h^T(\boldsymbol{x})$ and $\left(\frac{dg}{dy}\right)^* = \frac{dg}{dy}$. Moreover, each adjoint operation is local to the tensor itself, since it performed by simply inverting the graph constructed in the forward pass, i.e., by flipping the input-output connection. Therefore, the standard Automatic Differentiation implementation¹⁷ bundles the gradients of a tensor to the tensor itself, in order to reduce the memory consumption (no duplicate tensors) and exploit the same graph (see Figure 43).

¹⁷https://autodiff.github.io/

5.2 Counteract vanishing gradients with *ELU* and *SELU* activation functions

A valid alternative to ReLU is represented by Exponential Linear Unit [CUH16] ELU, depicted in Figure 9. The ELU function reads:

$$g(a) = ELU(a) = \begin{cases} a, a > 0 & (183) \\ \alpha \cdot (e^a - 1), a \le 0 & (184) \end{cases}$$

ELU is well known to prevent the so called *bias shift* [CUH16]. In order to explain the latter, the analogy between Empirical Loss function and negative log-likelihood exposed in Remark 22 must be retrieved. Provided this analogy, if one focuses on the k^{th} hidden layer, with $N_{\ell} \geq k > 1$ the derivative $\frac{\partial L_{\mathcal{D}_{XY}}}{\partial \theta_c^{(k)}}$ can be seen a the derivative of the negative log-likelihood $\frac{\partial \ln p_{\theta}(y|x)}{\partial \theta_c^{(k)}}$, for any label y in the database.

For the sake of clarity, let us consider layer the weights and biases of each of the $u^{(k+1)}$ neurons in the k^{th} layer, defined by the vector $\boldsymbol{\theta}^{(k)}$, that reads:

$$\boldsymbol{\theta}^{(k)} = \left[b^{(k)}, W_{11}^{(k)}, W_{12}^{(k)}, \dots, W_{1u^{(k)}}^{(k)}, W_{21}^{(k)}, \dots, W_{u^{(k+1)}u^{(k)}}^{(k)} \right] = \left[b^{(k)}, \boldsymbol{w}^{(k)} \right]$$
(185)

with a total amount of scalar weights/biases $n_{\theta}^{(k)} = u^{(k+1)} \cdot u^{(k)} + 1$. To access each entry of $\boldsymbol{\theta}^{(k)}$, a scalar indexing function *i* is defined as follows:

$$i(m,n) = 1 + n + u^{(k)} \cdot m, \quad 0 \le m \le u^{(k+1)}, \quad 0 \le n \le u^{(k)}$$
with
$$\theta_{i(0,0)}^{(k)} = b^{(k)}$$

$$\theta_{i(m,n)}^{(k)} = W_{mn}^{(k)}, \quad 1 \le m \le u^{(k+1)}, \quad 1 \le n \le u^{(k)}$$
(186)

considering Equation (97), Equation (104) and the derivative of the loss function with the respect to bias and weights expressed in Equation (99) and Equation (98), $\frac{\partial \ln p_{\theta}(y|\mathbf{X})}{\partial \theta_{i}^{(k)}}$ reads [CUH16]:

$$\frac{\partial \ln p_{\theta}\left(y|\boldsymbol{x}\right)}{\partial \theta_{i}^{(k)}} = \frac{\partial L_{\mathcal{D}_{XY}}\left(h_{\theta}\left(\boldsymbol{x}\right), y\right)}{\partial a_{i}^{(k)}} \cdot \frac{\partial a^{(k)}}{\partial \theta_{i}^{(k)}} = \delta(\boldsymbol{x}) \cdot \frac{\partial a^{(k)}}{\partial \theta_{i}^{(k)}}\left(\boldsymbol{x}\right)$$
(187)

 $\frac{\partial a^{(k)}}{\partial \theta_c^{(k)}}(\boldsymbol{x})$ is obtained by considering Equation (99) if $\theta_i^{(k)}$ is a bias and Equation (98) if $\theta_i^{(k)}$ is a weight:

$$\frac{\partial a^{(k)}}{\partial \theta^{(k)}_{c}} = \begin{cases} 1, & m = n = 0, \quad \theta^{(k)}_{i(0,0)} = b^{(k)} \\ b^{(k-1)} & 1 \le m \le w^{(k+1)}, \quad 1 \le m \le w^{(k)}, \quad 0^{(k)} \\ 0 \le 1, \dots, 0 \end{cases}$$
(188)

$$\partial \theta_c^{(\kappa)} = \left\{ h_n^{(k-1)}, \quad 1 \le m \le u^{(k+1)}, 1 \le n \le u^{(k)} \quad \theta_{i(m,n)}^{(k)} = W_{mn}^{(k)} \quad (189) \right\}$$

The Fisher Information Matrix (FIM, see Equation (24)) with the respect to the weights $\theta^{(k)}$ of the k^{th} neuron, reads:

$$\begin{pmatrix}
\begin{bmatrix}
\binom{k}{F}
\end{bmatrix}_{i_{r}i_{c}} = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\theta}} \begin{bmatrix}
\frac{\partial \ln p_{\theta}\left(Y|\boldsymbol{X}\right)}{\partial \theta_{i_{r}}^{(k)}} \cdot \frac{\partial \ln p_{\theta}\left(Y|\boldsymbol{X}\right)}{\partial \theta_{i_{c}}^{(k)}}\end{bmatrix} = \\
= \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\theta}} \begin{bmatrix}
\delta^{2}\left(\boldsymbol{X},Y\right) \cdot \frac{\partial a^{(k)}}{\partial \theta_{i_{r}}^{(k)}} \cdot \frac{\partial a^{(k)}}{\partial \theta_{i_{c}}^{(k)}}\end{bmatrix}$$
(190)

The Fisher information of two weights within the k^{th} layer depend on the value of $\delta^2(\boldsymbol{x}, y) \geq 0$, i.e., the probability of drawing samples (\boldsymbol{x}_i, y_i) is higher for higher values of $\delta(\boldsymbol{x}, y)$. The mean value in Equation (190) can be computed by observing that the samples y drawn via the MLP have a probability distribution $p_{\delta^2}(\boldsymbol{x}, y)$ that reads:

$$p_{\delta^{2}}(\boldsymbol{x}, y) = \frac{\delta(\boldsymbol{x}, y)^{2} \cdot p_{\theta}}{\int_{\mathcal{X} \times \mathcal{Y}} \delta(\boldsymbol{u}, u)^{2} \cdot p_{\theta} d\boldsymbol{u} d\boldsymbol{u}} = \frac{\delta(\boldsymbol{x}, y)^{2} \cdot p_{\theta}}{\mathbb{E}_{(\boldsymbol{x}, y) \sim p_{\theta}} \left[\delta(\boldsymbol{X}, Y)^{2}\right]}$$
(191)

Plugging Equation (191) in Equation (190), the latter can be rewritten as:

$$\left(\mathbb{I}_{F}^{(k)}\right)_{i_{r}i_{c}} = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\theta}}\left[\delta^{2}\left(\boldsymbol{X},Y\right)\right] \cdot \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\delta^{2}}}\left[\frac{\partial a^{(k)}}{\partial \theta_{i_{r}}^{(k)}} \cdot \frac{\partial a^{(k)}}{\partial \theta_{i_{c}}^{(k)}}\right]$$
(192)

The so called *bias shift* is defined as the 0^{th} column of the FIM, considering only the bias columns, as follows:

$$\boldsymbol{b}_{s} = \sum_{i_{r}=1}^{n_{\theta}^{(k)}} \left(\mathbb{I}_{F}^{(k)} \right)_{i_{r}i_{c}(0,0)} \boldsymbol{e}_{i_{r}} = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\theta}} \left[\delta^{2} \right] \cdot \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\delta^{2}}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}} a^{(k)} \right] = \\ = \mathbb{C}_{(\boldsymbol{x},y)\sim p_{\theta}} \left(\delta^{2}, \boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}} a^{(k)} \right) + \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\theta}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}} a^{(k)} \right] \cdot \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\theta}} \left[\delta^{2} \right]$$
(193)

 \boldsymbol{b}_s captures the statistical change of mean due the bias $b^{(k)}$ because of to the correlation among input and output units. As a matter of fact, this is confirmed by the fact that the bias shift drops to zero whenever $\mathbb{E}_{(x,y)\sim p_{\delta^2}}\left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}}a^{(k)}\right] = \mathbf{0}$ or, equivalently, whenever

$$\mathbb{C}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\theta}}}\left(\delta^{2}, \boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}}a^{(k)}\right) = -\mathbb{E}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\theta}}}\left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}}a^{(k)}\right] \cdot \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\theta}}}\left[\delta^{2}\right]$$
(194)

which implies that the derivative of the loss function at the output units δ and its gradient with the respect to the output of the input units are decorrelated [CUH16]. [CUH16] showed that, when computing the back-propagation (see solution (\mathcal{G})), one can mitigate the effect of the bias shift, by pre-multiplying the gradient $\nabla_{\boldsymbol{\theta}^{(k)}} L_{\mathcal{D}_{XY}}$ by the inverse of the FIM $\mathbb{I}_{F}^{(k)-1}$, obtaining a weight update vector $\boldsymbol{\Delta}\boldsymbol{\theta}^{(k)}$ that reads:

Gatti 317

$$\boldsymbol{\Delta w} = \left[\mathbb{I}_{F}^{(k)-1} \right]_{1:,1:} \left(\boldsymbol{\nabla}_{w} L_{\mathcal{D}_{XY}} - \Delta b^{(k)} \left[\boldsymbol{b}_{s} \right]_{1:} \right)$$
(195)

$$\Delta b^{(k)} = s \left(\frac{\partial L_{\mathcal{D}_{XY}}}{\partial b^{(k)}} - [\boldsymbol{b}_s]_{1:}^T \left[\mathbb{I}_F^{(k)-1} \right]_{1:1:} \boldsymbol{\nabla}_w L_{\mathcal{D}_{XY}} \right)$$
(196)

with

$$s = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\theta}}}^{-1} \left[\delta^{2} \right] + \\ + \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\theta}}}^{-1} \left[\delta^{2} \right] \cdot \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\delta}^{2}}}^{T} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}} a^{(k)} \right]$$

$$\cdot \mathbb{V}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\delta}^{2}}}^{-1} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}} a^{(k)} \right] \cdot \mathbb{E}_{(\boldsymbol{x},y)\sim p_{\boldsymbol{\delta}^{2}}} \left[\boldsymbol{\nabla}_{\boldsymbol{\theta}^{(k)}} a^{(k)} \right]$$

$$(197)$$

Whenever the bias shift drops to 0, the weight update in Equation (196) is equivalent to the standard update rule (see solution (\mathcal{G})). The correction of the bias shift strongly depends on the correlation of the incoming units which is captured by $\left[\mathbb{I}_{F}^{(k)}\right]_{11}^{-1}$ and it is equivalent to shift the mean activations of the incoming units toward zero (by a term $-\alpha \mathbb{E}_{(x,y) \sim p_{\delta^2}}\left[\nabla_{\boldsymbol{\theta}^{(k)}}a^{(k)}\right]$) and scaling up the bias unit (by a factor α) [CUH16]. The correction of the bias shift proposed in Equation (196) is effective but cumbersome to compute, since the FIM can become extremely large whenever the number of hidden neurons in the k^{th} layer is large. Therefore, [CUH16] proposed to adopt an *ELU* activation function to mitigate the bias shift instead. As a matter of fact, *ELU(a) < 0* for a < 0 and $\frac{\partial ELU(a)}{\partial a} \ge 0, \forall a$, saturating to 0 for $a \to -\infty$. The *ELU* units activate for a > 0, similarly to *ReLU*, but they keep active with a positive derivative even for a < 0 and negative small value that promote the bias shift correction toward zero mean and the bias scaling [CUH16]. Moreover, the *ELU* (as depicted in Figure 44) prevents the vanishing gradient at the same time.

An alternative to ELU is provided by the Scaled Exponential Linear Unit [Kla+17], SELU. The latter activation reads:

$$g(a) = SELU(a) = s\left(\max(0, a) + \min(0, \alpha (e^{a} - 1))\right)$$
(198)

and it is depicted, along with its derivative, in Figure 10. *SELU* has been conceived to prevent the occurrence of vanishing gradient phenomena by promoting the input/output self-centering of weights' means and variances towards 0 and 1 respectively, even in the presence of noise and perturbation, that could engender the overall gradient descent convergence. As a matter of fact, considering a generic \mathcal{MLP} layer ℓ , with pre-activation $\mathbf{a}^{(\ell)} = \sum_{j_1}^{u^{(\ell+1)}} a_j^{(\ell)} \mathbf{e}_j$, for the sake of simplicity one can assume that all the pre-activation units have the same average $\mathbb{E}[a_j] = \mu_a$, $1 \leq j \leq u^{(\ell+1)}$ and the same variance $\mathbb{V}[a_j] = \sigma_a^2$, $1 \leq j \leq u^{(\ell+1)}$. If one defines the mean and variance of the j^{th} activation

 $h_j(\ell) = SELU(a_j)$ respectively $\mu_h = \mathbb{E}[h_j]$ and $\sigma_h^2 = \mathbb{V}[h_j]$, then Figure 11 showed the effect of applying SELU on the mean and variance of each activation h_j [Kla+17]:

SELU makes the \mathcal{MLP} layer self-normalizing itself, since it a stable attracting fixed point solution, depending on the values of $w = \sum_{j_1}^{u^{(\ell+1)}} a_j^{(\ell)}$ and $v = \sum_{j_1}^{u^{(\ell+1)}} a_j^{(\ell)^2}$. Moreover, if the pre-activation mean and variance μ_a and σ_a^2 belong to a set $\mathcal{A} := \{(\mu, \sigma^2) | \mu \in [\mu_{\min}, \mu_{\max}], \sigma^2 \in [\sigma_{\min}^2, \sigma_{\max}^2]\}$, then $SELU(\mathcal{A}) \subseteq \mathcal{A}$, i.e., this inclusion is transitive across layers since $a_j^{(\ell)}$ is a function of $h_k^{(\ell-1)}$. SELU dampens the noise and perturbations which make w and v, recentering each neuron activation mean and variance.

A LTI, Fourier transform and convolution [RE-CAP]

This recap section is a summary of chapter II *Fourier Kingdom* in [Mal09], to which we remand for further details.

A linear time-invariant (LTI) filter is an operator equivariant with the respect to translation (or covariant):

$$\boldsymbol{y}(t) = \mathcal{L}(\boldsymbol{x}(t)) \iff \boldsymbol{y}(t-\tau) = \mathcal{L}(\boldsymbol{x}(t-\tau))$$
 (199)

with $\boldsymbol{y}(t) \in \mathbb{R}^n$. From now on, $\mathcal{L}(\cdot)$ is assumed to be hold weak continuity properties, so as to assure its stability against small deviations $\boldsymbol{x}(t) + \boldsymbol{\epsilon}$. If $\boldsymbol{x}(t) \in C^0(\mathbb{R}^n)$, then $\boldsymbol{y} \in C^0(\mathbb{R}^n)$ and they can be represented by:

$$\boldsymbol{x}(t) = \int_{\mathbb{R}} \boldsymbol{x}(\tau) \cdot \delta(t-\tau) dt$$

$$\boldsymbol{y}(t) = \mathcal{L}\left(\int_{\mathbb{R}} \boldsymbol{x}(\tau) \cdot \delta(t-\tau) dt\right) = \int_{\mathbb{R}} \sum_{k=1}^{n} x_{k}(\tau) \cdot \mathcal{L}\left(\delta(t-\tau)\boldsymbol{e}_{k}\right) dt$$
(200)

The term $\langle \mathcal{L}(\delta(t-\tau) \mathbf{e}_k), \mathbf{e}_h \rangle = H_{hk}(t-\tau)$ represents the impulse response of a LTI filter, whose components reads:

$$\boldsymbol{y}(t) = \int_{\mathbb{R}} \boldsymbol{H}(t-\tau) \boldsymbol{.} \boldsymbol{x}(\tau) d\tau = \int_{\mathbb{R}} \boldsymbol{H}(\tau) \boldsymbol{.} \boldsymbol{x}(t-\tau) d\tau$$
(201)

Therefore, the response $\boldsymbol{y}(t)$ is the *convolution* of the LTI response \boldsymbol{H} with the input signal $\boldsymbol{x}(t)$, denoted as $\boldsymbol{H} \star \boldsymbol{x}(t) = \boldsymbol{x} \star \boldsymbol{H}(t)$. A LTI filter implies the following properties:

• the convolution operator is commutative and translational covariant (see Section 3.2);
• the stability condition on Equation (201) implies that L is bounded if \boldsymbol{x} is bounded, i.e. if $\exists M \in \mathbb{R}^+$ such that $\int_{\mathbb{R}} \|\boldsymbol{H}(\tau)\| d\tau \leq M$ then:

$$\|\mathcal{L}(\boldsymbol{x}(t))\| \leq \int_{\mathbb{R}} \|\boldsymbol{H}(\tau)\| \cdot \|\boldsymbol{x}(t-\tau)\| d\tau \leq \sup_{t \in \mathbb{R}} \|\boldsymbol{x}(t)\| \cdot \int_{\mathbb{R}} \|\boldsymbol{H}(\tau)\| d\tau \leq \sup_{t \in \mathbb{R}} \|\boldsymbol{x}(t)\| \cdot M$$
(202)

• The response function can be written as :

$$\boldsymbol{H}(t) = \sum_{k=1}^{n} \boldsymbol{H}(t) \cdot \boldsymbol{e}_{k} \otimes \boldsymbol{e}_{k} = \sum_{k=1}^{n} \boldsymbol{h}_{k}(t) \otimes \boldsymbol{e}_{k}$$
(203)

• A filter is causal when $H_{hk}(t) < 0, \forall t < 0.$

Example 3. Some very well known LTI filters are:

• Amplification and delay:

$$\mathcal{L}(\boldsymbol{x}(t)) = \alpha \cdot \boldsymbol{x}(t-\tau), \quad \Rightarrow \quad \boldsymbol{H}(t) = \alpha \delta(t-\tau) \cdot \mathbf{Id}$$
 (204)

• Uniform moving average:

$$\mathcal{L}\left(\boldsymbol{x}\left(t\right)\right) = \frac{1}{T} \int_{t-\frac{T}{2}}^{t+\frac{T}{2}} \boldsymbol{x}\left(u\right) du = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} \boldsymbol{x}\left(t-u\right) du, \quad \Rightarrow \quad \boldsymbol{H}\left(t\right) = \frac{\chi\left[-\frac{T}{2};\frac{T}{2}\right]}{T} \mathbf{Id}$$
(205)
with $\chi_{t-\frac{T}{2}}$ = τ_{t} being the indicator function in the $\left[-\frac{T}{2};\frac{T}{2}\right]$ support

with $\chi_{\left[-\frac{T}{2};\frac{T}{2}\right]}$ being the indicator function in the $\left[-\frac{T}{2};\frac{T}{2}\right]$ support.

The LTI has a harmonic spectral decomposition, that relies on the following eigenvalue problem (see also Equation (203)):

$$\mathcal{L}\left(e^{i\omega t}\boldsymbol{e}_{k}\right) = \boldsymbol{h}_{k} \star e^{i\omega t} = \boldsymbol{\hat{h}}_{k}\left(\omega\right) \cdot e^{i\omega t}$$
(206)

where $e^{i\omega t}$ and $\hat{\mathbf{h}}_k$ are the eigenvector and eigenvalue of L, respectively. The linear operator has a harmonic decomposition, with $\hat{\mathbf{h}}_k(\omega)$ being the Fourier transform of $\mathbf{h}_k(t)$. The Fourier transform of a signal $\mathbf{x}(t) \in L^1(\mathbb{R})^{18}$, is defined as:

$$\mathcal{F}(\boldsymbol{x}(t)) = \hat{\boldsymbol{x}}(\omega) = \int_{\mathbb{R}} \boldsymbol{x}(u) \cdot e^{-i\omega u} du$$
(207)

If $\boldsymbol{x} \in L^1(\mathbb{R})$, then $\|\boldsymbol{\hat{x}}(\omega)\| \leq \int_{\mathbb{R}} \|\boldsymbol{x}(u)\| du < +\infty$. Moreover, since $f(t) = e^{-i\omega t} \in C^{\infty}(\mathbb{R})$ is continuous, if $\boldsymbol{x}(t) \in L^1(\mathbb{R})$, $\boldsymbol{\hat{x}}(\omega)$ is continuous, since the Fourier transform is a linear operator, which is bounded since $\|\boldsymbol{x}(t)e^{-i\omega t}\| \leq \|\boldsymbol{x}(t)\|$, so it's continuous.

¹⁸In \mathbb{R}^{n} , $\boldsymbol{x}\left(t\right)\in L^{1}\left(\mathbb{R}\right)$ iff

$$\int_{\mathbb{R}} \|\boldsymbol{x}\left(t\right)\| dt < +\infty$$

Example 4. Fourier transform of a Gaussian function.

A Gaussian function (such as the standard normal distributions) $p(u) = \frac{e^{-\frac{u^2}{2}}}{\sqrt{2\pi}}$, with $p(u) \in C^{\infty}(\mathbb{R})$ decaying very fast at infinite, has a Fourier transform that reads, by definition:

$$\hat{p}\left(\omega\right) = \frac{1}{\sqrt{2\pi}} \int_{\mathbb{R}} e^{-\frac{u^2}{2}} e^{-i\omega u} du$$

 $\hat{p}(0) = 1$ and $\hat{p}(\omega)$ can be differentiated, to obtain:

$$\frac{d\hat{p}}{d\omega}\left(\omega\right) = -\frac{i}{\sqrt{2\pi}} \int_{\mathbb{R}} u e^{-\frac{u^2}{2}} e^{-i\omega u} du$$

that integrated by part over \mathbb{R} provides the following differential equation solved by $\hat{p}(\omega)$:

$$\frac{d\hat{p}}{d\omega}\left(\omega\right) = -\frac{\omega}{\sqrt{2\pi}} \int_{\mathbb{R}} e^{-u^2} e^{-i\omega u} du = -\omega \cdot \hat{p}\left(\omega\right) \tag{208}$$

The solution of the simple ODE is $\hat{p}(\omega) = A \cdot e^{-\frac{\omega^2}{2}}$, with $\hat{p}(0) = A = 1$. Therefore, the Fourier transform of a Gaussian function is a Gaussian function too. With a similar approach, the function $x(t) = e^{-(a-ib)t^2}$ has a Fourier transform that reads

$$\hat{x}\left(\omega\right) = \sqrt{\frac{\pi}{a-ib}} e^{-\frac{-(a+ib)\omega^2}{4(a^2+b^2)}}$$

Therefore, if the LTI filter is stable, each eigenvalue reads:

$$\hat{\boldsymbol{h}}_{k}\left(\omega\right) = \int_{\mathbb{R}} \boldsymbol{h}\left(u\right) \cdot e^{-i\omega u} du < +\infty$$
(209)

and it represents the $L^1(\mathbb{R})$ -projection of $\boldsymbol{h}(t)$ on the harmonic of frequency ω . It follows that $\hat{\boldsymbol{H}}(\omega) = \sum_{k=1}^n \hat{\boldsymbol{h}}_k(\omega) \otimes \boldsymbol{e}_k$. If $\boldsymbol{x} \in L^1(\mathbb{R})$, its inverse $\hat{\boldsymbol{x}} \in L^1(\mathbb{R})$ is $\in L^1(\mathbb{R})$ too¹⁹ and it reads:

$$\boldsymbol{x}\left(t\right) = \frac{1}{2\pi} \int_{\mathbb{R}} \boldsymbol{\hat{x}}\left(\omega\right) \cdot e^{i\omega t} d\omega$$
(210)

¹⁹Adopting the definition of Fourier transform in Equation (207), the inverse Fourier transform can be written as $F^{-1}(\hat{x}(\omega)) = \frac{1}{2\pi} \int_{\mathbb{R}} \left(\int_{\mathbb{R}} \boldsymbol{x}(u) e^{i\omega(t-u)} du \right) d\omega$. However, $\int_{\mathbb{R}^2} \boldsymbol{x}(u) e^{i\omega(t-u)} du d\omega$ is not finite, so the Fubini's theorem - required to prove that the inverse Fourier transform is finite - does not apply directly. It is first necessary to multiply the integrand by a Gaussian kernel $\hat{g}_{\varepsilon}(\omega) = e^{-\frac{\varepsilon^2 \omega^2}{4}}$ - of the family $\mathcal{N}\left(0, \frac{2}{\varepsilon}\right)$, that converges to 1 for $\varepsilon \to 0$ - so to obtain the function: $\phi_{\varepsilon}(t) = \frac{1}{2\pi} \int_{\mathbb{R}} \int_{\mathbb{R}} \boldsymbol{x}(u) \hat{g}_{\varepsilon}(\omega) e^{i\omega(t-u)} du d\omega = \frac{1}{2\pi} \int_{\mathbb{R}} \hat{\boldsymbol{x}}(\omega) \hat{g}_{\varepsilon}(\omega) e^{i\omega t}$. In this case, $\|\hat{\boldsymbol{x}}(\omega) \hat{g}_{\varepsilon}(\omega) e^{i\omega t}\| \leq \|\hat{\boldsymbol{x}}(\omega)\|$, since the Gaussian kernel decays fast and $\hat{\boldsymbol{x}}(\omega) \hat{g}_{\varepsilon}(\omega) e^{i\omega t}$ converges to $\hat{\boldsymbol{x}}(\omega) e^{i\omega t}$ for $\varepsilon \to 0$. This implies that - by the theorem of dominated convergence - $\lim_{\varepsilon \to 0} \phi_{\varepsilon}(t) = \frac{1}{2\pi} \int_{\mathbb{R}} \hat{\boldsymbol{x}}(\omega) e^{i\omega t}$. Moreover, as shown

Equation (210) proves that the $\boldsymbol{x}(t)$ can be decomposed on a harmonic basis $\{e^{i\omega t}\}_{\omega\in\mathbb{R}}$. Again, since $f(t) = e^{i\omega t} \in C^{\infty}(\mathbb{R})$ is continuous, if $\hat{\boldsymbol{x}}(\omega) \in L^1(\mathbb{R})$, $\boldsymbol{x}(\omega)$ is continuous, since the inverse Fourier transform in Equation (210) is a linear operator, which is bounded since $\|\hat{\boldsymbol{x}}(t)e^{i\omega t}\| \leq \|\hat{\boldsymbol{x}}(t)\|$, so it's continuous. For discontinuous functions, the harmonic reconstruction in Equation (210) is not proved.

When both $\boldsymbol{x}(t), \boldsymbol{H}(t) \in L^{1}(\mathbb{R})$, the convolution in the Fourier domain becomes a contraction (or a product)²⁰:

$$\boldsymbol{y}(t) = \boldsymbol{H} \star \boldsymbol{x}(t) \iff \mathcal{F}(\boldsymbol{y}(t)) = \boldsymbol{\hat{y}}(\omega) = \boldsymbol{\hat{H}}(\omega) \cdot \boldsymbol{\hat{x}}(\omega) = \mathcal{F}(\boldsymbol{H}(t)) \cdot \mathcal{F}(\boldsymbol{x}(t))$$
(211)

This implies that, the response to a LTI filter is simply written as:

$$\boldsymbol{y}(t) = \frac{1}{2\pi} \int_{\mathbb{R}} \hat{\boldsymbol{H}}(\omega) \cdot \hat{\boldsymbol{x}}(\omega) e^{i\omega t} d\omega$$
(212)

Example 5. Fourier transform of a discontinuous functions

• The Fourier transform of the rectangle function $\boldsymbol{x}(t) = \frac{1}{T}\chi_{\left[-\frac{T}{2},\frac{T}{2}\right]}\boldsymbol{e}_{i}$

in Example 4, $\hat{g}_{\varepsilon}\left(\omega\right)$ is the Fourier transform of the function

$$\frac{1}{2\pi} \int_{\mathbb{R}} \hat{g}_{\varepsilon} \left(\omega \right) e^{i\omega t} d\omega = \frac{1}{\varepsilon \sqrt{\pi}} e^{-\frac{t^2}{\varepsilon^2}} = \frac{\sqrt{2}}{\varepsilon} g_1 \left(\frac{\sqrt{2}}{\varepsilon} t \right)$$

which is a Gaussian kernel that approximates the Dirac delta for $\varepsilon \to 0$, keeping the integral over \mathbb{R} equal to 1. This allows to rewrite $\phi_{\varepsilon}(t) = \int_{\mathbb{R}} x(u) \frac{\sqrt{2}}{\varepsilon} g_1\left(\frac{\sqrt{2}}{\varepsilon}(t-u)\right) du$. Therefore, the following expression concludes:

$$\begin{split} \lim_{\varepsilon \to 0} \int_{\mathbb{R}} \|\phi_{\varepsilon}\left(t\right) - \boldsymbol{x}\left(t\right)\|dt &= \int_{\mathbb{R}} \lim_{\varepsilon \to 0} \|\int_{\mathbb{R}} \frac{\boldsymbol{x}\left(u\right)}{\varepsilon \sqrt{\pi}} e^{-\frac{(t-u)^2}{\varepsilon^2}} du - \boldsymbol{x}\left(t\right)\|dt\\ \lim_{\varepsilon \to 0} \int_{\mathbb{R}} \|\phi_{\varepsilon}\left(t\right) - \boldsymbol{x}\left(t\right)\|dt &= \int_{\mathbb{R}} \|\lim_{\varepsilon \to 0} \phi_{\varepsilon}\left(t\right) - \boldsymbol{x}\left(t\right)\|dt = \int_{\mathbb{R}} \|\int_{\mathbb{R}} \boldsymbol{x}\left(u\right)\delta\left(t-u\right)du - \boldsymbol{x}\left(t\right)\|dt = 0\\ \lim_{\varepsilon \to 0} \phi_{\varepsilon}\left(t\right) &= \frac{1}{2\pi} \int_{\mathbb{R}} \hat{\boldsymbol{x}}\left(\omega\right)e^{i\omega t}d\omega = \boldsymbol{x}\left(t\right) \end{split}$$

 $^{20}\mathrm{As}$ a matter of fact, the Fourier transform of $\boldsymbol{y}\left(t\right)$ can be expanded as:

$$\hat{\boldsymbol{y}}\left(\omega\right) = \int_{\mathbb{R}} \int_{\mathbb{R}} \boldsymbol{H}\left(u\right) \cdot \boldsymbol{x}\left(t-u\right) e^{-i\omega t} du dt$$

This expression can be integrated by means of the Fubini's theorem because $\|\pmb{H}(t-u).\pmb{x}(u)\|\in L^1(\mathbb{R}^2)$ so

$$\hat{\boldsymbol{y}}\left(\omega\right) = \int_{\mathbb{R}^{2}} \boldsymbol{H}\left(v\right) . \boldsymbol{x}\left(u\right) e^{i\omega\left(u+v\right)} du dv = \hat{\boldsymbol{H}}\left(\omega\right) . \hat{\boldsymbol{x}}\left(\omega\right)$$

reads:

$$\hat{\boldsymbol{x}}(\omega) = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} e^{-i\omega t} dt \boldsymbol{e}_i = \frac{\sin\left(\frac{T\omega}{2}\right)}{\frac{T\omega}{2}} \boldsymbol{e}_i = \operatorname{sinc}\left(\frac{T\omega}{2}\right) \boldsymbol{e}_i \qquad (213)$$

• The Fourier transform of the Heaviside function $\boldsymbol{x}(t) = H(t - \tau) \boldsymbol{e}_i$:

$$\hat{\boldsymbol{x}}\left(\omega\right) = \int_{\tau}^{+\infty} e^{-i\omega t} dt = \frac{e^{-i\omega\tau}}{i\omega}$$
(214)

• The Fourier transform of the function $\boldsymbol{x}\left(t\right) = t \cdot H\left(t - \tau\right) \boldsymbol{e}_{i}$:

$$\hat{\boldsymbol{x}}(\omega) = \int_{\tau}^{+\infty} t e^{-i\omega t} dt \boldsymbol{e}_i = -e^{-i\omega \tau} \left(\frac{i\tau}{\omega} + \frac{1}{\omega^2}\right) \boldsymbol{e}_i \qquad (215)$$

The famous ReLU function can be written as $ReLU(t)=t\cdot H(t),$ whose Fourier transform is $ReLU(t)=-\frac{1}{\omega^2}$

• The Fourier transform of the symmetric triangle function

$$\boldsymbol{x}(t) = \left(H(-t) \cdot ReLU\left(t + \frac{T}{2}\right) + H(t) \cdot ReLU\left(-t + \frac{T}{2}\right)\right) \boldsymbol{e}_i$$

with $T \ge 0$ reads:

$$\hat{\boldsymbol{x}}(\omega) = \boldsymbol{e}_i \left(\int_{-\frac{T}{2}}^0 \left(t + \frac{T}{2} \right) e^{-i\omega t} dt + \int_0^{\frac{T}{2}} \left(\frac{T}{2} - t \right) e^{-i\omega t} dt \right)$$

By integrating by parts, one obtains:

$$\hat{\boldsymbol{x}}(\omega) = \frac{e^{-i\omega\frac{T}{2}} + e^{i\omega\frac{T}{2}} - 2}{\left(i\omega\right)^2} = T^2 \frac{\left(e^{-i\omega\frac{T}{2}} + e^{i\omega\frac{T}{2}}\right)^2}{\left(2i\omega\frac{T}{2}\right)^2} = T^2 \operatorname{sinc}^2\left(\frac{T\omega}{2}\right)$$
(216)

One can notice that the triangle function can be written as a convolution between two rectangular functions, as:

$$\boldsymbol{x}\left(t\right) = \chi_{\left[-\frac{T}{2},\frac{T}{2}\right]} \star \chi_{\left[-\frac{T}{2},\frac{T}{2}\right]}\boldsymbol{e}_{i}$$

This expression eases the computation of the Fourier transform, because of the convolution properties.

• The Fourier transform of the Dirac delta $\delta_{\tau}(t) = \delta(t - \tau)$ reads:

$$\hat{\delta}_{\tau}\left(\omega\right) = \int_{\mathbb{R}} \delta\left(t - \tau\right) e^{-i\omega t} d\tau = e^{-i\omega \tau}$$
(217)

and the Fourier transform of the Dirac comb

$$c_T(t) = \sum_{n \in \mathbb{Z}} \delta(t - nT)$$
(218)

reads:

$$\hat{c}_T(\omega) = \int_{\mathbb{R}} \sum_{n \in \mathbb{Z}} \delta(t - nT) e^{-i\omega t} d\tau = \sum_{n \in \mathbb{Z}} e^{-i\omega nT}$$
(219)

 $\hat{c}_T(\omega)$ is periodic of period $\frac{2\pi}{T}^{21}$. Moreover, the following theorem proves that $\hat{c}_T(\omega)$ can be rewritten as a Dirac comb in frequency domain (see Theorem 37).

In order to extend Fourier theory to discontinuous functions so to leverage its nice features, the continuity condition can be slightly released, focusing on functions defined over the Hilbert space $L^2(\mathbb{R})$ (finite energy), a complete space of functions endowed with a scalar product defined by²²:

$$\boldsymbol{x}(t), \boldsymbol{y}(t) \in L^{2}(\mathbb{R}) \Rightarrow \langle \boldsymbol{x}(t), \boldsymbol{y}(t) \rangle_{L^{2}(\mathbb{R})} = \int_{\mathbb{R}} \langle \boldsymbol{x}(u), \boldsymbol{y}^{*}(u) \rangle du$$

$$\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})} < +\infty; \quad \|\boldsymbol{y}(t)\|_{L^{2}(\mathbb{R})} < +\infty$$
(220)

All functions $\boldsymbol{x}(t), \boldsymbol{y}(t) \in L^{1}(\mathbb{R}) \cap L^{2}(\mathbb{R})$ inherits the following fundamental property:

Theorem 36. Parseval's and Plancherel's theorem

 $\forall \boldsymbol{x}(t), \boldsymbol{y}(t) \in L^{1}(\mathbb{R}) \cap L^{2}(\mathbb{R}) \Rightarrow Parseval formula:$

$$\langle \boldsymbol{x}(t), \boldsymbol{y}(t) \rangle_{L^{2}(\mathbb{R})} = \frac{1}{2\pi} \langle \hat{\boldsymbol{x}}(\omega), \hat{\boldsymbol{y}}(\omega) \rangle_{L^{2}(\mathbb{R})}$$
 (221)

If $x \equiv y \Rightarrow$ Plancherel formula:

$$\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2} = \frac{1}{2\pi} \|\boldsymbol{\hat{x}}(\omega)\|_{L^{2}(\mathbb{R})}^{2}$$
(222)

Proof. Considering $\boldsymbol{z}(t) = \boldsymbol{x}(t) \star \boldsymbol{y}^*(-t)$, then $\boldsymbol{z}(t) \in L^1(\mathbb{R}) \cap L^2(\mathbb{R})$ and $\boldsymbol{\hat{z}}(\omega) = \boldsymbol{\hat{x}}(\omega) \cdot \boldsymbol{\hat{y}}^*(\omega)^{23}$ with

$$\langle \boldsymbol{x} (t), \boldsymbol{y} (t) \rangle_{L^{2}(\mathbb{R})} = \boldsymbol{z}(0) = \int_{\mathbb{R}} \hat{\boldsymbol{z}} (\omega) d\omega =$$
$$= \int_{\mathbb{R}} \langle \hat{\boldsymbol{x}} (\omega), \boldsymbol{y}^{*} (\omega) \rangle d\omega = \langle \hat{\boldsymbol{x}} (\omega), \hat{\boldsymbol{y}} (\omega) \rangle_{L^{2}(\mathbb{R})}$$

.

 ${}^{21}e^{-i\omega nT}\cdot 1 = e^{-i\omega nT}\cdot e^{-i2\pi n} = e^{-i\left(\omega + \frac{2\pi}{T}\right)nT}$

 22 where * stands for complex conjugate

$${}^{23}\mathcal{F}(\boldsymbol{y}^{*}(-t)) = \int_{-\infty}^{+\infty} \boldsymbol{y}^{*}(-u) e^{-i\omega u} du = \left(-\int_{+\infty}^{-\infty} \boldsymbol{y}(-u) e^{(-i\omega(-u))} d(-u)\right)^{*} = \boldsymbol{y}^{*}(\omega)$$

For discontinuous functions $\boldsymbol{x}(t) \in L^2(\mathbb{R})$ but $\notin L^1(\mathbb{R})$, the inverse Fourier transform is computed as a limit of a suite of functions $\{\boldsymbol{x}_n(t)\}_{n\in\mathbb{Z}} \in L^1(\mathbb{R}) \cap L^2(\mathbb{R})^{24}$. An example is represented by the Dirac's comb, according to the following theorem.

Theorem 37. Poisson formula

Given two distributions $d_1(\omega)$ and $d_2(\omega)$ defined as:

$$d_1(\omega) = \sum_{n \in \mathbb{Z}} e^{-i\omega nT}, \quad d_2(\omega) = \frac{2\pi}{T} \sum_{n \in \mathbb{Z}} \delta\left(\omega - \frac{2\pi n}{T}\right)$$

 d_1 and d_2 are equal in the sense of distributions (or in a weak formulation), *i.e.*:

$$\int_{\mathbb{R}} d_1(\omega) \cdot \hat{\phi}(\omega) \, d\omega = \int_{\mathbb{R}} d_2(\omega) \cdot \hat{\phi}(\omega) \, d\omega, \quad \forall \phi \in C_0^{\infty}(\mathbb{R})$$

Proof. $d_1(t)$ is periodic of period $\frac{2\pi}{T}$, which implies that we can prove the theorem over the compact support $\left[-\frac{\pi}{T}, \frac{\pi}{T}\right]$, i.e.:

$$\int_{-\frac{\pi}{T}}^{\frac{\pi}{T}} \sum_{n \in \mathbb{Z}} e^{-i\omega nT} \cdot \hat{\phi}(\omega) \, d\omega = \int_{-\frac{\pi}{T}}^{\frac{\pi}{T}} \frac{2\pi}{T} \delta(\omega) \cdot \hat{\phi}(\omega) \, d\omega = \frac{2\pi}{T} \hat{\phi}(0), \quad \forall \phi \in C_0^{\infty}(\mathbb{R})$$
(223)

If one consider the truncated geometric series:

n

$$\sum_{n=-N}^{N} e^{-i\omega nT} = \sum_{n=0}^{N} \left(e^{-i\omega T} \right)^n + \sum_{n=0}^{N-1} \left(e^{i\omega T} \right)^{n+1}$$

with $|e^{i\omega T}| = |e^{-i\omega T}| \le 1$ over $\left[-\frac{\pi}{T}, \frac{\pi}{T}\right]$, the sum is equal to²⁵:

$$\sum_{n=-N}^{N} e^{-i\omega nT} = \frac{e^{i\omega\frac{T}{2}} + e^{-i\omega\left(N + \frac{1}{2}\right)T}}{e^{i\omega\frac{T}{2}} - e^{-i\omega\frac{T}{2}}} + \frac{e^{i\omega\frac{T}{2}} + e^{i\omega\left(N + \frac{1}{2}\right)T}}{e^{-i\omega\frac{T}{2}} - e^{i\omega\frac{T}{2}}}$$

Recalling that $2i\sin(\alpha) = (e^{i\alpha} - e^{-i\alpha})$ the truncated series results into:

$$\sum_{n=-N}^{N} e^{-i\omega nT} = \frac{\sin\left(T\omega\left(N+\frac{1}{2}\right)\right)}{\sin\left(\frac{T\omega}{2}\right)}$$

$${}^{25}\sum_{k=0}^{N} q^k = \frac{1+q^{N+1}}{1-q}$$

 $[\]begin{array}{c} \hline & 2^{4} \text{As a matter of fact } L^{1}(\mathbb{R}) \cap L^{2}\left(\mathbb{R}\right) \text{ is dense in } L^{2}\left(\mathbb{R}\right), \text{ which is a Hilbert space, so complete.} \\ \text{The suite of functions converges } \lim_{n \to +\infty} \| x_{n} - x \|_{L^{2}(\mathbb{R})} \text{ and this suite of function is a Cauchy's sequence } (\forall \varepsilon > 0, \exists M \in \mathbb{N} \text{ such that } \| x_{n} - x_{m} \|_{L^{2}(\mathbb{R})} < \varepsilon, \quad \forall n, m \geq M). \text{ Since the } x_{n} \in L^{1}\left(\mathbb{R}\right), \text{ their inverse Fourier transform exists and it is noted } \hat{x}_{n}\left(\omega\right). \text{ Therefore, for the Plancherel formula in Equation (222), it holds that } \| x_{n} - x_{m} \|_{L^{2}(\mathbb{R})}^{2} = \frac{1}{2\pi} \| \hat{x}_{n} - \hat{x}_{m} \|_{L^{2}(\mathbb{R})}^{2} \\ \text{ and therefore } \{ \hat{x}_{n} \}_{n \in \mathbb{Z}} \text{ is a Cauchy sequence too and (being } L^{2}\left(\mathbb{R}\right) \text{ complete}), \text{ it converges to } \\ \hat{x}\left(\omega\right) \in L^{2}\left(\mathbb{R}\right) \text{ which is the Fourier transform of } x\left(t\right). \text{ Plancherel's, Parseval's and properties of } L^{1}\left(\mathbb{R}\right) \text{ functions apply to } L^{2}\left(\mathbb{R}\right) \text{ too.} \\ \hline = \sum_{n=1}^{\infty} \sum_{k=1}^{N-1} |x_{n}|^{k+1}|^{k+1} \\ \hline = \sum_{k=1}^{N-1} \sum_{k=1}^{N-1} |x_{n}|^{k+1}|^{k+1} \\ \hline = \sum_{k=1}^{N-1} \sum_{k=1}^{N-1} |x_{n}|^{k+1}|^{k+1} \\ \hline = \sum_{k=1}^{N-1} \sum_{k=1}^{N-1} \sum_{k=1}^{N-1} \sum_{k=1}^{N-1} |x_{n}|^{k+1}|^{k+1} \\ \hline = \sum_{k=1}^{N-1} \sum_$

Therefore, Equation (223) can be rewritten as:

$$\lim_{N \to +\infty} \int_{-\frac{\pi}{T}}^{\frac{\pi}{T}} \sum_{n=-N}^{N} e^{-i\omega nT} \cdot \hat{\phi}(\omega) \, d\omega =$$

= $\frac{2\pi}{T} \lim_{N \to +\infty} \int_{\mathbb{R}} \hat{\psi}(\omega) \cdot \hat{\xi}_N(\omega) \, d\omega$ (224)

with

$$\hat{\psi}(\omega) = \frac{\chi_{\left[-\frac{\pi}{T},\frac{\pi}{T}\right]}(\omega) \cdot \hat{\phi}(\omega)}{\operatorname{sinc}\left(\frac{T\omega}{2}\right)}$$

and

$$\hat{\xi}_N(\omega) = \frac{\sin\left(T\omega\left(N + \frac{1}{2}\right)\right)}{\pi\omega} = \operatorname{sinc}\left(T\omega\left(N + \frac{1}{2}\right)\right)\frac{T\left(N + \frac{1}{2}\right)}{\pi}$$

Recalling that the inverse Fourier transform of the sinc function is the rectangle function (see Equation (213)), therefore

$$\frac{1}{2\pi} \int_{\mathbb{R}} \hat{\xi}_N(\omega) e^{i\omega t} d\omega = \frac{\chi[T(N+\frac{1}{2}), T(N+\frac{1}{2})]}{2\pi}$$

Thanks to the Parseval formula in Equation (221), Equation (224) becomes:

$$\frac{2\pi}{T}\lim_{N\to+\infty}\int_{\mathbb{R}}\hat{\psi}(\omega)\cdot\hat{\xi}_{N}(\omega)\,d\omega = \frac{4\pi^{2}}{T}\lim_{N\to+\infty}\int_{\mathbb{R}}\psi(t)\cdot\xi_{N}(t)\,dt = \\
=\frac{2\pi}{T}\lim_{N\to+\infty}\int_{T\left(N+\frac{1}{2}\right)}^{T\left(N+\frac{1}{2}\right)}\psi(t)\,dt = \frac{2\pi}{T}\hat{\psi}(0) = \int_{\mathbb{R}}\frac{2\pi}{T}\delta(u)\,\hat{\psi}(u)\,du$$
(225)

which proves the statement.

