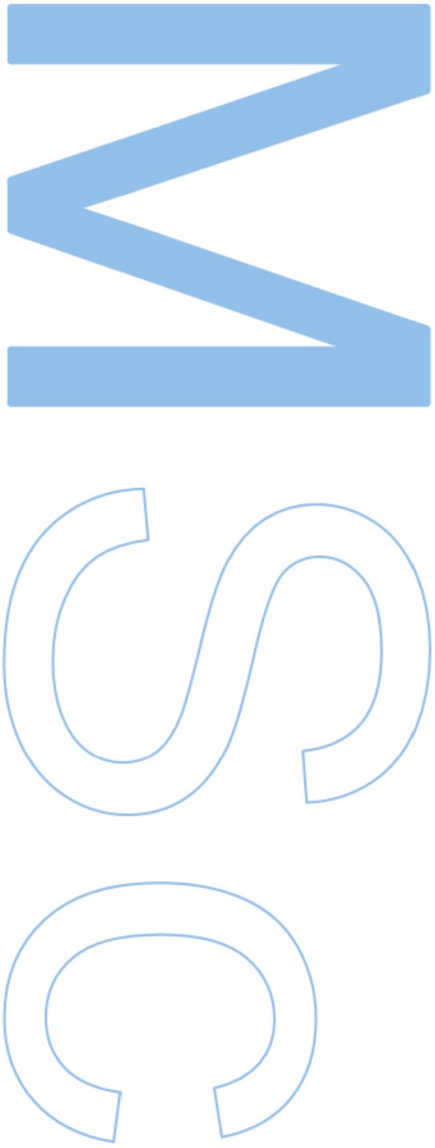
Automated physical law discovery from data using physics informed deep learning

José Miguel de Oliveira Bastos
Mestrado em Engenharia Física
Departamento de Física e Astronomía
2022

Supervisor
João Pedro Pedroso, Professor Associado, FCUP, Portugal

Co-supervisors
Erik Hostens, Senior Researcher, Flanders Make, Belgium
Agusmian Ompusunggu, Cranfield University, Belgium



Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ___/___/_____

Sworn Statement

I, José Bastos, enrolled in the Master's Degree in Physics Engineering at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this internship report reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this internship report, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This internship report does not include any content whose reproduction is protected by copyright laws.

I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

José Bastos
08-11-2022

Universidade do Porto

Masters Thesis

Automated physical law discovery from data using Physics Informed Machine Learning

Author:
José Bastos

Supervisor:
João Pedroso

Co-supervisor:
Erik Hostens

Co-supervisor:
Agusmian Ompusunggu

A thesis submitted in fulfilment of the requirements
for the degree of MSc. Engineering Physics

at the

Faculdade de Ciências da Universidade do Porto
Departamento de Física e Astronomia

January 29, 2023

" Fall in love with some activity, and do it! Nobody ever figures out what life is all about, and it doesn't matter. Explore the world. Nearly everything is really interesting if you go into it deeply enough. Work as hard and as much as you want to on the things you like to do the best. Don't think about what you want to be, but what you want to do. Keep up some kind of a minimum with other things so that society doesn't stop you from doing anything at all. "

Richard P.Feynman

Acknowledgements

It feels like yesterday that I was entering in FCUP, but six years have passed, and I'm about to become a Physics Engineer. These were six challenging, demanding, stimulating, and fun years. And in the end, although I know it is a cliché, I wouldn't be able to get to the finish line without all the help of family, friends, colleagues, and professors. So I leave here all my sincere thanks to all that helped me and made this journey fun, enriching, and amazing.

FCUP

To João Pedro Pedroso, for promptly accepting to be my supervisor and helping me do an internship abroad.

To all my friends, I want to thank you for all the fun and happy moments, and your company in the not so fun moments as well. You made all the stress-full moments easier, knowing I was not the only one struggling, that it was normal to feel lost, helped me relax and get through the difficult times.

Flanders Make

To Agusmian Ompusunggu, for accepting me at Flanders Make, making my goal of doing the Thesis Internship abroad possible. Thank you for introducing me to Physics Informed Machine Learning, and giving me a chance to work on this exciting topic. I am grateful for all the guidance and help when we worked together.

To Erik Hostens, for taking me under his supervision when he didn't have to. Thank you for all your help, discussions, and guidance. Thank you for challenging me to do better and improve.

To Kerem Eryilmaz, for all the headaches I gave you with my questions and technical problems. You helped me a lot with day-to-day issues, even if it meant interrupting more important work. Thank you for your patience and, most important, your friendliness.

To Jeroen Zegers, for all the ideas, input, and suggestions you contributed. Thank you for the patience you had to explain things to me (even with fewer hours of sleep due to your newborn daughter).

Thank you all in Flanders Make (and Leuven), for your sympathy, welcomeness, and hospitality. It was excellent working in such a young, dynamic environment. If the

weather and food were like here in Porto, I would not hesitate and say I would love to keep working with you.

Family and friends

To my parents, thank you for the unconditional support, the stability, and the flexibility you always gave me. Thank you for allowing me to make my decisions and let me learn with the outcomes. I know I am privileged to have you as parents. You gave me all the tools and opportunities I needed to be able to get to this point. I am forever grate-full.

To Carlota, thank you for always being with me. For your support, ability to make me laugh, and the patience. I know it weren't the easiest 6 years, but we made it.

To my friends, thank you for all the fun, silliness, jokes, and all the good and bad times we had together.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Física e Astronomia

MSc. Engineering Physics

Automated physical law discovery from data using Physics Informed Machine Learning

by José Bastos

We studied a novel Deep Learning framework called Physics Informed Neural Networks (PINNs), that adds physics related knowledge to Deep Learning models, improving generalisation and performance when compared to regular Deep Learning models. We used the PINNs framework to first determine the underlying physics (i.e. determining the underlying Partial Derivative Equation (PDE)) and then help make better predictions/forecasts of two systems: an axially moving vibrating string and weather. We studied these two systems to see how PINNs can be used in real-world scenarios and what are their advantages and disadvantages when compared to regular neural networks (without physical added knowledge) or existing PDE solvers (e.g. finite differences methods).