The Poisson formula is rather useful for discrete signals. Discrete digital signals over N points are defined through convolution with a uniform moving average function $s_N\left(\frac{n}{N}\right)$ of the type Equation (205), as follows:

$$\boldsymbol{x}_{N}[n] = s_{N}\left(\frac{n}{N}\right) \mathbf{Id} \star \boldsymbol{x}(t)$$
 (226)

In particular, recalling Equation (205), $s_N\left(\frac{n}{N}\right)$ samples $\boldsymbol{x}(t)$ with uniform interval $T = \frac{1}{N}$:

$$s_N(t) = \chi_{\left[0,\frac{1}{N}\right]}(t) \tag{227}$$

which implies that :

$$\boldsymbol{x}_{N}[n] = \int_{0}^{\frac{1}{N}} \boldsymbol{x}\left(\frac{n}{N} - u\right) du = \int_{\frac{n-1}{N}}^{\frac{n}{N}} \boldsymbol{x}\left(y\right) dy$$
(228)

If $\boldsymbol{x}(t) \in L^{1}(\mathbb{R})$, then, by Equation (226), the Fourier transform of a discrete signal reads:

$$\hat{\boldsymbol{x}}_{N}(\omega) = \hat{s}_{N}(\omega) \cdot \hat{\boldsymbol{x}}(\omega) = \frac{e^{\frac{\omega}{2N}}}{N} \operatorname{sinc}\left(\frac{\omega}{2N}\right) \cdot \hat{\boldsymbol{x}}(\omega)$$
(229)

Equation (229) proves that:

$$\hat{s}(\omega) = \frac{e^{\frac{i\omega}{2N}}}{N} \operatorname{sinc}\left(\frac{\omega}{2N}\right)$$
(230)

Analogously, if one considers the a sampling function with vanishing high frequencies, such as $\hat{s}_N(\omega) = \frac{1}{N}\chi_{\left[-\frac{N}{2},\frac{N}{2}\right]}$, since $\hat{s}_N(\omega) \in L^1 \cap L^2$, according to Equation (213):

$$s_N(t) = \operatorname{sinc}\left(\frac{tN}{2}\right) \tag{231}$$

Equation (231) paves the way to the following fundamental theorem.

Theorem 38. Nyquist-Shannon sampling theorem

Consider the space S_N of functions $L^2(\mathbb{R})$ whose Fourier coefficients vanish at high frequencies:

$$\mathcal{S}_{N} := \left\{ f\left(t\right) \in L^{2}\left(\mathbb{R}\right) | supp\left(\hat{f}\left(\omega\right)\right) \in \left[-\frac{N}{2}, \frac{N}{2}\right] \right\}$$

Then $\forall \boldsymbol{x} \in \mathcal{S}_N^n$:

$$\boldsymbol{x}(t) = \sum_{n \in \mathbb{Z}} \boldsymbol{x}\left(\frac{2\pi n}{N}\right) \operatorname{sinc}\left(\frac{N}{2}\left(t - \frac{2\pi n}{N}\right)\right)$$

Proof. $s_N(t) = \operatorname{sinc}\left(\frac{tN}{2}\right) \in \mathcal{S}_N$ corresponds to a rectangle function $\frac{\chi\left[-\frac{N}{2},\frac{N}{2}\right]^{(\omega)}}{N}$ in the Fourier domain. Moreover, $\boldsymbol{x}_N(t-nT) = \boldsymbol{x} \star s_N(t-nT) \operatorname{\mathbf{Id}}$, with a phase shift in the corresponding Fourier transform:

$$\mathcal{F}\left(s_{N}\left(t-nT\right)\right) = \hat{s}_{N}\left(\omega\right)e^{-i\omega nT} = \frac{\chi\left[-\frac{N}{2},\frac{N}{2}\right]\left(\omega\right)}{N}e^{-i\omega nT}$$

Therefore, on the support $\left[-\frac{N}{2}, \frac{N}{2}\right]$, $\left\{\hat{s}_N(\omega) e^{-i\omega nT}\right\}_{n \in \mathbb{Z}}$ is a orthonormal basis. Therefore, the Fourier transform of the functions $f \in S_N$ can be decomposed on this basis:

$$\hat{f}(\omega) = \hat{s}_N(\omega) \sum_{k \in \mathbb{Z}} \hat{f}_k \cdot e^{-i\omega kT}, \quad \hat{f}_k = \int_{-\frac{N}{2}}^{\frac{N}{2}} \hat{f}(\omega) \cdot e^{i\omega t} d\omega = f \star \operatorname{sinc}\left(\frac{kTN}{2}\right)$$

Thanks to the inverse Fourier formula in Equation (210),

$$f(t) = \sum_{k \in \mathbb{Z}} \hat{f}_k \cdot s_N \left(t - kT \right)$$

It follows that $\forall x \in \mathcal{S}_N^n$:

$$\boldsymbol{x}(t) = \sum_{n \in \mathbb{Z}} \hat{\boldsymbol{x}}_n \cdot \boldsymbol{s}_N (t - nT)$$

Since $s_N(mT - nT) = \delta_{m,n}$, $\hat{\boldsymbol{x}}_n = \boldsymbol{x}(nT)$. Finally, taking $T = \frac{2\pi}{N}$ proves the statement.

Remark 39. As a matter of fact, the Theorem 38 proves that a function with a compact support in the frequency domain can be approximated with a infinite number of cardinal sinuses. On the contrary, we could be tempted to the same thing in the inverse order, i.e., to interpolate a signal with a compact support in time, i.e., discontinuous functions in time. This is the case of the finite element method or, more in general, the spline approximation problem. The problem resides in the fact that, in analogy with Theorem 38, one needs an infinite amount of cardinal sinuses in the frequency domain to reconstruct this signal.

The continuity of a function is intimately related to the decay of its Fourier's coefficients. The following theorem [Mal09] helps understanding this point:

Theorem 40. A function $\boldsymbol{x} : \mathbb{R} \to \mathbb{R}^n$ is bounded and belongs to $\in C^p(\mathbb{R})$ if:

$$\int_{\mathbb{R}} \|\hat{\boldsymbol{x}}(\omega)\| (1+|\omega|^p) \, d\omega < +\infty \tag{232}$$

Proof. If $\hat{\boldsymbol{x}} \in L^1(\mathbb{R})$, the $\frac{d^k \boldsymbol{x}}{dt^k}$ corresponds to $(i\omega)^p \, \hat{\boldsymbol{x}}(\omega)$, since:

$$\frac{d^{p}\boldsymbol{x}}{dt^{p}}\left(t\right) = \frac{1}{2\pi} \int_{\mathbb{R}} (i\omega)^{p} \boldsymbol{\hat{x}}\left(\omega\right) e^{i\omega t} d\omega$$

with:

$$\|\boldsymbol{x}(t)\| \leq \frac{1}{2\pi} \int_{\mathbb{R}} \|\boldsymbol{\hat{x}}(\omega)\| |e^{i\omega t}| d\omega \leq \frac{1}{2\pi} \int_{\mathbb{R}} \|\boldsymbol{\hat{x}}(\omega)\| d\omega < +\infty$$

This inequality applies to all the derivatives of order $k \leq p$, under a condition:

$$\left\|\frac{d^{p}\boldsymbol{x}}{dt^{p}}\left(t\right)\right\| \leq \frac{1}{2\pi} \int_{\mathbb{R}} |\omega|^{p} \cdot \left\|\boldsymbol{\hat{x}}\left(\omega\right)\right\| d\omega \leq \frac{1}{2\pi} \int_{\mathbb{R}} \left(1 + |\omega|^{p}\right) \cdot \left\|\boldsymbol{\hat{x}}\left(\omega\right)\right\| d\omega$$

If $\frac{1}{2\pi} \int_{\mathbb{R}} (1 + |\omega|^p) \cdot \|\hat{\boldsymbol{x}}(\omega)\| d\omega$ is bounded by assumption, then all derivatives of order $k \leq p$ are bounded, which implies that $\boldsymbol{x} \in C^p(\mathbb{R})$

Theorem 40 proves that $\exists M>0$ such that , $\forall \varepsilon>0$

$$\|\hat{\boldsymbol{x}}(\omega)\| \le \frac{M}{1+|\omega|^{p+1+\varepsilon}}$$
(233)

then $\boldsymbol{x} \in C^p(\mathbb{R})$. If $\hat{\boldsymbol{x}}$ has a compact support, then $\boldsymbol{x} \in C^{\infty}(\mathbb{R})$. The regularity of \boldsymbol{x} depends on how fast its Fourier coefficients decay: the highest the decay rate p, the highest the degree of regularity (smoothness) of the function.

Example 6. Regularity of the function $\boldsymbol{x}(t) = \frac{1}{T}\chi_{\left[-\frac{T}{2},\frac{T}{2}\right]}(t) \boldsymbol{e}_i$. In this case, $\hat{\boldsymbol{x}}(\omega) = \operatorname{sinc}\left(\frac{T\omega}{2}\right)$ and $\|\hat{\boldsymbol{x}}(\omega)\| \leq |\omega|^{-1}$. Therefore, $\boldsymbol{x} \in C^0(\mathbb{R})$.

Theorem 40 can be revisited by introducing the Sobolev space $W^{k,p}$, defined in 1D as:

$$W^{k,p}\left(\mathbb{R}\right) := \left\{ \boldsymbol{x} : \mathbb{R} \to \mathbb{R}^{n}, D^{\alpha} \boldsymbol{x} \in L^{p}\left(\mathbb{R}\right) \forall 0 \le |\alpha| \le k \right\}$$
(234)

In Equation (234), the $D^{\alpha}x$ represents the weak derivative of x(t) of order $|\alpha| = k^{26}$, defined, $\forall \phi(t) \in C_C^{\infty}$ test functions with compact support, as:

$$\int_{\mathbb{R}} \left\langle \boldsymbol{x}^{*}\left(t\right), D^{\alpha}\boldsymbol{\phi}\left(t\right)\right\rangle dt = (-1)^{|\alpha|} \int_{\mathbb{R}} \left\langle \boldsymbol{\phi}^{*}\left(t\right), D^{\alpha}\boldsymbol{x}\left(t\right)\right\rangle dt \quad \forall \boldsymbol{\phi}\left(t\right) \in C_{C}^{\infty}$$

The Sobolev space defined in Equation (234) is often equipped with the following norm, valid for all $1 \le p < \infty$:

$$\|\boldsymbol{x}\|_{W^{k,p}(\mathbb{R})} = \left(\sum_{j=0}^{k} \|\boldsymbol{x}^{(j)}(t)\|_{L^{p}(\mathbb{R})}^{p}\right)^{\frac{1}{p}} = \left(\sum_{j=1}^{k} \int_{\mathbb{R}} \|\boldsymbol{x}^{(j)}(t)\|^{p} dt\right)^{\frac{1}{p}}$$
(235)

The extension of Equation (236) to $p = \infty$ reads:

$$\|\boldsymbol{x}\|_{W^{k,\infty}(\mathbb{R})} = \max_{j \in [0,k]} \|\boldsymbol{x}^{(j)}(t)\|_{\infty}$$
(236)

Remark 41. $W^{k,p}$ equipped with the norm $\|\cdot\|_{W^{k,p}(\mathbb{R})}$ is a Banach space, therefore a complete normed vector space. It can be proved that the norm $\|\boldsymbol{x}(t)\|_{W^{k,p}(\mathbb{R})}$ is equivalent to $\|\boldsymbol{x}(t)\|_{L^{p}(\mathbb{R})} + \|\boldsymbol{x}^{(k)}(t)\|_{L^{p}(\mathbb{R})}$.

Provided the norm equivalence stated in Remark 41, Theorem 40 can be revisited as follows:

Theorem 42. On the regularity of a function $\boldsymbol{x}(t) \in W^{k,2}$ is bounded and it belongs to $C^k(\mathbb{R})$ if its $W^{k,2}$ norm is bounded.

Proof. If $\boldsymbol{x}(t) \in W^{k,2}$, then all its derivatives of to the k^{th} one belongs to $L^2(\mathbb{R})$, with a Fourier transform that exists and it is equivalent to $(i\omega)^k \hat{\boldsymbol{x}}(\omega)$. Recalling Plancherel formula in Equation (222), the $W^{k,2}$ -norm of the function $\boldsymbol{x}(t)$ can be rewritten as:

$$D^{\alpha}\boldsymbol{x} = \frac{\partial^{|\alpha|}\boldsymbol{x}}{\partial^{\alpha_1}x_1\partial^{\alpha_m}x_m\dots\partial^{\alpha_m}x_m}$$

 $^{^{26}\}mathrm{In}$ general, for a function $\boldsymbol{x}:\mathbb{R}^m\to\mathbb{R}^m$

$$\|\boldsymbol{x}(t)\|_{W^{k,2}(\mathbb{R})} = \left(\sum_{j=0}^{k} \|\boldsymbol{x}^{(j)}(t)\|_{L^{2}(\mathbb{R})}^{2}\right)^{\frac{1}{2}} = \frac{\|\boldsymbol{\hat{x}}(\omega)\|_{L^{2}(\mathbb{R})}}{\sqrt{2\pi}} \left(1 + \sum_{\substack{j=1\\k>0}}^{k} |\omega|^{2j}\right)^{\frac{1}{2}} < +\infty$$
(237)

Considering the equivalence of the $W^{k,2}$ -norm with the norm defined as $\|\boldsymbol{x}(t)\|_{L^2(\mathbb{R})} + \|\boldsymbol{x}^{(k)}(t)\|_{L^2(\mathbb{R})}$. Therefore, \boldsymbol{x} is bounded if:

$$\|\boldsymbol{x}(t)\|_{W^{k,2}(\mathbb{R})} = \|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})} + \|\boldsymbol{x}^{(k)}(t)\|_{L^{2}(\mathbb{R})} = \frac{\|\boldsymbol{\hat{x}}(\omega)\|_{L^{2}(\mathbb{R})}}{\sqrt{2\pi}} \left(1 + |\omega|^{k}\right) < +\infty$$
(238)

which implies the condition defined by Theorem 40 and proves the statement. $\hfill \Box$

The Fourier transform is affected by the *Heisenberg uncertainty*, which implies that a function $\boldsymbol{x}(t)$ cannot have both a compact support in time and in the frequency domain, as stated by the following theorem.

Theorem 43. Heisenberg uncertainty [Mal09]

A function $\mathbf{x}(t) \in L^2(\mathbb{R})$ with a temporal mean u, a temporal variance σ_t^2 , a temporal mean ξ and a frequency variance σ_{ω}^2 that read:

$$u = \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2}} \int_{\mathbb{R}} t \|\boldsymbol{x}(t)\|^{2} dt \qquad \xi = \frac{1}{2\pi \|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2}} \int_{\mathbb{R}} \omega \|\hat{\boldsymbol{x}}(\omega)\|^{2} d\omega$$
$$\sigma_{t}^{2} = \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2}} \int_{\mathbb{R}} (t-u)^{2} \|\boldsymbol{x}(t)\|^{2} dt$$
$$\sigma_{\omega}^{2} = \frac{1}{2\pi \|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2}} \int_{\mathbb{R}} (\omega-\xi)^{2} \|\hat{\boldsymbol{x}}(\omega)\|^{2} d\omega$$

then

$$\sigma_t^2 \cdot \sigma_\omega^2 \geq \frac{1}{4}$$

Moreover, if $\exists (u, \xi, A, b) \in \mathbb{R}^2 \times \mathbb{C}^2$ such that $\boldsymbol{x}(t) = A \cdot e^{i\xi t - b(t-u)^2} \boldsymbol{e}_i$, then $\sigma_t^2 \cdot \sigma_\omega^2 = \frac{1}{4}$

Proof. u and ξ represent the time and frequency average values. Rearranging their definition and recalling the Plancherel's formula in Theorem 36, one obtains:

$$0 = \frac{1}{2\pi \|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2}} \int_{\mathbb{R}} (\omega - \xi) \|\boldsymbol{\hat{x}}(\omega)\|^{2} d\omega = \frac{1}{2\pi \|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2}} \int_{\mathbb{R}} \omega \|\boldsymbol{\hat{x}}(\omega + \xi)\|^{2} d\omega$$

The Fourier spectrum $\hat{\boldsymbol{x}} (\omega + \xi)$ corresponds to a function $\boldsymbol{y} (t) = \boldsymbol{x} (t) \cdot e^{-i\xi t}$, that has a nil frequency average and whose time average reads:

$$\frac{1}{2\pi \|\boldsymbol{y}(t)\|_{L^{2}(\mathbb{R})}^{2}} \int_{\mathbb{R}} t \|\boldsymbol{y}(t)\|^{2} dt = \frac{1}{2\pi \|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{2}} \cdot \int_{\mathbb{R}} t \|\boldsymbol{x}(t)\|^{2} dt = u$$

Therefore, the function $\boldsymbol{z}(t) = \boldsymbol{x}(t+u) \cdot e^{-i\xi t}$ has zero temporal and frequency average, which simplifies the proof that comes next, since without loss of generality, we can prove it for $u = \xi = 0$.

The product of the time and frequency variance reads:

$$\sigma_t^2 \cdot \sigma_\omega^2 = \frac{1}{2\pi \|\boldsymbol{x}(t)\|_{L^2(\mathbb{R})}^4} \int_{\mathbb{R}} t^2 \|\boldsymbol{x}(t)\|^2 dt \cdot \int_{\mathbb{R}} \omega^2 \|\boldsymbol{\hat{x}}(\omega)\|^2 d\omega$$

By applying the Plancherel's formula to $\boldsymbol{x}'(t)$, whose Fourier transform is $i\omega \hat{\boldsymbol{x}}(\omega)$, the product of the two variance can be rewritten as:

$$\begin{split} \sigma_{t}^{2} \cdot \sigma_{\omega}^{2} &= \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{4}} \int_{\mathbb{R}} t^{2} \|\boldsymbol{x}(t)\|^{2} dt \int_{\mathbb{R}} \|\boldsymbol{x}^{'}(t)\|^{2} dt = \\ &= \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{4}} \|t\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})} \cdot \|\boldsymbol{x}^{'}(t)\|_{L^{2}(\mathbb{R})} \end{split}$$

Because of the Hölder inequality, the product of the two variances is bounded by below as follows:

$$\begin{split} \sigma_{t}^{2} \cdot \sigma_{\omega}^{2} &\geq \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{4}} \left(\int_{\mathbb{R}} \left\langle t\boldsymbol{x}^{*}(t), \boldsymbol{x}^{'}(t) \right\rangle dt \right)^{2} \\ &= \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{4}} \left(\int_{\mathbb{R}} \frac{t}{2} \left(\left\langle \boldsymbol{x}^{*}(t), \boldsymbol{x}^{'}(t) \right\rangle + \left\langle \boldsymbol{x}^{*'}(t), \boldsymbol{x}(t) \right\rangle \right) dt \right)^{2} = \\ &= \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{4}} \left(\int_{\mathbb{R}} \frac{t}{2} \left(\|\hat{\boldsymbol{x}}(\omega)\|^{2} \right)^{'} dt \right)^{2} = \\ &= \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{4}} \left(\frac{t}{2} \left(\|\hat{\boldsymbol{x}}(\omega)\|^{2} \right)|_{\mathbb{R}} - \frac{1}{2} \int_{\mathbb{R}} \left(\|\hat{\boldsymbol{x}}(\omega)\|^{2} \right) dt \right)^{2} \end{split}$$

According to [Wey50], if one assumes that $\lim_{t \to +\infty} \sqrt{t} \| \boldsymbol{x}(t) \| = 0$, the first integral vanishes and the previous expression becomes:

$$\sigma_t^2 \cdot \sigma_\omega^2 \ge \frac{1}{4}$$

that proves the first statement.

The Heisenberg inequality becomes an equality if:

$$\sigma_{t}^{2} \cdot \sigma_{\omega}^{2} = \frac{1}{\|\boldsymbol{x}(t)\|_{L^{2}(\mathbb{R})}^{4}} \left(\int_{\mathbb{R}} \left\langle t\boldsymbol{x}^{*}(t), \boldsymbol{x}^{'}(t) \right\rangle dt \right)^{2}$$

An intuitive guess is the Gaussian function, i.e. $\exists (a, b) \in \mathbb{C}^2$ such that $\boldsymbol{x}(t) = Ae^{-bt^2}\boldsymbol{e}_i$ such that $\boldsymbol{x}' = -2bt\boldsymbol{x}(t)$ that satisfies the equality constraint. For $u \neq \xi \neq 0$, the above mentioned translation apply.

B Compendium of fundamental results in optimization [RECAP]

This section presents the basic fundamental results in optimization, that constitute the basis of Machine Learning algorithms. Those notes are a summary of the Optimization class notes of CentraleSupélec [CP19], Université Paris-Dauphine [Mul19], École Normale Supérieure [Pey20] and École Polytechnique [AE23]. Another fundamental reference is [CP11].

Definition 44. Domain of a function $f: K \to \mathbb{R}$, with $K \subset \mathbb{R}^n$

$$\operatorname{dom}(f) := \{ \boldsymbol{x} \in K | f(\boldsymbol{x}) < +\infty \}$$

$$(239)$$

Definition 45. Epigraph of a function $epi(f) = \{(x, y) \in dom(f) \times \mathbb{R} | f(x) \le y\}$ **Definition 46.** A function $f : K \to \mathbb{R}$, with $K \subset \mathbb{R}^n$, is said to be proper iff $dom(f) \neq \emptyset$

Definition 47. A set $K \subset H$, with H being a Hilbert space, is bounded iff²⁷

$$\exists r > 0, \boldsymbol{x}_0 \in H | K \subset B_r \left(\boldsymbol{x}_0 \right)$$

Definition 48. A set $K \subset H$, with H being a Hilbert space, is closed iff

$$\forall (\boldsymbol{x}_n)_{n \in \mathbb{N}} \in K \qquad \lim_{n \to +\infty} x_n = x \in K$$

Definition 49. A set $K \subset H$, with H being a Hilbert space, is compact iff

 $\forall (\pmb{x}_n)_{n\in\mathbb{N}}\in K \qquad \exists \ (\pmb{x}_{n_k})_{k\in\mathbb{N}} \text{ such that } \exists \pmb{x}\in K \quad \lim_{k\to+\infty} \pmb{x}_{n_k}=\pmb{x}$

Proposition 50. If $K \subset H$, with H being a Hilbert space, is compact \Rightarrow closed and bounded. The opposite holds iff dim $H < +\infty$

Definition 51. Convex set $K \subset \mathbb{R}^n$:

$$\forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2, \forall t \in [0, 1] \qquad \boldsymbol{x}_t = t\boldsymbol{y} + (1 - t)\boldsymbol{x} \in K$$

$$^{27}B_r(\boldsymbol{x}_0) \text{ represents a open ball of center } \boldsymbol{x}_0 \text{ and radius } r, \text{ i.e.}$$

$$(240)$$

$$B_r(x_0) := \{ x \in H \| x - x_0 \| < r \}$$

Definition 52. Convex functions $f: K \to \mathbb{R}$ on convex sets $K \subset \mathbb{R}^n$:

$$\forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2, \forall t \in [0, 1] \qquad f(\boldsymbol{x}_t) = f(t\boldsymbol{y} + (1 - t)\boldsymbol{x}) \le tf(\boldsymbol{y}) + (1 - t)f(\boldsymbol{y})$$
(241)

Definition 53. Strictly convex functions $f: K \to \mathbb{R}$ on convex sets $K \subset \mathbb{R}^n$:

$$\forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2, \forall t \in [0, 1] \qquad f(\boldsymbol{x}_t) = f(t\boldsymbol{y} + (1 - t)\boldsymbol{x}) < tf(\boldsymbol{y}) + (1 - t)f(\boldsymbol{y})$$
(242)

Definition 54. Strongly convex functions $f: K \to \mathbb{R}$ on convex sets K:

$$\forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2, \forall t \in [0, 1] \qquad f(\boldsymbol{x}_t) = f(t\boldsymbol{y} + (1-t)\boldsymbol{x}) \le tf(\boldsymbol{y}) + (1-t)f(\boldsymbol{y}) - \alpha t(1-t)\|\boldsymbol{y} - \boldsymbol{x}\|^2, \qquad \alpha > 0$$
(243)

Proposition 55. f convex on convex set $K \subset \mathbb{R}^n \Leftarrow f$ strictly convex on convex set $K \subset \mathbb{R}^n \Leftarrow f$ strongly convex on convex set $K \subset \mathbb{R}^n$

Proposition 56. f convex on convex set $K \subset \mathbb{R}^n$, $f \in C^1(K) \Leftrightarrow$

1.
$$f(\boldsymbol{y}) \ge f(\boldsymbol{x}) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \right\rangle \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2$$

 $\begin{array}{l} \textit{Proof.} \Rightarrow \\ f \text{ convex on convex set } K \subset \mathbb{R}^n, \ f \in C^1(K) \text{ then by Definition 52:} \\ \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2, \forall t \in [0, 1] \qquad f(\boldsymbol{x}_t) = f(t\boldsymbol{y} + (1-t)\boldsymbol{x}) \leq tf(\boldsymbol{y}) + (1-t)f(\boldsymbol{x}) \\ \text{If } f \in C^1(K): \end{array}$

$$\lim_{t \to 0^{+}} \frac{f(\boldsymbol{x}_{t}) - f(\boldsymbol{x})}{t} = \langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \leq f(\boldsymbol{y}) - f(\boldsymbol{x})$$
(244)

Proof.
$$\Leftarrow$$

 $f \in C^{1}(K)$ and $\langle \nabla_{x} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \leq f(\boldsymbol{y}) - f(\boldsymbol{x})$, then:
 $f(\boldsymbol{x}) \geq f(\boldsymbol{x}_{t}) + \langle \nabla_{x} f(\boldsymbol{x}_{t}), \boldsymbol{x} - \boldsymbol{x}_{t} \rangle$
 $f(\boldsymbol{y}) \geq f(\boldsymbol{x}_{t}) + \langle \nabla_{x} f(\boldsymbol{x}_{t}), \boldsymbol{y} - \boldsymbol{x}_{t} \rangle$

and therefore:

$$(1-t)f(\boldsymbol{x}) + tf(\boldsymbol{y}) \ge f(\boldsymbol{x}_t)$$
(246)

(245)

2.
$$\left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{y}) - \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \right\rangle \ge 0 \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2$$

Proof. \Leftrightarrow f convex on convex set $K \subset \mathbb{R}^n$, $f \in C^1(K)$ then:

$$f(\boldsymbol{y}) \ge f(\boldsymbol{x}) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \right\rangle \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^{2}$$

$$f(\boldsymbol{x}) \ge f(\boldsymbol{y}) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{y}), \boldsymbol{x} - \boldsymbol{y} \right\rangle \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^{2}$$
(247)

By summing the two inequalities, the proposition is proven.

From a geometric standpoint, Proposition 56 states that a convex function is always above its tangent plane in \boldsymbol{x} .

Proposition 57. f strongly convex on convex set $K \subset \mathbb{R}^n$, $f \in C^1(K)$ (elliptic $function) \Leftrightarrow$

1.
$$f(\boldsymbol{y}) \ge f(\boldsymbol{x}) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \right\rangle + \frac{\alpha}{2} \|\boldsymbol{y} - \boldsymbol{x}\|^2 \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2$$

Proof. \Rightarrow f strongly convex on convex set $K \subset \mathbb{R}^n$, $f \in C^1(K)$ then by Definition 54: $f(x_t) = f(ty + (1 - t)x) \le tf(y) + (1 - t)x$ $orall (oldsymbol{x},oldsymbol{y}) \in K^2, orall t \in [0,1]$ $t)f(\boldsymbol{x}) - \alpha t(1-t) \|\boldsymbol{y} - \boldsymbol{x}\|^2$ If $f \in C^1(K)$: $\lim_{t \to 0^+} \frac{f(\boldsymbol{x}_t) - f(\boldsymbol{x})}{t} + \alpha(1-t) \|\boldsymbol{y} - \boldsymbol{x}\|^2 = \langle \boldsymbol{\nabla}_x f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle + \alpha \|\boldsymbol{y} - \boldsymbol{x}\|^2 \le f(\boldsymbol{y}) - f(\boldsymbol{x})$ (248)

which proves the statement.

Proof.
$$\Leftarrow f \in C^{1}(K) \text{ and } \frac{\alpha}{2} \|\boldsymbol{y} - \boldsymbol{x}\|^{2} + \langle \boldsymbol{\nabla}_{x} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \leq f(\boldsymbol{y}) - f(\boldsymbol{x}), \text{ then:}$$

$$f(\boldsymbol{x}) \geq f(\boldsymbol{x}_{t}) + \langle \boldsymbol{\nabla}_{x} f(\boldsymbol{x}_{t}), \boldsymbol{x} - \boldsymbol{x}_{t} \rangle + \frac{\alpha}{2} \|\boldsymbol{x} - \boldsymbol{x}_{t}\|^{2}$$

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}_{t}) + \langle \boldsymbol{\nabla}_{x} f(\boldsymbol{x}_{t}), \boldsymbol{y} - \boldsymbol{x}_{t} \rangle + \frac{\alpha}{2} \|\boldsymbol{y} - \boldsymbol{x}_{t}\|^{2}$$

$$(249)$$

and therefore:

$$(1-t)f(\boldsymbol{x}) + tf(\boldsymbol{y}) - \alpha t(1-t)\|\boldsymbol{y} - \boldsymbol{x}\|^2 \ge f(\boldsymbol{x}_t)$$
(250)

2.
$$\left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f\left(\boldsymbol{y}\right) - \boldsymbol{\nabla}_{\boldsymbol{x}} f\left(\boldsymbol{x}\right), \boldsymbol{y} - \boldsymbol{x} \right\rangle \ge \alpha \|\boldsymbol{y} - \boldsymbol{x}\|^{2} \qquad \forall \left(\boldsymbol{x}, \boldsymbol{y}\right) \in K^{2}$$

Proof. \Leftrightarrow f convex on convex set $K \subset \mathbb{R}^n$, $f \in C^1(K)$ then:

$$f(\boldsymbol{y}) \ge f(\boldsymbol{x}) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \right\rangle + \frac{\alpha}{2} \|\boldsymbol{y} - \boldsymbol{x}\|^2 \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2$$

$$f(\boldsymbol{x}) \ge f(\boldsymbol{y}) + \left\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{y}), \boldsymbol{x} - \boldsymbol{y} \right\rangle + \frac{\alpha}{2} \|\boldsymbol{y} - \boldsymbol{x}\|^2 \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2$$

(251)

By summing the two inequalities, the proposition is proven. \Box

From a geometric standpoint, Item 2 states that a convex function is always above a quadratic function in \boldsymbol{y} , which is above the tangent plane in \boldsymbol{x} . Elliptic functions have higher curvature (if they belong to $C^2(K)$) since in this case property 1 in Item 2 adds to the Taylor expansion at the second order²⁸, as follows:

$$\begin{aligned}
& \oint \{ (\boldsymbol{y}) \geq f(\boldsymbol{x}) + \langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle + \frac{\alpha}{2} \| \boldsymbol{y} - \boldsymbol{x} \|^{2} \\
& \oint \{ (\boldsymbol{y}) = f(\boldsymbol{x}) + \langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle + \frac{1}{2} \langle \mathbf{H}_{f}(\boldsymbol{x}) (\boldsymbol{y} - \boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle + o(\| \boldsymbol{y} - \boldsymbol{x} \|^{2}) \\
& (252)
\end{aligned}$$

(253)

which implies that, at the second order:

$$\frac{1}{2} \langle \mathbf{H}_f(\boldsymbol{x}) (\boldsymbol{y} - \boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \geq \frac{\alpha}{2} \| \boldsymbol{y} - \boldsymbol{x} \|^2, \qquad \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^2$$
(254)

Remark 58. Equation (254) proves that C^2 strongly convex functions have "higher" curvature then a quadratic polynomial with coefficient α (whose Hessian reads $\alpha \mathbf{I}$). Moreover, since $\mathbf{H}_f(\mathbf{x})$ is real and symmetric, Equation (254) implies that $\mathbf{H}_f(\mathbf{x})$ is positive definite.

Proposition 59. The following properties holds for convex functions :

- A function $f: K \to \mathbb{R}$ is convex $\iff epi(f)$ is convex
- A function $f: K \to \mathbb{R}$ is convex $\iff dom(f)$ is convex
- All linear combinations $\sum_{i=1}^{N} a_i f_i(\boldsymbol{x})$ of convex functions $f_i : K \to \mathbb{R}$ defined on a convex set K and with positive coefficients $a_i > 0$ is convex (proof straightforward)
- For a set $(f_i)_{i \in I}$ of convex functions $f_i : K \to \mathbb{R}$, then $\sup_{i \in I} f_i$ is convex.

²⁸In this case, $\lim_{\boldsymbol{y}\to\boldsymbol{x}} \frac{o(\|\boldsymbol{y}-\boldsymbol{x}\|)}{\|\boldsymbol{y}-\boldsymbol{x}\|} = 0$

- All linear combinations $\sum_{i=1}^{N} a_i f_i(\boldsymbol{x})$ of convex functions $f_i : K \to \mathbb{R}$ defined on a convex set K and with positive coefficients $a_i = 1$ is convex (proof straightforward)
- The composition h = g ∘ f of a monotonically increasing convex function g: R → R and a convex function f: K → R on a convex set K is convex:

Proof. Since f is convex, $f(\boldsymbol{x}_t) \leq tf(\boldsymbol{y}) + (1-t)f(\boldsymbol{x})$. Since g is monotonically increasing, $g(f(t(\boldsymbol{y}) + (1-t)(\boldsymbol{x}))) \leq g(tf(\boldsymbol{y}) + (1-t)f(\boldsymbol{x}))$. Since g is also convex, $g(f(t(\boldsymbol{y}) + (1-t)(\boldsymbol{x}))) \leq g(tf(\boldsymbol{y}) + (1-t)f(\boldsymbol{x})) \leq tg(f(\boldsymbol{y})) + (1-t)g(f(\boldsymbol{x}))$

Definition 60. Coercive functions f on unbounded domains $K \subset \mathbb{R}^n$:

$$\lim_{\|\boldsymbol{x}_n\|_K \to +\infty} f(\boldsymbol{x}_n) = +\infty \tag{255}$$

Which implies that $\forall M \in \mathbb{N}, \exists N \in \mathbb{N}$ such that $f(\boldsymbol{x}_n) \geq M, \forall \boldsymbol{x}_n$ such that $\|\boldsymbol{x}_n\|_K \geq N$

Proposition 61. f strongly convex on convex unbounded set $K \subset \mathbb{R}^n$, $f \in C^1(K)$ (elliptic function) $\Rightarrow f$ is coercive

Proof. f elliptic:

$$\begin{split} f\left(\boldsymbol{y}_{n}\right) &\geq f\left(\boldsymbol{x}\right) + \langle \boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}\right), \boldsymbol{y}_{n} - \boldsymbol{x} \rangle + \frac{\alpha}{2} \|\boldsymbol{y}_{n} - \boldsymbol{x}\|^{2} \geq f\left(\boldsymbol{x}\right) - \|\boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}\right)\| \cdot \|\boldsymbol{y}_{n} - \boldsymbol{x}\| + \frac{\alpha}{2} \|\boldsymbol{y}_{n} - \boldsymbol{x}\|^{2} \\ &\text{If } \|\boldsymbol{x}_{n}\| = \|\boldsymbol{y}_{n} - \boldsymbol{x}\| \to +\infty, \text{ then } f\left(\boldsymbol{x} + \boldsymbol{x}_{n}\right) \geq f\left(\boldsymbol{x}\right) - \|\boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}\right)\| \cdot \|\boldsymbol{x}_{n}\| + \frac{\alpha}{2} \|\boldsymbol{x}_{n}\|^{2} \to +\infty \end{split}$$

Example 7. Some example of convex and coercive functions:

- $f: \mathbb{R} \to \mathbb{R}, f(x) = x^2$ is convex on $\mathbb{R}\left(\frac{df}{dx} = 2x \Rightarrow \left(\frac{df}{dx}(y) \frac{df}{dx}(y)\right)(y-x) \ge 0\right)$
- $f: \mathbb{R}^n \to \mathbb{R}, f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2$ is strongly convex on \mathbb{R} $(\boldsymbol{\nabla}_x f(\boldsymbol{x}) = 2\boldsymbol{x} \Rightarrow \langle \boldsymbol{\nabla}_x f(\boldsymbol{y}) \boldsymbol{\nabla}_x f(\boldsymbol{x}), \boldsymbol{y} \boldsymbol{x} \rangle = 2\|\boldsymbol{y} \boldsymbol{x}\|^2 \ge 0)$
- $f: \mathbb{R}^n \to \mathbb{R}, f(\boldsymbol{x}) = \frac{1}{2} \langle \mathbb{A}\boldsymbol{x}, \boldsymbol{x} \rangle + \langle \boldsymbol{b}, \boldsymbol{x} \rangle + c \text{ with } \mathbb{A} \in \mathcal{M}_n(\mathbb{R}), \boldsymbol{b} \in \mathbb{R}^n, c \in \mathbb{R}$ is convex on \mathbb{R}^n iff \mathbb{A} is semi-positive definite $(\boldsymbol{\nabla}_x f(\boldsymbol{x}) = \mathbb{A}\boldsymbol{x} + \boldsymbol{b} \Rightarrow \langle \boldsymbol{\nabla}_x f(\boldsymbol{y}) - \boldsymbol{\nabla}_x f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle = 2 \|\mathbb{A}(\boldsymbol{y} - \boldsymbol{x})\|^2 \ge 0)$
- $f: \mathbb{R}^n \to \mathbb{R}, f(\boldsymbol{x}) = \frac{1}{2} \langle \mathbb{A}\boldsymbol{x}, \boldsymbol{x} \rangle + \langle \boldsymbol{b}, \boldsymbol{x} \rangle + c \text{ with } \mathbb{A} \in \mathcal{M}_n(\mathbb{R}), \boldsymbol{b} \in \mathbb{R}^n, c \in \mathbb{R}$ is strongly convex on \mathbb{R}^n iff \mathbb{A} is positive definite and its minimum eigenvalue $\lambda_{\min} > 0$ ($\nabla_x f(\boldsymbol{x}) = \mathbb{A}\boldsymbol{x} + \boldsymbol{b} \Rightarrow \langle \nabla_x f(\boldsymbol{y}) - \nabla_x f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle = 2 \|\mathbb{A}(\boldsymbol{y} - \boldsymbol{x})\|^2 = 2 \sum_{i=1}^n \lambda_i |y_i - x_i|^2 \ge \lambda_{\min} \sum_{i=1}^n |y_i - x_i|^2 = \lambda_{\min} \|\boldsymbol{y} - \boldsymbol{x}\|^2 (\alpha = \lambda_{\min}))$

- $f : \mathbb{R}^n \to \mathbb{R}, f(\boldsymbol{x}) = \frac{1}{2} \langle \mathbb{A}\boldsymbol{x}, \boldsymbol{x} \rangle + \langle \boldsymbol{b}, \boldsymbol{x} \rangle + c \text{ with } \mathbb{A} \in \mathcal{M}_n(\mathbb{R}), \boldsymbol{b} \in \mathbb{R}^n, c \in \mathbb{R}$ is strongly convex on \mathbb{R}^n , with \mathbb{A} is positive definite and its minimum eigenvalue $\lambda_{\min} > 0, f$ est coercive (\mathbb{R}^n is unbounded, f is $C^1(\mathbb{R}^n)$ so elliptic and therefore coercive).
- All function bounded below by a coercive function are coercive

Definition 62. Infimum of a set $K \in \mathbb{R}$

$$m = \inf_{x \in K} x, \qquad m \in \left] -\infty, +\infty\right[\tag{256}$$

Iff:

- $\forall x \in K, \qquad m \leq x$
- $\forall y \in \mathbb{R}$ such that $\forall x \in K \quad y \leq x \Longrightarrow m \geq y$ or alternatively: $\forall y \in \mathbb{R}$ such that $m < y \Longrightarrow \exists x \in K$ such that y > xor alternatively: $\forall \varepsilon > 0 \Longrightarrow \exists x \in K$ such that $x - m < \varepsilon$ or alternatively: $\exists (x_n)_{n \in \mathbb{N}} \in \mathbb{R}$ such that $\forall \varepsilon > 0 \Longrightarrow \exists N \in \mathbb{N}$ such that $\forall n \geq N \quad |x_n - x| < \varepsilon$ or alternatively: $\exists (x_n)_{n \in \mathbb{N}} \in \mathbb{R}$, $\lim_{n \to +\infty} x_n = m$

The same consideration hold for $m = \inf_{\boldsymbol{x} \in K} f(\boldsymbol{x})$, with $f : K \to \mathbb{R}$ and $K \subset \mathbb{R}^{n}$.