We concluded that PINNs allowed to determine the PDE, up to an error of 3%, of the axially moving string system. However, when it came to a more complex system as the weather, PINNs could not determine the underlying PDE. In the system where the PDE was determined, PINNs greatly improved generalisation (forecasting) when compared to regular Deep Learning models. Nevertheless, when compared to finite differences methods, PINNs still show some drawbacks: accuracy, stability, computational costs, and convergence.

Finally, we conclude that PINNs seem very promising when it comes to determine the underlying physics of a system and improve existing Deep Learning models. We acknowledge there are disadvantages when using PINNs for solving PDEs, such as:

convergence, stability, and choosing an appropriate training scheme. However, we believe more research in this area should solve many of these issues, and thus PINNs will probably become a mainstream method of solving PDEs in the near future.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Física e Astronomia

Mestrado Integrado em Engenharia Física

Determinação automática de leis físicas a partir de data, usando Physics Informed Machine Learning

por José Bastos

Neste trabalho estudámos uma nova *framework* denominada *Physics Informed Neural Networks* (PINNs), que adiciona conhecimento físico a modelos de *Deep Learning*, melhorando a sua capacidade de generalizar e o seu desempenho no geral. Mais especificamente, usámos PINNs para primeiro determinar a física subjacente (i.e. usámos PINNs para determinar as Equações Diferenciais Parciais (PDEs)) e em seguida obter melhores previsões da evolução de dois sistemas: um fio com velocidade horizontal a vibrar e previsão metereológica. Estudámos estes dois sitemas para perceber quais as vantagens e desvatagens das PINNs em contexto industrial quando comparadas com redes neuronais "normais"(sem informação extra da física do sistema) e com métodos numéricos de resolução de PDEs (e.g método de diferencças finitas).

Conlcuímos que as PINNs permitem determinar a PDEs, até um erro de 3%, de um sistema como o do fio a vibrar. No entanto, que quando se tratava de um sistema mais complexo, como o da meterologia, não foi possível detemrinar o sistema de PDEs. No caso do sistema do fio a vibrar, em que conseguimos determinar a PDE, as PINNs melhoraram bastante a generalização comparativamente a modelos de *Deep Learning* normais, mas quando comparados a métodos de resolução de PDEs tradicionais, como o método das diferenças finitas, as PINNs ainda apresentam algumas desvantagens: precisão,estabilidade, convergência e tempo de treino.

Finalmente, conluímos que as PINNs são muito promissoras no que toca a determinar a física subjacente de um sistema e a melhorar modelos de *Deep Learning*. Temos consciência de que há disvantagens a usar PINNs para a resolução de PDEs, tais como,

precisão, convergência, estabilidade e escolha do esquema de treino correto. No entanto, pensamos que com a crescente investagação nesta área alguns destes problemas possam ser resolvidos num futuro próximo e fazer com a PINNs se tornem melhores métodos para resolver PDEs.

Contents

List of Figures

List of Tables

Glossary

FCUP	Faculdade de Ciências da Universidade do Porto
PINN(s)	Physical-Informed Neural Network(s)
PIML	Physics-Informed Machine Learning
PDE(s)	Partial Differential Equation(s)
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
NN	Neural Network
LT	Linear Threshold Unit
MLP	Multi layer perceptron
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
MSE	Mean Squared Error
MAE	Mean Absolute Error

Chapter 1

Internship context

Machine Learning (ML) has taken a huge role in our daily lives; it is present when one uses Google, Amazon, Twitter, when our email sorts the spam, or even when we interact with the support centre of most companies. Now, Machine Learning is becoming more and more prevalent in industrial settings, some examples are: detecting faults in production lines, quality control, drug discovery, robotics and much more. However, there is a gap from what is researched at Universities and top companies (Meta, Twitter, Amazon, Google) from what gets to be used in real-world scenarios in industry. Most of the Machine Learning methods are devised and built in "simulated" scenarios and when implemented in real settings fail, because of the lack of good quality data, computing power, time, or even real improvement when compared to what was being used before.

One of the main problems of ML, nowadays, is to be able to implement methods that are designed in Research and Academia environments in real-world and industrial settings. This has become one of the main focus of ML research, mainly because it is in those cases where the biggest possible gains are (increase productivity, save costs, and much more). This internship was made with this precise thought: trying to implement an ML framework to industrial problems; more precisely, a framework that combines Deep Learning (a sub-field of Machine Learning) with Physics knowledge called *Physics-Informed Machine Learning*.

Recently, I took an interest in Machine Learning because it combined Mathematics with Computer Science, which are two areas that I enjoy. During quarantine, I did many online courses and read more and more about it. Finally, I decided that I wanted to do my Master's Internship on something related to Machine Learning to really see if it was something that I liked and could go deeper into it.

I was lucky enough to find an internship opportunity that combined both Physics and Machine Learning. I could not hope for a better topic! I could still work with physics, but now I had the opportunity to add ML to the mix.

For 9 months I had the opportunity to learn much more about ML and, in particular, get to know a new area of research named Physics Informed Machine Learning, in a Research-like environment with close guidance and help from 3 researchers.