Definition 63. Infimum limit and supremum of a set $K \in \mathbb{R}$

$$\forall (x_n)_{n \in \mathbb{N}} \in \mathbb{R}, \quad \liminf x_n = \lim_{n \to +\infty} \inf_{k \ge n} (x_k) \tag{257}$$

$$\limsup x_n = -\liminf -x_n \tag{258}$$

Proposition 64. $\forall (x_n)_{n \in \mathbb{N}} \in \mathbb{R}$, $\lim_{n \to +\infty} x_n = m$ \iff $\liminf_{n \to +\infty} x_n = m$ $\lim_{n \to +\infty} x_n = m$ and $\lim_{n \to +\infty} x_n = m$

Proof. By definition, $\liminf x_n = m$ iff:

$$\forall \varepsilon > 0, \exists M_i \in \mathbb{N} \text{ such that } \forall n \geq M_i \quad |\inf_{k \geq n} x_k - m| < \varepsilon$$

By definition of $\limsup x_n = m$ iff:

$$\forall \varepsilon > 0, \exists M_s \in \mathbb{N} \text{ such that } \forall n \ge M_s \quad |\sup_{k \ge n} x_k - m| < \varepsilon$$

Since $\liminf x_n \leq \limsup x_n$, the following inequality holds:

 $\forall \varepsilon > 0, \exists M \ge \max\left\{M_s, M_i\right\} \in \mathbb{N} \text{ such that } \forall n \ge M \quad m + \varepsilon \le \inf_{k \ge n} x_k \le (x_k)_{k \ge n} \sup_{k \ge n} x_k < \varepsilon - m$

which proves the statement.

Definition 65. Lower semi-continuous functions (lsc) [CP19] $f: K \in \mathbb{R}$ with $K \in \mathbb{R}^n$ is lsc iff:

 $\exists (\boldsymbol{x}_n)_{n \in \mathbb{N}} \in K \text{ such that when } \lim_{n \to +\infty} \boldsymbol{x}_n = \boldsymbol{x} \Rightarrow \liminf f(\boldsymbol{x}_n) \ge f(\boldsymbol{x}) \quad (259)$

Proposition 66. $f : K \in \mathbb{R}$ with $K \in \mathbb{R}^n$ is lsc iff epi(f) is close. All continuous functions are lsc and so are the sum of lsc functions and $\sup_i f_i$ with f_i lsc [CP19].

Definition 67. \hat{x} is local minimizer of a proper function $f : K \to \mathbb{R}$ (see Definition 46) with a non-empty $K \subset H$ and H a Hilbert space if \exists open neighborhood $O(x) \in H$ such that:

$$f(\hat{\boldsymbol{x}}) \le f(\boldsymbol{x}) \qquad \forall \boldsymbol{x} \in O(\boldsymbol{x}) \cap K$$
 (260)

 $\boldsymbol{\hat{x}}$ is a strict local minimizer iff:

$$f(\hat{\boldsymbol{x}}) < f(\boldsymbol{x}) \qquad \forall \boldsymbol{x} \in (O(\boldsymbol{x}) \cap K) / \{\boldsymbol{x}\}$$
(261)

Definition 68. \hat{x} is global minimizer of a proper function $f : K \to \mathbb{R}$ (see Definition 46) with a non-empty $K \subset H$ and H a Hilbert space if:

$$f(\hat{\boldsymbol{x}}) \le f(\boldsymbol{x}) \qquad \forall \boldsymbol{x} \in K$$
 (262)

 $\boldsymbol{\hat{x}}$ is a strict global minimizer iff:

$$f(\hat{\boldsymbol{x}}) < f(\boldsymbol{x}) \qquad \forall \boldsymbol{x} \in K/\{\boldsymbol{x}\}$$
(263)

Definition 69. The minimum of a function $f : K \to \mathbb{R}$ with $K \subset \mathbb{R}^n$ is the value $m \in \mathbb{R}$ - if it exists - for which $\exists \hat{x} \in K$ such that:

$$\forall \boldsymbol{x} \in K \quad f(\boldsymbol{x}) \ge m \quad \text{and} \quad \min_{\boldsymbol{x} \in K} f(\boldsymbol{x}) = m = f(\hat{\boldsymbol{x}})$$
 (264)

The following theorems represent the theoretical framework within which the Machine Learning algorithms are defined. In particular, the following theorems assess the sufficient and necessary conditions to have local or global minimizers [CP19; Mul19].

Theorem 70. Bolzano-Weierstrass theorem (Rein analytischer Beweis, Bolzano, 1817 and Weierstrass, 1870): Sufficient condition of existence of a minimizer on a compact set

Given a function $f: K \to \mathbb{R}$ defined over a non-empty compact set $K \subset H$, with H being a finite dimensional Hilbert space, f proper and lsc on $K \Longrightarrow \exists \hat{x} \in K$ such that

$$f\left(\hat{\boldsymbol{x}}\right) = \inf_{\boldsymbol{x} \in K} f\left(\boldsymbol{x}\right) = m \in \mathbb{R}$$

ALERT Doctoral School 2023

Proof. If $\exists m = \inf_{\boldsymbol{x} \in K} f(\boldsymbol{x}), m \in \mathbb{R}$, by Definition 62 and by Proposition 64 $\exists (f(\boldsymbol{x}_n))_{n \in \mathbb{N}} \in H$ such that $m = \lim_{n \to +\infty} f(\boldsymbol{x}_n)$. Since K is compact $\exists (\boldsymbol{x}_{n_k})_{k \in \mathbb{N}}$ such that $\lim_{k \to +\infty} \boldsymbol{x}_{n_k} = \hat{\boldsymbol{x}} \in K$ (see Definition 49). Finally, since f is lsc (see Definition 65) $\liminf_{k \in \mathbb{N}} f(\boldsymbol{x}_{k \geq n}) = \liminf_{k \in \mathbb{N}} f(\boldsymbol{x}_{n_k}) \geq f(\hat{\boldsymbol{x}})$. But, by the property in Proposition 64,

$$\lim_{n \to \infty} f\left(\boldsymbol{x}_{n}\right) = m = \inf_{\boldsymbol{x} \in K} f\left(\boldsymbol{x}\right) \iff m = \liminf f\left(\boldsymbol{x}_{n}\right) \geq f\left(\boldsymbol{\hat{x}}\right)$$

Therefore, by definition of $\inf_{\boldsymbol{x}\in K} f(\boldsymbol{x})$ as the largest minimizer of $f(\boldsymbol{x})$ over K, it follows that $f(\boldsymbol{\hat{x}}) = m = \inf_{\boldsymbol{x}\in K} f(\boldsymbol{x})$ which proves the theorem. \Box

When K is closed but not bounded, the sufficient condition of existence of a minimizer requires f to be coercive. The following theorem is thus fundamental to prove the existence of minimizers on unbounded sets.

Theorem 71. Given a function $f : H \to \mathbb{R}$ defined over a non-empty finite dimensional Hilbert space H, f proper coercive and lsc on $H \Longrightarrow C = \arg\min_{\boldsymbol{x} \in K} f(\boldsymbol{x})$ is a non-empty compact set.

Proof. Since f is proper, $\exists \mathbf{x}_0 \in K | f(\mathbf{x}_0) < +\infty$. In this case, considering $B_r(\mathbf{x}_0) \in K$ the open ball in H of radius r > 0 and center \mathbf{x}_0 , then being f coercive (see Definition 60) then $\forall \mathbf{x} \notin B_r(\mathbf{x}_0)$ for which $||\mathbf{x} - \mathbf{x}_0|| > r$ it holds that $f(\mathbf{x}) > f(\mathbf{x}_0)$. On the contrary, $\bar{B}_r(\mathbf{x}_0) \cup K$ is a compact, on which the Weierstrass theorem applies, i.e., it exists a minimizer $\hat{\mathbf{x}} = \operatorname{arg\,inf}_{\mathbf{x} \in K} f(\mathbf{x}) \leq f(\mathbf{x}_0)$. Since f is coercive, then $\hat{\mathbf{x}}$ is a minimizer over H:

$$f\left(\boldsymbol{\hat{x}}\right) = \inf_{\boldsymbol{x}\in H} f\left(\boldsymbol{x}\right)$$

This result implies that $\arg \min_{\boldsymbol{x} \in H} f \subset K$ is bounded. Moreover, any sequence $(\boldsymbol{x}_n)_{n \in \mathbb{N}}$ converging to $\hat{\boldsymbol{x}} \in H$ satisfies the following property:

$$f\left(\boldsymbol{\hat{x}}\right) \leq \liminf f\left(\boldsymbol{x}_{n}\right) = \inf_{\boldsymbol{x} \in H} f\left(\boldsymbol{x}\right)$$

which implies that $\arg \min_{\boldsymbol{x} \in H} f(\boldsymbol{x})$ is also closed and then (since H is finite dimensional) $\arg \min_{\boldsymbol{x} \in H} f(\boldsymbol{x})$ is compact.

For any other open set $K \subset H$ is an open set, Theorem 71 does not apply. Therefore, does a minimizer of $f: K \to \mathbb{R}$ exist? The answer is positive, but under certain conditions.

Theorem 72. Sufficient condition of existence of a minimizer on an open set Given a function $f: K \to \mathbb{R}$ defined over a non-empty bounded open set $K \subset H$, with H being a finite dimensional Hilbert space, f proper and lsc on \overline{K} and $\exists x_0 \in K$ such that:

$$\forall \boldsymbol{x} \in \partial K, f\left(\boldsymbol{x}\right) > f\left(\boldsymbol{x}_{0}\right)$$

 $\Longrightarrow \exists \hat{x} \ such \ that$

$$f\left(\hat{\boldsymbol{x}}\right) = \inf_{\boldsymbol{x} \in K} f\left(\boldsymbol{x}\right) = m \in \mathbb{R}$$

Proof. \overline{K} is closed and bounded, therefore (since *H* is finite dimensional) \overline{K} is compact and the Weierstrass Theorem 70 applies:

$$\exists \hat{\boldsymbol{x}} \in \bar{K} \text{ such that } f(\hat{\boldsymbol{x}}) = \inf_{\boldsymbol{x} \in \bar{K}} f(\boldsymbol{x}) = m \qquad m \leq f(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \bar{K}$$

 $\hat{\boldsymbol{x}}$ belongs to $K=\operatorname{int}(K)$ (open set). The proof is obtained by reducing to absurd, i.e. by assuming that $\hat{\boldsymbol{x}} \in \partial K$. If it were so, provided the assumption that $\exists \boldsymbol{x}_0 \in K$ such that $\forall \boldsymbol{x} \in \partial K, f(\boldsymbol{x}) > f(\boldsymbol{x}_0)$, then

$$f\left(\hat{\boldsymbol{x}}\right) > f\left(\boldsymbol{x}_{0}\right) \quad \text{ and } \quad f\left(\hat{\boldsymbol{x}}\right) = m \leq f\left(\boldsymbol{x}_{0}\right), \boldsymbol{x}_{0} \in \bar{K}$$

The conditions are incompatible, which proves that $\hat{\boldsymbol{x}} \in K$ and $f(\boldsymbol{x}) \geq f(\hat{\boldsymbol{x}})$ $\forall \boldsymbol{x} \in \bar{K}$, included \boldsymbol{x}_0 .

The previous theorems assure the existence of a minimizer, but not the necessary condition required to search for one, nor the uniqueness of the minimizer. For such conditions, the convexity properties should be considered. The following results introduce such conditions.

Theorem 73. *Euler's inequality:* necessary condition for a minimizer on a convex set

Given a function $f : K \to \mathbb{R}$ defined over a non-empty convex set $K \subset H$, with H being a Hilbert space, f proper on K and $f \in C^1(K)$, if \hat{x} is a local minimizer of f on $K \Leftarrow$

$$\langle \boldsymbol{\nabla}_{\boldsymbol{x}} f\left(\boldsymbol{\hat{x}}\right), \boldsymbol{y} - \boldsymbol{\hat{x}} \rangle \geq 0, \qquad \forall \boldsymbol{y} \in K$$

Proof. Provided that K is convex $\forall t \in [0, 1] \ \boldsymbol{x}_t = t\boldsymbol{y} + (1 - t)\boldsymbol{\hat{x}} \in K, \forall \boldsymbol{y} \in K.$ Moreover, $f \in C^1(K)$ which implies that the following Taylor expansion on f holds:

$$f(\boldsymbol{x}_{t}) = T_{f} \hat{\boldsymbol{x}}(\boldsymbol{x}_{t}) + o\left(\|\boldsymbol{x}_{t} - \hat{\boldsymbol{x}}\|\right) = f\left(\hat{\boldsymbol{x}}\right) + \langle \boldsymbol{\nabla}_{\boldsymbol{x}} f\left(\hat{\boldsymbol{x}}\right), \boldsymbol{x}_{t} - \hat{\boldsymbol{x}} \rangle + o\left(\|\boldsymbol{x}_{t} - \hat{\boldsymbol{x}}\|\right) \ge f\left(\hat{\boldsymbol{x}}\right)$$

such that $\lim_{t\to 0^+} \frac{o(\|\boldsymbol{x}_t - \hat{\boldsymbol{x}}\|)}{\|\boldsymbol{x}_t - \hat{\boldsymbol{x}}\|} = 0$. Taking the following limit for $t \to 0$, the statement is proven:

$$\langle \boldsymbol{\nabla}_{x} f(\hat{\boldsymbol{x}}), \boldsymbol{y} - \hat{\boldsymbol{x}} \rangle =$$

$$= \lim_{t \to 0} \frac{\langle \boldsymbol{\nabla}_{x} f(\hat{\boldsymbol{x}}), \boldsymbol{x}_{t} - \hat{\boldsymbol{x}} \rangle + o\left(\|\boldsymbol{x}_{t} - \hat{\boldsymbol{x}}\| \right)}{t} = \lim_{t \to 0} \frac{f\left(\boldsymbol{x}_{t}\right) - f\left(\hat{\boldsymbol{x}}\right)}{t} \ge 0 \qquad \forall \boldsymbol{y} \in K$$

If K is open, the inequality in Theorem 73 becomes an equality and a sufficient and necessary condition for \hat{x} to be a minimizer. However, one must first acknowledge this preliminary result:

Theorem 74. *Euler's equality: necessary condition for a minimizer of a function on a convex open set*

Given a function $f: K \to \mathbb{R}$ defined over a non-empty convex set $K \subset H$, with H being a finite dimension Hilbert space, f proper on K and $f \in C^1(K)$, if \hat{x} is a local minimizer of f on $K \Leftarrow$

$$\boldsymbol{\nabla}_{x}f\left(\boldsymbol{\hat{x}}\right) = \boldsymbol{0}$$

Proof. \Rightarrow

Since $\hat{x} = \arg \min_{x \in K} f(x)$, $\exists O(\hat{x})$ open neighborhood that, according to Equation (260) in Definition 67:

$$f(\hat{\boldsymbol{x}}) \leq f(\boldsymbol{x}) \qquad \forall \boldsymbol{x} \in O(\boldsymbol{x}) \cap K$$

Therefore, if one considers $\boldsymbol{x} = \boldsymbol{\hat{x}} + h\boldsymbol{y}, h \in [0, r_0]^{29}$ such that $\boldsymbol{x} \in O(\boldsymbol{x}) \cap K, \forall h \in [0, r_0]$ and $\forall \boldsymbol{y} \in K$, then :

$$\lim_{h \to 0^{+}} \frac{f\left(\hat{\boldsymbol{x}} + h\boldsymbol{y}\right) - f\left(\hat{\boldsymbol{x}}\right)}{h} = \langle \boldsymbol{\nabla}_{\boldsymbol{x}} f\left(\hat{\boldsymbol{x}}\right), \boldsymbol{y} \rangle \ge 0 \quad \forall \boldsymbol{y} \in K$$

But $\langle \boldsymbol{\nabla}_{x} f(\hat{\boldsymbol{x}}), -\boldsymbol{y} \rangle \geq 0$ too, which implies the only solution: $\boldsymbol{\nabla}_{x} f(\hat{\boldsymbol{x}}) \equiv \boldsymbol{0}$. \Box

Proof. \Leftarrow If $\nabla_x f(\hat{x}) \equiv 0$, the statement is proven.

If K and f are both convex, the inequality in Theorem 73 becomes a sufficient and necessary condition for \hat{x} to be a minimizer.

Theorem 75. Euler's inequality for convex functions: sufficient and necessary condition for the existence of a global minimizer of a convex function on a convex set

Given a function $f : K \to \mathbb{R}$ defined over a non-empty convex set $K \subset \mathbb{R}^n$, with \mathbb{R}^n being a Hilbert space of finite dimension n, f proper and convex on Kand $f \in C^1(K)$

 $\hat{\boldsymbol{x}}$ is a global minimizer of f on $K \iff \langle \boldsymbol{\nabla}_x f(\hat{\boldsymbol{x}}), \boldsymbol{y} - \hat{\boldsymbol{x}} \rangle \ge 0, \quad \forall \boldsymbol{y} \in K$

 $\begin{array}{l} \textit{Proof.} \Rightarrow \\ \text{See Theorem 73} \end{array}$

Proof. \Leftarrow Since f is convex, Proposition 55 holds:

$$f\left(\boldsymbol{y}\right) \geq f\left(\boldsymbol{\hat{x}}\right) + \left\langle \boldsymbol{\nabla}_{x}f\left(\boldsymbol{\hat{x}}\right), \boldsymbol{y} - \boldsymbol{\hat{x}} \right\rangle$$

²⁹When $K \subset \mathbb{R}^n$, the open neighborhood is the open ball $B_r(\hat{x})$.

But, since \hat{x} is a minimizer, the following holds too:

$$f(\hat{\boldsymbol{x}})+2\langle \boldsymbol{\nabla}_{x}f(\hat{\boldsymbol{x}}), \boldsymbol{y}-\hat{\boldsymbol{x}}\rangle \geq f(\boldsymbol{y})+\langle \boldsymbol{\nabla}_{x}f(\hat{\boldsymbol{x}}), \boldsymbol{y}-\hat{\boldsymbol{x}}\rangle \geq f(\hat{\boldsymbol{x}})+\langle \boldsymbol{\nabla}_{x}f(\hat{\boldsymbol{x}}), \boldsymbol{y}-\hat{\boldsymbol{x}}\rangle$$
which proves the statement.

the state

Proposition 76. Provided that Theorem 75 holds for convex functions on convex subsets of \mathbb{R}^n , if K is open, Theorem 74 holds too, with the Euler's inequality that

$$\nabla_{x}f\left(\hat{x}\right) = \mathbf{0}$$

If f is strictly convex, the minimizer is unique.

Theorem 77. Sufficient and necessary condition for the existence of unique minimizer of a strictly convex function on a convex set Given a function $f: K \to \mathbb{R}$ defined over a non-empty convex set $K \subset H$, with H being a Hilbert space, f proper and strictly convex on $K \Longrightarrow$

$$\exists \mathbf{\hat{x}} \text{ such that } \mathbf{\hat{x}} = \inf_{\mathbf{x} \in K} f(\mathbf{x})$$

Proof. The proof is obtained by reducing to absurd and assuming that there are two different minimizers $\hat{x}_1 \neq \hat{x}_2$. In this case, assuming $t = \frac{1}{2}, \frac{\hat{x}_1 + \hat{x}_2}{2} \in K$ which is convex and therefore:

$$f\left(\hat{x}_{1}\right) < f\left(\frac{\hat{x}_{1} + \hat{x}_{2}}{2}\right) < \frac{1}{2}f\left(\hat{x}_{1}\right) + \frac{1}{2}f\left(\hat{x}_{2}\right) = f\left(\hat{x}_{1}\right)$$

which is not possible.

Finally, if f is strongly convex (see Definition 54), the minimum exists and it is unique and it satisfies the Euler's inequality.

Theorem 78. Euler's inequality for strongly convex functions: Sufficient and necessary condition for the uniqueness of a minimizer (if it exists) of a strongly convex function on a closed convex set

Given a function $f: K \to \mathbb{R}$ defined over a non-empty closed and convex set $K \subset H$, with H being a Hilbert space, f proper and strongly convex on K and $f \in C^1(K)$

$\hat{\boldsymbol{x}}$ is a unique minimizer of f on $K \iff \langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\hat{\boldsymbol{x}}), \boldsymbol{y} - \hat{\boldsymbol{x}} \rangle \geq 0, \quad \forall \boldsymbol{y} \in K$

Proof. Since f proper and $f \in C^1$, it is lsc on K. Moreover, f is strongly convex, therefore elliptic, then it is coercive (see Proposition 61). Being coercive, Theorem 71 holds and therefore the minimizer exists. Moreover, the Theorem 71 states that $\arg\min_{x \in K} f$ is compact. Again, since f is strongly convex,

it is also strictly convex (see Proposition 55), which implies - by Theorem 77 - that the existing minimizers is unique. Finally, f being strongly convex implies f being convex - again by Theorem 77 - which implies that Theorem 75 holds and therefore the Euler's inequality is proven.

Remark 79. In the following we will consider $K = \mathbb{R}^n$, which is convex. Therefore, the Euler's inequality in Theorem 73 holds and the vectors $\boldsymbol{y} - \hat{\boldsymbol{x}}$ span all the affine space [Mul19].

The strong convexity of the function f has some repercussions on its smoothness, defined by its β -Lipschitz continuity [Pey20]:

Definition 80. A function $f: K \to \mathbb{R}$ is said to be β -Lipschitz iff

$$\exists \beta \in \mathbb{R}^{+} \text{ such that } \forall (\boldsymbol{x}, \boldsymbol{y}) \in K^{2} \Rightarrow |f(\boldsymbol{y}) - f(\boldsymbol{x})| \leq \beta \|\boldsymbol{y} - \boldsymbol{x}\|$$
(265)

The function is Lipschitz, i.e. $f \in \text{Lip}(K)$ for $\beta = \sup_{y \neq x} \frac{|f(y) - f(x)|}{\|y - x\|}$ (Lipschitz constant).

Lipschitz strongly convex functions are sufficiently smooth since they are bounded by below and by above by a quadratic function, as shown in the following.

Proposition 81. A strongly convex function $f: K \to \mathbb{R}$, defined over a nonempty convex set $K \subset H$, with H being a Hilbert space, f proper and strongly convex on K, with $f \in C^2(K)$ and a gradient $\nabla_x f \in Lip(K)$ then $\forall (x, y) \in K^2$:

$$\frac{\alpha}{2} \|\boldsymbol{y} - \boldsymbol{x}\|^2 \le f(\boldsymbol{y}) - f(\boldsymbol{x}) - \langle \boldsymbol{\nabla}_x f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \le \frac{\beta}{2} \|\boldsymbol{y} - \boldsymbol{x}\|^2$$
(266)

or alternatively, since the Hessian operator $\mathbf{H}_{f}(\mathbf{x})$ is symmetric [Pey20]:

$$\alpha \mathbf{I} \preceq \mathbf{H}_f(\boldsymbol{x}) \preceq \beta \mathbf{I}, \qquad \forall \boldsymbol{x} \in K$$
(267)

with \leq indicating the natural order of symmetric matrices, i.e.:

$$\forall (\mathbb{A}, \mathbb{B}) \in Sym(\mathbb{R}^n)^2 \Rightarrow \mathbb{A} \preceq \mathbb{B} \iff \langle \mathbb{A}. \boldsymbol{u}, \boldsymbol{u} \rangle \leq \langle \mathbb{B}. \boldsymbol{u}, \boldsymbol{u} \rangle \forall \boldsymbol{u} \in \mathbb{R}^n$$

Proof. The lower bound has been proved in Item 2. For the upper bound, since $f \in C^2(K)$ and with $\boldsymbol{x}_t = t\boldsymbol{y} + (1-t)\boldsymbol{x}, t \in [0,1]$:

$$\begin{split} f\left(\boldsymbol{y}\right) - f\left(\boldsymbol{x}\right) &= \int_{0}^{1} D_{t\left(\boldsymbol{y}-\boldsymbol{x}\right)} f\left(\boldsymbol{x}\right) dt = \left\langle \boldsymbol{\nabla}_{x} f\left(\boldsymbol{x}\right), \boldsymbol{y}-\boldsymbol{x} \right\rangle + \\ &+ \int_{0}^{1} \left\langle \boldsymbol{\nabla}_{x} f\left(\boldsymbol{x}_{t}\right) - \boldsymbol{\nabla}_{x} f\left(\boldsymbol{x}\right), \boldsymbol{y}-\boldsymbol{x} \right\rangle dt \end{split}$$

Applying the Hölder inequality $\langle \boldsymbol{a}, \boldsymbol{b} \rangle \leq \|\boldsymbol{a}\| \cdot \|\boldsymbol{b}\|$ to the previous expression, coupled with the Lipschitz property of f, one obtains:

$$f\left(\boldsymbol{y}\right) - f\left(\boldsymbol{x}\right) - \left\langle \boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}\right), \boldsymbol{y} - \boldsymbol{x} \right\rangle \leq \int_{0}^{1} \left\|\boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}_{t}\right) - \boldsymbol{\nabla}_{x}f\left(\boldsymbol{x}\right)\right\| dt \cdot \left\|\boldsymbol{y} - \boldsymbol{x}\right\|$$

Applying the Lipschitz property of the gradient, the following expression proves the statement:

$$f(\boldsymbol{y}) - f(\boldsymbol{x}) - \langle \boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}), \boldsymbol{y} - \boldsymbol{x} \rangle \leq \beta \int_{0}^{1} t \cdot dt \cdot \|\boldsymbol{y} - \boldsymbol{x}\|^{2} \leq \frac{\beta}{2} \|\boldsymbol{y} - \boldsymbol{x}\|^{2}$$

Remark 82. If Proposition 81 holds, equivalently Equation (267) does too. In this case, Equation (267) can be rephrased as:

$$\alpha \|\boldsymbol{u}\|^{2} \leq \langle \mathbf{H}_{f}(\boldsymbol{x}) . \boldsymbol{u}, \boldsymbol{u} \rangle \leq \beta \|\boldsymbol{u}\|^{2}, \qquad \forall \boldsymbol{x} \in K, \forall \boldsymbol{u} \in H$$
(268)

Plus, as observed in Remark 58, $\mathbf{H}_{f}(\boldsymbol{x})$ is positive definite, i.e., all its eigenvalues are positive, thus, given the spectral representation of the symmetric real matrix $\mathbf{H}_{f}(\boldsymbol{x})$, Equation (268) reduces to:

$$(\min \lambda_i \left(\mathbf{H}_f \left(\boldsymbol{x} \right) \right) \ge \alpha \tag{269}$$

$$\max \lambda_{i} \left(\mathbf{H}_{f} \left(\boldsymbol{x} \right) \right) \leq \beta \tag{270}$$

Equation (270) implies that the Hessian matrix must be well conditioned. In other words, Equation (270) can be replaced by invoking the conditioning number of the Hessian matrix $\kappa (\mathbf{H}_f(\boldsymbol{x})) = \frac{\max \lambda_i(\mathbf{H}_f(\boldsymbol{x}))}{\min \lambda_i(\mathbf{H}_f(\boldsymbol{x}))}$ that must respect the condition $0 < \kappa (\mathbf{H}_f(\boldsymbol{x})) \leq \frac{\beta}{\alpha}$. This aspect is crucial for convergence analysis of gradient descent methods (see Section 4.3 for further details). In other words, the strongest is the convexity of the function, the more flexible is the conditioning of the Hessian matrix. On the contrary, a lower Lipschitz constant β on the gradient represents a more stringent conditioning of the Hessian matrix.

References

- [AE23] Allaire, Grégoire and Alexandre Ern. Optimisation et Contrôle. Cours de l'Ecole Polytechnique. 2023. URL: http://www.cmap. polytechnique.fr/~allaire/map435/poly435.pdf.
- [AW19] Azulay, Aharon and Yair Weiss. "Why do deep convolutional networks generalize so poorly to small image transformations?" In: *Journal of Machine Learning Research* 20 (2019), pp. 1–25.

- [Bar93] Barron, Andrew R. "Universal approximation bounds for superpositions of a sigmoidal function". In: *IEEE Transactions on Information theory* 39.3 (1993), pp. 930–945.
- [Bil95] **Billingsley, Patrick**. *Measure and Probability*. John Wiley and Sons: New York, 1995.
- [Cam18] Campagne, J.E. L'apprentissage face à la malédiction de la grande dimensionn. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2018). 2018. DOI: https://doi.org/ 10.4000/annuaire-cdf.15441.
- [Cam19] Campagne, J.E. L'apprentissage par réseaux de neurones profonds. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2019). 2019. DOI: https://doi.org/10. 4000/annuaire-cdf.16767. URL: https://www.di.ens.fr/ ~mallat/College/Cours-2019-Mallat-Jean-Eric-Campage. pdf.
- [Cam20] Campagne, J.E. Modè les multi-échelles et réseaux de neurones convolutifs. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2020). 2020. URL: https://www. di.ens.fr/~mallat/College/Cours2020-Mallat-Jean-Eric-Campagne.pdf.
- [Cam22] Campagne, J.E. Information et Complexite. Notes et commentaires au sujet des conférences de S. Mallat du Collè ge de France (2022). 2022. DOI: https://doi.org/10.4000/annuaire-cdf. 18077. URL: https://www.di.ens.fr/~mallat/College/Cours-2022-Mallat-Jean-Eric-Campagne.pdf.
- [Che+16] Chen, Xi et al. "InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets". en. In: arXiv:1606.03657 (June 2016). arXiv:1606.03657 [cs, stat]. URL: http://arxiv.org/abs/1606.03657.
- [Clo22] Clouteau, Didier. Statistical Learning for Mechanics Lecture notes. Cours de Modélisation des Incertitudes et Fiabilité des Ouvrages-Master 2 - Université Paris-Saclay. 2022.
- [Cou20] **Cournède, Paul-Henry**. Official handbook of the Statistics and Learning. "Ingénieur" Curriculum, CentraleSupélec. 2020.
- [CP11] Combettes, Patrick L. and Jean-Christophe Pesquet. "Proximal Splitting Methods in Signal Processing". en. In: Fixed-Point Algorithms for Inverse Problems in Science and Engineering. Ed. by Heinz H. Bauschke et al. Vol. 49. Springer Optimization and Its Applications. New York, NY: Springer New York, 2011, pp. 185–212. ISBN: 978-1-4419-9568-1. DOI: 10.1007/978-1-4419-9569-8_10. URL: https://link.springer.com/10.1007/978-1-4419-9569-8_10.
- [CP19] Chouzenoux, Emilie and Jean-Christophe Pesquet. Official handbook of the Optimization. "Ingénieur" Curriculum, Centrale-

Supélec. Class notes. 2019. URL: https://www-syscom.univmlv.fr/~chouzeno/ECP/index.htm.

- [CUH16] Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". en. In: arXiv:1511.07289 (Feb. 2016). arXiv:1511.07289 [cs]. URL: http://arxiv.org/abs/1511.07289.
- [Cyb89] Cybenko, G. "Approximation by superpositions of a sigmoidal function". en. In: Mathematics of Control, Signals, and Systems 2.4 (Dec. 1989), pp. 303-314. ISSN: 0932-4194, 1435-568X. DOI: 10. 1007/BF02551274. URL: http://link.springer.com/10.1007/BF02551274.
- [Fis22] Fisher, R. A. "On the mathematical foundations of theoretical statistics". en. In: 222 (Jan. 1922). URL: https://royalsocietypublishing. org/doi/10.1098/rsta.1922.0009.
- [For+15] **Forest, Samuel et al.** *Mécanique des milieux continus.* École des Mines de Paris, 2015.
- [GM98] Grenander, Ulf and Michael I. Miller. "Computational anatomy: an emerging discipline". en. In: Quarterly of Applied Mathematics 56.4 (1998), pp. 617–694. ISSN: 0033-569X, 1552-4485. DOI: 10. 1090/qam/1668732. URL: https://www.ams.org/qam/1998-56-04/S0033-569X-1998-1668732-7/.
- [Goo+14] Goodfellow, Ian J. et al. "Generative Adversarial Networks". In: arXiv:1406.2661 (June 2014). arXiv:1406.2661 [cs, stat]. URL: http://arxiv.org/abs/1406.2661.
- [HDD] Hecht-Nielsen, Robert, Oberlin Drive, and San Diego. "Kolmogorov's Mapping Neural Network Existence Theorem". en. In: ().
- [Hec92] Hecht-Nielsen, Robert. "Theory of the Backpropagation Neural Network". en. In: Neural Networks for Perception. Elsevier, 1992, pp. 65–93. ISBN: 978-0-12-741252-8. DOI: 10.1016/B978-0-12-741252-8.50010-8. URL: https://linkinghub.elsevier.com/ retrieve/pii/B9780127412528500108.
- [Hil79] Hill, Rodney. "Aspects of Invariance in Solid Mechanics". en. In: Advances in Applied Mechanics. Vol. 18. Elsevier, 1979, pp. 1–75. ISBN: 978-0-12-002018-8. DOI: 10.1016/S0065-2156(08)70264- 3. URL: https://linkinghub.elsevier.com/retrieve/pii/ S0065215608702643.
- [Jay57] Jaynes, E. T. "Information Theory and Statistical Mechanics". In: *Phys. Rev.* 106.4 (May 1957), pp. 620–630. DOI: 10.1103/PhysRev. 106.620. URL: https://link.aps.org/doi/10.1103/PhysRev. 106.620.
- [KD21] Kolev, B. and R. Desmorat. "An Intrinsic Geometric Formulation of Hyper-Elasticity, Pressure Potential and Non-Holonomic Constraints". en. In: *Journal of Elasticity* 146.1 (Sept. 2021), pp. 29– 63. ISSN: 0374-3535, 1573-2681. DOI: 10.1007/s10659-021-09853-

5. URL: https://link.springer.com/10.1007/s10659-021-09853-5.

- [Kla+17] Klambauer, Günter et al. "Self-Normalizing Neural Networks".
 en. In: arXiv:1706.02515 (Sept. 2017). arXiv:1706.02515 [cs, stat].
 URL: http://arxiv.org/abs/1706.02515.
- [KW22] Kingma, Diederik P. and Max Welling. "Auto-Encoding Variational Bayes". en. In: arXiv:1312.6114 (Dec. 2022). arXiv:1312.6114 [cs, stat]. URL: http://arxiv.org/abs/1312.6114.
- [LeC98] LeCun, Yann. "The MNIST database of handwritten digits". In: http://yann.lecun.com/exdb/mnist/ (1998).
- [Les+93] Leshno, Moshe et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". en. In: Neural Networks 6.6 (Jan. 1993), pp. 861-867. ISSN: 08936080. DOI: 10.1016/S0893-6080(05)80131-5. URL: https:// linkinghub.elsevier.com/retrieve/pii/S0893608005801315.
- [Mai99] Maiorov, V.E. "On Best Approximation by Ridge Functions". en. In: Journal of Approximation Theory 99.1 (July 1999), pp. 68–94. ISSN: 00219045. DOI: 10.1006/jath.1998.3304. URL: https:// linkinghub.elsevier.com/retrieve/pii/S0021904598933044.
- [Mak19] Makhzani, A. "Implicit Autoencoders". en. In: ICLR 2019 arXiv:1805.09804 (Feb. 2019). arXiv:1805.09804. URL: http://arxiv.org/abs/ 1805.09804.
- [Mal09] Mallat, S. G. A wavelet tour of signal processing: the sparse way.
 en. 3rd ed. Amsterdam; Boston: Elsevier Academic Press, 2009.
 ISBN: 978-0-12-374370-1.
- [Mal16] Mallat, Stéphane. "Understanding deep convolutional networks". en. In: Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 374.2065 (Apr. 2016), p. 20150203. ISSN: 1364-503X, 1471-2962. DOI: 10.1098/rsta. 2015.0203. URL: https://royalsocietypublishing.org/doi/ 10.1098/rsta.2015.0203.
- [MMD20] Mouton, Coenraad, Johannes C. Myburgh, and Marelie H. Davel. "Stride and Translation Invariance in CNNs". en. In: vol. 1342. arXiv:2103.10097 [cs]. 2020, pp. 267–281. DOI: 10.1007/ 978-3-030-66151-9_17. URL: http://arxiv.org/abs/2103. 10097.
- [Mul19] Mula, Olga. Optimisation et programmation dynamique. Master mention Mathématiques Appliquées, 1è re année Université Paris Dauphine. 2019.
- [Nes83] Nesterov, Yurii. "A method for unconstrained convex minimization problem with the rate of convergence O (1/k²)". In: *Doklady* an ussr. Vol. 269. 1983, pp. 543–547.
- [Pey20] Peyré, Gabriel. Course notes on Optimization for Machine Learning. Notes de cours de l'École Normale Supérieure. 2020. URL: https://mathematical-tours.github.io.

- [Pin99] Pinkus, Allan. "Approximation theory of the MLP model in neural networks". en. In: Acta Numerica 8 (Jan. 1999), pp. 143–195. ISSN: 0962-4929, 1474-0508. DOI: 10.1017/S0962492900002919. URL: https://www.cambridge.org/core/product/identifier/ S0962492900002919/type/journal_article.
- [Rio18] Rioul, Olivier. "Une théorie mathématique de la communication".
 In: BibNum (Jan. 2018). ISSN: 2554-4470. DOI: 10.4000/bibnum.
 1190. URL: http://journals.openedition.org/bibnum/1190.
- [SG20] Semih Kayhan, Osman and Jan C. van Gemert. "On Translation Invariance in CNNs: Convolutional Layers Can Exploit Absolute Spatial Location". en. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). Seattle, WA, USA: IEEE, June 2020, pp. 14262–14273. ISBN: 978-1-72817-168-5. DOI: 10.1109/CVPR42600.2020.01428. URL: https://ieeexplore. ieee.org/document/9156444/.
- [Sha48] Shannon, C. E. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.
- [TN04] Truesdell, C. and W. Noll. "The Non-Linear Field Theories of Mechanics". en. In: The Non-Linear Field Theories of Mechanics. Ed. by Stuart S. Antman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–579. ISBN: 978-3-642-05701-4. DOI: 10.1007/ 978-3-662-10388-3_1. URL: http://link.springer.com/10. 1007/978-3-662-10388-3_1.
- [Wey50] Weyl, Hermann. The theory of groups and quantum mechanics. Courier Corporation, 1950.



Figure 8: $g(\varepsilon)$, with $\varepsilon = \frac{\prod_{k=1}^{k} ERT_{x}}{\|x_{k}-\hat{x}\|}$ for different values of the Lipschitz constant β . The contour plots represent different values of δ_{k} . Blue-graded lines represent values of $\delta_{k} > \frac{1}{\beta}$. Orange-graded lines represent values of $\frac{2}{\beta} < \delta_{k} \leq \frac{1}{\beta}$. Blue solid line represents $\delta = \frac{1}{\beta}$ and green line $\delta_{k} = \frac{2}{\beta}$.



Figure 9: ELU activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.



Figure 10: SELU activation function g(a) and its derivative $\frac{\partial g}{\partial a}$.



Figure 11: Self-centering properties of the *SELU* activation function, reprinted from [Kla+17]. μ_a and μ_h represent the average pre-activation and activation of any neuron at layer ℓ of a \mathcal{MLP} , whereas σ_a^2 and σ_h^2 represent their variances.

Physics-informed and thermodynamics-based neural networks

Filippo Masi¹, Ioannis Stefanou²

¹ The University of Sydney, Australia

² Nantes Université, Ecole Centrale Nantes, CNRS, Institut de Recherche en Génie Civil et Mécanique (GeM), UMR 6183, F-44000, Nantes, France

This chapter aims to offer a comprehensive introduction to the integration of prior knowledge stemming from physics and/or thermodynamics into deep learning algorithms. By means of motivating examples, we explore the capabilities and strengths of (i) Physics-Informed Neural Networks for the discovery, driven by data, of partial differential equations and of (ii) Thermodynamics-based Artificial Neural Networks for the discovery of constitutive equations of complex materials.

1 Introduction

Fuelled by a continuously increasing flow of data, Machine Learning (ML) has been offering promising solutions to diverse problems in a broad spectrum of disciplines. As a result, science has experienced a shift of paradigm, where data are no longer confined to a mere supporting role, but have instead taken center stage as main protagonists in the scientific narrative. This transformative shift can be considered as the advent of a "*fourth paradigm*" [HTT⁺09], which coexists with and complements the three traditional scientific paradigms: experimental, theoretical, and computational.