1.1 Goals of the Internship

The full title of the Internship was ***Automated physical laws discovery in noisy measurement data using a deep learning technique to solve industrial problems***, and it clearly states that the main goal of the project was to discover the physical laws of systems placed on an industrial setting using Deep Learning. Discovering the physical laws was not the only goal; on top of it, it was intended to use the same Deep Learning approach to compute correct predictions of the system's future behaviour. Particularly, particularly, 2 cases were studied:

1. Determining the underlying physics of a wire with axial horizontal speed vibrating between 2 pulleys.

2. Weather forecast on an offshore wind farm.

In both cases, the goal was to determine the physical equations governing the system using a novel machine learning framework called Physics-Informed Machine Learning (PIML). At the same time, we intended to be able to determine the behaviour of the system in the future. In the vibrating wire problem, we wished to correctly predict the wire's position after we stopped "recording" it. On the weather forecast problem, it was important to get the governing system of Partial Derivatives Equations (PDEs), then correctly forecast weather was more of an ambitious goal and an additional goal (in the results chapters we will see why).

The **vibrating wire case** reflects many real-world scenarios, for example: a wire vibrating inside a machine, a belt vibrating in a production line, a wire between two electricity poles and much more. The fact that this problem could have real-world applications was the main reason why it was chosen. The other important reason was that Flanders Make has a client/partner, Bekaert, which is a "world market and technology

leader in steel wire transformation". Flanders Make had already worked with Bakaert on problems of predicting a wire's behaviour, so this was another approach to try to tackle this problem.

The **weather forecast problem** is obviously a real-world problem, but it was chosen because Flanders Make participated in a project called WaterEye. WaterEye, an European funded project, had the goal to improve monitoring and general health of offshore wind-farms. Flanders Make had the task of testing different possible ways of predicting weather at wind farm locations, making it possible to predict future corrosion and estimate the power generated by wind turbines. As such, we studied the performance of PINNs as a weather forecasting method.

The internship had a duration of 9 months, starting in September 2021 and ending in June 2022. It was carried out on site in Leuven, Belgium. Halfway through, because of the pandemic situation, and work-from-home was imposed during the month of December 2021, and I returned to the offices in January 2022.

During the full period of the internship, the work pattern consisted of biweekly (twice per month) tasks/feedback from the supervisors, followed by actually doing the work by myself until the next week's meeting. Then I would get new feedback and repeat the process.

In January 2022 my original supervisor, Agusmian Ompusunggu, left Flanders Make, and Erik Hostens took his place. The work flow remained unchanged, except that the meetings became more frequent and less spaced between each other. In June 2022 the internship ended, and to mark the end of the internship I gave a presentation, to some Flanders Make personnel, describing the work I had done during those 9 months.

1.2 Overview of the Company - Flanders Make

Flanders Make is a strategic Research Centre for the manufacturing industry. They focus on industry-driven technological research and innovation together with and to the benefit of SMEs (Small and Medium Enterprises) and large companies within the Flemish manufacturing industry. Their focus is on open innovation through excellent research in the field of Mechatronics, Machine Learning, and various methods for developing products. Flanders Make employs around 750 people and is located in the Flemish part of Belgium, mainly in Leuven, Kortrijk, and Lommel. Flanders Make is focussing more and more on using Machine Learning to tackle 4 types of problems that industry faces:

1. Sensing, monitoring, control, and decision-making for products and production.
2. Design and optimisation of products and production.
3. Product specification, architecture, and validation.
4. Flexible assembly specification, architecture, and validation.

Flanders Make mostly aims to shorten the gap between research and industry in Machine Learning, by building real-world ML frameworks for its clients/partners. This was the main reason why I choose to do my Internship in Flanders Make, it would allow me to both gain ML theoretical knowledge while, at the same time, get the chance to do research that could also be ”applied” and it not only theoretical. Flanders Make also has close ties to the Katholieke Universiteit Leuven (KU Leuven), one of the best Universities of Europe. This means that most of its researchers come from the KU Leuven and many projects are done in partnership with it. In fact, Flanders Make was ”born” inside the KU Leuven and still is a Research Center rather than a normal company.

1.3 Programming Language, Frameworks and Code Availability

It is important to refer that ***Python*** was the programming language used. Due to the large flexibility of Python it was possible to do almost all the work (simulating data, training Deep Learning models, download open source datasets) only using it. We decided to use ***PyTorch*** [1] as Deep Learning framework because it is more flexible, allows easier customization, and it is more geared towards research. PyTorch allowed us to easily build Deep Learning training schemes without having to build everything from scratch, giving us the ability to focus only on the essential. In addition, we also used ***PyTorch Lightning*** [2], which is a framework built on top of PyTorch to help to simplify training Deep Learning models even further; when one already has a working training scheme and wants to serialize it, (for example for hyperparameter optimization) PyTorch Lightning is tremendously helpful.

All the code is available on GitHub at `https://github.com/Jose-Bastos/Master-Thesis` . Most of it was written in ***Jupyter Notebooks*** because it is easier to run and debug. The repository is organized by the following structure:

```
/
└── String
    └── Notebooks
```

```
        ├── String_Data_Generation.ipynb
        ├── Coefficients.ipynb
        ├── Extrapolate.ipynb
        └── GIFS.ipynb
    ├── GIFs
    ├── Resolution
    ├── Spatial
    │   ├── Data ...  This directory holds the data files for this
    │   │             specific parent folder.
    │   └── Models ...  This directory holds the models files for every
    │                   run for this specific parent folder.
    ├── Temporal
    │   ├── Data
    │   └── Models
    ├── Noise
    │   ├── Data
    │   └── Models
    └── C_and_v
        ├── Data
        └── Models
└── Weather
    ├── Notebooks
    │   ├── Weather1D.ipynb
    │   └── Weather2D.ipynb
    ├── deepymod ...  This directory contains the DeePyMoD package source code.
    ├── Images
    ├── Dataframes ...  This directory holds dataframes files with the data.
    ├── requiremts.txt
    └── README
```

Although folder and notebook names are self-explanatory, here are some important details:

- String/Notebooks/GIFs.ipynb generates GIFs (and images) given model and data files

- Each data and model folder under String/Resolution, String/Noise, and String/C_and_b has subfolder for Coefficients and PINNs runs.
- It could be the case that some code does not run because of the way the folders were rearranged, fixing whatever directory naming issue raised should be enough.

1.4 Dissertation Outline

After this brief introduction comes a theoretical introduction to Machine Learning (ML) and Deep Learning (DL). This chapter is fundamental if the reader is not familiarized with basic concepts of ML and DL, which are required to then understand what are PINNs. It was written having in mind that my graduation area differs, Physics, from Computer Science and ML in particular, so a small introduction is needed for readers that come from the natural sciences area. Then, in Chapter 3 comes an introduction to, details, and current research on PINNs. In Chapters 4 and 5 we discuss the work done and the results in the vibrating wire and weather forecast cases respectively.

Finally, we conclude this work with final remarks and future work in Chapter 6.

All results, neural networks used, hyperparameters, and string parameters are present in the Appendix.

Chapter 2

Theoretical Introduction

In this chapter, I will introduce the fundamental to understand the work done in this dissertation. I will start by introducing Machine Learning (ML) in general and then Deep Learning (DL). In the end, I will introduce the novel Physics Informed Neural Networks (PINNs), which are the main focus of the work done throughout the internship. In each area, I will go through the fundamental ideas and how they are connected to each other.

2.1 Artificial Intelligence

First, I want to clearly distinguish between Artificial Intelligence, Machine Learning, and Deep Learning.

A concise definition of Artificial Intelligence(AI) would be the following: *The effort to automate intellectual tasks normally performed by humans.*[3] AI is a general field that encompasses others such as Machine Learning, Deep Learning, hard-coded rules, and Symbolic AI. In the beginning of AI (mid-1950s) up until the 1990s, Symbolic AI was the dominant paradigm. Symbolic AI programs are based on creating explicit structures and behaviour rules. Symbolic AI programs learn to understand the world by forming internal symbolic representations of its "world".

Although Symbolic AI proved suitable for solving well-defined logical problems, such as playing chess, it turned out to be inadequate when used to figure out explicit rules for solving more complex problems, such as image classification, speech recognition, and language translation. A new approach emerged to replace symbolic AI: machine learning.

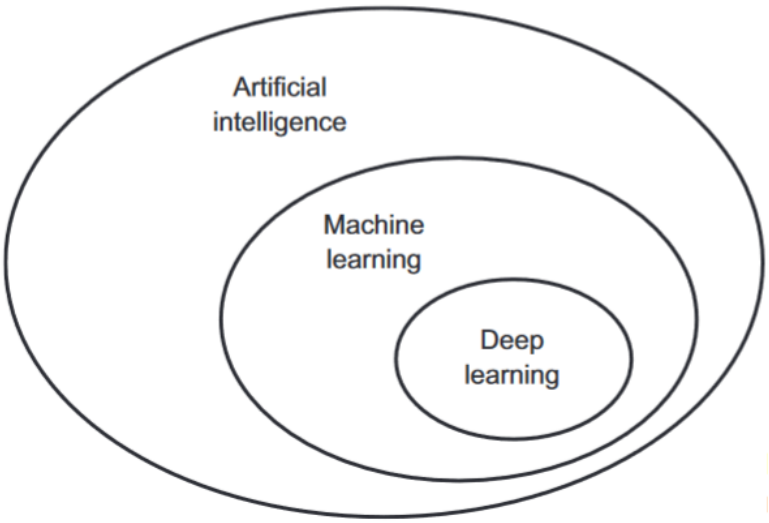


FIGURE 2.1: Artificial Intelligence, Machine Learning and Deep Learning [3]

2.2 Machine Learning

Could we have programs/systems that could learn how to perform certain tasks, instead of being just a static program that only knows how to do what we order it to do?

This question led to a new paradigm in programming: Instead of giving explicit rules to our systems, we now could *train* them to come up with rules/strategies that better suit the problem at hand.

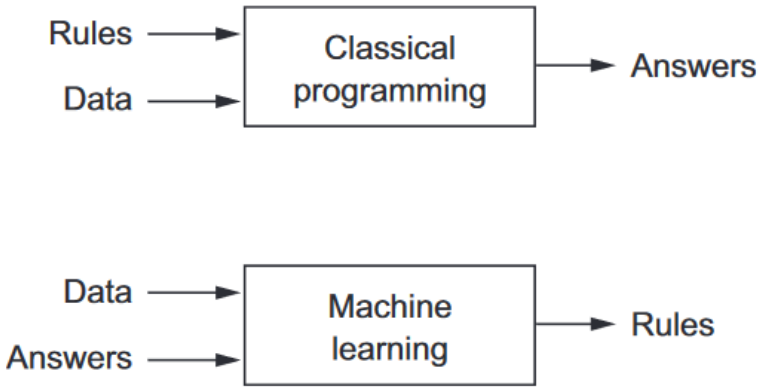


FIGURE 2.2: Machine Learning: a new paradigm[3].

Fig. 2.1 shows the paradigm introduced by Machine Learning. In classical programming, we would hard-code rules that our programme would apply to the data. In Machine Learning we let our programs learn patterns from data. In a more formal definition:

> *Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed.*
>
> (Arthur Samuel, 1959)

The introduction of Machine Learning enabled computers to tackle problems involving knowledge of the real world and make decisions that appear subjective. But still, how does a programme really *learn*?