Within such a context, the quest for accurate and predictive data-intensive (or datacentric) models has received and continues to receive enormous attention. Such models are often characterized by complex, highly specialized algorithms, e.g. deep neural networks, with numerous interconnected parameters that are optimized to learn those patterns and those correlations proper to a training data set. However, while dataintensive models may fit observations well, many of them face limitations in extracting interpretable information and knowledge from vast amounts of data. In addition, they can suffer from physical inconsistencies and poor generalization [KKL⁺21]. *Consistency* refers here to the fulfilment of laws of physics and thermodynamics. *General*

ization refers to the ability of making predictions for data that do not belong to the particular data set used in the learning process (cf. Chapter 2).

To address these challenges, there has been a surging desire, in the last four-to-five years, to switch the focus from a pure data-centric vision towards a hybrid one that also accounts for first principles – namely, physics and/or thermodynamics. That is, leverage the high expressivity (i.e. approximation properties) of ML models but also embed prior knowledge (or physics-based bias) stemming from our physical understanding of the world in order to reach improved performance, reliability, and interpretability. This gave birth to numerous successful methods, sometimes referred to as *physics-informed machine learning*¹. These approaches enable the enforcement of prior knowledge through the introduction of appropriate *observational, inductive,* and/or *learning biases* [KKL⁺21].

Observational biases use data augmentation procedures by leveraging physical principles that dictate the generation of the latter² [LJP⁺21]. *Inductive biases* resort to the design of specialized neural network architectures that, by construction, embed prior assumptions or physical models, e.g. [HBG⁺21, MSVMB21]. *Learning biases* consists of introducing physical constraints, rather than via specialized architectures, in a soft manner, by appropriately penalizing the loss functions of conventional neural networks, e.g. [RPK19]. For more, we refer to [KKL⁺21].

In this chapter, we focus on two emblematic examples of physics-informed machine learning. The first one consists of the seminal work carried out by Raissi, Perdikaris, and Karniadakis on "Physics-Informed Neural Networks" (PINN) [RPK19]. PINN are a class of supervised deep learning algorithms capable of encoding, through learning biases, physical laws that govern a given data set, and can be described by partial differential equations.

The second example is provided by "Thermodynamics-based Artificial Neural Networks" (TANN) [MSVMB21], that, through the hardwiring of the laws of thermodynamics within the architecture of neural networks (inductive and learning bias), enable the discovery of constitutive equations of complex materials.

After studying this chapter, we hope that the reader will be able to

- Grasp the limitations of machine learning in providing reliable and accurate descriptions of physical phenomena and, at the same time, the need for developing ML approaches accounting for physics/thermodynamics principles.
- Understand, by means of hands-on and pedagogic examples, how to introduce physical knowledge in the form of learning biases (e.g. PINN) and inductive biases (e.g. TANN) to construct high-fidelity physical representations from data.

The codes related to the hands-on example are available on ALERT Geomaterial GitHub (repository alert-geomaterials/2023-doctoral-school), while those

¹There seems to be no strong consensus on the definition of ML models leveraging prior knowledge, mainly due to the fact that this is a very fast moving field. In this Chapter, we follow the broad definition suggested in [KKL⁺21]. Note that this is not the sole possibility, of course.

²For instance, augment the data by leveraging frame indifference and/or symmetries.

related to TANN can be found at github.com/filippo-masi/TANN-multiscale and filippo-masi/Thermodynamics-Neural-Networks.

Updated versions of this chapter are also available at filippo-masi.github.io.

Below, we adopt the following notation: $\mathbf{a} \cdot \mathbf{b} = a_i b_i$, $\mathbf{P} : \mathbf{F} = P_{ij} F_{ij}$, and div $\mathbf{y} = \frac{\partial y_i}{\partial x_j}$, with \mathbf{x} the spatial coordinates, i, j = 1, 2, 3. Einstein's summation is implied for repeated indices.

2 Physics-informed neural networks

Physics-Informed Neural Networks (PINN) are a class of supervised deep learning algorithms capable of encoding physical laws governing given data sets whose evolution can be described by partial differential equations.

The setting considers parametrized and nonlinear partial differential equations (PDE) of the form

$$\dot{\boldsymbol{u}} + \mathcal{N}_{\gamma}[\boldsymbol{u}] = 0, \tag{1}$$

where $\boldsymbol{u} = \hat{\boldsymbol{u}}(\boldsymbol{x},t)$ is the latent (hidden) solution and $\dot{\boldsymbol{u}}$ its partial derivative with respect to time – the superposed caret in $\hat{\boldsymbol{u}}$ serves to distinguish the solution from its values, $\mathcal{N}_{\gamma}[\cdot]$ is a nonlinear differential operator parametrized by γ , with $\boldsymbol{x} \in \Omega \subset \mathbb{R}^n$ the spatial coordinates and t the time coordinate. Note that expression (1) encapsulates a wide range of problems in physics including conservation laws, diffusion processes, advection-diffusion-reaction systems, and kinetic equations. For instance, the heat transfer equation³ can be retrieved by selecting $N_{\gamma}[\boldsymbol{u}] = \gamma \Delta \boldsymbol{u}$, where $\Delta \cdot$ denotes the Laplacian.

2.1 Methodology

By leveraging the high expressivity of neural networks and prescribed physical equations during the learning process – by means of learning biases – PINN provide a powerful and general framework, driven by data, to (i) discover solutions of PDE and/or (ii) the underlying differential equations, by leveraging expression (1).

2.1.1 Solution of partial differential equations

The data-driven solution of PDE aims at finding an approximation of \hat{u} such that (1) holds true for fixed model parameters γ , by means of a deep neural network u_{θ} , parametrized with respect to the network parameters θ – that is, the set of weights and biases of each layer (see Chapter 7).

In doing so, the expression (1) is reformulated as

$$\varrho(\boldsymbol{x},t) \equiv \dot{\boldsymbol{u}}_{\boldsymbol{\theta}} + \mathcal{N}[\boldsymbol{u}_{\boldsymbol{\theta}}], \qquad (2)$$

 $^{{}^{3}\}dot{\boldsymbol{u}} = \alpha \Delta \boldsymbol{u}$, where α is the thermal diffusivity.

where $\rho(x, t)$ is a residual. The residual is computed by applying the chain rule for differentiating compositions of functions using automatic differentiation (cf. [BPRS18] and Chapters 7 and 8), see Figure 1. The learning process consists of the minimization of a loss function composed of a purely data-driven term and a physics-based one, namely

$$\mathcal{L} = \lambda_u \mathcal{L}_u + \lambda_{\text{PDE}} \mathcal{L}_{\text{PDE}},\tag{3}$$

where λ_u and λ_{PDE} are weighting parameters that can be tuned to balance the interplay between the two losses. \mathcal{L}_u and \mathcal{L}_{PDE} are, respectively, a supervised loss related to the (data) measurements u from the initial and boundary conditions and an unsupervised loss of the PDE, namely

$$\mathcal{L}_{u} = \frac{1}{N_{u}} \sum_{i=1}^{N_{u}} \left\| \boldsymbol{u}_{\boldsymbol{\theta}} \left(\boldsymbol{x}^{(i)}, t^{(i)} \right) - \boldsymbol{u}^{(i)} \right\|$$

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_{\text{PDE}}} \sum_{k=1}^{N_{\text{PDE}}} \left\| r \left(\boldsymbol{x}^{(k)}, t^{(k)} \right) \right\|,$$
(4)

 $\|\cdot\|$ being a metric based on an arbitrary norm (cf. Chapter 2). Here $\{(x^{(i)}, t^{(i)})\}$ denote the initial and boundary training data on $\hat{u}(x, t)$, while $\{(x^{(k)}, t^{(k)})\}$ specify the set of collocation points sampled in the entire domain for $\varrho(x, t)$.



Figure 1: NN versus PINN: learning the projectile motion from

2.1.2 Discovery of partial differential equations

In the presence of noisy and/or incomplete measurements u of the state of the system, PINN can be deployed for the data-driven discovery of PDE by means a of neural network u_{θ} and learning model parameters γ that best describe the measurement data. The expression (1) is reformulated as

$$\varrho(\boldsymbol{x},t) \equiv \dot{\boldsymbol{u}}_{\boldsymbol{\theta}} + \mathcal{N}_{\gamma}[\boldsymbol{u}_{\boldsymbol{\theta}}].$$
⁽⁵⁾

Exactly as for the discovery of the solution of PDE, the network is trained over the minimization of the loss (3), where the unknown parameters γ are part of the optimization problem.
While there is no theoretical guarantee of convergence to a global minimum of the loss function (3), many benchmarks and applications, e.g. [RPK19], demonstrated that PINN can achieve accurate predictions when applied to well-posed PDE with unique solutions, both in terms of data-driven discovery of solutions and of PDE. PINN are nowadays used to solve not only partial differential equations, but also fractional equations, integral-differential equations, and stochastic PDE, see [CDCG⁺22] for an extensive review.

The underlying success depends on accounting for physical biases, employing a sufficiently expressive neural network architecture, and an adequate number of collocation points (N_{PDE}) – being the sampling choice a crucial aspect, cf. [LP21].

2.2 Hands-on example

Let us consider as an example the (one-dimensional) motion of a projectile, governed by the ordinary differential equation (ODE) $\ddot{u}(t) + g = 0$, that has the following general solution

$$u(t) = u(0) + \dot{u}(0)t - \frac{1}{2}gt^2,$$

where u, \dot{u} , and \ddot{u} are the trajectory, velocity, and acceleration of the projectile, respectively; t is the time, and g is the gravitational acceleration.

This example should ring a bell to those who read Chapter 2, on regression methods. There, we considered exactly the same problem but addressed it with regularized polynomial regression. Here, we aim at learning the same governing equation but instead by using traditional artificial neural networks and physics-informed ones.

We proceed by reusing the data set generated in Chapter 2, Section 7, and add a Gaussian measurement noise on the data representing the trajectory, $u^{(i)}$ – following the formalism introduced above. In doing so, we use the code hereinafter:

```
t_i = X[0:50:2].copy() # from Chapter 2, Sect. 7
u_i = y[0:50:2].copy() # from Chapter 2, Sect. 7
u_i += 0.1 * np.multiply(np.random.normal(0,1,25), u_i) # add noise
u_i[0] = 0.0 # noise-free initial condition
t_i = torch.tensor(t_i)[:,None] # expand dim
u_i = torch.tensor(u_i)[:,None] # expand dim
```

Figure 2 displays the exact solution of the ODE for $t \in [0, 1]$ s and the noisy data measurements $u^{(i)}$ used for training, $t \in [0, 0.5]$ s.

Then, we continue by defining a neural network (in PyTorch), that we will train (i) in the traditional way and (ii) following the general setting provided by PINN. The code below serves to construct a general neural network architecture with inputs of dimension i_dim, outputs of dimension o_dim, hidden layers with h_dim nodes, and Tanh activations:

import numpy as np
import torch
torch.manual_seed(123)



Figure 2: Projectile motion: exact solution of the ordinary differential equation $\ddot{u} + q = 0$ and noisy data measurements $u^{(i)}$ used for training, $t \in [0, 0.5]$ s.

```
np.random.seed(123)
class NeuralNet(torch.nn.Module):
     '''Feed-forward neural network'''
    def __init__(self,hyper_params,dtype=torch.float32):
        super(NeuralNet, self).__init__()
        self.dtype = dtype
        self.NN = self.constructor(hyper_params) # neural net
        self.gamma = torch.nn.Parameter(torch.tensor(1.)) # gamma in PINN
    def constructor(self, hyper_params):
         '''Construct neural net''
        i_dim,o_dim,h_dim = hyper_params
        dim = i_dim
        layers = torch.nn.Sequential()
        for hdim in h_dim:
            layers.append(torch.nn.Linear(dim, hdim, dtype=self.dtype))
            layers.append(torch.nn.Tanh())
            dim = hdim
        layers.append(torch.nn.Linear(dim, o_dim, dtype=self.dtype))
        return layers
    def forward(self, x):
         '''Forward pass
        return self.NN(x)
```

In order to train the neural network, without any learning bias, we proceed as it follows:

```
dtype=torch.float32
NN_params = [1,1,[12,12]] # [i_dim, o_dim, [h_dim, ...]]
model = NeuralNet(NN_params,dtype) # build neural net
optimizer = torch.optim.Adam(model.parameters(),lr=1e-2) # Adam optimizer
verbose_training = 1000 # display loss
n_epochs = 10000 # number of epochs
for i in range(n_epochs):
    optimizer.zero_grad()
    u_i_pred = model(t_i) # make predictions
    MSE_u = torch.mean((u_i_pred-u_i)**2) # MSE_u
    MSE_u.backward() # backpropagate loss
    optimizer.step()
    if i % verbose_training == 0:
        print('epoch:{:5.0f}'.format(i),
        '- train_loss:{:8.6f}'.format(MSE_u.item()))
```

Note that the architecture is composed of two hidden layers, with 12 nodes each. The evolution of the predictions of the network, at varying of the number of epochs, is shown in Figure 3(a). Without any surprise, the neural network (progressively) massively overfits the training data, and the associated noise. As a result, it cannot neither generalize nor extrapolate. Indeed, we can notice that the predicted "projectile" seems to find itself amidst a tempestuous whirlwind, in the training range ($t \in [0, 0.5]$ s), while it *magically* levitates, defying common sense and the laws of physics (i.e., gravity), in the extrapolation range (t > 0.5 s).

Let us see, at this point, what happens if we insert prior knowledge based on physics. We train the PINN analog of the same neural network and in order to discover the underlying governing equation, we assume the following physical model:

$$\varrho(t) \equiv \ddot{u}_{\theta} + \gamma,$$

where γ is a trainable parameter – in the code snippet above, NeuralNet.gamma. Reformulating, we are telling the network that the second order derivative of the solution is equal to an undetermined constant, that the network will eventually identify. During the learning process, we sample uniformly distributed time (collocation) points $t^{(k)} \in [0, 1]$ s. Automatic differentiation is used to compute \ddot{u}_{θ} relying on PyTorch's functionality torch.autograd.grad():

```
t_k = torch.linspace(0,1,20).view(-1,1).requires_grad_(True) # collocation pt
model = NeuralNet(NN_params,dtype) # build PINN
optimizer = torch.optim.Adam(model.parameters(),lr=1e-2) # Adam optimizer
verbose_training = 1000 # display loss
n_epochs = 10000 # number of epochs
lambda_PDE = 0.2 # weighting parameter
for i in range(n_epochs):
    optimizer.zero_grad()
    u_i_pred = model(t_i) # make predictions for ti
    MSE_u = torch.mean((u_i_pred-u_i)**2) # MSE_u
    u_k_pred = model(t_k) # make predictions for tk
    dudt = torch.autograd.grad(u_k_pred, t_k, torch.ones_like(u_k_pred),
    create_graph=True)[0] # computes du/dt
    dduddt = torch.autograd.grad(dudt, t_k, torch.ones_like(dudt),
    create_graph=True)[0] # computes d^2u/dt^2
    physics = dduddt+model.gamma # computes the residual
    MSE_PDE = lambda_PDE*torch.mean(physics**2) # MSE_PDE
    loss = MSE_u + MSE_PDE # add two loss terms together
    loss.backward() # backpropagate joint loss
    optimizer.step()
    if i % verbose_training == 0:
        print('epoch:{:5.0f}'.format(i),
         - train_loss:{:8.6f}'.format(MSE_u.item()))
```

Figure 3(b) displays the evolution of the predictions of the physics-informed network, as the training advances: much better, isn't it?

Thanks to the (physical) learning bias we introduced, the physics-informed network not only possesses good generalization capabilities, but is also able to extrapolate⁴

⁴A disclaimer is necessary concerning the capability of the trained PINN model to extrapolate. In general, there is no guarantee, as for all ML models, that the network enables extrapolation. In this example,

with high accuracy. By evaluating, at the end of the learning process, the γ parameter, we can indeed notice that the network finds a good approximation of the gravitational acceleration, namely gamma = 10.58 m/s².



Figure 3: Learning the projectile motion from noisy measurement data using a traditional neural network (NN) (a) and a physics-informed neural network (PINN) (b).

Before concluding this example, it is worth drawing an interesting analogy between the addition of physical biases and regularization strategies in ML. In particular, we have seen in Chapter 2, paragraph 7.1, how regularized regression models based on the minimization of the ℓ_1 and ℓ_2 norm can provide similar accuracies compared to the physics-informed neural network we have deployed for predicting the projectile motion. It is important to understand that such a similarity is not simply a coincidence. To fix the ideas, let us evaluate, in Figure 4, the evolution of the ℓ_1 and ℓ_2 norms of the parameters of the traditional neural network and its physics-informed sibling. The latter is characterized by a much smaller norm compared to the traditional neural network – with a 70% decrease. This means that the physical bias acts as a sort of regularizer, shrinking the value of most of the network parameters θ to zero. The opposite is also true: the benefits of regularization techniques lie in the fact that they enable the discovery of simple models – which, most of the time, are more physical

extrapolation in the domain $t \in [0, 1]$ s is made possible thanks to the adopted sampling strategy for the collocation points, i.e. $t^{(k)} \in [0, 1]$ s.

than very sophisticated ones. In an essence, both techniques (physics and regularization) allow for the implementation of a "*lex parsimoniae*" (see Chapter 2, Section 7, and [KB22]).



Figure 4: Evolution, at training, of the ℓ_1 and ℓ_2 norm (scaled with respect to the maximum value) of the neural network parameters with and without learning biases based on physics, respectively PINN and NN.

3 Thermodynamics-based neural networks

Accurate models for the behavior of materials are of fundamental importance in material science and (geo-)mechanics. However, heuristic constitutive models, traditionally derived from first principles (thermodynamics) and empirical approaches (to ensure calibration over experiments), can hardly describe the behavior of all complex materials that display path-dependency and possess multiple inherent scales. In such contexts, multiscale approaches are commonly preferred as they enable to capture the effects of the fine material scales (here denoted with *microstructure*) on the average (*macroscopic*) behavior. However, the bottleneck of these approaches lies in the cumbersome and time-consuming ways with which the auxiliary problem is being solved which eventually hinders their application in real-case scenarios.

In recent years, ML, particularly deep learning, has been offering new ways to reduce the computational burden of multiscale approaches. Indeed, taking advantage of the high expressivity of neural networks (see Chapters 7 and 8), deep learning can enable the modeling of several aspects of the behavior of complex materials and show promise in building high-fidelity replicas or *digital twins*⁵.

Yet, constitutive modeling based on neural networks has been facing two major issues

⁵Let us define the concepts of *virtual* and *digital twins*. We do so by adopting the definitions proposed in [CCAC⁺20]. A *virtual twin* refers to the emulation of a physical system by one or more mathematical models that describe the behavior of the former. To fix the ideas, numerical models, such the Finite Element Method (FEM) or the Discrete Element Method (DEM), are examples of virtual twins. A *digital twin* – term originally coined by General Electric to depict the digital replica of an engine produced in their manufacturing facilities – is a data-intensive representation (or model) of the same physical system. As opposite to its virtual siblings, digital twins are usually characterized by reduced computational complexity and can be deployed for real-time decision-making.

- inconsistency and poor generalization, cf. Section 1 – that hinder the applicability of classical "black-box" (i.e., physics-agnostic) approaches [GS98, LS03, LBS09].

To this, new developments were recently proposed to tackle the lack of physical consistency in neural networks and the consequent constitutive models. In particular, it has been demonstrated that is possible to structure neural networks in a way that they deliver, though inductive and/or learning biases (cf. Section 1), thermodynamics-/physics-consistent constitutive representations, see e.g. [HBG⁺21, YZYK22, KFM⁺22]. Among these approaches, the Thermodynamics-based Artificial Neural Networks (TANN, [MSVMB21, MS22, MS23]), based on the theory of internal state variables, are able to uncover constitutive equations from the laws of thermodynamics. As a direct consequence of the thermodynamically-consistent ar-chitecture, TANN deliver accurate predictions of the material behavior and show remarkable generalization abilities in presence of unseen loading paths, both within the range of the training set (interpolation) and outside (extrapolation).

Herein, we focus on presenting the methodology and the underlying theoretical framework of TANN. Through a pedagogical, example we illustrate how thermodynamicsbased neural networks enable to construct accurate, robust, and reliable digital twins/replicas of granular media. This approach not only captures the overall macroscopic material behavior but also allows to discover the (hidden) internal variables and to characterize the time evolution of the microstructure at extremely reduced computational cost. Finally, we demonstrate how to deploy the same framework for speedingup state-of-the-art multi- and fine-scale simulations.

3.1 Theoretical framework

We start by briefly recalling the general continuum thermodynamic setting. The local form of the energy balance and dissipation rate inequality, at the level of an arbitrary unit volume element⁶ \mathcal{V} and with respect to the reference configuration, read

$$\dot{e} + \operatorname{div} \boldsymbol{q} = \boldsymbol{P} : \boldsymbol{F} + \boldsymbol{r}, \tag{6}$$

$$d = \dot{\eta} - \left(T^{-1}\boldsymbol{r} - \operatorname{div}\left(T^{-1}\boldsymbol{q}\right)\right) \ge 0,\tag{7}$$

where e and \dot{e} are the volume density of the internal energy and its local time derivative; P is the first Piola-Kirchhoff stress tensor; F is the deformation gradient and \dot{F} its rate of change; q is the heat flux vector; r is the volume density of possible external source terms; d is the volume density of the total (mechanical plus thermal) dissipation rate; η the volume density of entropy; and T the absolute temperature. Note that all above quantities are functions of the time t, i.e. $F \equiv F(t)$, but for the sake of simplicity, we voluntarily omitted, and continue to do so, such explicit way of writing.

⁶The notion of unit volume element or unit cell, stems from considerations related to the mathematical theory of asymptotic homogenization. All quantities of interest, except when explicitly written, are herein expressed in the form of volume averages, e.g. $\psi = \frac{1}{|\mathcal{V}|} \int_{\mathcal{V}} \psi(\boldsymbol{x}) d\boldsymbol{x}$, where $\boldsymbol{x} \in \mathcal{V}$ are the spatial coordinates and $|\mathcal{V}|$ is the volume. For more, we refer to [Mie02, MS22].

From the combination of the above two expressions – that is, the first and second law of thermodynamics – we obtain the Clausius-Duhem inequality, namely

$$\begin{aligned} d &= \boldsymbol{P} : \dot{\boldsymbol{F}} - \dot{\boldsymbol{e}} + T\dot{\eta} - T^{-1}\boldsymbol{q} \cdot \boldsymbol{m} \\ &= \boldsymbol{P} : \dot{\boldsymbol{F}} - \left(\dot{\psi} + \dot{T}\eta\right) - T^{-1}\boldsymbol{q} \cdot \boldsymbol{m} \ge 0 \end{aligned}$$
(8)

the latter involving the Helmholtz free energy density (per unit volume), $\psi = e - Ts$, and denoting with m the temperature gradient. Whenever m = 0, the Clausius-Duhem inequality becomes

$$d = \mathbf{P} : \dot{\mathbf{F}} - \dot{\psi} - \dot{T}\eta \ge 0, \tag{9}$$

which has to hold at each time t, and where d is the internal (i.e., mechanical) dissipation rate.

Relying on the aforementioned expression, we derive constitutive restrictions on material processes. In particular, we follow the seminal work of Coleman and Gurtin [CG67] and introduce the internal state variables z_j , with $j = 1, 2, ..., n_z$ – where n_z is the number of internal state variables. Note that the internal state variables or, internal variables for short, are introduced to account for the influence of (dissipative) micromechanisms on the constitutive behavior of materials. At this point, we continue by assuming the free energy density as a function of the internal variables and the state variables, F and T – that is,

$$\psi = \psi \left(T, \boldsymbol{F}, \boldsymbol{z} \right), \tag{10}$$

where the superposed caret in $\hat{\psi}$ serves to distinguish the free energy function from its values and z denotes the internal state vector $z = (z_1, z_2, \dots, z_{n_z})$. Notice that, alternatively, one may consider, rather than F, its elastic part in Eq. (10), see e.g. [Rub01, Ein12, Daf22]. However, here we disregard such a choice as we aim at developing a constitutive modeling approach that is independent of the requirement of the decomposition of the deformation gradient, or its rate of change, into an elastic and a plastic part.

It should also be mentioned that, for a given material, the free-energy function is not unique for a given deformation gradient (for example, the free energy values ψ and $\psi + c$, with c a constant, will give the same stress-strain response and dissipation rate). From Equation (10), it follows that

$$\dot{\psi} = \partial_{F} \hat{\psi} : \dot{F} + \partial_{T} \hat{\psi} \dot{T} + \partial_{z} \hat{\psi} \cdot \dot{z}, \qquad (11)$$

where $\partial_F \hat{\psi} = \frac{\partial \hat{\psi}}{\partial F}$. We continue by substituting expression (11) into the Clausius-Duhem inequality (9), and obtain the following expression for the internal dissipation rate

$$(\boldsymbol{P} - \partial_{\boldsymbol{F}}\hat{\psi}) : \dot{\boldsymbol{F}} - (\eta + \partial_{T}\hat{\psi})\dot{T} - \partial_{\boldsymbol{z}}\hat{\psi}(T, \boldsymbol{F}, \boldsymbol{z}) \cdot \dot{\boldsymbol{z}} - d = 0, \quad \forall \ \dot{T}, \dot{\boldsymbol{F}}, \dot{\boldsymbol{z}}$$
(12)

with $-\partial_{\boldsymbol{z}}\hat{\psi}(T, \boldsymbol{F}, \boldsymbol{z})$ being the thermodynamic forces. It follows that, for satisfying equality (12) for any $\dot{T}, \dot{\boldsymbol{F}}, \dot{\boldsymbol{z}}$, every admissible thermodynamic process must satisfy

the following constitutive restrictions hereinafter, holding true at any time t,

$$\boldsymbol{P} = \partial_{\boldsymbol{F}} \hat{\psi} \left(T, \boldsymbol{F}, \boldsymbol{z} \right), \tag{13}$$

$$\eta = -\partial_T \hat{\psi} \left(T, \boldsymbol{F}, \boldsymbol{z} \right), \tag{14}$$

$$d = -\partial_{\boldsymbol{z}}\hat{\psi}\left(T, \boldsymbol{F}, \boldsymbol{z}\right) \cdot \dot{\boldsymbol{z}} \ge 0.$$
(15)

A last ingredient allows the closure of the constitutive relationship: the evolution equation for the internal variables, based on (10) and (12-15) takes the form

$$\dot{\boldsymbol{z}} = \boldsymbol{f}\left(T, \boldsymbol{F}, \boldsymbol{z}\right) \tag{16}$$

such that the dissipation inequality (15) holds true. The above equation can cover a wide variety of constitutive models, including rate-dependent and rate-independent plasticity theory through the introduction of Lagrange multipliers (see [EHN07]). Note that, in some particular cases, e.g. when postulating the existence of a dissipation potential, analytical expressions for f that satisfy the dissipation inequality by construction, can be retrieved [MM94, EHN07]. Similarly, evolution equations could be formulated as Onsagerian conductivity equations satisfying the dissipation inequality (see [Gur96, Ván03, EL18], among others). Yet, the above structure does not originate directly from thermodynamic considerations and may not be general enough, as it requires ad-hoc assumptions on the form of the internal variables rates and the thermodynamic forces. Thus, in the following, we opt for the more general expression (16).

3.1.1 The quest for internal state variables

The aforementioned thermodynamics framework builds upon the knowledge of the internal variables, *z*. And, whilst TANN can effectively leverage any a priori identified set of internal variables (as we will see in the next paragraph), the key question is: *how can we identify internal state variables*? Traditionally, to determine the number and nature of internal variables, we must identify the internal mechanisms and phenomena that influence the behavior of the particular material at hand. This approach, referred to as the *Art of modelling* [Mau15], requires adaptation to each specific application and material and is hardly scalable. To face this difficulty, there has been a recent interest for devising approaches that automatically reveal – or discover – the internal variables independent of the type of the material, see e.g. [VBE08, HBG⁺21, MS22].

Herein, we adopt the approach developed in [MS22], where internal variables are identified based on the knowledge of the internal degrees of freedom of the microstructure of the material under investigation. To this end, we introduce a new quantity, referred to as *internal coordinates* $\boldsymbol{\xi}$. These internal coordinates describe the material behavior at the microscopic scale⁷ and encompass variables such as displacement, velocity, momentum fields, and internal force networks.

⁷Herein, we adopt the notion of microscopic scale as the finest scale of a material. Depending on the degree of fidelity one desires to achieve, *microscopic* may refer, for instance, to the scale of the grains composing a granular material or the atoms composing each grain.

According to this formalism, the identification of internal variables passes from learning low-rank representations of the internal coordinates by identifying an operator h, such that

$$\boldsymbol{z} \equiv \boldsymbol{h}(\boldsymbol{\xi}) \quad \text{and} \quad \boldsymbol{\psi} = \hat{\boldsymbol{\psi}}(T, \boldsymbol{F}, \boldsymbol{z}).$$
 (17)

In parallel and without any loss of generality, a pseudoinverse operator g can also be postulated, such that

$$\hat{\boldsymbol{\xi}} = \boldsymbol{g}\left(\boldsymbol{h}\left(\boldsymbol{\xi}\right)\right) = \boldsymbol{g}\left(\boldsymbol{z}\right),$$
(18)

where the superposed tilde serves to distinguish a low-rank approximation of the internal coordinates from the internal coordinates themselves. To rephrase, the adopted framework implies that the determination of the internal variables boils down to identifying the operator h – and, additionally, the operator g – from the knowledge of the microstructural internal degrees of freedom of a material and without any a priori constitutive choice⁸.

3.2 Methodology

The thermodynamic framework presented above can be employed in the architecture of neural networks to learn physical and generalizable constitutive models. Such neural networks are referred to as Thermodynamics-based Artificial Neural Networks (TANN) [MS22]. TANN can be declined into two configurations, depending whether the internal state variables are priori known [MSVMB21] or not [MS22]. Both differential discrete-time and continuous-time formulations can be employed [MS23].

Herein, we briefly recall the main building blocks of the network (for more, we refer to [MS23]). TANN are composed of two building blocks: the free energy density network and the evolution equation network. The free energy network, shown in Figure 5, is responsible of the prediction of the material stress, from the fulfilment (inductive bias) of the thermodynamics restrictions (13-15). It consists of one network trained to predict the value of the free energy density – that is, $\psi = \hat{\psi}_{\theta}(T, F, z)$, parametrized by the neural networks parameters θ . Material stress, entropy, and internal dissipation rate are computed, using the constitutive restrictions (13-15), by relying on the automatic differentiation of the operator $\hat{\psi}_{\theta}$ with respect to its inputs. The free energy



Figure 5: Thermodynamics block. All quantities refer at time t.

⁸It is worth noticing that we only consider a finite number of internal variables. Whilst this is the case for all constitutive models developed so far, one may question whether the state space of a real, non-idealized, material can be proven to be finite or not.

network is trained by minimizing a loss composed of three contributions, and namely

$$\mathcal{L}_{\text{energy}} \equiv \mathcal{L}_{\psi} + \mathcal{L}_{\nabla\psi} + \lambda_{\text{reg}} \mathcal{L}_{\text{reg}}, \qquad (19)$$

where λ_{reg} is a weighting parameter. The three terms ensure, respectively, that (i) the neural network can predict the free energy density, (ii) the gradients of the latter coincide with the material stresses, entropy density, and intrinsic dissipation rate, and (iii) the fulfilment (in the form of a learning bias) of the dissipation inequality, namely

$$\mathcal{L}_{\psi} \equiv \left\| \psi - \hat{\psi}_{\theta} \left(T, F, z \right) \right\|,$$

$$\mathcal{L}_{\nabla \psi} \equiv \left\| P - \partial_{F} \hat{\psi}_{\theta} \left(T, F, z \right) \right\| + \left\| \eta + \partial_{T} \hat{\psi}_{\theta} \left(T, F, z \right) \right\| + \left\| d + \partial_{z} \hat{\psi}_{\theta} \left(T, F, z \right) \cdot \dot{z} \right\|$$

$$\mathcal{L}_{\text{reg}} \equiv \left\| \left[\partial_{z} \hat{\psi}_{\theta} \left(T, F, z \right) \cdot \dot{z} \right] \right\|,$$
(20)

with $\|\cdot\|$ being an error metric based on an arbitrary norm (cf. Chapter 2), averaged over all data points, and $[\cdot]$ being the Macaulay brackets. Note that the loss associated with the free energy density, \mathcal{L}_{ψ} , is not a necessary condition for the fulfilment of the laws of thermodynamics and the accurate prediction of the material response. The same holds true for the loss associated with the dissipation rate, which can be omitted in those cases where there is no information related to the values of the dissipation (cf. paragraph 3.3). Indeed, it can be easily proved that, in absence of the aforementioned loss terms, the free energy density network, trained only through its gradients, will still respect the restrictions (13-15), because they are hardwired in the network architecture.

The evolution equation network, shown in Figure 6, is responsible for learning the evolution equations of the internal variables by means of a neural network $f_{\theta}(T, F, z)$. The training is performed by minimizing the error between the outputs and the rates of change of the internal variables,

$$\mathcal{L}_{\dot{\boldsymbol{z}}} = \| \dot{\boldsymbol{z}} - \boldsymbol{f}_{\boldsymbol{\theta}} \left(T, \boldsymbol{F}, \boldsymbol{z} \right) \|.$$
(21)

The above evolution equation can be identified either relying on the aforementioned (time-continuous) formulation or, rather, by relying on a discrete-time, incremental one.



Figure 6: Evolution law block for a priori determined internal variables. All quantities refer at time t.

3.2.1 Data-driven identification of internal variables

To the aforementioned building blocks, a third one adds whenever no constitutive assumptions are made on the nature of the internal variables. At the heart of the identification of the internal variables and their evolution equations lies the search for an appropriate form of the functions g and h^9 . Here, we identify the latter as the neural networks of a standard autoencoder¹⁰, being the operator $h_{\theta}(\xi)$ the encoder and the operator $g_{\theta}(z)$ the decoder.

While autoencoders allow the reconstruction of the internal coordinates and their rates, a full reconstruction may not be needed to characterize the material response (see paragraph 3.3). Alternatively, one may prefer other dimensionality reduction techniques, e.g. principal component analysis, among others [Gér19, PMSJ22]. The proposed approach is general and independent on the particular choice made to identify the latent representations of the internal coordinates.

Figure 7 shows the architecture for the data-driven identification of the internal variables and evolution equations. The architecture is composed of an autoencoder, the free energy density network, and the evolution equation network. The identification of the internal variables is driven by the minimization of the reconstruction loss,

$$\mathcal{L}_{\text{recon}} = \left\| \boldsymbol{\xi} - \boldsymbol{g}_{\boldsymbol{\theta}} \left(\boldsymbol{h}_{\boldsymbol{\theta}} \left(\boldsymbol{\xi} \right) \right) \right\|, \qquad (22)$$

which ensures that the autoencoder can reconstruct the internal coordinates from the latent representations. Note that, at the end of the training, these latent representations, $h_{\theta}(\xi)$, will coincide with the internal variables, i.e., $z \equiv h_{\theta}(\xi)$. The identification of the evolution equations follows depending on whether a discrete- or continuous-time formulation is adopted, see [MS23].

It is worth noticing that, the training of the networks composing TANN can also be performed individually. In this case, the autoencoder and the free energy density network can be first trained together, by minimization of the weighted sum of the losses $\mathcal{L}_{recon}, \mathcal{L}_{\psi}, \mathcal{L}_{\nabla\psi}$, and \mathcal{L}_{reg} , then, the evolution equation network is trained by minimization of the loss $\mathcal{L}_{\dot{z}}$.

3.2.2 Inference

After having trained TANN, they can be used at inference as classical constitutive models that can further be deployed in displacement-based formulations, where loading paths are expressed in terms of strain and temperature loads, i.e.

$$\boldsymbol{P}(t) = \mathbf{TANN}_{\boldsymbol{\theta}} (T(t), \boldsymbol{F}(t), \boldsymbol{z}(t)) \quad \forall t,$$
(23)

⁹We implicitly assume the existence of such functions, but this might not be always the case (see differential inclusions.

¹⁰In their simplest form, autoencoders are unsupervised learning algorithms that map inputs to latent representations of minimum dimensionality and then back to themselves. Given an input $\boldsymbol{\xi} \in \mathbb{R}^n$, we want to learn a latent representation $\boldsymbol{z} \in \mathbb{R}^l$ – where $l \ll n$ – which is mapped back into $\tilde{\boldsymbol{\xi}} \in \mathbb{R}^n$ by minimizing $\|\boldsymbol{\xi} - \tilde{\boldsymbol{\xi}}\|$. This parametrization is implemented by two functions: an encoder that maps $\boldsymbol{\xi}$ into \mathbb{R}^l and a decoder that performs the opposite transformation.



Figure 7: Evolution TANN for the data-driven identification of internal variables and governing equations. All quantities refer at time t. The internal variables are identified as the latent representations of the encoder, i.e., $z \equiv h_{\theta}(\xi)$. The internal variables rate can be computed either in terms of finite-difference or by means of the automatic differentiation of the encoder with respect to the internal coordinates, i.e. $\dot{z} = \partial_{\xi} h_{\theta}(\xi) \cdot \dot{\xi}$.

where the time evolution of the internal variables is computed by solving the following initial value problem

$$\dot{\boldsymbol{z}}(t) = \boldsymbol{f}_{\boldsymbol{\theta}} \big(T(t), \boldsymbol{F}(t), \boldsymbol{z}(t) \big), \quad \boldsymbol{z}(t_0) = \boldsymbol{z}_0.$$
(24)

The latter equation can be solved using the artillery of well-known and studied numerical integration methods. Additionally, an alternative path consists of directly learning the integral form of the evolution equations [ME23].

It is also worth noticing that, in the case where one opts for a stress-based formulation, the proposed approach and formalism of TANN can still be used, but selecting the Gibbs free energy density instead of the Helmholtz free energy density.

For the case of a priori selected internal variables, only the evolution equation and free energy networks are needed. Instead, in the case where the internal variables are discovered, these are conclusively identified as the latent representations of the encoder, which is then removed completely from the architecture. However, the decoder may be kept to map z back to the internal coordinates, i.e. $\tilde{\xi}(t) = g_{\theta}(z(t))$, if the particular application at hands requires it.

In summary, TANN enable robust predictions of material responses, even for unseen data and in the presence of noise, thanks to the hardwiring of thermodynamic principles. Extensive studies on the benefits of a thermodynamics-based approach against a purely data-driven one can be found in [MSVMB21]. For the wide applicability

of the framework, we refer to [MS23] where TANN are demonstrated to accurately model a broad spectrum of complex material behaviors, from plasticity to damage and viscosity (and combination of them).

3.3 Digital twins of granular materials: a pedagogic example

Let us investigate, in the form of an example, how the framework of thermodynamicsbased neural networks can be used to build high-fidelity thermodynamics-based digital twins of a granular system. To this end, we rely on high-fidelity replicas of the latter (i.e., a virtual twin) to generate virtual experiments that will be used in the learning process.

3.3.1 Virtual twin

The reference granular material is modeled by infinitely rigid, spherical discrete particles, relying on the Discrete Element Method (DEM). The particles interact with each others through inter-particle constitutive laws of friction and their motion is governed by Newton second law of motion. The numerical analyses are conducted using the open-source platform YADE-Open DEM [KD08, SAC+21] where the equations of motion are integrated in time explicitly, using a central finite difference approximation algorithm.

Herein, we consider a purely frictional (cohesionless) granular medium, where the grains interact with each other through normal and tangential forces. The grains have a mean diameter $d_{50} = 200 \,\mu\text{m}$, drawn from a uniform distribution $d_{50} \,\mathcal{U}(1 - 0.0707, 1 + 0.0707)$, and are composed of an isotropic elastic material with density $\rho = 1800 \,\text{kg/m}^3$, Young modulus $E = 300 \,\text{MPa}$, and poisson ratio $\nu = 0.3$. The grains interact through Coulomb friction interfaces, with friction angle $\varphi = 30^{\circ}$.

The granular packing is first subjected to consolidation to match prescribed stress conditions and then undergoes monotonous and cyclic drained triaxial compression. Accordingly, we first generate a periodic specimen by randomly depositing spherical particles. At the beginning, the material is prescribed with a frictional angle equal to 0° in order to obtain an ultimate dense packing. We proceed with an isotropic compaction of the packing to achieve a dense state and a consolidation to reach the desired homogeneous volumetric stress level, p = 100 kPa, and zero deviatoric stress, q = 0 kPa. The former is defined as $p = \frac{1}{3}\sigma_{ii}$, while the latter as the invariant of the deviatoric stress tensor, $\sigma'_{ij} = \sigma_{ij} - p$, namely $q = \sqrt{2\sigma'_{ij}\sigma'_{ij}} - \text{with } i, j = 1, 2, 3$. The final material properties are then assigned to the particles, along with their existing interactions. At this point, the packing is subjected to isotropic extension to generate states at different confining pressures, i.e., $p \in (20, 40, 60, 80, 100)$ kPa.

Virtual experiments Using the consolidated granular packing obtained following the aforementioned protocol, we perform virtual experimental tests to generate data necessary for training, validation, and testing of TANN, and namely:

- five monotonous drained triaxial compression tests with initial confining pressures p ∈ (20, 40, 60, 80, 100) kPa up to a total axial deformation approximately equal to 20%, with constant strain rate equal to 125 s⁻¹;
- 10 cyclic drained triaxial compression tests with initial confining pressure equal to 100 kPa, where the strain rates are assumed piecewise constant, in time, equal to $\pm 125 \text{ s}^{-1}$ and of random direction.

To simulate these conditions, we implement the model by setting $\dot{\varepsilon}_{11}$ and determining, through a servo-controller, $\dot{\varepsilon}_v = \dot{\varepsilon}_{ii}$ at each timestep to satisfy $\dot{p} = \dot{q}/3$, see Figure 8. The inertia number $I = \dot{\gamma} d_{50} \sqrt{\rho/p_c}$, with $\dot{\gamma}$ the shearing rate, is approximatively equal to 10^{-5} in all tests. Accordingly, the material is in a quasi-static regime, with negligible strain-rate effects [MiD04, RD11].

Stochastic representation Granular systems are characterized by inherent spatial heterogeneity of the material properties and of the microstructure topology. As a result, the determination of the size of a representative volume element is not trivial due to the subsequent topological randomness, see [Ngu21, SAS04].

An alternative way consists of resorting to statistical ensemble approaches that leverage stochastic elementary volumes [OS06], containing a sufficient number of particles to be representative and having characteristic size exceeding that of individual grains. Following this framework, analyses based on the stochastic interpretation of multiple small, periodic stochastic elementary volumes, of characteristic size l, lead, after averaging, to the identification of the representative volume element, with characteristic size L, see Figure 8 and [PSS23]. To determine the number of stochastic elementary volumes necessary for an adequate description of the material response, Monte-Carlo analyses must be performed, cf. [PSS23].