The Learning Part of Machine Learning

To have a functioning ML algorithm, we need 3 key things:

1. ***Input data*** - Data fed to the algorithm, for example, emails when we are trying to build a spam filter.
2. ***Expected output*** - The ground truth, by which we can compare the output of our model and draw conclusions.
3. ***A way to measure how good our algorithm is performing*** - This is fundamental to check how our algorithm is doing and how it could be doing better. By comparing the expected output to the output, it is possible to correct the mistakes being made by the model. This serves as a feedback signal to adjust the way the algorithm works. This adjustment is called *learning*.[3]

By continuously adjusting its outputs in order to get a better result in the performance measurement, our algorithm "learnt" a way of transforming input data into correct outputs. Thus:

> *Machine-learning models are all about finding appropriate representations for their input data—transformations of the data that make it more amenable to the task at hand, such as a classification task.*
>
> (François Chollet,2017 in [3])

So, that's what machine learning technically is: searching for useful representations of some input data, within a predefined space of possibilities, using the guidance of a feedback signal. In other words, when our algorithm learns, it just finds a way to model the data it has seen. This simple idea gave us image classification, speech recognition, forecasting revenues, detecting anomalies and much more.

2.3 Deep Learning

Biology Inspiration

Biological neurons release short electrical impulses called *action potentials* (or signals) which travel along its axons. Upon reaching the end of its axons, called *synaptic terminals*, it releases chemical signals called *neurotransmitters*. The following neuron, which is connected through synapses to the one releasing the neurotransmitters, upon receiving a sufficient amount of these neurotransmitters releases its own electrical impulses.

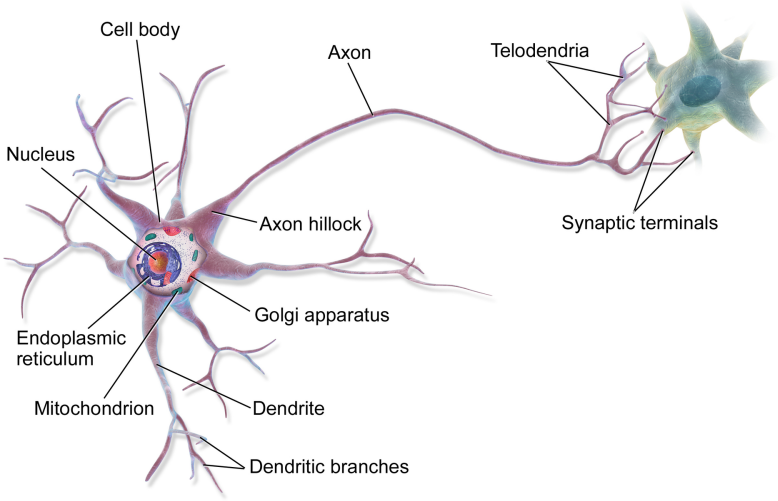


FIGURE 2.3: Neural Circuit [4].

Although a single neuron behaves in a simple way, in our body, a single neuron is usually connected to thousands of other neurons, making it possible to "build" and "transmit" very complex messages. This grouping and connection of many neurons is called a ***Biological Neural Network***(BNN).

2.3.1 The 1^{st} Artificial Neuron

In 1943, McCulloch and Pitts [5] proposed a new model of the biological neuron, this model would latter become known as the *artificial neuron*. Their artificial neuron would activate its outputs when more than a certain number of its inputs were active. In this historical paper, they were able to build networks (groups of neurons) that performed simple logical computations (OR, AND, NAND, and more).

2.3.2 The Linear Threshold Unit

In 1957, Frank Rosenblatt slightly modified the Artificial Neuron and created the ***linear threshold unit***(LTU), also known as ***Perceptron*** or just ***neuron***. Now, the inputs to the neuron are numbers (instead of binary on/off values), and each input is associated with its own weight. A representation of the LTU can be seen in Fig. 2.4. Instead of being active if a certain number of its inputs are active, the LTU computes the weighted sum of its inputs:

$$z = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n + b = \boldsymbol{W}^T \boldsymbol{X} + b \quad [1] \tag{2.1}$$

With z being the output of the weighted sum, $x_1, x_2, \ldots, x_n$ the inputs of the neuron, and $w_1, w_2, \ldots, w_n$ the corresponding weight of each input.The ***weight matrix***, $\boldsymbol{W} = [w_1, w_2, \ldots, w_n]^T \in \mathbb{R}^{n \times 1}$ [2], is transposed to match the dimensions of the matrices. A ***bias term*** is also added, we are going to see why in the following paragraphs. I decided to omit the bias terms from the images to make them simpler, but keep in mind that every neuron (LTU) has a respective bias term that is added to its inputs.

After the weighted sum, the LTU the applies a ***step function***, also called the Heaviside step function. Outputting the following result:

$$h_{w,b}(x) = Y = step(\boldsymbol{W}^T \boldsymbol{X} + b) \tag{2.2}$$

With $h_w(x)$ and Y representing the output of the LTU. Subscripts w and b are used to show that the output is also dependent on the weights and biases. We say h(x) is parameterised by w and b. Here we can see that the bias term shifts the step function horizontally, this allows to fit the data better because we can now predict functions of the type $y = ax + b$ and not just $y = ax$. The bias term should be seen as just another parameter of the neuron, just as weights.

[1] when variables appear in bold, like in $\boldsymbol{X}$ and $\boldsymbol{W}$, it's to show that they are not scalar variables but rather vectors/matrices.

[2] Some prefer to define $\boldsymbol{W} = [w_1, w_2, \ldots, w_n]$ and not transpose it twice. It's a matter of taste, as long as the dimensions match correctly.

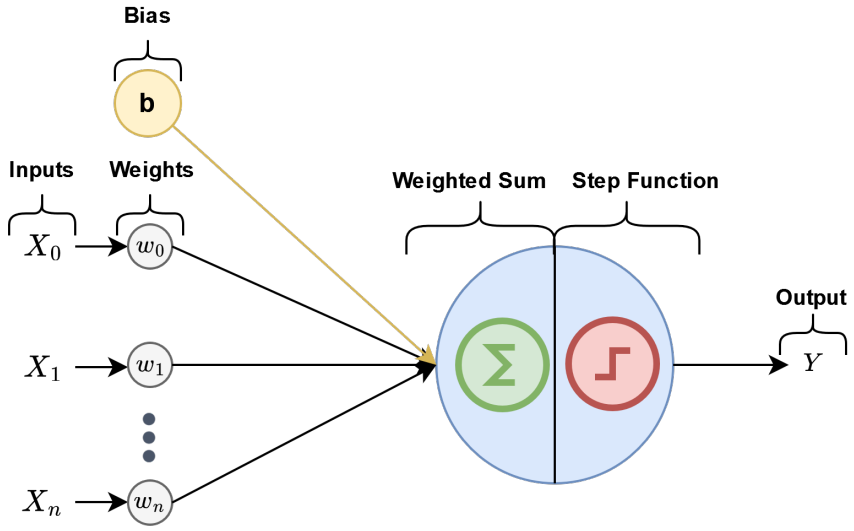


FIGURE 2.4: Scheme of a Linear Threshold Unit.

The key idea in LTUs is the flow of the inputs, which can be described as the following:

$$Inputs \rightarrow Weighting \rightarrow Summing \rightarrow Applying\ a\ function(e.g\ step\ function) \rightarrow Output \tag{2.3}$$

Remember that in Machine Learning, and consequently in Deep Learning, we want to model the data/inputs. If we are using the LTU, $h_w(x)$ is our attempt to model the input x. And $h_w(x)$ is what comes out of the step function, described by eq. 2.4, i.e. the step function restricts what we can model. With the Heaviside function, we could build a binary classifier; it would compute a linear combination of the inputs, and if the result exceed a threshold, it would output a positive value. Otherwise, it would output a negative or null value.

$$heaviside(z) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ if } z \geq 0 \end{cases} \tag{2.4}$$

The non-linear function that is applied after the sum is called the ***activation function***, in our case the activation function was the Heaviside function. Instead of using the Heaviside function as the activation function, we could use other functions that better suit the problem at hand. Later, we will see that it is better, in certain cases, to use differentiable functions (which the Heaviside function is not) as activation functions.

One of the greatest advantages of the LTU was the fact that by using ***non-linear*** activation functions we could model non-linear behaviour/data, something that isn't possible in most other Machine Learning approaches. Also, while gaining this capacity of modelling non-linear behaviour, the LTU can still model linear behaviour because of the linear combination that occurs when multiplying each input by its weight and then summing all the terms together. In this linear/non-linear nomenclature, eq. 2.3 can be written as:

$$Inputs \rightarrow Linear\ Transformation \rightarrow Non\text{-}Linear\ Transformation \rightarrow Output \tag{2.5}$$

By using different activation functions for different goals, Deep Learning became very versatile and powerful. The recent boom (2010s) in Deep Learning interest and usage came when new activation functions (combined with new neural networks architectures) were used and shown to solve many of the, until then, existent problems.

2.3.3 The Feed Forward Neural Network

The term ***Perceptron*** is used interchangeably with LTU, but sometimes it is used to refer to a ***layer*** of LTUs. An example of such layer is shown in Fig. 2.5.

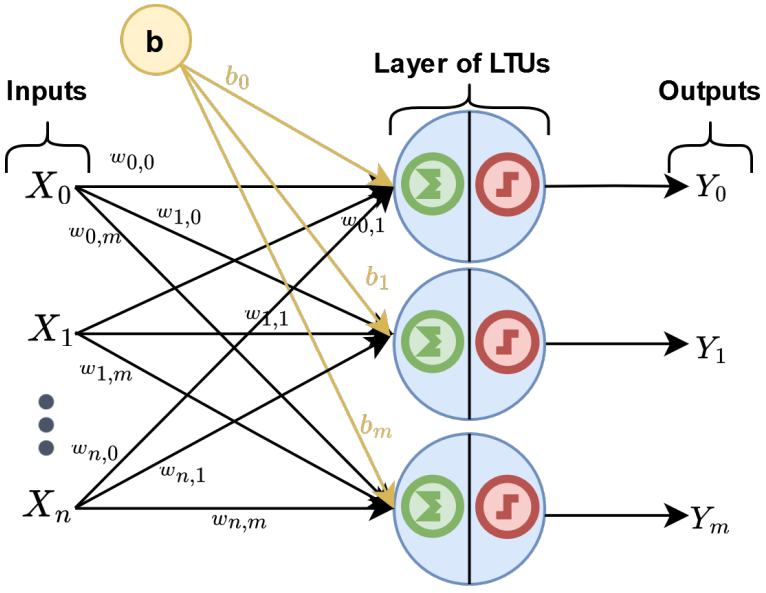


Figure 2.5: Single Layer of LTUs.

To compute the output of each LTU, eq. 2.2 still holds. But now each connection between input and neuron has a corresponding weight, so now $\boldsymbol{W} \in \mathbb{R}^{n \times m}$, with m being the number of neurons in the layer. Note that n and m are not equal most of the

time, that is: *the number of neurons in the layer does not have to be the same as the number of inputs*. The ***bias matrix*** is the column vector of all the bias terms, i.e.:

$$\boldsymbol{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \tag{2.6}$$

By grouping LTUs in layers, we add more complexity to the way we can model the inputs; in other words, by adding more LTUs (plus weights and biases), we add more ways in which we can combine the data to build truthful representations of the input. Using this simple concept of LTUs we can model very complex patterns/behaviour in data.