Figure 8: (re-adapted from [PSS23]) The material behavior of a granular system in terms of the representative elementary volume of size L, approximated by a stochastic ensemble of a series of N stochastic elementary volumes of size l, at the level of the microstructure.

Herein, we shall limit ourselves to only four different volumes, with size $l \approx 8d_{50}$, composed of 2^{10} particles – the number of particles is kept at minimum in order to generate data in a short amount of time. The number of stochastic elementary volumes is obviously not enough for having realist representations of the heterogeneous nature of granular systems (as Monte-Carlo analyses would reveal). Yet, the statistical average of the four responses of the DEM models leads to a smooth material behavior, see Figure 9, not dominated by *local stick-slip* motion [PSS23] and representative enough of the response of a granular system within the limits of this pedagogic example.



Figure 9: Monotonous and cyclic drained triaxial compression response (a), (b) of each individual stochastic elementary volume and averaged, where ε_s is the invariant of the deviatoric strain tensor, $\varepsilon'_{ij} = \varepsilon_{ij} - \frac{\varepsilon_v}{3}$, namely $\varepsilon_s = \sqrt{\frac{1}{2}\varepsilon'_{ij}\varepsilon'_{ij}}$. (c) Spatial distribution of the grains diameter in each stochastic volume ($d_{50} = 0.2$ mm).

3.3.2 Digital twin

Relying on the aforementioned generated virtual experiments, we proceed with the training of a digital twin based on TANN, using a discrete-time formulation (cf. [MS22]). In particular, we select as test set one monotonous (at initial confining pressure p = 80 kPa) and one cyclic path, from those previously generated. The remaining data are shuffled and split into training and validation set.

The internal coordinates are selected to be the three-dimensional displacement fields of all grains composing the system (averaged over the stochastic elementary volumes). The free energy density network consists of one hidden layer with GELU activations. The evolution equation network has three hidden layers with GELU activations. While the encoder and decoder networks are composed of two hidden layers and RELU activations.

The learning process is split in two phases. First, the energy network and the autoen-

coder are trained by minimizing the reconstruction error (22), the error on the stress predictions, i.e., $\|\boldsymbol{\sigma} - \partial_{\boldsymbol{\varepsilon}} \hat{\psi}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}, \boldsymbol{h}(\boldsymbol{\xi}))\|$, and the dissipation inequality, rather than the total loss function (19). Such a choice is made to mimic real case scenarios where the energy and dissipation rate values might not be readily available. Second, after having identified the internal variables, according to Eq. (17), we proceed by training the evolution equation network.

It is worth mentioning that the number of internal variables necessary for an accurate reconstruction of the internal coordinates – i.e., the microscopic displacement fields – and the description of the evolution of the material stresses is not a priori known. To this end, we set the number of hidden dimensions (output of the encoder) equal to 2^6 and use an activity regularization based on the ℓ_1 norm to have an accurate low-rank reconstruction $\tilde{\xi}$ with the smallest number of hidden variables. By doing so, we find that the autoencoder can accurately reconstruct the internal coordinate fields with only seven (non-zero) hidden variables – that is, compressing the information of the three-dimensional displacement fields of 2^{10} particles by a factor of approximately $440 (32^{10}/7)$.

Despite such compression, we should note that not all the so-determined hidden variables may be needed for characterizing the material response. Thus, we introduce, in parallel, a penalty based on the ℓ_1 norm for the weights of input layer of the free energy network (cf. Chapter 2). This allows to promote parsimony and identify only those (internal) variables that are effectively needed for predicting the material response. In doing so, we find that only four internal variables are necessary for the stress paths probed in this example.

3.3.3 Results

We show in Figure 10 the evolution of the volume average behavior of the reference granular material and the predictions, at inference, of TANN, for both the monotonous and the cyclic unseen path. The digital twin is found to accurately describe the complex material behavior, correctly accounting for the underlying inelastic phenomena and yielding an accurate representation of the dissipative nature of the system, subjected to multiple loading and unloading.

Particularly interesting is to observe the evolution of the discovered internal variables, shown in Figure 11, for the same monotonous and cyclic test paths. There exist two essentially different behaviors. The first two internal variables are mainly responsible for the description of the irreversible phenomena taking place before the stress peak, see Figure 11(b,e). This is particularly evident for the monotonous path (b), where the variables rapidly evolve until they reach an equilibrium state (post-peak). A similar behavior is also found for the cyclic path (e), where the multiple unloading and (re)loading repeatedly perturb the equilibrium state. The remaining two internal variables, whose evolution is depicted in Figure 11(c,f), display a behavior essentially different from that of the former as they continuously evolve, both in loading and unloading, suggesting that they may represent a macroscopic measure of dissipation mechanisms taking place at the frictional inter-particle contacts.

Note that the physical nature of the discovered internal state variables can be examined though feature extraction methods [LKS20].

Additionally, internal variables can be decoded to obtain the internal coordinates of the microstructure – the displacement fields of the grains. To demonstrate the full capabilities of TANN, we show such reconstruction in Figure 12 for the monotonous test set. The three-dimensional reconstruction of the granular packing in Figure 12(a) is made by using the predicted particles displacements (and accounting for the diameter distribution and initial grains position of one of the four stochastic elementary volumes). Figure 12(b) additionally compares the reference and predicted time evolution of the displacements of some of the grains. The framework allows to deliver accurate descriptions of the material microstructure. Note, again, that the proposed approach does not require to operate (on-the-fly) with the high-dimensional microscopic fields, but only with the identified hidden variables.



Figure 10: Comparison of the predictions of TANN with respect to unseen, test responses obtained from the statistical average of four granular stochastic elementary volumes: monotonous drained triaxial (DT, left) and cyclic (right). (a), (c) $q - \varepsilon_s$ response and (b), (d) p - q response. For the monotonous path, we show both the training and validation sets (train), and test (test) set.

3.4 Speed-up multiscale simulations

We have seen so far how TANN can be deployed to construct reliable and accurate digital twins of complex and intricate materials, thanks to the universal approximation power provided by neural networks and the rigorous theoretical setting offered by



Figure 11: Evolution of the discovered internal variables for the test monotonous (left) and cyclic (right) loading paths: (a,d) reference deviatoric stress path, and (b-c,e-f) discovered internal state variables.

thermodynamics. In addition, the same framework can be used to speed-up, in a reliable manner, fine-scale simulations, relying on a multiscale approach.

Multiscale approaches integrate a number of nested computational methods at various scales, such as the Finite Element Method (FEM) and the Discrete Element Method (DEM), to obtain iterative solutions to boundary value problems at the unit cell of the microstructure (*auxiliary problem*) and, through upscaling, retrieve the effective macroscopic response [Fey03, NCDD11, NCCD14].

To fix the ideas, let us consider a large scale, macroscopic structure, whose microstructure is made of spatially (quasi-)periodic distributions of a representative unit cell \mathcal{V} . We leverage scale separation, i.e., we assume the existence of two independent scales \mathbf{x} and $\mathbf{y} = \mathbf{x}/\epsilon$, with $\epsilon \ll 1$ being the dimension of the unit cell. The first scale, \mathbf{x} , is associated to the macroscale, while \mathbf{y} refers to the scale of the material microstructure. Under such considerations, we can resort to the rigorous mathematical framework of *asymptotic homogenization* [BP12] for scales bridging and, in particular, by leveraging its extension to nonlinear problems by means of an incremental formulation, see [Mie02].

Note that whenever the assumption of scale separation does not hold, for in-



Figure 12: Internal coordinates (particles displacements) from the DEM simulations and as reconstructed from the set of discovered internal variables. (a) Reconstruction for one of the stochastic volume of the granular packing from the knowledge of the displacement fields, radii distribution, and initial position. (b) Comparison of the time evolution of (some of) the grains displacements and the predictions obtained by integrating in time the evolution equation, $f_{\theta}(\cdot)$, and decoding the internal variables, $g_{\theta}(\cdot)$.

stance, in presence of strain localization phenomena, homogenization cannot be used. However, a remedy consists of resorting to higher order continuum theories, see [SSV10, GSS⁺17, Var18] among others.

According to asymptotic homogenization, TANN should be trained to identify the volume average behavior of the unit cell – that is, predict the solution of the auxiliary problem. At this point, we should note that the previous example already considers training data sets related to the unit cell (of a granular material) with periodic boundary conditions. Thus the network can be directly used at inference to perform multiscale analyses. To do so, we rely on the FEM×TANN approach, as developed in [MS22]. In FEM×TANN, we perform Finite Element analyses by a straightforward replacement of classical constitutive models, at the Gauss points, with the digital twin provided by the trained network (cf. asymptotic homogenization). The tangent matrix is computed, at each Gauss integration point, by virtue of the automatic differentiation of TANN, cf. Chapter 7. For a detailed discussion on the computational accelerations available with the FEM×TANN approach, we refer to [MS22].

3.4.1 FEM×TANN: an example

To fix the ideas, let us consider the problem of a panel, fixed on one end, and subjected, quasi-statically, to a shearing load at the other end. Figure 13(a) depicts the initial geometry and boundary conditions. Plane strain conditions are adopted.

Initially, a force equal to 12 N/mm (per unit cell in width) is applied linearly in time until $t_{\text{load}} = 0.9$ ms. The loading time is selected in order to have strain rates of the order of 1 s⁻¹, corresponding to those developed due to impact loading. The force is then maintained over a prescribed time interval, up to t_{steady} , and, finally, the structure is unloaded gradually within a time interval equal to that of the loading phase, see Figure 13(e). To capture relaxation effects due to the viscous behavior of the microstructure, the unloaded phase is maintained for an additional time interval, up to t_{rel} .

Here, the microstructure is not that of a granular material, but rather the one of an elasto-plastic lattice material whose microstructure is made of bars with an elasto-viscoplastic behavior and isotropic hardening (with Perzyna-type viscosity), see Figure 13(b). The material has the following properties K = 167 GPa, G = 77 GPa, c = 100 MPa, H = 10 MPa and $\mu = 25$ s, where K and G are the bulk and the shear modulus, c is the material strength in simple shear, H the hardening modulus, and μ the viscosity parameter. The microstructural bars have a constant circular cross-section equal to 1 mm².

Following the procedure detailed in [MS23], the digital twin of the lattice cell is built using TANN, trained over randomly generated strain-driven paths applied in the form of periodic boundary conditions. In total, 33 internal variables are identified from the microscopic total and inelastic deformation fields.

As it follows, we investigate and compare the solutions obtained from the (exact) micromechanical model with those from the homogenized one, relying on TANN. For the homogenized case, we consider a FE model consisting of 80 linear tetrahedral elements in length and 8 in height (with crossed diagonals) – the number of elements was determined by mesh convergence analyses.

First, we restrict the analysis exclusively to the loading phase and verify that the micromechanical solution converges to the homogenized one, for ϵ tending to zero, as expected from asymptotic homogenization [Mie02]. To this end, we compare the results of the micromechanical model, with different sizes of the unit-cell ϵ , with those obtained with the FEM×TANN approach. Figure 14(a,b) depicts the total energy and dissipation rate in function of ϵ . For $1/\epsilon \ge 6$, the maximum relative error is as low as 1.7% (in energy) and 2.3% (in dissipation). The convergence of the response of the homogenized model to the response of the micromechanical simulations is achieved at $\epsilon = 0.1$ (with an error approximately equal to 0.5%).

Then, we continue by considering the full time scale of the analysis and select the microscopic model with $\epsilon = 0.1$ as the reference solution. Figure 13(c,d) depicts the deformed shapes of the micromechanical and homogenized models (magnified by a factor of 10), at the end of the steady phase. Contours of the free energy density in the



Figure 13: Multiscale problem: (a) homogenized and micromechanical model of a panel, with length 10 mm and height 2 mm, under plan strain conditions, subjected to a shearing load and (b) force-displacement response of the micromechanical ($\epsilon = 0.1$) and the FEM×TANN model.

homogenized model identify the areas where high stresses develop.

As far as it concerns the kinematics of the problem, we compare, in Figure 14(c), the force-displacement response of the homogenized model with the micromechanical reference solution. For the former, we consider the (zeroth-order) approximation of the displacements, while for the latter, the microscopic displacements at the nodes are considered as reference. A good agreement between the two models is observed. During the steady phase, the material displays viscous effects that alter the displacements and deformation fields. This is captured by the homogenized model with a relative error in the displacement of 0.25%. In addition, the model correctly predicts the residual (non-zero) vertical displacement, cf. Figure 14(c).

In parallel, TANN enable the characterization of microscopic fields, specifically the internal coordinates $\boldsymbol{\xi}$, within the homogenized model, known as *localization* in asymptotic homogenization (not to be confused with strain localization). Unlike the classical procedure [PdCOTD09], the proposed approach allows straightforward reconstruction of microscopic fields by decoding the internal variables.

Figure 15 compares the microscopic deformations at the intrados of the panel in the micromechanical model ($\epsilon = 0.1$) with those obtained from the decoding of the internal variables. An excellent agreement is observed. The FEM×TANN approach accurately captures deformation redistribution during the steady phase caused by viscosity. Additionally, it provides not only overall agreement with the micromechanical



Figure 14: Total energy (a) and dissipation rate (b) of the micromechanical (ref) and homogenized (FEM×TANN) models ($t = t_{load}$), at varying of the unit cell size, ϵ .

solution for microscopic inelastic deformations but also precise prediction of the plastic region location.

Furthermore, the proposed method successfully captures complex micromechanical mechanisms, such as the phenomenon of *trapped*, locked elastic energy. At the end of the unloading phase, the inelastic deformation exceeds the total microscopic deformation (cf. Figure 15). Elastic deformations remain trapped within the plastified region of the structure, from the clamped end to the end of the plastic region ($x \in [-5, 1.625]$ mm), persisting even without external force. Conversely, for x > 1.625 mm, total and inelastic deformations vanish during unloading.

It is worth noticing that predictions near the fixed end (x = -5 mm) of the panel exhibit minor differences compared to the micromechanical solution due to high strain gradients with wavelengths comparable to the unit cell size. Remedies for these boundary layer effects exist (see e.g. [BP12]). Nonetheless, despite the limitations of first-order asymptotic homogenization theory, the FEM×TANN approach yields excellent results compared to the micromechanical problem and holds potential for higher-order homogenization schemes.

4 Conclusions

Deep learning and, in general, data-intensive (or data-centric) approaches have the potential of revolutionizing the way we perceive models in physics and in mechanics. However, a purely data-centric vision is intrinsically hindered by poor generalization and interpretability of the consequent discovered models.

Particularly inspiring, in recent years, has been the possibility of hardwiring (in soft and hard ways) first principles into deep learning algorithms and learn interpretable, robust, and high-fidelity models that respect the inherent physics [KKL⁺21, HBG⁺21, MSVMB21, KFM⁺22, CC22].

In this chapter, we presented two emblematic methodologies, namely Physics-Informed Neural Networks (PINN) and Thermodynamics-based Artificial Neural Net-



Figure 15: Microscopic deformations (total and inelastic) at the intrados of the panel at the end of the loading, steady, and unloading plus relaxation phase. In the micromechanical model ($\epsilon = 0.1$), the microscopic deformations are computed from the deformations of the bars with axis parallel to the intrados. In the homogenized model, the microscopic deformations are obtained from the decoding of the identified internal variables at the Gauss points (no interpolation).

works (TANN). The former enable accounting for physical laws in the solution and discovery of partial different equations, from data. Physical knowledge is introduced though ad-hoc learning biases that consists in the minimization of a loss function related to the residual of the underlying governing equations. The latter consist, instead, of hardwiring the first and second laws of thermodynamics (i.e., energy and entropy balance) within the architecture of neural networks, via inductive and learning biases, to discover constitutive equations of complex materials, from data.

Relying on an hands-on example, we draw an interesting analogy between the integration of physics into deep learning algorithms and regularization techniques (based on ℓ_1 and ℓ_2 norms) promoting parsimony and model simplicity (see paragraph 2.2). Then, we demonstrated the possibility of identifying high-fidelity and interpretable digital replicas of complex granular materials relying on the framework of thermodynamics-based neural networks (paragraph 3.3). The latter not only capture the average material behavior but also allow to uncover the (hidden) internal state variables and efficiently track the microstructure evolution at reduced computational cost. Finally, the same approach can be deployed to accelerate computationally intensive multiscale simulations, herein exemplified by means of an application to lattice structures (paragraph 3.4).

The aforementioned examples and applications demonstrate the underlying idea of TANN in learning the constitutive behavior of intricate materials and achieving rapid

predictions at inference, thus allowing for massive accelerations of multiscale simulations. For instance, with reference to the example involving granular materials, it was observed that the computational time required by TANN to forecast the material response under a cyclic drained triaxial compression test was roughly three orders of magnitude lower compared to the extensive fine-scale simulations carried out using the discrete element method, specifically 1200/0.8 s/s.

In conclusion, the combination of physical principles with data-centric models offers a pathway to tame and describe, with high-accuracy, the complex response of intricate systems. Yet, some challenges remain to be addressed. Indeed, in mechanics, we are most of the times faced, rather than with vast amounts of data (*big data*), with very limited volumes of measurements (*small data*), characterized by an inherent heterogeneity and noise. Particular interesting is the development of physics-based deep learning algorithms that can address such a problem and deliver robust predictive models able to operate with reduced/partial information.

In addition, machine learning approaches can be employed to automatically determine meaningful variables of the system at hands and discover novel formulations, as we have seen in the example involving granular media (cf. paragraph 3.3). The same approach, driving the discovery of hidden latent (internal) variables, could be extended to address open challenges related to the choice of the descriptors of the state space of complex materials or any other physical system.

Acknowledgements

The author F.M. would like to acknowledge Prof. Itai Einav for the fruitful discussions which have greatly enriched this chapter.

The authors would like to acknowledge the support of the European Research Council (ERC) under the European Union Horizon 2020 research and innovation program (Grant agreement ID 757848 CoQuake).

References

- [BP12] NS Bakhvalov and G Panasenko. Homogenisation: averaging processes in periodic media: mathematical problems in the mechanics of composite materials, volume 36. Springer Science & Business Media, 2012.
- [BPRS18] AG Baydin, BA Pearlmutter, AA Radul, and JM Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.
- [CC22] E Cueto and F Chinesta. Thermodynamics of learning physical phenomena. *arXiv preprint arXiv:2207.12749*, 2022.

- [CCAC⁺20] F Chinesta, E Cueto, E Abisset-Chavanne, JL Duval, and FE Khaldi. Virtual, digital and hybrid twins: a new paradigm in data-based engineering and engineered data. Archives of computational methods in engineering, 27:105–134, 2020.
- [CDCG⁺22] S Cuomo, VS Di Cola, F Giampaolo, G Rozza, M Raissi, and F Piccialli. Scientific machine learning through physics–informed neural networks: where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.
- [CG67] BD Coleman and ME Gurtin. Thermodynamics with internal state variables. *Journal of Chemical Physics*, 47(2):597–613, 1967.
- [Daf22] YF Dafalias. Split stress rate plasticity formulation. *International Journal of Solids and Structures*, page 111494, 2022.
- [EHN07] I Einav, GT Houlsby, and GD Nguyen. Coupled damage and plasticity models derived from energy and dissipation potentials. *International Journal of Solids and Structures*, 44(7-8):2487–2508, 2007.
- [Ein12] I Einav. The unification of hypo-plastic and elasto-plastic theories. International Journal of Solids and Structures, 49(11-12):1305–1315, 2012.
- [EL18] I Einav and M Liu. Hydrodynamic derivation of the work input to fully and partially saturated soils. *Journal of the Mechanics and Physics of Solids*, 110:205–217, 2018.
- [Fey03] F Feyel. A multilevel finite element method (FE²) to describe the response of highly non-linear structures using generalized continua. *Computer Methods in Applied Mechanics and Engineering*, 192(28-30):3233–3244, 2003.
- [Gér19] A Géron. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, 2019.
- [GS98] J Ghaboussi and DE Sidarta. New nested adaptive neural networks (NANN) for constitutive modeling. *Computers and Geotechnics*, 22(1):29–52, 1998.
- [GSS⁺17] M Godio, I Stefanou, K Sab, J Sulem, and S Sakji. A limit analysis approach based on Cosserat continuum for the evaluation of the inplane strength of discrete media: Application to masonry. *European Journal of Mechanics-A/Solids*, 66:168–192, 2017.
- [Gur96] ME Gurtin. Generalized ginzburg-landau and cahn-hilliard equations based on a microforce balance. *Physica D: Nonlinear Phenomena*, 92(3-4):178–192, 1996.

- [HBG⁺21] Q Hernandez, A Badias, D Gonzalez, F Chinesta, and E Cueto. Deep learning of thermodynamics-aware reduced-order models from data. *Computer Methods in Applied Mechanics and Engineering*, 379:113763, 2021.
- [HTT⁺09] AJG Hey, S Tansley, KM Tolle, et al. *The fourth paradigm: dataintensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.
- [KB22] JN Kutz and SL Brunton. Parsimony as the ultimate regularizer for physics-informed machine learning. *Nonlinear Dynamics*, 107(3):1801–1817, 2022.
- [KD08] J Kozicki and FV Donze. A new open-source software developed for numerical simulations using discrete modeling methods. *Computer Methods in Applied Mechanics and Engineering*, 197(49-50):4429– 4443, 2008.
- [KFM⁺22] DK Klein, M Fernández, RJ Martin, P Neff, and O Weeger. Polyconvex anisotropic hyperelasticity with neural networks. *Journal of the Mechanics and Physics of Solids*, 159:104703, 2022.
- [KKL⁺21] GE Karniadakis, IG Kevrekidis, L Lu, P Perdikaris, S Wang, and L Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.
- [LBS09] M Lefik, DP Boso, and BA Schrefler. Artificial neural networks in numerical modelling of composites. *Computer Methods in Applied Mechanics and Engineering*, 198(21-26):1785–1804, 2009.
- [LJP⁺21] Lu Lu, Pengzhan Jin, Guofei Pang, Zhongqiang Zhang, and George Em Karniadakis. Learning nonlinear operators via deeponet based on the universal approximation theorem of operators. *Nature Machine Intelli*gence, 3(3):218–229, 2021.
- [LKS20] PY Lu, S Kim, and M Soljačić. Extracting interpretable physical parameters from spatiotemporal systems using unsupervised learning. *Physical Review X*, 10(3):031056, 2020.
- [LP21] R Leiteritz and D Pflüger. How to avoid trivial solutions in physicsinformed neural networks. *arXiv preprint arXiv:2112.05620*, 2021.
- [LS03] M Lefik and BA Schrefler. Artificial neural network as an incremental non-linear constitutive model for a finite element code. *Computer methods in applied mechanics and engineering*, 192(28-30):3265–3283, 2003.
- [Mau15] GA Maugin. The saga of internal variables of state in continuum thermo-mechanics (1893–2013). *Mechanics Research Communications*, 69:79–86, 2015.

- [ME23] F Masi and I Einav. Physics-based neural constitutive equations. In *Hybrid Computational Methods in Geotechnics*. The University of Melbourne, 2023.
- [MiD04] GDR MiDi. On dense granular flows. *The European Physical Journal E*, 14:341–365, 2004.
- [Mie02] C Miehe. Strain-driven homogenization of inelastic microstructures and composites based on an incremental variational formulation. *International Journal for Numerical Methods in Engineering*, 55(11):1285– 1322, 2002.
- [MM94] GA Maugin and W Muschik. Thermodynamics with internal variables. Part I. General concepts. *Journal of Non-Equilibrium Thermodynamics*, 19:217–249, 1994.
- [MS22] F Masi and I Stefanou. Multiscale modeling of inelastic materials with Thermodynamics-based Artificial Neural Networks (TANN). *Computer Methods in Applied Mechanics and Engineering*, 398:115190, 2022.
- [MS23] F Masi and I Stefanou. Evolution TANN and the identification of internal variables and evolution equations in solid mechanics. *Journal of the Mechanics and Physics of Solids*, 174:105245, 2023.
- [MSVMB21] F Masi, I Stefanou, P Vannucci, and V Maffi-Berthier. Thermodynamics-based Artificial Neural Networks for constitutive modeling. *Journal of the Mechanics and Physics of Solids*, 147:104277, 2021.
- [NCCD14] TK Nguyen, G Combe, D Caillerie, and J Desrues. FEM × DEM modelling of cohesive granular materials: Numerical homogenisation and multi-scale simulations. *Acta Geophysica*, 62(5):1109–1126, aug 2014.
- [NCDD11] M Nitka, G Combe, C Dascalu, and J Desrues. Two-scale modeling of granular materials: a DEM-FEM approach. *Granular Matter*, 13(3):277–281, 2011.
- [Ngu21] T-K Nguyen. On the Representative Volume Element of dense granular assemblies made of 2D circular particles. In *Structural Health Monitoring and Engineering Structures: Select Proceedings of SHM&ES* 2020, pages 499–508. Springer, 2021.
- [OS06] M Ostoja-Starzewski. Material spatial randomness: From statistical to representative volume element. *Probabilistic engineering mechanics*, 21(2):112–132, 2006.
- [PdCOTD09] J Pinho-da Cruz, JA Oliveira, and F Teixeira-Dias. Asymptotic homogenisation in linear elasticity. Part I: Mathematical formulation

and finite element modelling. *Computational Materials Science*, 45(4):1073–1080, 2009.

- [PMSJ22] G Piunno, F Masi, I Stefanou, and C Jommi. Multi-scale modelling of natural composites via Thermodynamics-based Artificial Neural Networks. In *Congrès Français de Mécanique (CFM)*. AFM, 2022.
- [PSS23] E Papachristos, I Stefanou, and J Sulem. A discrete elements study of the frictional behavior of fault gouges. *Journal of Geophysical Research: Solid Earth*, 128(1):e2022JB025209, 2023.
- [RD11] F Radjai and F Dubois. *Discrete-element modeling of granular materials*. Wiley-Iste, 2011.
- [RPK19] M Raissi, P Perdikaris, and GE Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.
- [Rub01] MB Rubin. Physical reasons for abandoning plastic deformation measures in plasticity and viscoplasticity theory. Archives of Mechanics, 53(4-5):519–539, 2001.
- [SAC⁺21] V Smilauer, V Angelidakis, E Catalano, R Caulk, B Chareyre, W Chèvremont, S Dorofeenko, J Duriez, N Dyck, J Elias, et al. YADE Documentation 3rd ed. *The Yade Project*, 2021.
- [SAS04] M Stroeven, H Askes, and LJ Sluys. Numerical determination of representative volumes for granular materials. *Computer Methods in Applied Mechanics and Engineering*, 193(30-32):3221–3238, 2004.
- [SSV10] I Stefanou, J Sulem, and I Vardoulakis. Homogenization of interlocking masonry structures using a generalized differential expansion technique. *International Journal of Solids and Structures*, 47(11-12):1522– 1536, 2010.
- [Ván03] P Ván. Weakly nonlocal irreversible thermodynamics. Annalen der Physik, 515(3):146–173, 2003.
- [Var18] I Vardoulakis. *Cosserat continuum mechanics: with applications to granular media*, volume 87. Springer, 2018.
- [VBE08] P Van, A Berezovski, and J Engelbrecht. Internal variables and dynamic degrees of freedom. *Journal of Non-Equilibrium Thermodynamics*, 33(3):235–254, 2008.
- [YZYK22] M Yin, E Zhang, Y Yu, and GE Karniadakis. Interfacing finite elements with deep neural operators for fast multiscale modeling of mechanics problems. *Computer Methods in Applied Mechanics and Engineering*, page 115027, 2022.

Introduction to Reinforcement Learning with Applications in Geomechanics

Alexandros Stathas, Diego Gutiérrez-Oribio and Ioannis Stefanou

Nantes Université, École Centrale Nantes, CNRS, GeM, UMR 6183, F-44000 Nantes, France

Reinforcement learning (RL) is a subfield of Machine Learning (ML) that focuses on the development of software agents that are capable of making optimal decisions in dynamic and uncertain environments. It is a powerful learning paradigm that enables machines to learn from their own interactions with the environment, rather than relying on explicit instructions or labeled datasets. In RL, an agent learns through a trial-and-error process, where it takes actions in an environment, receives feedback in the form of rewards or penalties, and adjusts its behaviour to maximize the cumulative reward over time. In this chapter, we present the basic concepts of RL and introduce model-based and model-free methods for deterministic and stochastic cases. Then, we present two applications of Geomechanics: the spring-slider and a geothermal reservoir. On both systems, an RL algorithm is used to design a controller able to prevent seismic events.

1 Introduction

In this chapter, we introduce *Reinforcement Learning* (RL), which is a family of *Machine Learning* (ML) algorithms that allow the learner (*i.e.* software *agent*) to determine an *optimal behaviour* inside an *environment*¹ that will provide the maximum cumulative *reward*² (see Figure 1).

RL is neither a *Supervised Learning* nor an *Unsupervised Learning* technique since the agent does not require the existence of a labelled dataset to train on, nor it extracts underlying patterns from already available data (see Figure 2). Instead, the agent

¹The environment is the set of all states, actions and rewards the agent can take.

²A reward is feedback signal from the environment (real value number) reflecting how well the agent is performing.



Figure 1: Schematic representation of the key components of *Reinforcement learning*. The agent interacts with the *environment* through *actions* and receives *feedback* through *observations* and *rewards*.

learns by performing $actions^3$ inside the environment and getting a reward. The objective of the RL algorithms is to measure the *value* of the actions taken by the agent and formulate a *policy* (*i.e.* an algorithm that chooses an action from an available set of actions at each state) so that it maximises the cumulative reward.



Figure 2: Schematic diagram between the Supervised learning, Unsupervised learning and Reinforcement learning techniques of Machine Learning. Deep learning with the help of neural networks can be applied to all these techniques.

Initial advances in RL took place in the framework of *Dynamic Programming* (DP) through the use of the *Bellman equations* and the *Bellman optimality conditions* [Bel54]. These methods allow us to construct powerful algorithms that given enough information about the environment, are capable of planning a suitable winning strategy (*i.e.* a *policy*) that will maximize the reward they receive [Sut99]. In essence, the problem or RL can be cast into a form of finding the optimal policy function, $\pi(s_t)^4$ that maximizes the expected accumulated reward.

This type of problem can be solved iteratively⁵ by applying the Bellman optimality conditions and the fundamental algorithms of *Value Iteration*, *Policy Iteration* and

³Input from an agent to the environment.

⁴Equivalently a stochastic policy, $\pi(\alpha_t|s_t)$, indicating the probability of the agent performing action α_t given the state s_t can be used.

⁵Such a method is called fixed point iteration.

Generalized Policy Iteration [Sut99, BT95].

The challenge in these fundamental methods of DP (value iteration and policy iteration) is their scalability to real-world applications. Although game examples such as chess or backgammon have a relatively small number of discrete states the agent can find itself in and possible actions it can perform, real world examples are much more complex in the sense that their state and action space can be continuous, rendering the classical techniques intractable.

A solution to this problem comes from the *Function Approximation* branch of ML. These are techniques of supervised learning (unsupervised learning can also be used), which estimate the state value and action value functions using vectors of interpolation weights.

In recent years, the significant increase in computational power together with the advent of *Deep Artificial Neural Networks* (DANNs) and newer programming environments (see [C⁺15, ABC⁺16, PGM⁺19], among others), which allow for the conceptually simpler implementation of such complicated nonlinear interpolation functions, permitted the combination of DANNs with the classical DP techniques (see [Gér22, VSS⁺19, Lap18]). This combination of modern computational power and past wisdom allowed researchers to significantly advance in the field of RL by applying their advanced algorithm architectures to previously intractable problems. These methods have been applied successfully to applications that involve:

- **Control applications:** These include a signal the agent observes from the environment and/or a reward that informs the agent that an action needs to be taken. This simple framework applies both to simple applications such as thermostat or pressure switches controllers [Sut90, SBW92] and to fancier controllers, like controlling the movement of robotic components [SSZ⁺22], or even the movement of plasma in Tockamak fusion reactors! [DFB⁺22].
- Game applications: These include software agents that learn to play optimally in games involving high dimensional state and action spaces such as Go [SHS⁺17, Pic17] and Starcraft [VEB⁺17].
- Generative AI: These applications of RL include a software agent that learns to create human interpretable content. This allows for the creation of AI art [Euc23], customer service and all-around helper Chatbots, like ChatGPT [Ope23, GBGM23].

Considering the use of RL in geomechanics, we focus on a recent control application from fault mechanics based on earthquake control (see [Ste19, PS21, ST22, GOTSP23, GOOSP23, GOSP22], among others). Roughly speaking, earthquakes are dynamic instabilities caused by frictional weakening. Such instability leads to fastslip (earthquake-like behaviour) creating waves that travel through the earth's crust and become catastrophic events. The cited works study the influence of injected fluid to avoid such behaviour, by designing strategies based on control theory. Here we solve this problem using RL.

The chapter is structured as follows. We first introduce the fundamental components of RL and their relations to the Bellman equations presenting an example on *policy evaluation*: a miner trapped inside a trembling mine finding his way to the gold (reward). Next, we present the fundamental model-based methods of DP: policy evaluation, state value iteration and policy iteration algorithms. For the stochastic frame, we show the most known *model-free* methods, which are useful for more practical applications when the states and actions are large and the probability distribution is unknown. Then we extend the applicability of the above methods with the use of function approximation tools, introducing policy approximation methods and discussing the application of policy gradient for finding an optimal policy. At last, we comment on the actor-critic popular approach in RL, which is used for problems with continuous state and action spaces, which is the case for the two geomechanics applications that we present. Inspired by [PS21], two earthquake control applications using a *Deep* Neural Network Actor-Critic (DNNAC) architecture are shown. The first application is the so-called spring-slider system, whereas the second is a geothermal reservoir. In both cases, the RL algorithm should have to adequate the gains of a linear controller, adjusting the fluid pressure inside the earth's crust. This will lead to an aseismic response of both systems, *i.e.*, the prevention of seismic events.

2 Reinforcement Learning: The basics

2.1 Basic Definitions: Deterministic case

Consider the game of a miner of Left Figure 3. The miner (shown as a black dot and known as the *agent*) moves inside the mine (known as the *environment*) to find gold (the *reward*) while avoiding the cliff (known as *penalties*). Each movement constitutes a different *action*. For this example, the miner can perform four actions: go right (\rightarrow), go up (\uparrow), go left (\leftarrow) and go down (\downarrow). Therefore, the movements will be discrete. The environment is constituted of 16 cells defined as C1, C2, ..., C16 and known as *states*. The *state space* of the environment is denoted as $S = \{C1, C2, ..., C16\}$.

The game will finish when the miner reaches either the cliff (located at cell C1) or the gold (located at cell C16). Due to their special nature, these two cells will be known as *terminal states*. When the miner reaches a terminal state, it completes an *episode*.

We define as *trajectory* the set of states, s_t , actions, α_t , and rewards, r_t , taken on every time step, t, of an episode (see Right Figure 3). A trajectory is then denoted as

$$T_r = \{s_0, \alpha_0, r_1, s_1, \alpha_1, \dots, s_{T-1}, \alpha_{T-1}, r_T\},\tag{1}$$

for an episode of time t = 0 to t = T. Note how time is also a discrete variable. Furthermore, the first state, s_0 , does not provide any reward and the final state, s_T , and final action, α_T , are not written because it is when the game finishes.

As in every game, we would like to receive the biggest reward in each episode. For that purpose, we will define the total accumulated reward, G_t , at the time t as the sum



Figure 3: Left: The mine (*environment*) and the miner (*agent*, black dot). The miner can move in the surrounding cells known as *states*. In order to move, the miner can perform four *actions*: go right (\rightarrow), go up (\uparrow), go left (\leftarrow) and go down (\downarrow). Two *terminal states* are present, C16, with a positive reward of +100 and C1 with a negative reward of -20. Right: A trajectory, *T*, followed by the agent. The miner started at cell C6 and followed a path to reach the gold, receiving a reward after taking an action. For this episode, the trajectory is defined as $T_r = \{s_0 = C6, \alpha_0 = \rightarrow, r_1 = 0, s_1 = C7, \alpha_1 = \uparrow, r_2 = 0, s_2 = C11, \alpha_2 = \uparrow, r_3 = 0, s_3 = C15, \alpha_3 = \rightarrow, r_4 = 100\}.$

of all the rewards from the reward r_{t+1} until the reward of the terminal state, r_T , *i.e.*

$$G_t = \sum_{k=0}^{T-1} r_{t+k+1},$$
(2)

where $r_{t+k+1} = r_{t+k+1}(s_{t+k}, \alpha_{t+k})$ is the intermediate reward between the two adjacent states s_{t+k} and s_{t+k+1} in the trajectory. We note that T-1 is the state before reaching the end of the game. Moreover we follow the convention that rewards later than the final reward r_T at the terminal time T are set to zero (*i.e* they are ignored). Between two states the total accumulated reward can also be written as

$$G_t = r_{t+1} + \sum_{k=0}^{T-2} r_{t+k+2}$$
(3)

where $r_{t+1} = r_{t+1}(s_t, \alpha_t)$ is the intermediate reward between the current state s_t and the next state s_{t+1} in the trajectory.

2.1.1 Policy evaluation

In order to move inside the mine, the miner will follow a set of rules specifying which *action* to perform at each cell (*state*) of the mine. This mapping between states and actions is called a *policy*. In this example, we will consider *deterministic* policies (*e.g.*, if the miner is in state $s_t = C2$ do action $\alpha_t = \uparrow$) and they are denoted as $\alpha_t = \pi(s_t)$.

Based on the policy of the miner we can expect that different states exhibit different cumulative rewards, *i.e.*, starting from an initial state, the miner might either reach the gold (C16) or fall in the cliff (C1). Moreover, depending on the given policy, there could be two problematic cases. In the first case, the miner might get stuck in a loop between two states and in the second case, the miner may hit the boundaries of the mine and stay there indefinitely as well (see Figure 4).



Figure 4: Problematic cases of the environment. Left: The agent falls in a loop between two states. Right: The policy forces the agent to hit the boundaries.

In order to handle these cases, we will force the episode to finish if the time reaches a time limit T_{max} . This will ensure that every episode will terminate and it won't stay in an infinite loop. Moreover, we penalize the agent with a negative reward -10 if it chooses to hit the boundaries of the mine.

We can evaluate the expected accumulated reward G_t during a trajectory where the miner follows a given policy $\pi(s_t)$, starting from the state s_t . This will be known as the value of the state:

Definition 1 [SB18] The value of the state, $V_{\pi}(s_t)$, is written as

$$V_{\pi}(s_t) = \sum_{k=0}^{T-1} r_{t+k+1}(s_{t+k}, \pi(s_{t+k})), \quad \text{or}$$

$$V_{\pi}(s_t) = r_{t+1}(s_t, \pi(s_t)) + \sum_{k=0}^{T-2} r_{t+k+2}(s_{t+k+1}, \pi(s_{t+k+1})). \tag{4}$$

From the definition, we note that the actions taken at each subsequent state s_{t+k} are prescribed by the policy $\pi(s_{t+k})$. The same is also true for the rest of the future actions $\pi(s_{t+k+1})$ where they are only dependent on the future state, s_{t+k+1} , and not on the past states. We will see in the general stochastic setting of DP, that this is a fundamental assumption known as the *Markov property*.

Based on the definition of the *value of the state* in (4), we can replace the previous sum over the future rewards with the value of the future state $V_{\pi}(s_{t+1})$ as

$$V_{\pi}(s_t) = r_{t+1}(s_t, \pi(s_t)) + V_{\pi}(s_{t+1}).$$
(5)

Such equation is known as the *Bellman equation* for the value function $V_{\pi}(s_t)$.

Let's consider now three deterministic policies, $(\pi_A(s_t), \pi_B(s_t), \pi_C(s_t))$, the miner can follow in the environment (see Figure 5). Policy $\pi_A(s_t)$ could be considered as bad because there have three states (C2, C6, C10) that lead to the cliff. Policy $\pi_B(s_t)$ and policy $\pi_C(s_t)$ seem to be good because all of the states will lead to the gold. One may think, which is the best of the three?



Figure 5: Example of three different policies.

We will define a rule to check which policy is better depending on the value of the state as follows:

Definition 2 A policy $\pi'(s_t)$ is better than policy $\pi(s_t)$ if for every state, s_t , the value of the state under policy π' is larger or equal to the value of the state under policy π , *i.e.*, $V_{\pi'}(s_t) \ge V_{\pi}(s_t)$ for every s_t .

With this definition, we can compare our three policies in order to check which may be the best. We only need to evaluate the value of each state for each policy,

 $(V_{\pi_{A}}(s_{t}), V_{\pi_{B}}(s_{t}), V_{\pi_{C}}(s_{t}))$, applying sequentially equation (5) for each state $s_{t} \in S$. The results can be found in Table 1 in the columns below $\gamma = 1$ (whose definition will be clarified a few lines later).

	$\gamma = 1$			$\gamma = 0.9$	
	$V_{\pi_{\mathrm{A}}}(s_0)$	$V_{\pi_{\mathrm{B}}}(s_0)$	$V_{\pi_{\rm C}}(s_0)$	$V_{\pi_{\mathrm{B}}}(s_0)$	$V_{\pi_{\rm C}}(s_0)$
C1	-20.0	-20.0	-20.0	-20.0	-20.0
C2	-20.0	+100.0	+100.0	+65.0	+65.0
C3	-10.0	+1000.	+100.0	+72.0	+72.0
C4	-10.0	+100.0	+100.0	+81.0	+81.0
C5	0.0	+100.0	+100.0	+65.0	+65.0
C6	-20.0	+100.0	+100.0	+59.0	+72.0
C7	+100.0	+100.0	+100.0	+81.0	+81.0
C8	+100.0	+100.0	+100.0	+90.0	+90.0
C9	0.0	+100.0	+100.0	+72.0	+72.0
C10	-20.0	+100.0	+100.0	+81.0	+81.0
C11	+100.0	+100.0	+100.0	+90.0	+90.0
C12	+100.0	+100.0	+100.0	+100.0	+100.0
C13	-10.0	+100.0	+100.0	+81.0	+81.0
C14	-10.0	+100.0	+100.0	+90.0	+90.0
C15	-10.0	+100.0	+100.0	+100.0	+100.0
C16	+100.0	+100.0	+100.0	+100.0	+100.0

Table 1: Comparison between the values of the state value function $V_{\pi}(s)$ for deterministic policies $(\pi_A(s_t), \pi_B(s_t), \pi_C(s_t))$. Without the discount parameter $(\gamma = 1)$, we can see that policy A is the worst but we cannot decide which is better between policies B and C. Including the discount parameter $(\gamma = 0.9)$, we can finally say that policy C is the best of the three because it has a larger accumulated reward value for state $(s_0 = C6)$, while for the rest of the states the values remain the same.