The term ***Multi-layer Perceptron*** (MLP) is used when referring to a stack of multiple layers of LTUs, and they are the simpler/fundamental ***neural networks*** used in Deep Learning. MLPs are also called ***feed forward neural networks***. The term neural network stems from the similarity that these stacks of layers of artificial neurons have with the huge networks of neurons in brains.

The ***deep*** in ***Deep Learning*** comes from this idea of successive layers in a neural network. The number of layers in Deep Learning models is called the ***depth*** of the model.

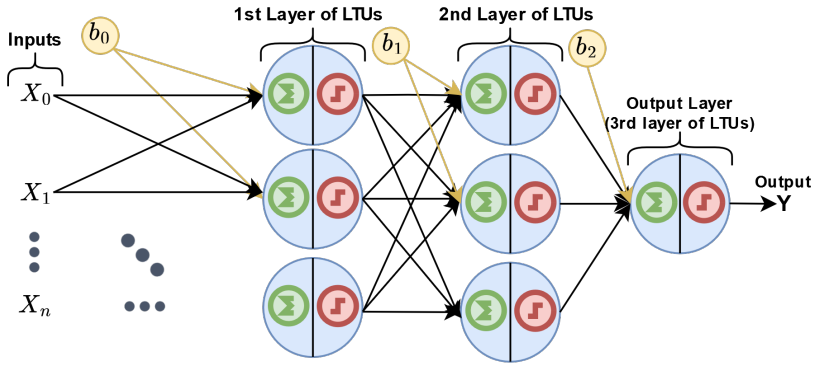


FIGURE 2.6: A MLP or feed forward neural network.

We write the output of each neuron just like in 2.2, but after the 1st layer of neurons, the inputs to each neuron in the layer after become the outputs of the previous ones

multiplied by each corresponding weight (plus bias term). So, we can write the output of the *l-th* layer as the following:

$$\boldsymbol{h}_{\boldsymbol{w}}^{(\boldsymbol{l})} = \sigma^{(l)}(\boldsymbol{W}^{\boldsymbol{T}(l)}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(\boldsymbol{l})}) \tag{2.7}$$

The term $\boldsymbol{h}^{(l-1)} \in \mathbb{R}^{n^{(l-1)}}$ being the vector of outputs from the previous layer (l-1) and σ is the activation function of the *l-th* layer (so far we have worked only with the step function as the activation function, but there are many more used). The weight matrix $\boldsymbol{W} \in \mathbb{R}^{n^{(l-1)} \times n^{(l)}}$, is a matrix that combines all the weights for the layer *(l)* in the following way:

$$\boldsymbol{W}^{(\boldsymbol{l})} = \begin{bmatrix} w_{0,0}^{(l)} & w_{0,1}^{(l)} & \dots & w_{0,n^{(l)}}^{(l)} \\ w_{1,0}^{(l)} & w_{1,1}^{(l)} & \dots & w_{1,n^{(l)}}^{(l)} \\ \dots & \dots & \ddots & \dots \\ w_{n^{(l-1)},0}^{(l)} & w_{n^{(l-1)},1}^{(l)} & \dots & w_{n^{(l-1)},n^{(l)}}^{(l)} \end{bmatrix} \tag{2.8}$$

Meaning, the (i,j) element of this matrix is the weight of the connection that goes from the neuron i in layer *l-1* to the neuron j in layer l. The superscript *(l-1)* in n is to show that $n^{(l-1)}$ is the number of neurons in the *l-th* layer. The same notation applies to the number of neurons in the *l-th* layer, $n^{(l)}$.

The bias matrix of the *l-th* layer, $\boldsymbol{b} \in \mathbb{R}^{n^{(l)}}$is the following:

$$\boldsymbol{b}^{(\boldsymbol{l})} = \begin{bmatrix} b_0^{(l)} \\ b_1^{(l)} \\ \vdots \\ b_{n^{(l)}}^{(l)} \end{bmatrix} \tag{2.9}$$

In which the *i-th* element is the bias for the neuron i in layer l. Each neuron has its own bias term.

2.3.4 The *learning* part of Deep Learning

So far we have seen how the building block of Deep Learning, the LTU, works and how can we stack multiple layers of LTUs to form a neural network. But how does the *learning* part of Deep Learning happen?

Recall that, in the end, the goal of a Multi-layer Perceptron is to approximate some function f^*. This is the same to say that, it defines a mapping from its inputs to the

outputs, such that $\boldsymbol{y'} = f(\boldsymbol{X}; \boldsymbol{\theta})$. When saying that our MLP learns is to say that it found the value of the parameters $\boldsymbol{\theta}$ that result in the best approximation of f^*. And these parameters of a neural network $\boldsymbol{\theta}$ are nothing other than the weights and biases of each layer. Something that can be visualised in Fig. 2.7. The process of gradually adjusting the value of the weights of a neural network is called ***training*** a neural network.

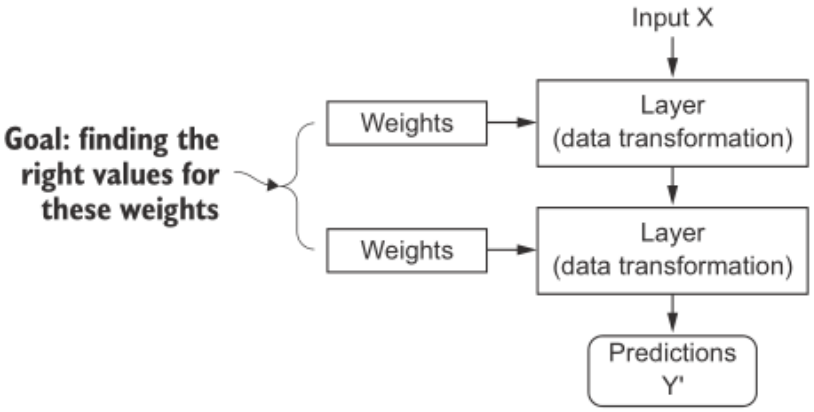


FIGURE 2.7: The goal of training a neural network [3].