We can confirm that the worst policy is $\pi_A(s_t)$ because it presents lower values for all states. However, we cannot decide between policies $\pi_B(s_t)$, $\pi_C(s_t)$ because they have the same values at every state. Therefore, we need to include something that unties the result in this kind of situation. For this purpose, we will consider also the number of moves the miner needed for getting the reward. To do this we will apply a *discount*
rate, γ , to the cumulative reward of each state as

$$V_{\pi}(s_t) = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}(s_{t+k}, \pi(s_{t+k})),$$

= $r_{t+1}(s_t, \pi(s_t)) + \gamma \sum_{k=0}^{T-2} \gamma^k r_{t+k+2}(s_{t+k+1}, \pi(s_{t+k+1})),$
= $r_{t+1}(s_t, \pi(s_t)) + \gamma V(s_{t+1}).$ (6)

You can see that if $\gamma \in (0, 1)$, the value of the state at every time step will decrease, penalizing then the number of moves done by the miner before getting the reward. Calculating again the value of the state for policy B and C (see results in Table 1 under $\gamma = 0.9$), we can see that policy C is better because it has at least one value of the state bigger than policy B at the state C6, while all the others are the same.

The question that we face now is:

How can I know if the policy $\pi_C(s_t)$ is the best policy that I could find?

In other words:

Can I find an optimal policy π^* for which the cumulative reward at the end of the episode becomes maximum?

This question will be answered in the following sections.

2.1.2 Action Value Function

We will introduce another value function called the *action value function*, which indicates the value⁶ of a specific action, α_t , given that the miner is at state s_t .

Definition 3 The action value function, $Q_{\pi}(s_t, \alpha_t)$, at a state, s_t , corresponds to the immediate reward, $r_{t+1}(s_t, \alpha_t)$, that will be obtained performing one of the available actions, α_t , at such state, plus the expected value of the state according to a given policy starting from the state s_{t+1} , i.e.,

$$Q_{\pi}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma \sum_{k=0}^{T-2} \gamma^k r_{t+k+2}(s_{k+t+1}, \pi(s_{k+t+1})),$$

= $r_{t+1}(s_t, \alpha_t) + \gamma V_{\pi}(s_{t+1}).$ (7)

From the definition of the value of the state, $V_{\pi}(s_t)$, in (6) and the action value function, $Q_{\pi}(s_t, \alpha_t)$, in (7), we can see that both expressions are the same at s_{t+1} and when $\alpha_t = \pi(s_{t+1})$, *i.e.*, $V_{\pi}(s_{t+1}) = Q_{\pi}(s_{t+1}, \pi(s_{t+1}))$. Therefore, we can get a recursive expression of the action value function as

$$Q_{\pi}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma Q_{\pi}(s_{t+1}, \pi(s_{t+1})).$$
(8)

⁶also known as the quality

In the case of our miner, from the Policy B of Figure 5, we could calculate the action value for the state $s_t = C6$ at every possible action $\alpha_t = \{ \rightarrow, \uparrow, \leftarrow, \downarrow \}$ (see Figure 6). We took again $\gamma = 0.9$ so we still get the same values of $V_{\pi}(s_t)$ shown in Table 1. The action value functions are then calculated as

$$Q_{\pi_{\rm B}}(C6, \to) = r_{t+1}(C6, \to) + 0.9V_{\pi_{\rm B}}(C7) = 0 + 0.9(81) = 72,$$

$$Q_{\pi_{\rm B}}(C6, \uparrow) = r_{t+1}(C6, \uparrow) + 0.9V_{\pi_{\rm B}}(C10) = 0 + 0.9(81) = 72,$$

$$Q_{\pi_{\rm B}}(C6, \leftarrow) = r_{t+1}(C6, \leftarrow) + 0.9V_{\pi_{\rm B}}(C5) = 0 + 0.9(65) = 59,$$

$$Q_{\pi_{\rm B}}(C6, \downarrow) = r_{t+1}(C6, \downarrow) + 0.9V_{\pi_{\rm B}}(C2) = 0 + 0.9(65) = 59.$$
(9)

We can see that there are at least two actions (\uparrow, \rightarrow) that give more cumulative reward than what the current policy says (we will obtain 59 if we take the action \leftarrow of Policy B). In other words, there is room for improvement! This is called *Policy Improvement*:

Definition 4 [SB18] Any policy π for which $V_{\pi}(s_t) \leq Q_{\pi}(s_t, \alpha_t)$, $\forall s_t \in S$, is a policy that we can improve on. That is we can choose action α_t at state s_t , formulating a new policy π' in the process, for which $V_{\pi}(s_t) \leq V_{\pi'}(s_t)$.



Figure 6: Evaluation of the action value function starting at state $s_t = C6$ with $\gamma = 0.9$. For every possible action in state C6 ($\alpha_t = \{ \rightarrow, \uparrow, \leftarrow, \downarrow \}$), we evaluate $Q_{\pi_{\rm B}}(C6, \alpha_t)$ as the immediate reward for performing this action and then moving according to policy $\pi_{\rm B}(s_t)$.

The task now is to find a method to choose the right action at every state, s_t , obtaining the maximum accumulated reward at the end of every episode.

2.1.3 Bellman's Optimality conditions

As we saw before, from all the possible actions, α_t , that we can take in a state, s_t , there is one special action that maximises the immediate reward. Bellman showed that if we recursively take the best action at every state of the episode, then the expected accumulated reward for each state becomes maximum [Bel54]. In other words, the

policy that maximizes the accumulated reward at each state⁷ is the optimal policy, $\pi^*(s_t)$, that we were looking for, *i.e.*, $V_{\pi^*}(s_t) = \sum_{k=0}^T r_{t+k+1}(s_{t=K}, \pi^*(s_{t=k})) = \sum_{k=0}^T r_{t+k+1} \max_{\alpha_{t+k}} r(s_{t+k}, \alpha_{t+k})$. This means that the value of the state and the action value function will have the same value if we follow the optimal policy, *i.e.*,

$$V_{\pi^{\star}}(s_t) = \max_{\alpha_t} Q_{\pi^{\star}}(s_t, \alpha_t), \tag{10}$$

which, according to equation (7), leads to

$$V_{\pi^{\star}}(s_t) = \max r_{t+1}(s_t, \alpha_t) + \gamma V_{\pi^{\star}}(s_{t+1}).$$
(11)

On the other hand, if we take the optimal policy in equation (7), it turns into

$$Q_{\pi^{\star}}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma V_{\pi^{\star}}(s_{t+1}), \tag{12}$$

which together with equation (10) reads as

$$Q_{\pi^{\star}}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma \max_{\alpha_{t+1}} Q_{\pi^{\star}}(s_{t+1}, \alpha_{t+1}).$$
(13)

Equations (11) and (13) are known as *Bellman's Optimality conditions* and they can be used iteratively to find the optimal state and action value functions. Moreover, the max operator on the right-hand side of the equations prevails over any policy π that is used for the evaluation of $V_{\pi^*}(s_t), Q_{\pi^*}(s_t, \alpha_t)$. As a result, the optimal values of the state and action, $V^*(s_t), Q^*(s_t, \alpha_t)$ are independent of the optimal policy! In othger words, there can be many policies π^* that lead to the optimal values of the state and action value functions $(V_{\pi^*}(s_t), Q_{\pi^*}(s_t, \alpha_t))$, but these values are unique and equal to $(V^*(s_t), Q^*(s_t, \alpha_t))$. We will present the basic algorithms for finding such optimal state and action value functions in the following.

2.1.4 Maximizing the reward: Model-based Methods

Based on the evaluation of the optimal policy and the *Bellman optimality conditions* we can discern two types of algorithms for maximizing the reward. These are the *Policy iteration algorithms* and the *Value iteration algorithms* (see Figure 7)

Both methods work excellently in deterministic or stochastic cases where the number of available states and actions is small. However, these methods require complete knowledge of the states, actions and rewards of the environment to work. This is why they are called model-based methods⁸. For our miner example, we need to know all possible states inside the mine and their corresponding rewards. The agent (miner) must *plans* its actions before they are taken.

We will now explain in more detail how both algorithms work.

⁷*i.e.* the value of the state $V_{\pi}(s_t)$

⁸We note that the notion of *model* in RL differs from that of supervised or unsupervised learning in the sense that the RL model is used to provide states and rewards to the agent and not to perform a regression task or find a pattern in the data (*i.e.*, in RL we train the agent, not the model). To describe the term model used in (supervised and unsupervised learning) the term *function approximator* is used instead (see [SB18] and Section 2.3).



Figure 7: The basic learning algorithms for finding the unique optimal value and action functions, $V^*(s_t)$, $Q^*(s_t, \alpha_t)$, and one optimal policy, π^* , (of many): Policy Iteration (blue)) and Value iteration (green). The difference between the two lies in the update strategy. Policy iteration calculates a better policy for all states and then evaluates the value functions according to this policy. Value iteration updates only one action at a time based on the Bellman optimality condition.

Policy Iteration (PI) method

In this algorithm, we achieve the optimal action values and the respective optimal policy by modifying the policy $\pi(s_t)$. In particular, starting from a random initialization of $\pi(s_t)$, we evaluate the value function over each state $V_{\pi}(s_t)$. Then we calculate the action value function for each state-action pair $Q_{\pi}(s_t, \alpha_t)$.

Next, we compare between the different $Q_{\pi}(s_t, \alpha_t)$ and find the value of $\max_{\alpha_t} Q_{\pi}(s_t, \alpha_t)$. Then, we extract the action that maximises the action value function $Q_{\pi}(s_t, \alpha_t)$, *i.e.*, $\alpha'_t = \operatorname{argmax}_{\alpha_t} Q_{\pi}(s_t, \alpha_t)$. Finally, we replace the action a_t in state s_t with the better action α'_t . This leads to a new policy $\pi'(s_t)$.

Once the policy has been updated, we repeat the evaluation of the value functions $V_{\pi'}(s_t), Q_{\pi'}(s_t)$. The next iterations for the next prediction of the optimal state and action value $V_{\pi^{\star}}^{k+1}(s_t), Q_{\pi^{\star}}^{k+1}(s_t, \alpha_t)$ functions are given by

$$V_{\pi^{\star}}^{k+1}(s_t) = r_{t+1}(s_t, \pi'(s_t)) + \gamma V_{\pi'}^k(s_t), \tag{14}$$

$$Q_{\pi^*}^{k+1}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma Q_{\pi'}^k(s_{t+1}, \pi'(s_{t+1})).$$
(15)

When the policy $\pi'(s_t)$ converges for all states, the PI algorithm has converged to an optimal policy $\pi^*(s_t) = \pi'(s_t)$ and yields the optimal $V_{\pi^*}(s_t)$, $Q_{\pi^*}(s_t, \alpha_t)$.

We emphasize that the optimal state value function, $V_{\pi^*(s_t)}$, and the optimal action value function, $Q_{\pi^*}(s_t, \alpha_t)$, are unique. However, the policies that yield the optimal reward are not, *i.e* $V^*(s_t) = V_{\pi^*}(s_t)$, $Q^*(s_t, \alpha_t) = Q_{\pi^*}(s_t, \alpha_t)$. Moreover, we note that in this algorithm the evaluation of the future state-action value functions $(V_{\pi'}^k(s_{t+1}), Q_{\pi'}^k(s_{t+1}, \alpha_{t+1}))$ does not happen according to the initial policy $\pi(s_t)$

(see also section 2.1.1) but the new updated policy $\pi'(s_t)$ is used in each iteration (see also Figure 7).

Application to the miner example

We can apply the above algorithm for finding the optimal state and action-value functions, $V^*(s_t)$, $Q^*(s_t, \alpha_t)$, in the example of the miner. In this case, we see in Figure 8, that the policy iteration algorithm converges after four iterations to the optimal values $V^*(s_t)$ for each state s_t inside the mine (see Table 2).



Figure 8: Policy iteration example for the example of the miner. The algorithm evaluates according to the policy $\alpha_t = \pi_i(s_t)$ the accumulated reward starting from each state separately $V_{\pi_i}(s_t) = V_{\pi_i}(s_t, \pi^i(s_t))$ and compares with the accumulated reward for the different actions at each state $Q_{\pi_i}(s_t, \alpha_t)$. Then the algorithm updates the actions in order to optimise $Q_{\pi_i}(s_t, \alpha_t)$.

	$V_{\pi^0}(s_t)$	$V_{\pi^1}(s_t)$	$V_{\pi^2}(s_t)$	$V_{\pi^{\star}}(s_t)$
C1	-20	-20	-20	-20
C2	+65	+65	+65	+65
C3	+72	+72	+72	+72
C4	+81	+81	+81	+81
C5	+65	+65	+65	+65
C6	+59	+72	+72	+72
C7	+81	+81	+81	+81
C8	+90	+90	+90	+90
C9	+72	+72	+72	+72
C10	+81	+81	+81	+81
C11	+90	+90	+90	+90
C12	+100	+100	+100	+100
C13	+81	+81	+81	+81
C14	+90	+90	+90	+90
C15	+100	+100	+100	+100
C16	+100	+100	+100	+100

Table 2: Values of the state value function for the different iterations of the policy iteration algorithm for the example of the miner (the discount parameter was chosen $\gamma = 0.9$). The optimal policy is reached after the first iteration. The final optimal policy and the first iteration are both optimal policies because they lead to the optimal values of the state. The change in the actions happens because of the way the algorithm selects the maximum action value (the right action, \rightarrow , is selected first).

Generalised Policy Iteration methods (GPI)

In the previous algorithm (PI) we evaluated the state values and action values of the optimal policy $V_{\pi'}(s_t)$, $Q_{\pi'}(s_t, \alpha_t)$ by iteratively changing the policy $\pi'(s_t)$ which is used for the evaluation of $V_{\pi'}(s_{t+1})$, $Q_{\pi'}(s_{t+1}, \alpha_{t+1})$. We will now use Bellman's optimality condition to perform the updates on the values of the state value and action value functions without performing a policy update. These learning algorithms are called GPI methods.

These methods stem from the property of the optimal state value, $V^*(s_t)$, and action value, $Q^*(s_t, \alpha_t)$, to be independent of the optimal policy π^* . This property extends also to the updates of the GPI algorithm, therefore, we don't need to explicitly guess the optimal policy π^* in order to obtain the optimal $V^*(s_t)$, $Q^*(s_t, \alpha_t)$. We can instead apply the maximum operator at each state repeatedly, find the optimal values $V^*(s_t)$, $Q^*(s_t, \alpha_t)$ and then deduce the optimal policy π^* . We present below the methods of the *Value iteration* and *Q-Value iteration*.

• Value Iteration method

The simplest method to use in order to find the optimal $V^*(s_t)$ and a corresponding

optimal policy π^* , is to randomly initialize the values of $V^0(s_t)$ and then iterate over each state, s_t , and update the value of the state value function $V^{k+1}(s_t)$ according to the rule,

$$V^{k+1}(s_t) = \max_{\alpha_t} \left[r_{t+1}(s_t, \alpha_t) + \gamma V^k(s_{t+1}) \right], \tag{16}$$

where $V^{k+1}(s_t)$, $V^k(s_{t+1})$ are the next update of the state value function at the current state, s_t , and the current update of the state value function at the subsequent states, s_{t+1} , respectively.

After sufficient iterations over each state choosing the best action α_t at each state, the values of the state value function will converge to the optimal values, *i.e.*, $V^{k+1}(s_t) \rightarrow V^*(s_t)$.

At this point, the optimal policy, π^* , can be found by choosing to move along the path of optimal state values and therefore to higher cumulative reward, *i.e.*, $\pi^* = \operatorname{argmax}_{\alpha_t} [r_{t+1}(s_t, \alpha_t) + \gamma V_{\pi}^k(s_{t+1})].$

Q-Value Iteration \ Greedy-Policy Improvement algorithm

Due to the equivalence between the optimal state values and the optimal action values $(V^*(s_t) = \max_{\alpha_t} Q^*(s_t, \alpha_t))$ the above state value iteration algorithm can be converted to its action value form as

$$Q^{k+1}(s_t, \alpha_t) = r_{t+1}(s_t, \alpha_t) + \gamma \max_{\alpha_{t+1}} Q^k(s_{t+1}, \alpha_{t+1}),$$
(17)

where $Q^k(s_{t+1}, \alpha_{t+1})$ is the current update k of the future action state values (s_{t+1}, α_{t+1}) and $Q^{k+1}(s_t, \alpha_t)$ is the new update k+1 of the action value function for the present state action pair s_t, α_t .

After sufficient iterations over each state-action pair (s_t, α_t) , choosing the best action α_{t+1} for each next state s_{t+1} , the values of the action value function will converge to the optimal values, *i.e.*, $Q^{k+1}(s_t, \alpha_t) \rightarrow Q^*(s_t, \alpha_t)$.

At this point, the optimal policy π^* can be found by choosing the actions that have the highest action value and equal the higher cumulative reward, *i.e.*, $\pi^* = \operatorname{argmax}_{\alpha_t} Q^*(s_t, \alpha_t)$.

2.2 Probabilistic environment and Stochastic policies

2.2.1 Motivation

In the deterministic example described above, there are two subtle assumptions. First, we assumed that the environment does not influence directly the movement of the miner (*i.e.*, there are no *extraneous actions*⁹ and that at each state the miner is moving exactly as the deterministic function of the policy $\pi(s_t)$ dictates. Moreover, each

 $^{{}^{9}}$ By extraneous actions we refer to actions that are outside the control of the agent and depend on the environment dynamics (see [SB18]).

state-action pair (s_t, α_t) leads to exactly one new state, s_{t+1} , and one corresponding reward $r_{t+1}(s_t, \alpha_t)$.

However, this is not always the case. In most practical applications there are *uncertainties* that influence both the environment dynamics *extraneous actions* and the movement of the agent. This means that the movement of the agent does not lead always to the intended new state and reward s_{t+1} , $r_{t+1}(s_t, \alpha_t)$.

For our miner example, we can take account of these uncertainties in the environment and the policy of the miner. More specifically, we assume that the mine is trembling (uncertainties in the dynamics of the environment) forcing the miner to miss a step and that the lamp of the miner is not working at all times (it flickers) so that the miner mixes its moves (see Figure 9).



Figure 9: In the stochastic framework the move of the miner is decided based on the conditional probability of its current state. In this example the miner in state $s_t = C7$ selects more frequently the \uparrow action. The next tile the miner will transition to (new state s_{t+1}), depends on the transition probabilities of the miner for its current state s_t and selected action α_t . In this example, we consider only two non-zero transition probabilities based on the state-action pair $(s_t, \alpha_t) = (C7, \uparrow)$, these are: $p(C13, 0|C7, \uparrow)$ and $p(C11, 0|C7, \uparrow)$. In both cases, the intermediate reward for the transition is zero $(r_{t+1} = 0)$. We know that the environment favours the transition to C13 more, *i.e* it is more probable that the agent's next state will be $s_{t+1} = C13$ with a reward $r_{t+1} = 0$.

We implement these uncertainties in the policy of the agent and the dynamics of the environment by considering:

- The conditional probability the miner will perform an action, α_t, given the previous history of states and actions it performed in the past s_t, α_{t-1}, s_{t-1}, ..., α₀, s₀, *i.e.*, the *policy function* becomes a probability function of all previous states and actions, π(α_t|s_t, α_{t-1}, s_{t-1}, ..., α₀, s₀).
- The conditional probability the miner will visit a new state, s_{t+1} , and receive the reward, r_{t+1} , given the previous history of states actions and rewards

 $s_t, \alpha_t, r_t, \alpha_{t-1}, s_{t-1}, ..., r_1, \alpha_0, s_0$. This is the next state and reward probability function $p(s_{t+1}, r_{t+1}|s_t, \alpha_t, r_t, \alpha_{t-1}, s_{t-1}, ..., r_1, \alpha_0, s_0)$.

The knowledge of the previous history significantly complicates the problem to be solved, but as in the deterministic case, we will consider a special type of history-independent process, called a *Markov Decision Process* [WIW89].

2.2.2 Markov Processes

A Markov process is a formal way to describe a problem that involves different states that can be observed by the agent and change sequentially¹⁰. At the time t during the process, the agent observes the features of a state, s_t , and then transitions to another state, s_{t+1} , receiving an immediate reward, r_{t+1} . The total reward the agent accumulates is the sum of all immediate rewards. This is called a *Markov Reward Process* (MRP).

The Dynamics of an MRP process are completely characterised by a *Markov transition* matrix $p(s_{t+1}, r_{t+1}|s_t)$ that shows the probability of the agent observing the next state, s_{t+1} , and receiving a reward, r_{t+1} , given the agent's current observation of state s_t .

In most applications when the agent lies in its current state s_t , it also chooses an action α_t that changes the probability of the agent's next state s_{t+1} , r_{t+1} . This is called a *Markov Decision Process*. We note that if we assign the action, α_t , inside the state, s_t , observed by the agent, we obtain a state action pair. When we consider transitions between the different state action pairs, we retrieve again an MRP. The Markov Decision Process (MDP) is also described by a transition matrix between the agent's current state and action pair (s_t, α_t) and the next state and reward (s_{t+1}, r_{t+1}) .

Considering that the process is history independent, the transition probability matrix implies that the current state, s_t , and action, α_t , contain all information needed for the transition to the next state and the next reward (this is called the *Markov property* [WIW89])¹¹.

Using the Markov property, the previous history doesn't matter and we can assert that

$$\pi(\alpha_t|s_t, \alpha_{t-1}, s_{t-1}, ..., \alpha_0, s_0) = \pi(\alpha_t|s_t),$$
(18)

$$p(s_{t+1}, r_{t+1}|s_t, \alpha_t, r_t, \alpha_{t-1}, s_{t-1}, \dots, r_1, \alpha_0, s_0) = p(s_{t+1}, r_{t+1}|s_t, \alpha_t).$$
(19)

For the case of our miner, our analysis then simplifies significantly, because we need to only specify two functions for the DP framework to work. These are:

• The conditional probability that the miner will perform an action, α_t , given the current state, s_t , it is in, *i.e.*, the *policy function* becomes a probability function $\pi(\alpha_t|s_t)$.

¹⁰Sequentially means that the agent can only find itself in one state at each time. Moreover, only the past states influence the next state *i.e* in a Markov Process the future does not influence the past.

¹¹In the case where the processes are history dependent and don't have the Markov property, we choose to enhance the *features* of the state by appending available information from previous states to the current state. The number of states will grow drastically but the Markov property will be restored.

The conditional probability the miner will visit a new state, s_{t+1}, and receive the reward, r_{t+1}, given that in the current state, s_t, the miner performed an action, α_t. This is the next state and reward probability function p(s_{t+1}, r_{t+1}|s_t, α_t).

We present a general example of an MDP with four states $s_t \in \{C1, C2, C3, C4\}$ and three actions $\alpha_t \in \{d1=\leftarrow, d2=\uparrow, d3=\rightarrow\}$ in Figure 10. The transition probability matrix characterising this MDP is given in Table 3.

As we did in the deterministic case, we will focus on episodic processes. In order for episodic cases to be possible, the Markov process should contain at least one terminal state, in which the agent will remain with probability 1. In this case, the Markov process is called *absorbing*.



Figure 10: Left: A Markov Decision Process. At each state transition, $s_t \rightarrow s_{t+1}$, the agent performs an action, α_t , and receives a reward, r based on its final state s_{t+1} . The dynamics of the environment, *i.e.*, the rule that associates the rewards and the next states according to the current state and actions might be known (deterministic or stochastic) or unknown. We can also treat each state-action pair as a separate state. This is useful in model-free methods when the environment dynamics are not known a priori. In this case, we obtain A Markov Reward Process. For a specific trajectory realization, the state-action-reward of the MDP is shown together with its corresponding state action pair-reward MRP.

2.2.3 Value of the State and Action Value Function

In the general stochastic framework of DP, we seek to maximize the expected value of the accumulated reward starting from a given state s_t , given by

$$V_{\pi}(s_{t}) = \mathbb{E}_{\pi}[G_{t}|s_{t}]$$

= $\mathbb{E}_{\pi}[r_{t+1}|s_{t}] + \gamma \mathbb{E}_{\pi}[G_{t+1}|s_{t}]$
= $\sum_{\alpha_{t}\in\mathcal{A}} \pi(\alpha_{t}|s_{t}) \sum_{\substack{s_{t+1}\in\mathcal{S}\\r_{t+1}\in\mathcal{R}}} p(s_{t+1}, r_{t+1}|s_{t}, \alpha_{t})r_{t+1} + \gamma \mathbb{E}_{\pi}[G_{t+1}|s_{t}],$ (20)

$p(s_{t+1}, r_{t+1} s_t, \alpha_t)$	C1	C2	C3	C4
C1	[0.5,1.0,0.0]	[0.5,0.0,0.0]	[0.0,0.0,0.0]	[0.0,0.0,1.0]
C2	[1.0,0.0,0.0]	[0.0,1.0,0.0]	[0.0,0.0,1.0]	[0.0,0.0,0.0]
C3	[0.6,0.3,0.0]	[0.4,0.7,0.0]	[0.0,0.0,0.0]	[0.0,0.0,1.0]
C4	[0.0,0.0,0.0]	[0.0,0.0,0.0]	[0.0,0.0,0.0]	[1.0,0.0,0.0]

Table 3: Markov Decision Process (MDP) Transition Matrix. This probability matrix shows us the transition probabilities between the states in the MDP $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$ of Figure 10, i.e the probabilities of reaching a new state $s_{t+1} \in$ $\{C1, C2, C3, C4\}$ from the current state $s_t \in \{C1, C2, C3, C4\}$ and receiving the next reward r_{t+1} given that we perform action $\alpha_t \in \{\leftarrow, \uparrow, \rightarrow\}$. Note that in each row summing over a specific action the total probability is 1.

where $r_{t+1} = r_{t+1}(s_{t+1}, \alpha_t, s_t)$, *i.e* the reward at next state s_{t+1} given the current state s_t and action α_t and $\mathbb{E}_{\pi}[\cdot]$ is an operator signifying the *mean* of the *expected* conditional probability given the policy π .

More precisely, the *expected value* of a discrete random variable X, $\mathbb{E}[X]$ is given by

$$\mathbb{E}[X] = \sum_{x_i \in X} x_i p(X = x_i).$$
(21)

where x_i are the values of the random variable and $p(X = x_i)$ are the corresponding probability of the random variable X taking the value x_i .

When the random variable is expressed in the form of conditional probabilities *e.g.* $p(X|s_t, \alpha_t)$, the average value is generalized to the *expected conditional probability*

$$g(s_t, \alpha_t) = \mathbb{E}[X|s_t, \alpha_t] = \sum_{x_i \in X} x_i p(X = x_i|s_t, \alpha_t)$$
(22)

. We can expand the above relation using the *general product rule* of probability theory to obtain

$$g(s_t) = \mathbb{E}[X|s_t, \alpha_t] = \sum_{x_i \in X} x_i p(X = x_i|s_t, \alpha_t) \pi(\alpha_t|s_t) p(s_t)$$
(23)

. Because s_t can be any state in the state space S we get that $p(s_t) = 1$.

We can then derive the mean value of the expected conditional probability as

$$\mathbb{E}_{\pi}[X|s_t] = \mathbb{E}[\mathbb{E}[X|s_t, \pi(\alpha_t|s_t)]] = \sum_{\alpha_t \in \mathcal{A}} \sum_{x_i \in X} x_i p(X = x_i|s_t, \alpha_t) \pi(\alpha_t|s_t).$$
(24)

In order to find a policy, $\pi^*(\alpha_t|s_t)$, that maximizes the agent's expected cumulative reward, $\pi^* = \operatorname{argmax}_{\pi} \mathbb{E}_{\pi}[G_t|s_t]$, at the end of the episode, it is useful to evaluate the

value function of the state the agent finds itself in, $V_{\pi}(s_t)$, and the value function of the available actions the agent can take, $Q_{\pi}(s_t, \alpha_t)$, as we did for the deterministic case.

For a given policy, π , and starting from the state, s_t , the value function of the state can be evaluated using the Bellman equation [Bel54, BT95]

$$V_{\pi}(s_{t}) = \sum_{\alpha_{t} \in \mathcal{A}} \pi(\alpha_{t}|s_{t}) \sum_{\substack{s_{t+1} \in \mathcal{S}, \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_{t}, \alpha_{t}) r_{t+1} + \gamma \mathbb{E}_{\pi} \left[\sum_{k=0}^{T-2} \gamma^{k} r_{t+k+2} |s_{t} \right]$$
$$= \sum_{\alpha_{t} \in \mathcal{A}} \pi(\alpha_{t}|s_{t}) \sum_{\substack{s_{t+1} \in \mathcal{S}, \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_{t}, \alpha_{t}) r_{t+1}$$
$$+ \gamma \sum_{\substack{s_{t+2}, \dots, s_{T} \in \mathcal{S}, \\ \alpha_{t+1}, \dots, \alpha_{T_{1}} \in \mathcal{A}, \\ r_{t+2}, \dots, r_{T} \in \mathcal{R}}} p(Tr, s_{t+1}, \alpha_{t+1}, s_{t}, \alpha_{t}) \sum_{k=0}^{T-2} \gamma^{k} r_{t+k+2}, \quad (25)$$

where $p(Tr, s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) = p(s_T, r_T \dots s_{t+2}, r_{t+2}, s_{t+1}, \alpha_{t+1}, s_t, \alpha_t)$ is the joint probability of the specific trajectory realization. Using again the *general product rule* from probability theory we get

$$p(Tr, s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) = p(s_T, r_T \dots s_{t+2}, r_{t+2} | s_{t+1}, \alpha_{t+1}, s_t, \alpha_t) p(s_{t+1} | s_t, \alpha_t) \pi(\alpha_t | s_t)$$

Replacing and grouping the terms we obtain,

$$V_{\pi}(s_{t}) = \sum_{\alpha_{t} \in \mathcal{A}} \pi(\alpha_{t}|s_{t}) \sum_{\substack{s_{t+1} \in \mathcal{S}, \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_{t}, \alpha_{t}) \\ \times \left[r_{t+1} + \gamma \sum_{\substack{s_{t+2}, \dots, s_{T} \in \mathcal{S} \\ \alpha_{t+1}, \dots, \alpha_{T_{1}} \in \mathcal{A}, \\ r_{t+2}, \dots, r_{T} \in \mathcal{R}}} p(T_{r}|s_{t+1}, \alpha_{t+1}, s_{t}, \alpha_{t}) \sum_{k=0}^{T-2} \gamma^{k} r_{t+k+2} \right].$$
(26)

Moreover, we note that

$$\gamma^{k} r_{t+k+2} | s_{t}, \alpha_{t}, r_{t+1}, s_{t+1} = \sum_{\substack{s_{t+2}, \dots, s_{T} \in \mathcal{S} \\ \alpha_{t+1}, \dots, \alpha_{T_{1}} \in \mathcal{A}, \\ r_{t+2}, \dots, r_{T} \in \mathcal{R}}} p(T_{r} | s_{t+1}, \alpha_{t+1}, s_{t}, \alpha_{t}) \gamma^{k} r_{t+k+2}$$

, which leads to

$$V_{\pi}(s_{t}) = \sum_{\alpha_{t} \in \mathcal{A}} \pi(\alpha_{t}|s_{t}) \sum_{\substack{s_{t+1} \in \mathcal{S}, \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_{t}, \alpha_{t}) \\ \times \left[r_{t+1} + \gamma \sum_{k=0}^{T-2} \gamma^{k} r_{t+k+2}|s_{t}, \alpha_{t}, r_{t+1}, s_{t+1} \right],$$
(27)

However, since the process is an MDP, the last sum (which denotes the expectation of the reward r_{t+2} for the next state) is only dependent on the next state s_{t+1} . This yields

$$V_{\pi}(s_{t}) = \sum_{\alpha_{t} \in \mathcal{A}} \pi(\alpha_{t}|s_{t}) \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} \left[r_{t+1}(s_{t}, \alpha_{t}, s_{t+1}) + \gamma \sum_{k=0}^{T-k-2} \gamma^{k} r_{t+k+2}|, s_{t+1} \right]$$
$$= \sum_{\alpha_{t} \in \mathcal{A}} \pi(\alpha_{t}|s_{t}) \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} \left[r_{t+1}(s_{t}, \alpha_{t}, s_{t+1}) + \gamma V_{\pi}(s_{t+1}) \right].$$
(28)

Similarly for the action value function $Q_{\pi}(s_t, \alpha_t)$ we derive

$$Q_{\pi}(s_{t}, \alpha_{t}) = \mathbb{E}_{\pi}[G_{t}|s_{t}, \alpha_{t}]$$

$$= \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_{t}, \alpha_{t})$$

$$\times \left[r_{t+1} + \gamma \sum_{\alpha_{t+1} \in \mathcal{A}} \pi(\alpha_{t+1}|s_{t+1}) \sum_{k=0}^{T-k-2} \gamma^{k} r_{t+k+2}|s_{t}, \alpha_{t}, r_{t+1}, s_{t+1}, \alpha_{t+1} \right]$$

$$= \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_{t}, \alpha_{t}) \left[r_{t+1} + \gamma \sum_{\alpha_{t+1} \in \mathcal{A}} \pi(\alpha_{t+1}|s_{t+1}) Q_{\pi}(s_{t+1}, \alpha_{t+1}) \right].$$
(29)

The relationship between $V_{\pi}(s_t)$, $Q_{\pi}(s_t, \alpha_t)$ can be shown starting from equation 27 and replacing the initial definition of the action-value function 29, obtaining

$$V_{\pi}(s_t) = \sum_{\alpha_t \in \mathcal{A}} \pi(\alpha_t | s_t) Q_{\pi}(s_t, \alpha_t).$$
(30)

The above mathematical derivation for the stochastic Bellman equation can be summarized in Figure 11.



Figure 11: The schematic diagrams above indicate how the state value function, $V_{\pi}(s_t)$, and the action value function, $Q_{\pi}(s_t, \alpha_t)$, are evaluated. This is done through the definition of the average value $V(s_t) = \mathbb{E}_{\pi}[G_t|s = s_t]$, $Q(s_t, \alpha_t) = \mathbb{E}_{\pi}[G_t|s = s_t, \alpha = \alpha_t]$, the dynamics of the environment $p(s_{t+1}, r_{t+1}|s = s_t, \alpha = \alpha_t)$, and the policy of the agent $\pi(\alpha_t|s_t)$.

2.2.4 Bellman's Optimality conditions

The logic we presented in the deterministic case is also valid for the stochastic case considering the maximization of the expected value of the accumulated reward $\max_{\pi} \mathbb{E}_{\pi}[G_t]$. More specifically the Bellman optimality conditions and the policy improvement theorem also hold in the stochastic case [SB18]. The general formulae for the optimal state value function, $V^*(s_t)$, and action-value function, $Q^*(s_t, \alpha_t)$, are given by

$$V^{\star}(s_{t}) = \max_{\alpha_{t}} \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} (s_{t+1}, r_{t+1} | s_{t}, \alpha_{t}) \left[r_{t+1} + \gamma V_{\pi^{\star}}(s_{t+1}) \right],$$
(31)

$$Q^{\star}(s_{t},\alpha_{t}) = \sum_{\substack{s_{t+1} \in S \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1} | s_{t}, \alpha_{t}) r_{t+1} + \gamma \max_{\alpha_{t+1}} Q(s_{t+1}, \alpha_{t+1}), \quad (32)$$

where $r_{t+1} = r_{t+1}(s_t, \alpha_t, s_{t+1})$.

2.2.5 Policy Improvement Theorem

For the general stochastic case, the *Policy Improvement Theorem* [SB18] reads that any policy π' for which

 $V_{\pi}(s_t) \leq \sum_{\alpha_t \in \mathcal{A}} \pi(s_t, \alpha_t) Q(s_t, \pi'(\alpha_t | s_t))$ is a policy that we can improve on.

We note that in the above theorem the action value, $Q_{\pi}(s_t, \pi'(\alpha_t|s_t))$, belongs to the set of action values evaluated with the initial policy, π , at each state, $s_t \in S$, but it is selected based on policy $\pi'\alpha_t, s_t$. The above relation is equivalent to $V_{\pi}(s_t) \leq V'_{\pi}(s_t)$.

The Bellman equations applied to the framework of MDPs, lend themselves to a straightforward application of *Dynamic Programming*, which provides basic algorithms to maximize the state (Value iteration) and/or action values (Q-Value iteration) of the environment and consequently find the optimal policy for maximizing the expected reward of the agent.

Moreover, the notions of MDPs and the Bellman Optimality conditions play a central role in *Reinforcement Learning* (RL). In particular, they constitute the basis for many learning methods that seek to maximize the agent's expected reward.

Based on whether or not we have complete knowledge of the dynamics of the environment, $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$, we separate these learning methods into two categories namely model-based and model-free methods.

2.2.6 Model-based Methods

In the case of model-based methods we can find the optimal policy using a *Policy Iteration* algorithm or a *Generalised policy iteration algorithm* (see Figure 7). These methods are the equivalent of the deterministic model-based methods discussed in section 2.1.4. Consequently, they work well in ideal cases where the number of available states and actions are small and the MDP probability distribution function $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$ is known a priori. In this case the agent *plans* its actions before they are taken.

Policy Iteration (PI) method

Similar to what we explained in the deterministic case, In this algorithm, we achieve the optimal action values and the respective optimal policy by modifying the policy π .

At first, we evaluate the policy π . This evaluation can be done by direct inverting the Markov system. However, this takes a lot of time when the number of states increases. We can evaluate the solution of the system faster by using an iterative solution strategy.

In particular, using a *Fixed Point Iteration algorithm* we are able starting from a random initialization of $V^0(s_t)$, $Q^0(s_t, \alpha_t)$ (which does not respect necessarily the Bellman equations (28),(29)) and applying iteratively the state value definition of equation (28), for a randomly generated policy π , to evaluate the values of the state value function $V_{\pi}^{k+1}(s)$ at each state of the environment.

Once the values of the state value function $V_{\pi}(s_t)$ (respectively state value function, $Q_{\pi}(s_t, \alpha_t)$, based on policy π , have converged for each state s_t (and action α_t), we retrieve the action α'_t that is responsible for the maximum value of the action value function i.e α'_t = argmax_{α}, $Q(s_t, \alpha_t)$.

In the case of deterministic policies, if this action α'_t is different than the action calculated by the policy $\alpha_t = \pi(\alpha_t)$, *i.e.*, $\alpha'_t \neq \pi(s_t)$, we update the action of policy algorithm at the specific state s_t equal to α'_t .

Moreover, in the stochastic framework, we can also consider an ε – greedy policy. This policy can be applied according to the following rule

$$\pi^{k+1}(\alpha_t|s_t) = \begin{cases} 1 - \varepsilon - \frac{\varepsilon}{|\mathcal{A}|}, & \alpha_t = \alpha_t^* \\ \frac{\varepsilon}{|\mathcal{A}|} & \alpha_t \neq \alpha_t^*, \end{cases}$$
$$\alpha_t^* = \operatorname{argmax}_{\alpha_t} Q_{\pi}^{k+1}(s_t, \alpha_t), \tag{33}$$

where ε is a small parameter and A is the action space of the system.

Essentially, this policy update rule indicates that the action that had the highest action value will be sampled among all possible other actions with probability $1 - \varepsilon$. In contrast, the rest of the actions will be sampled with a probability of ϵ . This policy may only partially exploits the best action according to a specific state in the MDP, *i.e.*, it is not a purely greedy policy.

However, it has been shown to converge towards the optimal value and as a result to an optimal policy (see [SB18]). This stochastic policy is very useful in the case of model-free learning, where the agent does not know the transition probabilities of the MDP.

Once the policy π' has been updated, we evaluate the new updates of the state value and action-value functions $V^{k+1}_{\pi'}(s_t), Q^{k+1}_{\pi'}(s_t, \alpha_t)$ as

$$V_{\pi'}^{k+1}(s_t) = \sum_{\alpha_t \in \mathcal{A}} \pi'(\alpha_t | s_t) \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} [r_{t+1} + \gamma V_{\pi'}^k(s_{t+1})]$$
(34)
$$Q_{\pi'}^{k+1}(s_t, \alpha) = \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1} | s_t, \alpha_t)$$
$$\times \left(r_{t+1} + \gamma \sum_{\alpha_{t+1} \in \mathcal{A}} \pi'(\alpha_{t+1} | s_{t+1}) Q_{\pi'}^k(s_{t+1}, \alpha_{t+1}) \right).$$
(35)

When the values of the state-value function under policy $\pi'(\alpha_t|s_t)$ have converged in all states, we say that the PI algorithm has converged to an optimal policy $\pi^*(s_t, \alpha_t) = \pi'(s_t, \alpha_t)$ and yields the optimal values $V^*(s_t) = V_{\pi^*}(s_t)$, $Q^*(s_t, \alpha_t) = Q_{\pi^*}(s_t, \alpha_t)$.



Figure 12: Stencil of the Policy iteration algorithm of Dynamic Programming, we present the case where the algorithm evaluates the values of the state value function $V^*(s_t)$. The states and actions inside the rectangle are taken into account during the update of the value of the state function.

We present in Figure 12 the stencil for the model-based dynamic programming algorithm. We note that the updates involve only the current state-action pair, s_t , α_t , and the next state s_{t+1} .

We emphasize that the optimal state value function, $V_{\pi}^{\star}(s)$, and the optimal action value function, $Q_{\pi}^{\star}(s, \alpha)$, values are unique. However, the policies that yield the optimal reward are not. In this algorithm, the evaluation of the future state-action value functions $(V_{\pi'}^k(s_{t+1}), Q_{\pi'}^k(s_{t+1}, \alpha_{t+1}))$ does not happen according to the initial policy π , but the new updated policy $\pi'(\alpha_t|s_t)$ is used directly (see also Figure 7).

Generalised Policy Iteration methods (GPI)

Similar to what we discussed in the deterministic case, we can also use directly Bellman's optimality condition to perform the updates on the values of the state value and action value functions without performing a policy update. These learning algorithms are called GPI methods.

In these methods, we apply directly the nonlinear max operator for the Bellman optimality condition. As a result, the system that emerges is not linear and cannot be solved by a direct inversion method. We are only left with the iterative approach.

• Value Iteration method

The simplest method to use in order to find the optimal $V^*(s_t)$ and the corresponding optimal policy π^* , is to iterate over each state, s_t , using the \max_{α_t} operator, and

update the value of the state, $V^k(s_t)$, according to the rule

$$V^{k+1}(s_t) = \max_{\alpha_t} \sum_{\substack{s_{t+1} \in S \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1} | s_t, \alpha_t) \left[r_{t+1} + \gamma V^k(s_{t+1}) \right], \quad (36)$$

where $V^k(s_t)$, $V^k(s_{t+1})$ are the current and subsequent state values at iteration k and $V^{k+1}(s_t)$ is the new state value for the next iteration. We emphasize that these values are not evaluated based on a policy $\pi(\alpha_t|s_t)$ but by direct application of the maximum \max_{α_t} operator at each state. After sufficient iterations over each state choosing the best action, α_t , at each state, the values of the state value function will converge to the optimal values, $V^*(s_t)$. At this point, the optimal policy, π^* , can be found by choosing to move along the path of optimal state values and therefore to higher cumulative reward, *i.e.*, $\pi^* = \operatorname{argmax}_{\alpha_t} \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1}|s_t, \alpha_t) [r_{t+1} + \gamma V^*(s_{t+1})].$

• Q-Value Iteration \ Greedy-Policy Improvement algorithm

Because of the equivalence between the optimal state values and the optimal action values $(V^*(s_t) = \max_{\alpha} Q^*(s_t, \alpha))$ the above state value iteration algorithm can be converted to its action value form

$$Q^{k+1}(s_t, \alpha_t) = \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r_{t+1} \in \mathcal{R}}} p(s_{t+1}, r_{t+1} | s_t, \alpha_t) \left(r_{t+1} + \gamma \max_{\alpha_{t+1}} Q^k(s_{t+1}, \alpha_{t+1}) \right).$$
(37)

where $Q^k(s_{t+1}, \alpha_{t+1})$ are the future action state values at iteration k and $Q^{k+1}(s_t, \alpha_t)$ is the new update of the state action value function. At this point, the optimal policy, π^* , can be found by choosing the actions that have the highest action value and equal the higher cumulative reward, *i.e.*, $\pi^* = \operatorname{argmax}_{\alpha_t} Q^*(s_t, \alpha_t)$.

2.2.7 Model-free methods

These methods are used when the MDP probability distribution function,

 $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$, is not known. This is the case in most applications of practical geomechanical interest, where the number of states and actions and thus the dimensions of the *Markov transition matrix* can become prohibitively large for planning strategies to be practical.

Therefore, model-free algorithms that *learn* action to return associations are more useful. These methods can again be cast into the framework of value iteration methods including Monte Carlo Control, Temporal differences learning (SARSA), Q-learning, Deep Q Networks, Policy gradient methods, and Actor-Critic Networks, to name a few (see [BT95, SB18, Dee22]). Especially in the case of *Deep Learning*, different methods and networks can be combined together to yield better optimal policies and higher cumulative rewards (see also [Gér22, Fu16]).

The general idea of the model-free methods is that the agent does not know the probability function of the MDP, $p(s_{t+1}, r_{t+1|s_t,\alpha_t})$, that connects future states, s_{t+1} , and rewards, r_{t+1} , to current actions, α_t , and states s_t . For this reason, the action value of the state, s_t , according to the second Bellman equation (29) cannot be computed directly. Instead, the action value $Q_{\pi}(s_t, \alpha)$ at state s_t for any action, α , selected by the policy, π , will have to be *evaluated* through averaging the cumulative rewards of multiple episodes for different states of the MDP. This stage of learning is called *prediction*.

After a step of prediction, the new estimates of the state and action values, $Q_{\pi}(s_t, \alpha)$, are available. Then the algorithm will adjust its policy to π' in order to choose with higher probability the actions with the highest action value at each state $Q_{\pi'}(s_t, \alpha_t)$.

Monte Carlo method

Let us consider that the agent finds itself in an environment with a small space of available states, $s_t \in S$, and that the agent takes actions, $Q_{\pi}(s_t, \alpha_t)$, based on some random policy, π , The probability function of the MDP, $p(s_{t+1}, r_{t+1|s_t,\alpha})$, is unknown. For this reason the value function of a state, s_t , and the action value function of a state-action pair (*i.e.*, $(s_t, \alpha_t), V_{\pi}(s_t), Q_{\pi}(s_t, \alpha_t)$) cannot be computed directly.

In the Monte Carlo (MC) method we choose to *estimate* the action value, $Q_{\pi}(s_t, \alpha_t)$, at state s_t for any action, α_t , selected by the policy, $\pi(\alpha_t|s_t)$, by using equation (29), *i.e.*, by running multiple episodes of numerical experiments, generating a sufficient number of trajectories and finally averaging over the expected reward for each state-action pair¹².

We do this procedure in batches and in each batch, m episodes are run. In order to average the expected reward at each state-action pair, we take note of the agent's trajectory in each episode of the batch. Then for each state-action pair, (s_t, α_t) , in the agent's trajectory, we calculate the accumulated reward (see Figures 13 and 14).



Figure 13: Three realizations of the miner's trajectory inside the mine

¹²This way we render the *Markov Decision Process* into a *Markov Reward Process*, where the agent visits different state-action pairs during an episode and then, its receives a reward r_{t+1} . Thus, each state-action pair is treated as a separate state leading to a different reward.



Figure 14: Accumulated reward counting in multi-pass Monte Carlo. We examine the realization of the agent's trajectory of Figure 13. In a *Markov Reward Process* the states are the state-action pairs of the corresponding *Markov Decision Process*. Every time the state s_t is visited, the reward after each state is summed. Here states $(C8,\uparrow)$ and $(C9,\uparrow)$ were visited twice in the same episode during trajectories b, and c, respectively. And state $(C2,\rightarrow), (C3,\uparrow), (C4,\rightarrow)$ visited twice in different episodes. The trajectories have a degree of randomness due to the stochastic policy $\pi(\alpha_t|s_t)$ and the environment dynamics. The points where the action selected by the agent was not followed through are shown with a circle and the intended action is shown with a dotted line.

For each state-action pair, we add the accumulated reward over the different episodes of the batch and then we divide by the number of times the state-action pair was present in the batch. The action value then is given by

$$Q_{\pi}^{k+1}(s_t, \alpha_t) = \mathbb{E}_{\pi}[G_t|s_t, \alpha_t] = \frac{1}{C((s_t, \alpha_t))} \sum_{m=1}^{M} \sum_{t=0}^{T_m-1} \mathbb{I}(s = s_t, \alpha_t) G_{t_m}, \quad (38)$$

where G_{t_m} is the accumulated reward at the end of the episode *m* starting counting at state-action pair (s_t, α_t) , $C((s_t, \alpha_t))$ is the count of each state-action pair inside the batch of trajectories *M* and $\mathbb{I}(s = s_t, \alpha_t)$ is the assignment matrix that selects only the accumulated returns of the selected state-action pair (s_t, α_t) (see Table 4).

	$G, T_{r,a}$	$G, T_{r,b}$	$G, T_{r,c}$	$C((s_t, \alpha_t))$	$\mathbb{E}[G]$
$(C2, \rightarrow)$	+90	+100	0.0	2	+95
$(C2,\uparrow)$	0.0	0.0	-20	1	-20
(C3,↑)	+90	+100	0	2	+95
$(C4, \rightarrow)$	+90	+100	0	2	+95
(C4,↑)	+100	0.0	0.0	1	+100
(C4, ←)	+100	0.0	0.0	1	+100
$(C5,\downarrow)$	0.0	0.0	-20	1	-20
$(C6,\uparrow)$	0.0	0.0	-20	1	-20
$(C7, \rightarrow)$	0.0	+100	0.0	1	+100
(C7,↑)	+100	0.0	0.0	1	+100
$(C8,\uparrow)$	0.0	+200	0.0	2	+100
(C8, ←)	+100	0.0	0.0	1	+100
(C9,↓)	0.0	0.0	-40	2	-20
(C10,↓)	0.0	0.0	-20	2	-20
(C12,↑)	+100	0.0	0.0	1	+100

Table 4: Value of a state-action pair (Q-values) for three different trajectory realizations ($\gamma = 1.0$). $C((s_t, \alpha_t))$ is the number of times (count) the specific action-value pair was encountered in the batch of trajectories (in this example the batch size is 3). The expected value is given by summing the rewards following the specific stateaction pair over the count of the state-action pair in the batch of trajectories.

Instead of evaluating the new average after each batch of trajectories, we could evaluate the running average of $Q_{\pi}(s_t, \alpha)$ after the agent realizes one trajectory. This reduces significantly the memory storage requirements of the algorithm. The above relation (38) can then be written as

$$Q_{\pi}^{m+1}(s_t, \alpha_t) = Q_{\pi}^m(s_t, \alpha_t) + \frac{1}{C((s_t, \alpha_t))} \left(G_{t_m} - Q_{\pi}^m(s_t, \alpha) \right).$$
(39)

The update parameter $\frac{1}{C((s_t,\alpha_t))}$ that based on the number of counts of the pair (s_t, α_t) on the batch samples, can be also replaced by a constant step-size learning hyperparameter a^{13} and we obtain

$$Q_{\pi}^{m+1}(s_t, \alpha_t) = Q_{\pi}^m(s_t, \alpha_t) + a \left(G_t^m - Q_{\pi}^m(s_t, \alpha_t) \right).$$
(40)

We show in Figure 15 the stencil of the Monte-Carlo algorithm, during the policy evaluation.

¹³The action evaluation (prediction) method then is called "constant-a MC".



Figure 15: Stencil of the Monte-Carlo Learning algorithm. The black rectangles indicate terminal states. We note that a whole episode needs to be computed before any update can take place.

We note that the effect of the step-size parameter, a, in the convergence towards the optimal $\max_{\alpha_t} Q_{\pi}(s_t, \alpha_t)$ is similar to the effect of the gradient update parameter in the stochastic batch gradient descend algorithm used in supervised learning tasks. In essence, for large values of the hyperparameter a, the optimal action values oscillate without really converging to a value. *i.e.*, quick but noisy learning. On the contrary, smaller values of a will converge at the optimal value slowly but in an accurate manner (see Figure 16).

In order to perform the control task in RL, *i.e.*, reach the optimal values of the action value function $Q^*(s_t, \alpha_t)$ at each state-action pair, we can use a default greedy policy that immediately selects the best action based on the updated values of the action value function.

Notice that the use of such greedy policy implies that another policy π has been used during the prediction of the action value function in each state-action pair (s_t, α_t) . This type of learning where another policy generates the data and makes predictions and another policy controls the maximization of the action value function towards the optimal action values, $Q^*(s_t, \alpha_t)$, is called *Off-policy learning* and requires the use of statistical techniques, like the so-called *Importance sampling*, in order to remove any bias from the learned data.

This second problem comes from the fact that after the first values of $Q_{\pi}(s_t, \alpha_t)$ have been evaluated, the greedy policy will always exploit the same actions that yield the

Stathas, Gutiérrez-Oribio & Stefanou 413



number of episodes: m

Figure 16: Representation of the convergence error estimate between updates of the optimal action value function $Q^*(s_t, \alpha_t) - Q_{\pi^*}^{m+1}(s_t, \alpha_t)$ for all state-action pairs (s_t, α_t) . Red curve: higher values of the step hyperparameter a_1 make faster the convergence but higher the variance. Blue curve: lower values of the learning hyperparameter a_2 lead to slower convergence but less variance.

highest $Q_{\pi}(s_t, \alpha_t)$ at each s_t in order to maximise the expected accumulated reward¹⁴. Because $Q_{\pi}(s_t, \alpha_t)$ is not known for all state-action pairs, the greedy policy runs the risk of getting trapped in a local maximum of the expected reward and not reaching the global maximum. This problem is also known as the *Exploration vs Exploitation tradeoff*.

Alternatively, the agent can use an ε -greedy policy as in the case of the Q-value modelbased method for the prediction and the action value function and the optimization of the reward (see equation (33)).

In particular, this policy considers that all possible actions are included during the action selection stage, *i.e.*, $\pi(\alpha|s) > 0, \forall s \in S, \forall \alpha \in \mathcal{A}(f)^{15}$. Therefore, given enough tries by the agent, it is guaranteed to visit all possible states of the problem, spending more time on the interesting parts of the problem where higher expected cumulative rewards are present.

Thus, the Monte Carlo algorithm *learns to associate* actions to expected accumulated rewards achieving a balance between exploration and exploitation of the problem state space.

We note that for a larger number of states, the Monte Carlo Control method becomes impractical. This is because the action values are updated at the end of each episode which can take a long time to complete.

¹⁴Considering the example of the mine, a greedy miner won't explore the mine and will head straight for the exit, where the principal reward is, without ever searching for any hidden or secondary rewards that would increase the final cumulative reward.

¹⁵In this case the policy is characterized as *soft*.

On policy vs Off policy methods

In RL we make a distinction between the policy the agent uses in order to generate the data (behaviour policy $b(s_t|\alpha_t)$) and the policy we improve/evaluate in order to find the optimal policy (target policy $\pi(s_t|\alpha_t)$)) and maximize the expected cumulative reward. When we use the target policy to also generate the data in subsequent episodes, the RL method is called an *On policy* method, else, the method is called *Off policy*.

The need for *Off policy* methods becomes clear when we don't want to compromise the exploratory capabilities of the agent and more importantly when we want to expand our dataset with trajectory samples from third parties. This is common because RL applications take quite a lot to run and produce an adequate number of new trajectories for statistics applications. Therefore, the use of already available data is always welcome.

We can convert the action values of the initial dataset produced under the policy $b(\alpha_t|s_t)$, to action values obtained through the new policy $\pi(\alpha_t|s_t)$, by using *Importance sampling* provided that in each policy the available actions over each state are common, *i.e.*, if $\pi(s|\alpha) > 0$ then $b(s|\alpha) > 0$. The action values under policy π can be found using the formula

$$Q_{\pi}(s_t, \alpha_t) = \mathbb{E}_b[\rho G_t | s_t, \alpha_t], \tag{41}$$

where:

$$\rho = \Pi_{\tau=t+1}^{T-1} \frac{b(\alpha_{\tau}|s_{\tau})}{\pi(\alpha_{\tau}|s_{\tau})}.$$
(42)

Temporal Difference learning: SARSA

When Monte Carlo is used, the update to the new action values happens only after the agent has completed one episode. When the number of states becomes large, an episode may take a long time to complete and therefore, convergence to the maximum action values, $Q_{\pi^*}(s_t, \alpha_t)$, and optimal policy, $\pi^*(s_t, \alpha_t)$, becomes slow. Temporal Difference algorithm (TD) or State-Action-Reward-State-Action algorithm (SARSA), solves the update problem of the Monte Carlo control algorithm by updating the action values during the episode. The update happens every *n*-steps inside the episode *m* using the estimate of the expected accumulated reward. This is done assuming the agent acts optimally after a future state that is *n*-steps in front of the current state s_{t_m} . We consider as an optimal policy the ε -greedy policy (see equation (33)). We call this estimate a target value $G_{t_m}(n)^{16}$. Inside an episode *m*, the update of the action value $Q_{\pi}^{k+1}(s_{t_m}, \alpha_{t_m})$ is given by

$$Q_{\pi}^{k+1}(s_{t_m}, \alpha_{t_m}) = Q_{\pi}^k(s_{t_m}, \alpha_{t_m}) + a\left(G_{t_m}(n) - Q_{\pi}^k(s_{t_m}, \alpha_{t_m})\right), \quad (43)$$

¹⁶A technique that uses estimates of state and/or action values in order to evaluate some target values is called *Bootstraping*.

Stathas, Gutiérrez-Oribio & Stefanou 415



Figure 17: Stencil of the Temporal Difference Learning algorithm for 1 number of steps into the future (SARSA). The black rectangles indicate terminal states. The algorithm needs only the next state in order to perform the update of the current action value.

where:

$$G_{t_m}(n) = r_{t_m+1} + \gamma r_{t_m+2} + \dots \gamma^{n-1} r_{t_m+n} + \gamma^n Q_{\pi}(s_{t_m+n}, \alpha_{t_m+n}).$$
(44)

We show in Figure 17 the stencil of the SARSA algorithm, for policy evaluation and policy iteration tasks.

We note that in equation (44) the update of the action value, $Q_{\pi}^{k+1}(s_{t_m}, \alpha_{t_m})$, does not coincide with the next episode m + 1. The update happens every n steps inside the episode depending on the *n*-step evaluation of the target value $G_{t_m}(n)$. We also note two special cases in the TD method depending on the hyper-parameter n:

- n = 1: In this case, the action values are updated after the new state is reached. This allows for maximum learning during an episode.
- n → ∞: In this case, the action values are updated only after the episode ends. This defaults to the usual Monte Carlo learning method of the previous section.

In the TD method, learning is affected by the pair of hyperparameters n and a (see Figure 18). In particular, we note that a low parameter n (*e.g.*, n = 1) yields faster learning rates when the higher values of the step size parameter a are considered. Moreover, with respect to the MC method, the convergence results are less noisy. This indicates a fundamental difference between the MC and TD learning methods.



number of episodes: m

Figure 18: Comparison of the temporal difference algorithm for different values of the hyperparameter n based on their convergence estimates between updates of the action value function $Q^*(s_t, \alpha_t) - Q^{m+1}(s_t, \alpha_t)$ for all s_t, α_t . Red curve $n_1 = 100$: The algorithm is equivalent to a Monte Carlo method. Blue curve $n_2 = 1$: The algorithm reaches a lower minimum for the convergence error. The two methods do not converge to the same optimal point because of the different estimates for the expected reward $\mathbb{E}[G_t]$.

Let us consider that we train the MC and TD algorithms on a batch of m episodes until convergence to the optimal state, $V^*(s_t)$, and action values, $Q^*(s_t, \alpha_t)$, for this batch. This way we eliminate the influence of the step size a. According to [SB18], the two methods will converge to different estimates of $Q^*(s_t, \alpha_t)$! In particular, the MC method will converge towards values of the state $V_{\pi'MC}(s)$ that minimize the mean square error between the values of the state and the limit, while the TD method will tend towards the estimate that maximizes the likelihood of high expected reward. Therefore, TD is a better method for approximating the maximum expected accumulated reward max $\mathbb{E}[G_t]$ when the assumptions of an MDP hold.

Q-learning

In the TD case, we used the estimate of a ε – greedy policy for the accumulated reward, at step n inside the episode m. Another approach is to update the accumulated reward estimate using the maximum action value function of the n-step action value pair (s_{t+n}, α_{t+n}) . This leads to the following algorithm for learning the optimal action value function

$$Q_{\pi}^{k+1}(s_{t_m}, \alpha_{t_m}) = Q_{\pi}^k(s_{t_m}, \alpha_{t_m}) + a\left(G_{t_m}(n) - Q^k(s_{t_m}, \alpha_{t_m})\right),$$
(45)

where:

$$G_{t_m}(n) = r_{t_m+1} + \gamma r_{t_m+2} + \dots \gamma^{n-1} r_{t_m+n} + \gamma^n \max_{\alpha_{t_m+n}} Q^k(s_{t_m+n}, \alpha_{t_m+n}).$$
(46)

We note that Q-learning is an off-policy method because the $\max_{\alpha_t} Q(s_t, \alpha_t)$ is not known a priori but deduced from the evaluation of the $Q_{\pi}(s_t, \alpha_t)$ values after *n*-steps in the trajectory.

Expected SARSA

Another variant of the Q-learning method can be made using the expected value of the action under policy $\pi(\alpha_t|s_t)$, *i.e.*,

$$Q_{\pi}^{k+1}(s_{t_m}, \alpha_{t_m}) = Q_{\pi}^k(s_{t_m}, \alpha_{t_m}) + a\left(G_{t_m}(n) - Q_{\pi}^k(s_{t_m}, \alpha_{t_m})\right), \quad (47)$$

where:

$$G_{t_m}(n) = r_{t_m+1} + \gamma r_{t_m+2} + \dots \gamma^{n-1} r_{t_m+n} + \gamma^n \sum_{\alpha_{t_m+n} \in \mathcal{A}} \pi(\alpha | s_{t_m+n}) Q^k(s_{t_m+n}, \alpha_{t_m+n}).$$
(48)

We note that this can be an on-policy method because the action value $Q_{\pi}^{k}(s_{t}, \alpha_{t})$ is evaluated the same way as its update $Q_{\pi}^{k+1}(s_{t}, \alpha_{t})$.

2.3 Function approximation

When the state space is large (*e.g.*, a continuous system with infinite states) it is not practical -let alone feasible- to perform iterations over each state to find the optimal state and action values. We can move around this difficulty considering that the optimal state value function, $V_{\pi^*}(s_t)$, and the optimal action value function, $Q_{\pi^*}(s_t, \alpha_t)$, can be approximated by other functions $\hat{v}(s_t, \boldsymbol{w}) = \boldsymbol{w}\boldsymbol{x}$ and $\hat{q}(s_t, \alpha_t, \boldsymbol{w}) = \boldsymbol{w}\boldsymbol{\psi}$, respectively. Such functions are a linear combination of the vector of weight parameters, \boldsymbol{w} , and the vectors of data features $\boldsymbol{x}, \boldsymbol{\psi}$. By data features, we mean the data itself or patterns from the data that can arise from any classification method (see **Chapter 4: Classification Techniques in Machine Learning** for more details). Any number of weights and data features can be implemented and the task of state evaluation now is to minimize, \overline{VE} (respectively \overline{QE}), which can be the square error (or the Huber norm) between the target function $V_{\pi^*}(s_t)$ and its approximation $\hat{v}(s_t, \boldsymbol{w})$ (respectively, $Q_{\pi^*}(s_t, \alpha_t)$ and its approximation $\hat{q}(s_t, \alpha_t, \boldsymbol{w})$) as

$$\overline{VE} = \sum_{s \in S} \mu(s) \left[V_{\pi^{\star}}(s) - \hat{v}_{\pi}(s, \boldsymbol{w}) \right]^2,$$
(49)

$$\overline{QE} = \sum_{s \in S, \alpha \in \mathcal{A}} \mu(s, \alpha) \left[Q_{\pi^{\star}}(s_t, \alpha_t) - \hat{q}_{\pi}(s_t, \alpha_t, \boldsymbol{w}) \right]^2,$$
(50)

where $\mu(s), \mu(s, \alpha)$ are measures of the frequency a state or state-action pair is visited by the agent.

Once the mean (square) error, \overline{VE} , (\overline{QE}) , has been calculated we apply an optimisation algorithm (*e.g.*, batch gradient descent, stochastic gradient descent Adam or

Nesterov, see Chapter 7 - Artificial Neural Networks: layer architectures, optimizers and automatic differentiation for more details) to calculate the gradient of the error and update the weights.

In the framework of Gradient descent, an update rule for the weights can be given by

$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^k + a\left(U_t - \hat{v}(s_t, \boldsymbol{w}^k)\right) \nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w}^k), \tag{51}$$

$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^k + a\left(U_t - \hat{q}(s_t, \alpha_t, \boldsymbol{w}^k)\right) \nabla_{\boldsymbol{w}} \hat{q}(s_t, \alpha_t, \boldsymbol{w}^k), \tag{52}$$

where the quantity U_t is called the target. The target U_t is an estimation of $V_{\pi^*}(s)$, (or $Q_{\pi^*}(s_t, \alpha_t)$). Typically, we can use as a target the accumulated reward at the end of the episode, G_t . We note that as long as the expected target value is equal to the state or action value¹⁷, the update w^{k+1} leads to a local minimum of $\overline{VE}, \overline{QE}$.

In practice, since G_t is not known a priori, we use an estimate of the future rewards after the current state, s_t , by bootstrapping G_t as

$$G_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w}), \tag{53}$$

$$G_t = r_{t+1} + \gamma \hat{q}(s_{t+1}, \alpha_{t+1}, \boldsymbol{w}),$$
(54)

respectively. The weight updates can then be given by

$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^{k} + a\left(r_{t+1} + \gamma \hat{v}(s_{t+1}, \boldsymbol{w}) - \hat{v}(s_{t}, \boldsymbol{w}^{k})\right) \nabla_{\boldsymbol{w}} \hat{v}(s_{t}, \boldsymbol{w}^{k}),$$
(55)
$$\boldsymbol{w}^{k+1} \leftarrow \boldsymbol{w}^{k} + a\left(r_{t+1} + \gamma \hat{q}(s_{t+1}, \alpha_{t+1}, \boldsymbol{w}) - \hat{q}(s_{t}, \alpha_{t}, \boldsymbol{w}^{k})\right) \nabla_{\boldsymbol{w}} \hat{q}(s_{t}, \alpha_{t}, \boldsymbol{w}^{k}),$$
(56)

respectively. We note that these estimates are not unbiased because they contain the influence of $\hat{v}(s_t, \boldsymbol{w}), \hat{q}(s_t, \alpha_t, \boldsymbol{w}), i.e.$, they are dependent on \boldsymbol{w} . Therefore, there is no guarantee that the update with this estimate will help convergence towards a local minimum. However, in the case of a 1-step approximation of the target, the point that the algorithm converges is close to the local minimum [SB18].

Of course, this is a regression task that can be solved by many different supervised learning algorithms of ML (see **Chapter 2: Introduction to regression methods** for more details). Such a regression can be performed with the help of deep neural networks (DNNs). In the model-free approaches where the action values optimal $Q_{\pi^{\star}}(s_t, \alpha_t)$ are approximated, we call the DNN a Deep Q-Network (DQN).

Once the target action value function $Q_{\pi^*}(s_t, \alpha_t)$ (respectively state value function $V_{\pi}^*(s_t)$) have been successfully approximated using $\hat{q}^k(s_t, \alpha_t, \boldsymbol{w})$ (respectively action value $\hat{v}^k(s_t, \boldsymbol{w})$) we can apply any of the previously described learning methods to obtain an estimate of the new target optimal action value $Q_{\pi^*}^{k+1}(s_t, \alpha_t)$ and or the state value $V_{\pi^*}^{k+1}(s)$. Usually, the Q-learning method for n = 1 is used to update the target values as

$$Q_{\pi^{\star}}^{k+1}(s_{t_m}, \alpha_{t_m}) = r_{t_m+1} + \gamma \max_{\alpha_{t+1}} \hat{q}^k(s_{t+1}, \alpha_{t+1}, \boldsymbol{w}).$$
(57)

¹⁷Under this assumption the target is *unbiased*.

When a DQN is used for the function approximation of the action value $Q^k(s, \alpha)$ that will later be used by the RL learning algorithm (*e.g.*, Q-learning) for updating the target $Q^{k+1}(s, \alpha)$, is called *Deep Q-Learning*.

2.3.1 Policy gradients

Until now, we focused on learning the optimal action values, $Q(s_t, \alpha_t)$, that maximise the expected accumulated reward, $\max_{\alpha_t} \mathbb{E}[G_t]$. We then established that the optimal policy, $\pi^*(\alpha_t | \alpha_t)$, simply selects among the actions with the highest action value $\alpha_t^* = \operatorname{argmax}_{\alpha_t} Q(s_t, \alpha_t)$. However, we have also seen cases where the policy can also be tuned to get higher rewards more frequently (*e.g.*, Monte Carlo control). Now, we advance one step further and introduce a special class of policy gradient algorithms, in which we assume that the optimal policy function can itself be approximated by a linear combination of features θ_i , which constitute a vector $\boldsymbol{\theta}$. Therefore, this is a method that uses function approximation, *i.e.*, $\pi(\alpha_t | s_t) \approx \hat{\pi}(\alpha_t | s_t, \boldsymbol{\theta})$.

We will describe the policy gradient algorithm (which is called *REINFORCE* algorithm, see [Wil92]), which provided a conceptually simple rule for updating the vector of policy parameters $\boldsymbol{\theta}$. This rule aims at improving the expected cumulative reward at each time step of the trajectory by modifying the parameter vector $\boldsymbol{\theta}$ of the policy $\hat{\pi}(\alpha_t|s_t, \boldsymbol{\theta})$.

More specifically, during a trajectory, the cumulative reward for each state-action pair $s_t - \alpha_t$ is given by $G_t(s_t, \alpha_t) = \sum_{k=0}^{T'-1} \gamma^k r_{t+k-1}(s_{t+k}, \alpha_{t+k})$. For a stochastic policy $\pi(\alpha_{t+k}|s_{t+k})$ together with stochastic dynamics of the environment $p(s_{t+k}, \alpha_{t+k}, s_{t+k+1})$ the expected cumulative reward is equal to

$$\mathbb{E}_{\pi} \left[G_t | s_t, \alpha_t \right] = \sum_{k=0}^{T-1} p(s_{t+k+1}, \dots, s_0) \gamma^k r_{t+k+1}(s_{t+k}, \alpha_{t+k}, s_{t+k+1})$$
(58)

where $p(s_{t+k+1}, \alpha_{t+k}, s_{t+k}, ..., s_0)$ is the joint probability of a specific trajectory realization up to state action pair s_{t+k}, α_{t+k} .

We note that since the above process is a Markov process of independent events, the above joint probability of reaching the future state s_{t+1} following the trajectory $T_r = \{s_0, \alpha_0, r_1, s_1, \alpha_1, ..., r_{t+1}, s_{t+1}\}$ can be written as

$$p(s_{t+1}, r_{t+1}, \alpha_t, s_t, \dots, \alpha_0, s_0) = p(s_0)\pi(\alpha_0|s_0)p(s_1, r_1|s_0, \alpha_0)\pi(\alpha_1|s_1)p(s_2, r_2|s_1, \alpha_1) \\ \dots \pi(\alpha_{t-1}|s_{t-1})p(s_t, r_t|s_{t-1}, \alpha_{t-1})\pi(\alpha_t|s_t)p(s_{t+1}, r_{t+1}|s_t, \alpha_t).$$

This means that the joint probability of a trajectory reaching future state s_{t+1} and receiving reward r_{t+1} is the same as the product of conditional probabilities between the sequential states, (s_t, s_{t+1}) , following the policy of the agent $\pi(\alpha_t|s_t)$ and the dynamics of the environment, $p(s_{t+1}, r_{t+1}|s_t, \alpha_t)$ *i.e.* finding state s_{t+1} after performing action α_t in state s_t . Inserting equation 59 into equation 58 we obtain

$$\mathbb{E}_{\pi} \left[G_t | s_t, \alpha_t \right] = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}(s_{t+k}, \alpha_{t+k}, s_{t+k+1}) p(s_1, r_1 | s_0, \alpha_0) \dots \times \pi(\alpha_{t+k} | s_{t+k}) p(s_{t+k+1}, r_{t+k+1} | s_t, \alpha_t)$$
(59)

Normally because of the environment dynamics, this information is known only for model-based methods. We choose to approximate the agent's policy with a set of parameters θ , *i.e.*, $\pi(\alpha_t|s_t) \approx \pi_{\theta}(\alpha_t|s_t, \theta)$, therefore equation (59) can be approximated as

$$\mathbb{E}_{\pi_{\theta}} \left[G_t | s_t, \alpha_t \right] = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}(s_{t+k}, \alpha_{t+k}, s_{t+k+1}) p(s_{t+k+1}, r_{t+k+1} | \alpha_{t+k}, s_{t+k}) \pi_{\theta}(\alpha_{t+k} | s_{t+k}, \theta) \dots \times p(s_1, r_1 | \alpha_0, s_0) \pi_{\theta}(\alpha_0 | s_0 \theta) p(s_0).$$
(60)

In order to update the parameters vector, θ , towards the policy that maximises the expected total accumulated reward at the end of the episode, we will consider the gradient of the approximated expected accumulated value of a trajectory with respect to this vector as

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[G_t | s_t, \alpha_t \right] \approx \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}(s_{t+k}, \alpha_{t+k}, s_{t+k+1}) \dots \\ \times \nabla_{\boldsymbol{\theta}} \left[p(s_{t+k+1}, r_{t+k+1} | \alpha_{t+k}, s_{t+k}) \pi_{\boldsymbol{\theta}}(\alpha_{t+k} | s_{t+k}, \boldsymbol{\theta}) \dots p(s_1, r_1 | \alpha_0, s_0) \pi_{\boldsymbol{\theta}}(\alpha_0 | s_0, \boldsymbol{\theta}) p(s_0) \right].$$
(61)

Using the logarithmic property $\nabla_{\theta} \left(\log(f(\theta)) \right) = \frac{\nabla_{\theta} f(\theta)}{f(\theta)}$ for a given function $f(\theta)$, the last expression becomes

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[G_t | s_t, \alpha_t \right] \approx \sum_{k=0}^{T-1} \gamma^k r_{t+k+1}(s_{t+k}, \alpha_{t+k}, s_{t+k+1}) \\ \times \left[p(s_{t+k+1}, r_{t+k+1} | \alpha_{t+k}, s_{t+k}) \pi_{\boldsymbol{\theta}}(\alpha_{t+k} | s_{t+k}, \boldsymbol{\theta}) \dots p(s_1, r_1 | \alpha_0, s_0) \pi_{\boldsymbol{\theta}}(\alpha_0 | s_0 \boldsymbol{\theta}) p(s_0) \right] \\ \times \nabla_{\boldsymbol{\theta}} \log \left[p(s_{t+k+1}, r_{t+k+1} | \alpha_{t+k}, s_{t+k}) \pi_{\boldsymbol{\theta}}(\alpha_{t+k} | s_{t+k}, \boldsymbol{\theta}) \dots p(s_1, r_1 | \alpha_0, s_0) \pi_{\boldsymbol{\theta}}(\alpha_0 | s_0, \boldsymbol{\theta}) p(s_0) \right], \\ = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[G_t \nabla_{\boldsymbol{\theta}} \left(\log \pi_{\boldsymbol{\theta}}(\alpha_{t+k} | s_{t+k}, \boldsymbol{\theta}) + \dots + \log \pi_{\boldsymbol{\theta}}(\alpha_0 | s_0, \boldsymbol{\theta}) \right) \right], \\ = \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[G_t \nabla_{\boldsymbol{\theta}} \sum_{k=0}^{T-1} \log \pi_{\boldsymbol{\theta}}(\alpha_{t+k} | s_{t+k}, \boldsymbol{\theta}) \right].$$
(62)

We note that in this form the gradient update does not require any knowledge of the extraneous environment dynamics, *i.e.*, $p(s_{t+1}, r_{t+1}|\alpha_t, s_t, ..., \alpha_0, s_0)$ is not needed to evaluate the gradient of the expected accumulated return.

In practice, we cannot compute the expectation directly and therefore we estimate the mean by summing over different trajectories. Thus we obtain

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[G_t | s_t, \alpha_t \right] \approx \frac{1}{M} \sum_{m=1}^M \sum_{t_m=0}^{T_m} \gamma^{t_m} r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) \nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m} | s_{t_m}, \boldsymbol{\theta}) \right]$$
(63)

Now, we can update the values of the parameter vector $\boldsymbol{\theta}$ towards increasing the expected return $\boldsymbol{\theta} = \max_{\pi_{\boldsymbol{\theta}}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_t]$ as

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^M \sum_{t_m=0}^{T_m} \gamma^{t_m} r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) \nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m+1} | s_{t_m+1}, \boldsymbol{\theta}) \right],$$
(64)

where a is a hyperparameter (see MC method), t_m is the current time at episode m, T_m is the final time at episode m, and M is the batch size.

2.3.2 Baseline correction and Advantage function

The main problem with the *REINFORCE* algorithm presented above is the fact that the estimate of the gradient update contains a lot of variance due to the limited number of samples the method averages over. Moreover, the method inherently presents a problem when it averages over states that lead to high rewards because of *vanishing gradients*¹⁸.

We can further increase the convergence rate of the policy gradient by reducing the variance of the gradient without changing its expected value. This can be done by adding a baseline function $b(S_t)$ into the function of the cumulative reward G. Since this function is not a function of the policy parameters θ_i , it does not influence the gradient. The addition of the baseline allows us to better differentiate between good and very good actions. A good baseline function is the use of the state value function $V(s_t)$. This function is independent of the actions α_t and consequently from the policy parameters θ_i .

Using the Policy Gradient Theorem and the state value function $V(s_t)$ we can replace the cumulative reward $\gamma^t r(s, \alpha)$ with the Advantage function,

$$A_{\pi_{\theta}}(s_t, \alpha_t) = Q_{\pi_{\theta}}(s_t, \alpha_t) - V_{\pi_{\theta}}(s_t).$$
(65)

The values of the parameter vector $\boldsymbol{\theta}$ can then be updated using

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^M \sum_{t_m=0}^{T_m} \left(Q_{\pi_{\boldsymbol{\theta}}}(s_{t_m}, \alpha_{t_m}) - V_{\pi_{\boldsymbol{\theta}}}(s_{t_m}) \right) \nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m+1} | s_{t_m+1}, \boldsymbol{\theta}) \right]$$
(66)

¹⁸In a region of the state-action space where all accumulated rewards are high, the gradient of the cumulative reward estimate between the different states vanishes.

Replacing the action value function $Q_{\pi_{\theta}}(s_t, \alpha_t)$ we obtain,

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^{M} \sum_{t_m=0}^{T_m} \left(r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) + \gamma V_{\pi_{\boldsymbol{\theta}}}(s_{t_m+1}) - V_{\pi_{\boldsymbol{\theta}}}(s_{t_m}) \right)$$
$$\nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m} | s_{t_m}, \boldsymbol{\theta}) \right], \tag{67}$$

and when the action value function is used, we obtain,

$$\boldsymbol{\theta}^{k+1} \leftarrow \boldsymbol{\theta}^k + a \sum_{m=1}^M \sum_{t_m=0}^{T_m} \delta(s_{t_m}, \alpha_{t_m}, r, s_{t_m+1}, \alpha_{t_m+1}) \nabla_{\boldsymbol{\theta}} \log \left[\pi_{\boldsymbol{\theta}}(\alpha_{t_m} | s_{t_m}, \boldsymbol{\theta}) \right],$$
(68)

where

$$\delta(s_{t_m}, \alpha_{t_m}, r, s_{t_m+1}, \alpha_{t_m+1}) = r(s_{t_m}, \alpha_{t_m}, s_{t_m+1}) + \gamma Q_{\pi_{\theta}}(s_{t_m}, \alpha_{t_m+1}) - Q_{\pi_{\theta}}(s_{t_m}, \alpha_{t_m}).$$

We note that the quantity $\delta(s_{t_m}, \alpha_{t_m}, r, s_{t_m+1}, \alpha_{t_m+1})$ is the *TD-error* of the 1 step TD learning (SARSA) algorithm (see Section 2.2.7).

2.4 Summing up

In the configuration of section 2.3.1 all the tasks, *i.e.*, policy iteration (calculation of the gradients) and policy evaluation (prediction, calculation of the action values) are performed by the same algorithm. This is similar to the *policy iteration* scheme we visited in the classic DP approaches (see Section 2.1.4).

The difference between these approaches and the policy gradient method is that in order for the system to learn a policy of continuous actions we approximated the policy $\pi(\alpha_t|s_t) \approx \pi_{\theta}(\alpha_t|s_t, \theta)$ with a set of policy interpolation weights θ .

The biggest problem in the above approach lies in calculating the values of the action value and state value functions $(Q_{\pi\theta}(\alpha_t, s_t), V_{\pi\theta}(s_t))$, respectively. These functions are not known for the whole state-action space, and performing the steps of policy evaluation and value iteration is inefficient.

Instead of calculating the exact $Q_{\pi_{\theta}}$ we can use an approximation - *yet again* - of the action/state value function $Q_{\pi_{\theta}}(s_t, \alpha_t) \approx \hat{q}(s_t, \alpha_t, \boldsymbol{w}), \quad (V_{\pi_{\theta}}(s_t) \approx \hat{v}(s_t, \boldsymbol{w}))$ (see Section 2.3). In this way, the maximum value of the action value function approximation $\max_{\alpha_t} Q_{\pi\theta}(s_t, \alpha_t) \approx \max_{\alpha_t} \hat{q}(s_t, \alpha_t, \boldsymbol{w})$ can be used for the estimates of the model-free algorithms (*e.g.*, Q-learning algorithm).

Notice that the above procedure introduces three problematic elements in comparison to the original methods of DP^{19} . These are:

¹⁹These are also called *The deadly Triad*, see [SB18].

- Bootstrapping
- Off policy learning (e.g. due to the application of the max operator)
- Function approximations for the policy and value functions

Under these conditions, convergence is not assured and the algorithm might show signs of oscillations in the accumulated reward and even diverge. This general difficulty in RL applications leads to the need for large data sets and large training times. Moreover, specific regression architectures need to be put in place for the combination of the policy $\pi_{\theta}(\alpha_t|s_t, \theta)$ and action function value $\hat{q}(s_t, \alpha_t, w)$.

2.5 Actor-Critic Network (AC)

This is the most common method for handling Policy iteration and Value approximation performed by the DNN. This involves an actor network that takes actions and based on those actions accumulates rewards. In essence the actor network initializes with a random policy θ , $\pi_{\theta}(\alpha_t|s_t, \theta)$ and runs a set of trajectories inside the state-action space (s_t, α_t) of the problem, accumulating rewards, r_{t+1} , and visiting different future state-action pairs (s_{t+1}, α_{t+1}) .

As we discussed in the policy gradients and in the summary sections (see Sections 2.3.1 and 2.4), the approximation of the predicted action values

 $Q_{\pi_{\theta}}(s_t, \alpha_t) \approx \hat{q}(s_t, \alpha_t, w)$ needs to be used. This is done thanks to the separate DQN which is called the critic.

The DQN is also initialized with random weights w and uses the next states, s_{t+1} , actions, α_{t+1} , and rewards, r_{t+1} , obtained by the actor network²⁰, to find the maximum action value estimates $\hat{q}^k(s_{t+1}, \alpha_{t+1}, w)$.

The DQN network then calculates the maximum target action values $\hat{q}^{k+1}(s_t, \alpha_t, \boldsymbol{w})$ based on one of the algorithms of RL (Control, e.g Q-Value iteration, MC-control, SARSA and Q-learning)²¹. The DQN uses and applies the loss function between the target values and the actual predicted values of the action value function

 $L\left(\hat{q}_{\boldsymbol{w}}^{k+1}(s_t, \alpha_t) - \hat{q}_{\boldsymbol{w}}^k(s_{t+1}, \alpha_{t+1})\right)$, where $L = \|\cdot\|^p$ is a norm. Then it calculates the gradient of the loss to train the weights (see also **Chapter 7 - Artificial Neural Networks: layer architectures, optimizers and automatic differentiation** and **Chapter 8 - Artificial Neural Networks: advanced topics** for more details).

The converged values of the critic are then used to predict the values of action value function for the current state of the actor $\hat{q}_{w}^{k}(s_{t}, \alpha_{t})$ in the policy gradient estimation. The procedure repeats until the maximum reward is reached.

There are different variants of the AC network. For instance, instead of calculating the loss function of the actor with respect to the action value approximation, we can use the advantage function (see Section 2.3.2). The change between the actor-critic

²⁰We emphasize that the actor uses the policy $\pi_{\theta}(\alpha_t | s_t, \theta)$ to get $s_{t+1}, \alpha_{t+1}, r_{t+1}$.

²¹We emphasize that the policy used by the DQN is an ε -greedy policy when Q-Value iteration, MCcontrol or SARSA are used and a plain greedy policy when Q-learning is used.

(AC) and the Advantage actor-critic A2C is minimal if someone uses the SARSA algorithm in the critic network (see Section 2.2.7) and monitors the updates between $\hat{q}_{\boldsymbol{w}}^{k+1}(s_t, \alpha_t) - \hat{q}_{\boldsymbol{w}}^k(s_{t+1}, \alpha_{t+1})$. In the advantage actor-critic (A2C) the updates of the policy gradients are done using the error δ of the SARSA algorithm in the policy gradient. We will use this network in the next Section for controlling two different applications in Geomechanics.

3 Applications to Geomechanics

In this Section, we will present two interesting applications of earthquake prevention. In the first, a reduced model of earthquakes (the so-called *spring-slider* model) will be presented for avoiding a fast-slip behaviour. In the second, a geothermal reservoir is introduced to prevent seismicity rate while keeping energy production. In both examples, an RL method will be used to design a linear control and prevent earthquake-like behaviour. But first, we will start by giving a short introduction to Control Theory.

3.1 Control Theory: The basics

Control theory, a discipline within control engineering and applied mathematics, focuses on managing dynamical systems in engineered processes and machines. The primary goal of control theory is to establish models or algorithms that govern the application of system inputs, enabling the system to achieve a desired state or behaviour.

In this chapter, we will only talk about closed-loop control²². Such kind of control requires measuring an *output* from the system, calculating an error with respect to a desired reference, and then taking actions with an *input* depending on such signal, taking the states to a desired reference. The algorithm that gets the output as a *feedback* from the system and generates the signal for the input of the system is called a *controller*. This explains the name of *closed-loop* control (see Fig. 19). Therefore, in real applications, a control process requires elements to measure signals (sensors) and actuators to be the input of the system (*e.g.*, motors and valves).

Ktesibios of Alexandria is recognized as the inventor of float valves during the 3rd century BC. These valves were designed to regulate the water level of water clocks, marking a significant milestone as the earliest recorded instance of a controlled process in human history. In such an example, the water clock is the system to-be-controlled, the output is the position of the float valves, the reference is the desired water level, and the valve is the actuator.

Control theory can be classified depending on the system (linear and nonlinear), the number of inputs and outputs (Single-Input-Single-Output, Multiple-Input-Multiple-Output and its combinations), the design (classical using transfer functions or modern using state-space representation), or if there exist perturbations in the model (robust and non-robust), to name a few (see [Oga10, Kha02, SEFL14] for more details). We

²²Open-loop control, in contrast, is the one that does not depend on the output of the system.



Figure 19: A typical control system diagram.

will only introduce the most known controller: the Proportional-Integral-Derivative (PID) control.

Consider the next (very simple) second-order linear system

$$\dot{x}_1 = x_2,$$

 $\dot{x}_2 = u,$
 $y = [x_1, x_2]^T$
(69)

where y is the output and u is the input of the system. Such a system can be seen as a mechanical system, where x_1 represents a mass position, x_2 its velocity and u represents an acceleration that we can control. The objective is to take the displacement to a **constant** reference, r, by designing correctly the control input u. A classical approach is to take the system to its error dynamics by introducing the next change of coordinates: $e_1 = x_1 - r$, $e_2 = x_2 - \dot{r} = x_2$, obtaining the next system

$$\begin{aligned} \dot{e}_1 &= e_2, \\ \dot{e}_2 &= u. \end{aligned} \tag{70}$$

System (69) and (70) seem to be exactly the same, but there is a fundamental difference: if we can design u in system (70) to take the states, e_1, e_2 , to zero, we will solve the main goal of taking the original states, x_1, x_2 , in system (69) to the desired reference r. This is due to the introduced change of coordinates. For that purpose, let us present the PID control as

$$u = -k_1 e_1 - k_2 e_2 - k_3 \zeta, \dot{\zeta} = e_1,$$
(71)

where k_1, k_2, k_3 are control gains to be designed. Note how the PID takes the output $y = [x_1, x_2]^T$ in its design. This linear control takes the *present* (the proportional part e_1), the *past* (the integral part ζ) and the *future* (the derivative part e_2) of the error to drive it to zero. The question is, how can I design the three gains? Let us obtain the *closed-loop system* by substituting the PID control (71) in system (70) as

$$\dot{e}_1 = e_2,$$

 $\dot{e}_2 = -k_1 e_1 - k_2 e_2 - k_3 \zeta,$ (72)
 $\dot{\zeta} = e_1.$

The first thing that we notice is that the PID added another state to the system due to the integral part. Using simple linear algebra, we can obtain the characteristic equation of the system as $\lambda^3 + \lambda^2 k_2 + \lambda k_1 + k_3 = 0$, where λ represents the characteristics values of the system. In order to have exponential stability, *i.e.*, the three states e_1, e_2, ζ will tend to zero exponentially, the characteristic values must have a negative real part. This can be solved, *e.g.* by the Routh-Hurwitz criterion [Oga10], obtaining the next design rules for the gains

$$k_1 > 0, \quad k_2 > 0, \quad k_3 < k_1 k_2.$$
 (73)

Note how the selection of the gains to have exponential stability is very general and the final choice can be done by adding more requests for the stabilization, like choosing the exact characteristic values of the system by selecting desired time response (how fast we want to reach the origin) and the overshoot (maximum peak value of the response curve measured from the desired reference), to name a few.

In more realistic applications, the gain selection is not as easy to perform due to many factors, like nonlinearities, perturbations, unmodelled dynamics or noise. In the following, we will apply the PID control for two different applications of Geomechanics. Due to the challenging nature of such systems, we will use an RL algorithm to select the best gains for each application.

3.2 Reduced Model for Earthquakes: The spring-slider

The dynamics of earthquakes can be represented, in average/energetical sense, with the spring-slider analogue system (see [Ste19, Ste20, Sch02, KB04, TBS21, GOTSP23]) depicted in Fig. 20.



Figure 20: Reduced mechanical model for reproducing earthquake-like instabilities. Figure extracted from [GOTSP23].

This mechanical system consists of a mass, m, which slides on a frictional interface (equivalent to a seismic fault). The mass is connected to a Kelvin-Voigt configuration composed of a spring with stiffness k (equivalent to the apparent elasticity of the host rock) and a dashpot with damping coefficient η (equivalent to the apparent viscosity of the host rock). At the other extremity of the Kelvin-Voigt configuration a constant
velocity, ν_{∞} , is applied (equivalent to the far-field motion of the tectonic plates). It is assumed Coulomb friction with a friction coefficient $\mu(\delta, \dot{\delta})$ that depends on the slip δ (block's displacement) and the slip-rate $\dot{\delta}$ (block's velocity). According to Terzaghi's principle, the frictional stress τ takes the following form: $\tau = \mu(\delta, \dot{\delta}) (\sigma'_n - p)$, where σ'_n is the constant/reference average effective stress (*e.g.*, the overburden due to the weight of the rocks and the interstitial fluid pressure) and p the fluid pressure developed due to injecting fluid. p is the input to the system for which the controllers will be designed and tested.

According to [Sch02] and [KB04], approximately a rock mass of volume $L_{\rm ac}^3$ is mobilized during an earthquake event, where $L_{\rm ac}$ is equal to the length of the seismic fault. Therefore, the mobilized mass during an earthquake event is $m \approx \rho L_{\rm ac}^3$, where ρ is the density of the surrounding seismic fault rocks. The fault length can be calculated as $L_{\rm ac} = G/\bar{k}$, where G is the shear-modulus of the host rock and $\bar{k} = k/L_{\rm ac}^2$, its apparent normalized elastic stiffness. The damping coefficient η is given by $\eta = 2\zeta m\omega_n$, where ζ is the damping ratio and $\omega_n = \sqrt{k/m}$, the natural frequency of the reduced system.

The system in Fig. 20 can be represented in a state representation by the following mathematical model [Ste19, Ste20, TBS21, GOTSP23]

$$x_1 = x_2,$$

$$\dot{x}_2 = -[\mu(x_1, x_2) - \mu^*]\hat{N}\sigma'_n + \mu(x_1, x_2)\hat{N}p - \hat{k}x_1 - \hat{\eta}x_2,$$
(74)

where $x_1 = \delta$ and $x_2 = \dot{\delta}$ are the state variables, $\hat{N} = A/m$, $\hat{k} = k/m$, $\hat{\eta} = \eta/m$, and $A \approx L^2_{\rm ac}$ is the effective contact area (fault rupture area). The term μ^* represents an initial shifting in the frictional term which results in the system being on the verge of slip (it will start moving) if $\mu^* = \max \mu(x_1, x_2)$.

In this work, the friction coefficient $\mu(x_1, x_2)$ is defined as

$$\mu(x_1) = \mu_{res} - \Delta \mu \cdot e^{-x_1/d_c},\tag{75}$$

with $\Delta \mu < 0$. Such function is defined as a slip-weakening friction law [KB04] and it evolves from an initial value $\mu_{max} = \mu_{res} - \Delta \mu$ (static friction coefficient), to a residual one μ_{res} (kinetic friction coefficient) in a characteristic slip d_c . This will define our term μ^* as $\mu^* = \mu_{max} = \mu_{res} - \Delta \mu$. Notice that such a choice of friction coefficient was made for this academic example. Indeed, the exact frictional rheology is not known in real applications, which creates the need for the design of robust controllers²³. All the parameters of the reduced model for earthquakes (see eq. (74)) are given in Table 5.

Note that system (74) has an equilibrium point²⁴ at the origin $x_1^* = x_2^* = 0$, when p = 0. According to the nominal studies of [Ste19, Sch02, Die79, GOTSP23], a

²³A controller that copes with some uncertainties, *i.e.*, unknown system parameters, or disturbances, *i.e.*, an external dynamics affecting the system, is called a robust controller.

²⁴We call as equilibrium point to the point, x, in the state space where the derivatives are zero, *i.e.*, $\dot{x} = 0$.

Parameter	Description	Value
ρ	density	2500 [kg/m ³]
G	shear modulus	30 [GPa]
η	damping coefficient	$5 imes 10^{14}$ [kg/s]
L_{ac}	activated fault length	5 [km]
σ'_n	effective normal stress	50 [MPa]
μ_{res}	residual friction	0.2353
$\Delta \mu$	friction drop	-0.1
d_c	characteristic slip distance	276.35 [mm]
d_{max}	maximum displacement	785 [mm]
t_{op}	operation time	600 [s]
T_s	sampling time	0.5 [s]
M_0	seismic moment	6.25×10^{17} [Nm]
M_w	seismic magnitude	5.8

Table 5: Mechanical and frictional properties adopted for the simulations

dynamic instability (a fast-slip behaviour) will take place when the elastic unloading of the springs or the apparent viscosity of the host rock cannot be counterbalanced by friction. This is exactly what happens for system (74), (75), when there is no control (p = 0) and when the system parameters are chosen as in Table 5. Fig. 21 shows the fast-slip (earthquake-like) behaviour of such a system, corresponding to an earthquake of magnitude $M_w = 5.8$. The slip, x_1 , evolves from an initial state to a maximum displacement, d_{max} , due to the high value of the slip rate ($x_2 \approx 0.075$ [m/s]). The simulation was performed in Python with an implicit solver (BDF) and saving values every $T_s = 0.5$ [s]. The prevention of such fast slips is the main goal in the following.



Figure 21: Earthquake-like behaviour in the spring-slider model. The elastic stored energy is released abruptly in the form of kinetic energy, creating a fast slip. Note that such behaviour lasts only a couple of seconds.

3.2.1 Control Objective

As shown in system (74) and Fig. 20, the fluid pressure p is the only input acting on the dynamics of the mechanical system. In a real-scale scenario, fluid injections in the earth's crust change the fluid pore pressure over seismic faults [CSC⁺19]. As shown in [TBS21, GOTSP23], among others, this can destabilize the fault system and induce/trigger larger earthquakes. To prevent this, one could adequately adjust the fluid pressure (input p) by employing control techniques in order to stabilize the system (74) origin and/or track a reference input signal, releasing the stored elastic energy smoothly (x_1 evolving slowly), and not abruptly.

Therefore, the objective in the following is to design a control law p driving x_1 and x_2 in system (74) to follow some desired predefined references of slow slip rate, $r(t), \dot{r}(t)$, resulting in a slow-aseismic response, without the knowledge of the system parameters or its dynamics. The designed control will reduce the natural response time of the system slowing its energy dissipation and eliminating bursts of kinetic energy (earthquake phenomenon).

The desired reference for the output $y = x_1$ is a smooth function reading as

$$r(t) = d_{max}s^3(10 - 15s + 6s^2), \quad \dot{r}(t) = 3d_{max}\frac{s^2}{t_{op}}(10 - 20s + 10s^2), \quad (76)$$

where $s = t/t_{op}$, d_{max} the target displacement and t_{op} the operational time of the tracking strategy. The constant d_{max} is the distance the fault slides dynamically in order to reach the next stable equilibrium point. Notice that the parameter t_{op} is free to be decided depending on the earthquake control scenario that one wants to apply. Nevertheless, t_{op} has to be sufficiently high with respect to the characteristic time of the earthquake phenomenon, but low enough to achieve aseismic slip with higher velocity than the far-field velocity (v_{∞} in Fig. 20), for the control scenario to make sense. The chosen parameters of the reference can be seen in Table 5.

The choice of the reference output $y = x_1$ is motivated by the need to control the average slip over the fault. This average slip is directly connected with the magnitude of an earthquake through the seismic moment [KB04]. Therefore, by controlling the rate of the average slip, the system is forced to release its energy in a quasi-static way, *i.e.*, aseismically. See [Ste19, ST22, GOTSP23] for more details.

3.2.2 Control Design

As we did before, and following [GOTSP23], we will define the tracking error variables as

$$e_1 = x_1 - r, \quad e_2 = x_2 - \dot{r}.$$
 (77)

Note that if we design the input p in (74) such that e_1, e_2 tend to zero, we will do exactly what we want it, to force the slip and slip-rate to follow a reference that allows the slow dissipation of energy.

Let us study the error dynamics to see how the control p affects the new states e_1, e_2 . Such dynamic is written as

$$\dot{e}_1 = e_2, \dot{e}_2 = -[\mu(e_1 + r) - \mu^*]\hat{N}\sigma'_n + \mu(e_1 + r)\hat{N}p - \hat{k}(e_1 + r) - \hat{\eta}(e_2 + \dot{r}) - \ddot{r},$$
(78)

which is practically the same as (74) except for the new terms due to the reference and its derivatives.

If the exact knowledge of the system parameters and the system dynamics would be available, all the known dynamics in \dot{e}_2 could be fought with the input

$$p = \frac{u + [\mu(e_1 + r) - \mu^*] \hat{N} \sigma'_n + \hat{k}(e_1 + r) + \hat{\eta}(e_2 + \dot{r}) + \ddot{r}}{\mu(e_1 + r) \hat{N}},$$
(79)

to get the same (very simple) error system as in (70). For that case, we know that if we design the control input u as in (71), the errors would tend to zero by selecting the gains as in (73). This is obviously not the case in real applications where we can not know exactly the system parameters or its dynamics, making it impossible to implement the control (71),(79).

In [GOTSP23], a feedback control was designed with the knowledge of some nominal values of the system parameters and with some knowledge of the dynamics. However, the objective here is to design a controller, *p*, without any information on the system.

We will define our control, p, as the PID of equation (71), i.e.,

$$p = -k_1 e_1 - k_2 e_2 - k_3 \zeta,
\dot{\zeta} = e_1.$$
(80)

As discussed before, such linear control has three parts in order to drive the errors to zero, **if** the reference signal to be followed is constant. This is clearly not our case since the reference was chosen as (76). Nevertheless, as we will show later, such control will be enough to keep the states, x_1, x_2 , close to the references, r, \dot{r} , avoiding the earthquake-like behaviour.

Due to the fact that we do not know anything about the error dynamics (78), we can not design the gains of the control (80) in a conventional way (*e.g.*, pole location, Lyapunov-based or frequency-based [Oga10, Kha02]), For that purpose, we will use an RL approach in order to select the best possible gains to maximize a reward based on the error variables, dissipating slowly the stored energy of an earthquake. We will start by creating our environment, the Spring-Slider.

This application can be seen as the continuation of [PS21], where an RL approach was also used to avoid earthquake-like behaviour in the spring-slider, but without designing a control algorithm *a priori*, like the PID in (80).

3.2.3 Spring-Slider as environment

1 2

3 4 5

6

7

8

In order to train and implement our RL algorithm, an *OpenAI Gym*-based environment of the Spring-Slider will be constructed (see https://www.gymlibrary.dev/ content/environment_creation/ for more details on creating a custom environment).

For the spring-slider system (74) and the control input (80), we define as observations the states x_1, x_2 and the integral action ζ . The actions will be defined as the control gains k_1, k_2, k_3 . Therefore, the first step is to define the space of the observations and actions.

Note how we chose high values for the gains to try to compensate for all the unknown dynamics. Next, we define a function for the dynamics of the system, where we have used all the system parameters shown in Table 5:

```
1
     def rhs(t,y,p):
        #System parameters
2
3
         rho = 2500 #[kg/m^3]
         G = 30e9 #[Pa]
4
         eta = 5e14 #[kg/s]
5
6
        Lac = 5e3 #[m]
         sigma = 50e6 #[Pa]
7
         mu_res = 0.2353 #[-]
8
        delta_mu = -0.1
9
10
        dc = 276.35e-3 #[m]
         A = Lac**2 #[m^2]
11
         m = rho*Lac**3 #[kg]
12
         k = Lac*G \#[N/m]
13
14
15
         #Reference signal
         s = t/top
16
         ref = dmax*s**3*(10-15*s+6*s**2)
17
18
         #Spring-Slider states
19
20
         x1 = y[0]
         x2 = y[1]
21
22
         #Tracking error
23
         e1 = x1-ref
24
25
         #Friction coefficient
26
27
         mu = mu_res-delta_mu*np.exp(-np.abs(x1)/dc)
28
```

```
29 #Spring-Slider dynamics
30 x1d = x2
31 x2d = -(mu-(mu_res-delta_mu))*A*sigma/m +mu*A*p/m - k*x1/m - eta*x2/m
33 #Integral extension
34 integral = e1
35
36 return np.hstack((x1d,x2d,integrald))
```

Then, we can define our class (environment) "SpringSlider" as

```
class SpringSlider(gym.Env):
1
2
         def __init__(self):
            self.action_space = springslider_action_space
3
4
             self.observation_space = springslider_obs_space
5
             self.dmax = dmax #[m]
6
             self.tau = tau #[s]
7
             self.top = top #[s]
8
9
         def observation(self):
10
11
             return np.array([self.state[o] for o in self.observations])
12
         def reset(self):
13
14
            self.state = np.array([np.random.uniform(0,5e-4),np.random.uniform(0,1e-4),0])
             self.t = 0
15
             return self.state
16
17
         def step(self, action):
18
19
             #States and gains
             x1, x2, integral = self.state
20
             k1, k2, k3 = action
21
22
             #Reference signals and errors
23
             s = self.t/self.top
24
             ref = self.dmax*s**3*(10-15*s+6*s**2)
25
             refd = 3*self.dmax*s**2*(10-20*s+10*s**2)/self.top
26
27
             #Tracking errors
28
29
             e1 = x1-ref
             e2 = x2-refd
30
31
             #PID control
32
             p = -k1*e1 - k2*e2 -k3*integral
33
34
             #Solution of the system dynamics at every step tau
35
36
             sol = solve_ivp(rhs, y0=np.hstack((x1,x2,integral)),
             t_span=[self.t, self.t+self.tau], args=[p], method="BDF", t_eval=[self.t+self.tau])
37
38
             x1,x2,integral=sol.y[:,-1]
39
             self.t += self.tau
             self.state = (x1, x2, integral)
40
41
42
             #Norm of the errors and its condition for the reward
             Gamma = np.sqrt(e1**2+e2**2)
43
```

```
reward1 = np.exp(-Gamma)
44
45
              #Norm of the gains and its condition for the reward
46
47
              Gamma_k = np.sqrt(k1**2+k2**2+k3**2)
              max_k = np.sqrt(env.action_space.high[0]**2+env.action_space.high[1]**2
48
49
                     +env.action_space.high[2]**2)
              reward2 = 1-Gamma_k/(max_k)
50
51
52
              #Total reward
53
              alpha = 0.5
              reward = (1-alpha)*reward1+alpha*reward2
54
55
56
              #Conditions to stop the simulation
57
              done = (self.t >= self.top or np.abs(x2)>3e-3)
58
59
              return np.squeeze(self.state), reward, done, {}
60
          def close(self):
61
              self.isopen = False
62
```

We defined the observation and the reset functions, where we choose random numbers for the slip and slip-rate, and zero for the integral action, as initial conditions. The solution of the spring-slider and the integral extension dynamics will be obtained at every step using an implicit solver ("BDF"). The reference and its derivative have been designed as (76) and the control p has been designed as (80). Finally, the reward system has been selected depending on the norm of the errors

$$\Gamma(t) = \sqrt{e_1^2 + e_2^2},$$
(81)

and the norm of the control gains

$$\Gamma_k(t) = \sqrt{k_1^2 + k_2^2 + k_3^2},\tag{82}$$

as

reward =
$$(1 - \alpha)e^{-\Gamma(t)} + \alpha \left(1 - \frac{\Gamma_k(t)}{max_{\Gamma_k}}\right), \quad \alpha \in [0, 1].$$
 (83)

The first term gives rewards depending on how close the states are to the references, while the second gives rewards for using less control magnitude (smaller control gains). Furthermore, the conditions to finish the episode (the variable called "done") are when the time is equal to the operational time, t_{op} , or the velocity is bigger than a certain value. This last condition will force the ML algorithm to learn to avoid the fast-slip behaviour.

One can perform a simulation for the environment without control with the next code lines:

#Define the environment as the Spring Slider and check the initial state

2 env = SpringSlider()

1

```
3 obs = env.reset()
```

```
#Define the case without control, i.e., with the gains equal to zero
5
6
     action=np.array([0,0,0])
     nsteps=int(top/tau)
7
8
     #Store the states and the time in a variable
9
     x1sc = np.zeros(nsteps+1); x1sc[0] = obs[0]
10
     x2sc = np.zeros(nsteps+1); x2sc[0] = obs[1]
11
     integralsc = np.zeros(nsteps+1); integralsc[0] = obs[2]
12
13
     tsc = np.arange(nsteps+1)*tau
14
     #Simulation from t=0 to t=t_{op}
15
     for step in range(nsteps):
16
         obs. reward. done. info = env.step(action)
17
18
         x1sc[step+1]=obs[0]
19
         x2sc[step+1]=obs[1]
         integralsc[step+1]=obs[2]
20
```

In this case, the states and the time are stored in the variables x1sc, x2sc, tsc, where plots can be obtained to check the earthquake-like behaviour of the system (Fig. 21 has been obtained with this procedure).

3.2.4 Deep Deterministic Policy Gradient algorithm

The DDPG [LHP⁺19] was chosen as RL to select the best control gain, k_1 , k_2 , k_3 in (80) to maximize the reward system described previously. Such algorithms present an Actor-Critic network in order to train a **continuous** action. This is our case because the PID gains must be chosen according to (73). As explained in Section 2.5, the NN that parametrizes the Q-function is called "the critic", but it exists also the NN of the policy, which is called "the actor". The policy is basically the agent behaviour, a mapping from state to action (in the case of deterministic policy) or a distribution of actions (in the case of stochastic policy).

The parameters of the policy network have to be updated in order to maximize the expected accumulated reward $\mathbb{E}[G_t]$ defined in the policy gradient theorem, while the parameters of the critic network are updated in order to minimize the temporal difference loss (see Section 2.2.7 for more details on TD).

The actor takes the state as input to give an action as output, while the critic takes both state and action as input to give as output the value of the Q function. The critic uses gradient temporal-difference learning while the actor parameters are learned following the policy gradient theorem. The main idea behind this architecture is that the policy network acts producing an action and the Q-network criticizes that action.

The DDPG has been trained for 200 episodes and implemented in the constructed environment. These simulations were made in Python using a sampling time of $T_s = 0.5$ [s]. The results are shown in Figs. 22-23. The slip and slip-rate are close to the reference and its derivative, maintaining the norm of the errors below a certain value. The slip-rate has been made slower (one order of magnitude smaller!) than the

earthquake-like behaviour shown in Fig. 21, obtaining a slow aseismic behaviour. The control signal generated is shown in Fig. 23 and it is the result of the RL algorithm. Although the states always present an error in the tracking (see norm of the error in Fig. 23), these results fulfilled the task of avoiding the fast slip of the original system, by designing a control without the knowledge of any of the system parameters or its nonlinear dynamics.



Figure 22: Output tracking of a slow reference due to the linear control and the reinforcement learning approach. Note how the slip and slip-rate follow the desired reference releasing the stored energy one order of magnitude slower than the fast slip behaviour of Fig. 21.



Figure 23: Norm of the tracking error, $\Gamma(t)$ and control signal generated by the PID controller. The control signal was generated by the selection of gains performed by the DDPG algorithm, keeping the norm of the tracking error very close to zero.

3.3 Controlling induced seismicity in a Geothermal Reservoir

In the context of an Enhanced Geothermal System (EGS) [Cor19], one of the objectives is to enhance the permeability between two wells by creating a network of small cracks. These cracks facilitate the circulation of fluids between the wells, promoting

efficient energy extraction. However, the creation of these cracks is often accompanied by localized microseismic activity in the vicinity of the wells. This microseismicity is considered desirable as long as it does not significantly impact the overall seismicity rate over the larger region of the reservoir.

Let's consider an underground reservoir situated approximately 4 [km] below the Earth's surface, as illustrated in Figure 24. This reservoir consists of a porous rock formation that allows the flow of fluids through its pores and fractures. In our example, the reservoir has a thickness of approximately 100 meters and extends horizontally across a square surface with dimensions of approximately 5 [km] by 5 [km]. Within the reservoir, there are various injection points where wells are used to inject and/or extract fluids, such as water, as depicted in Figure 24. For the sake of simplicity, the term "injection of fluids" will encompass both the injection and extraction of fluids from the reservoir.



Figure 24: A diagram illustrating an underground reservoir with a thickness of approximately 100 meters and covering a square surface of dimensions approximately 5 [km] by 5 [km]. There are injection points marked within the reservoir where fluids are being injected and/or extracted using wells.

When fluids are pumped deep into the reservoir, it induces the circulation of fluids within the reservoir, resulting in the deformation of the surrounding porous rock. The hydro-mechanical behaviour of the reservoir, caused by the injection of fluids at depth, can be effectively described by Biot's theory [Bio41]. According to this theory, the diffusion of fluid and the deformation of the porous rock are dynamically coupled processes. However, under certain conditions, if the injection rates are sufficiently slow compared to the characteristic times of the system, considering the effects of inertia, and if the volumetric strain rate of the porous rock is negligible, the diffusion of fluid in the host rock due to fluid injections can be accurately approximated using a three-dimensional (3D) diffusion equation [ZCB80].

It is nowadays well established that injecting fluids into the Earth's crust can lead to the creation of new seismic faults and the reactivation of existing ones, resulting in significant earthquakes (refer to [RM15], [KSAC13], and [Zas19] for further information). The underlying physical mechanism behind these induced seismic events is

Parameter	Description	Value and Units
c_{hy}	Hydraulic diffusivity	$3.6 \times 10^{-4} [\text{km}^2/\text{hr}]$
D	Reservoir length	5 [km]
D_z	Reservoir depth	0.1 [km]
Q_{s_1}	Static flux	0.32 [m ³ /hr]
β	Mixture compressibility	1.2×10^{-4} [1/MPa]
V_w	Volume of the well	0.1 [km ³]
μ	Friction coefficient	0.5 [-]
$\dot{ au_0}$	Background stressing rate	1×10^{-6} [MPa/hr]
t_a	Characteristic decay time	500100 [hr]

Table 6: Diffusion and Seismicity rate system parameters

closely related to changes in stress within the surrounding rock caused by the fluid injections. These changes can either increase the loading on existing faults or reduce the friction along these faults or newly formed discontinuities. In other words, fluid injections contribute to an elevated seismicity rate in a particular region, referring to the increased number of earthquakes occurring within a given time frame (for more detailed information on seismicity rate, refer to [SL15] and [Die94]).

To illustrate this mechanism, let us consider an injection of $Q_{s_1} = 0.32 \text{ [m}^3/\text{hr]}^{25}$ through a single injection well as shown in Figure 25. In this numerical example, we consider the parameters of Table 6. We then calculate the seismicity rate, R_1, R_2 , over two distinct regions, one over the whole reservoir and one close to the injection point (see regions V_1, V_2 , respectively, in Figure 25, bottom).

In Figure 25 (top) we plot the seismicity rate in both regions as a function of time. We observe that the maximum seismicity rate over V_1 is equal to $R_1 = 45.91$, which means that 45.91 more earthquakes of a given magnitude are expected over region V_1 . The seismicity is even higher (but finite) close to the injection well.

3.3.1 Control Design

In this application, the control problem focuses on achieving a specific control objective: to deliberately increase the seismicity rate in a small region surrounding certain wells while maintaining a constant seismicity rate over the broader area of the reservoir. The aim is to actively manage and control the localized seismic activity, ensuring that it remains within acceptable limits while optimizing the circulation of fluids and energy production in the EGS.

Consider two new control wells, Q_{c_1}, Q_{c_2} , that will be the input of the system (see Fig. 26 for the localization of such control wells in the reservoir). We then perform a similar procedure as in the Spring-Slider application to tune two PID control (one

²⁵We use the notation Q_s for a static or fixed flux input and Q_c for a controlled pressure flux input. Do not confuse them with the action value function $Q(s_t, \alpha_t)$.



Figure 25: Seismicity rate in both regions, V_1, V_2 (left). Pressure's reservoir after 3 [years] (right). Both results were obtained with a constant injection rate, Q_{s_1} .

for each control input) with the DDPG algorithm to regulate the two seismicity rates of the previously detailed regions V_1, V_2 in two constant values $r_1 = 5, r_2 = 1$. The results are shown in Figs. 26–27. We achieve the goal of maintaining a constant seismicity rate of 5 in the small region V_1 while avoiding induced seismic events in the surrounding region V_2 . These results are obtained much faster than the case without control, where the steady state is achieved in 3 [years] (36 [months]) and in this case, it is achieved in 30 [months] (see Fig. 25 for comparison). Therefore, our strategy is able to change the response time of the system! The control signals generated are the result of the DDPG as before, and the norm of the tracking errors is not zero but is maintained very close to the origin.



Figure 26: Seismicity rate in both regions, V_1 , V_2 (left). Pressure's reservoir after 30 [months] (right). Both results were obtained by adding two control wells.



Figure 27: Static flux input Q_{s_1} and controlled flux inputs Q_{c_1}, Q_{c_2} (left). Norm of the tracking error, y_e (right).

This work is a preliminary result. All the details of the calculations and more interesting case studies of the geothermal reservoir are presented in [GOS23].

4 Conclusions

In this chapter, we have introduced the basic concepts of Reinforcement Learning (RL). Starting from the framework of Dynamic programming, we presented the *Bellman Equations*, the *Policy Improvement Theorem* and the *Bellman optimality conditions*. Finally, we presented the model-based methods of *policy iteration* and *value iteration* for finding the optimal state, action values and the determination of the optimal policy.

We expanded the presentation of Dynamic programming to the stochastic case and we introduced the framework of *Markov Decision processes* which generalises the notions of dynamic programming to the stochastic case.

When we don't have complete knowledge of the dynamics of the environment the *model based* methods of dynamic programming cannot be used. For this reason, we presented a second class of *model free* algorithms that can optimise the state and action value functions of the Dynamic Programming problem. Here we presented the algorithms of Monte-Carlo, Temporal differences and Q-learning.

When the number of states and actions in the problem at hand becomes large, the optimization of the values of the state and action value functions of the problem by the above model-free methods becomes intractable. For this reason, function approximation techniques are used to predict the values of the state and action value functions. These values are then used together with the model-free algorithms to improve the estimates of the state and action value functions.

Finally, based on the function approximation of the policy we presented the method of *policy gradients* that aims to improve the policy in order to maximize the expected

state and action values. The methods of direct function approximation of the optimal value functions and the policy gradient algorithm can be used together to provide a powerful actor-critic algorithm suitable for problems with continuous state and action spaces.

Based on an actor-critic network algorithm, we have presented two applications of geomechanics for preventing seismic events without the knowledge of the system dynamics or its parameters. In the first one, a reduced-order model for earthquakes is introduced and the designed control is able to release the stored energy in a slow manner, preventing the fast slip of an earthquake. In the second application, a geothermal reservoir is studied and controlled for preserving energy production, while preventing seismic events in the surrounding area.

The present chapter opens the possibility of more applications in the earthquake prevention field (presence of delays, noise in the sensors, uncertainties in the system parameters, to name a few), thanks to the ability to handle complex and unstructured environments of the RL algorithms.

Acknowledgment

The authors would like to acknowledge the European Research Council's (ERC) support under the European Union's Horizon 2020 research and innovation program (Grant agreement no. 757848 CoQuake). Furthermore, the authors would like to thank Dr Efthymios Papachristos for his support and feedback on the realization of this chapter.

References

- [ABC⁺16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), 2016.
- [Bel54] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.
- [Bio41] M. A. Biot. General theory of three-dimensional consolidation. *Journal of Applied Physics*, 12(155):155–164, 1941.
- [BT95] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE conference on decision and control*, volume 1, pages 560–564. IEEE, 1995.
- [C⁺15] Francois Chollet et al. Keras. https://github.com/fchollet/ keras, 2015.

- [Cor19] François H. Cornet. The engineering of safe hydraulic stimulations for EGS development in hot crystalline rock masses. *Geomechanics for Energy and the Environment*, 1(1):1, October 2019.
- [CSC⁺19] F. Cappa, M.M. Scuderi, C. Collettini, Y. Guglielmi, and J.-P. Avouac. Stabilization of fault slip by fluid injection in the laboratory and in situ. *Science Advances*, 5(3):eaau4065, 2019.
- [Dee22] Google Deepmind. Deepmind x ucl rl lecture series introduction to reinforcement learning. https://www.youtube.com/watch?v= TCCjZe0y4Qc&list=PLqYmG7hTraZDVH599EItlEWsU0sJbAodm, 2022.
- [DFB⁺22] Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414– 419, 2022.
- [Die79] J. H. Dieterich. Modeling of rock friction: 1. experimental results and constitutive equations. *Journal of Geophysical Research: Solid Earth*, 84(B5):2161–2168, 1979.
- [Die94] James H. Dieterich. A constitutive law for rate of earthquake production and its application to earthquake clustering. *Journal of Geophysical Research*, 99(B2):2601–2618, 1994.
- [Euc23] Jim Euchner. Generative ai. *Research-Technology Management*, 66(3):71–74, 2023.
- [Fu16] Michael C Fu. AlphaGo and Monte Carlo tree search: the simulation optimization perspective. In *2016 Winter Simulation Conference (WSC)*, pages 659–670. IEEE, 2016.
- [GBGM23] Roberto Gozalo-Brizuela and Eduardo C. Garrido-Merchan. ChatGPT is not all you need. A State of the Art Review of large Generative AI models, 2023.
- [Gér22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* "O'Reilly Media, Inc.", 2022.
- [GOOSP23] Diego Gutiérrez-Oribio, Yury Orlov, Ioannis Stefanou, and Franck Plestan. Robust boundary tracking control of wave PDE: Insight on forcing slow-aseismic response. Systems & Control Letters, 178:105571, 2023.
- [GOS23] Diego Gutiérrez-Oribio and Ioannis Stefanou. Robust Tracking for a 3D Diffusion Equation: Controlling Seismicity Rate in Geothermal Reservoirs. *Control Engineering Practice (submitted)*, 2023.

- [GOSP22] D. Gutiérrez-Oribio, I. Stefanou, and F. Plestan. Passivity-based control of underactuated mechanical systems with Coulomb friction: Application to earthquake prevention. *arXiv:2207.07181*, 2022.
- [GOTSP23] Diego Gutiérrez-Oribio, Georgios Tzortzopoulos, Ioannis Stefanou, and Franck Plestan. Earthquake control: An emerging application for robust control. theory and experimental tests. *IEEE Transactions on Control Systems Technology*, 31(4):1747–1761, 2023.
- [KB04] H. Kanamori and E. E. Brodsky. The physics of earthquakes. *Reports* on Progress in Physics, 67(8):1429–1496, 2004.
- [Kha02] H. Khalil. Nonlinear Systems. Prentice Hall, New Jersey, U.S.A., 2002.
- [KSAC13] K. M. Keranen, H. M. Savage, G. A. Abers, and E. S. Cochran. Potentially induced earthquakes in Oklahoma, USA: Links between wastewater injection7 and the 2011 Mw 5.7 earthquake sequence. *Geology*, 41(6):1060–1067, 2013.
- [Lap18] Maxim Lapan. Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more. Packt Publishing Ltd, 2018.
- [LHP⁺19] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv:1509.02971, 2019.
- [Oga10] Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, New Jersey, USA, 2010.
- [Ope23] OpenAI. Gpt-4 technical report. arXiv:2303.08774, 2023.
- [PGM⁺19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.
- [Pic17] Moxie Pictures. AlphaGo The Movie. https://www.youtube.com/ watch?v=WXuK6gekU1Y, 2017.
- [PS21] Efthymios Papachristos and Ioannis Stefanou. Controlling earthquakelike instabilities using artificial intelligence. *arXiv:2104.13180*, 2021.
- [RM15] J. L. Rubinstein and A. B. Mahani. Myths and facts on wastewater injection, hydraulic fracturing, enhanced oil recovery, and induced seismicity. *Seismological Research Letters*, 86(4):1060–1067, 2015.

- [SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.
- [SBW92] R.S. Sutton, A.G. Barto, and R.J. Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine*, 12(2):19–22, 1992.
- [Sch02] C. H. Scholz. The Mechanics of Earthquakes and Faulting. Cambridge University Press, USA, 2002.
- [SEFL14] Y. Shtessel, C. Edwards, L. Fridman, and A. Levant. Sliding Mode Control and Observation. Intuitive theory of sliding mode control. Birkhauser, New York, USA, 2014.
- [SHS⁺17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815, 2017.
- [SL15] Paul Segall and S Lu. Injection-induced seismicity: Poroelastic and earthquake nucleation effects. *Journal of Geophysical Research: Solid Earth*, 120(7):5082–5103, July 2015.
- [SSZ⁺22] Md Tanzil Shahria, Md Samiul Haque Sunny, Md Ishrak Islam Zarif, Jawhar Ghommam, Sheikh Iqbal Ahamed, and Mohammad H Rahman. A comprehensive review of vision-based robotic applications: Current state, components, approaches, barriers, and potential solutions. *Robotics*, 11(6):139, 2022.
- [ST22] Ioannis Stefanou and Georgios Tzortzopoulos. Preventing instabilities and inducing controlled, slow-slip in frictionally unstable systems. *Journal of Geophysical Research: Solid Earth*, 127(7):e2021JB023410, 2022.
- [Ste19] I. Stefanou. Controlling anthropogenic and natural seismicity: Insights from active stabilization of the spring-slider model. *Journal of Geophysical Research: Solid Earth*, 124(8):8786–8802, 2019.
- [Ste20] I. Stefanou. Control instabilities and incite slow-slip in generalized Burridge-Knopoff models. *arXiv:2008.03755*, 2020.
- [Sut90] Richard S Sutton. Integrated modeling and control based on reinforcement learning and dynamic programming. In R.P. Lippmann, J. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3. Morgan-Kaufmann, 1990.
- [Sut99] Richard S Sutton. Reinforcement learning: Past, present and future. In Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning, SEAL'98 Canberra, Australia,

November 24–27, 1998 Selected Papers 2, pages 195–197. Springer, 1999.

- [TBS21] G. Tzortzopoulos, P. Braun, and I. Stefanou. Absorbent porous paper reveals how earthquakes could be mitigated. *Geophysical Research Letters*, 48(3):e2020GL090792, 2021.
- [VEB⁺17] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft ii: A new challenge for reinforcement learning, 2017.
- [VSS⁺19] Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants, and Valentino Zocca. Python Deep Learning: Exploring deep learning techniques and neural network architectures with Pytorch, Keras, and TensorFlow. Packt Publishing Ltd, 2019.
- [Wil92] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.
- [WIW89] Chelsea C White III and Douglas J White. Markov decision processes. *European Journal of Operational Research*, 39(1):1–16, 1989.
- [Zas19] M. Zastrow. South Korea accepts geothermal plant probably caused destructive quake. *Nature*, 2019.
- [ZCB80] O. C. Zienkiewicz, C. T. Chang, and P. Bettess. Drained, undrained, consolidating and dynamic behaviour assumptions in soils. *Geotechnique*, 30(4):385–395, 1980.

©ALERT Geomaterials Laboratoire 3SR / Bâtiment Galilée CS 40700 38 058 Grenoble cedex 9 France

ISBN 978-2-9584769-1-5

Fon: +33 (0) 456 528 621 Fax: +33 (0) 476 827 043 president@alertgeomaterials.eu http://alertgeomaterials.eu

All rights reserved. No part of this book may be reproduced in any form without written permission from the publisher or author.

ALERT Doctoral School 2023

Machine Learning (ML) in Geomechanics

Editors: I. Stefanou & F. Darve

Stefanou

Overview of Machine Learning

Masi

Introduction to regression methods

Jakse

Unsupervised Learning: Basic Concepts and Application to Particle Dynamics

Jakse

Classification Techniques in Machine Learning

Karapiperis

Data-Driven Modeling in Geomechanics

Sun

Non-Euclidean machine learning for geomechanics

Gatti

Artificial Neural Networks: layer architectures, optimizers and automatic differentiation

Gatti

Artificial Neural Networks: advanced topics

Masi & Stefanou

Physics-informed and thermodynamics-based neural networks

Stathas, Gutiérrez-Oribio & Stefanou

Introduction to Reinforcement Learning with Applications in Geomechanics

ISBN 978-2-9584769-1-5