As said before, in order to train a neural network, one needs to be able to measure how far the current output is from the expected output. This is the job of the ***loss function***, also called the ***objective function***. The loss function takes the predictions of the network and the true target and computes a ***distance score***, which says how well the network has performed in this specific input/target pair. With the value of the loss function, we can adjust the value of the weights a little in a direction that will lower the loss score. This adjustment is the job of the ***optimiser*** which implements what is called the ***Gradient Descent*** algorithm. Gradient Descent is the engine of Deep Learning, is what makes the clogs move, is what makes the network's parameters converge to the optimal values, essentially is what allows the network to *learn*.

Note that in our case, the network learns by comparing its output with the true value (which are also called ***labels***). We are learning a function that maps an input to an output based on example input-output pairs.

2.3.4.1 Gradient Descent and Backpropagation

So far we have seen that our neural network, composed of multiple layers of LTUs, gets the inputs, applies linear and non-linear transformations to it in each layer, and then

outputs one (or multiple) values. After it, we compare that output with the expected value in some kind of distance measure (the loss function). But how do we adjust the parameters(i.e. the weights) of our neural network?

The answer is intuitive and could be easily understood with the following analogy: Imagine being stuck on top of a mountain and it is really foggy, such that you can not see much of the path down the mountain. So, in order to get down, you look at what you can see around you and take a small step in the direction that goes down the most (and it's safe!), i.e. the direction with the steepest descent. Eventually, if you repeat this process for many steps, you would arrive at the bottom of the mountain.

In neural network models, the mountain is the loss function, and we want to get to its minimum value (the bottom). So, we compute the loss function, then its ***gradient***, which is the direction and rate of the fastest increase of a differentiable function. Then, we take a step in *the opposite direction* of the gradient. After doing these steps many times, we should eventually get to a point where the loss function reaches a minimum value.

What does "taking a step" mean? The set of weights and biases of each layer is what really makes a neural network change its output; when we say taking a step, we mean slightly adjusting all the weights and biases of our network. In order to do this, we need to compute the gradient of the loss function with respect to the weights and biases of all layers of our network. The way we do it is using the ***Backpropagation*** algorithm, which leverages the ***Chain Rule*** to propagate the derivatives from the outputs to the inputs (the name Backpropagation comes from this idea of computing the derivatives starting from the end and propagating them till the beginning of the neural network).

In a more formal definition, gradient descent requires the calculation of the gradient of the loss function $L(\boldsymbol{X}, \theta)$ with respect to the weights $w_{i,j}^{(l)}$ and biases $b_i^{(l)}$ of the *l-th* layer. Then, applying the desired ***learning rate*** α, which is just a way to define the size of the steps we want to take. Each iteration of gradient descent updates the weights and biases, collectively denoted by θ, using the following equation:

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial L(\boldsymbol{X}, \theta^t)}{\partial \theta} \tag{2.10}$$

Where θ^t denotes the parameters of the neural network at iteration t of gradient descent. The derivative term, also called ***Gradient Vector***, is just the vector that stores the derivative of the loss with respect to all network parameters (weights and biases):

$$\frac{\partial \mathscr{L}(\boldsymbol{X}, \theta^t)}{\partial \theta} = \nabla \mathscr{L}(\boldsymbol{X}, \theta^t) = \begin{bmatrix} \frac{\partial \mathscr{L}}{\partial w^{(0)}} \\ \frac{\partial \mathscr{L}}{\partial b^{(0)}} \\ \vdots \\ \frac{\partial \mathscr{L}}{\partial w^{(L)}} \\ \frac{\partial \mathscr{L}}{\partial b^{(L)}} \end{bmatrix}_t \tag{2.11}$$

Each derivative term in 2.11 is computed using the backpropagation algorithm that we will see now. Recalling that the Chain Rule allows us to compute the derivatives of composite functions and can be written as the following:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} \tag{2.12}$$

Here, f is a function of g, which in turn is a function of x, so we want to compute the derivative of f(g(x)). If we focus on the output/last layer (L) of our network and define the relevant variables as:[2]

$$\boldsymbol{z}^{(L)} = \boldsymbol{W}^{(L)}\boldsymbol{h}^{(L-1)} + b^{(L)} \tag{2.13}$$

$$\boldsymbol{h}^{(L)} = \sigma(\boldsymbol{z}^{(L)}) \tag{2.14}$$

Where z is the result of the linear transformation, the superscript L-1 denotes the layer before the last, and h is the output of the layer after the non-linear transformation. With the output of the last layer we can compute a loss value, using ***Mean Squared Error(MSE)*** (which is the most common metric used), we write:

$$\mathscr{L} = (\boldsymbol{h}^{(L)} - y)^2 \tag{2.15}$$

With $\mathscr{L}$ denoting the loss and y the target value. The MSE simply computes the squared difference between the true value and the output of our model. MSE is used when dealing with regression problems. Recall that the goal is to compute the derivative of the loss with respect to the weights and biases, and then compute eq. 2.10.

We will focus on the weights, but the thinking is the same for the biases. We want to compute $\frac{\partial \mathscr{L}}{\partial w^L}$, which is the derivative of the loss with respect to the weights of the last layer L. Is it recommended to first look at this as if we only have 2 layers, the last

[2]It is usual to define the last layer of a neural network as L, but do not mix it with the loss written as $\mathscr{L}$.

and penultimate layers of a network, each with 1 neuron, and then add more layers and neurons after we understand the fundamental ideas. The computational graph is the following:

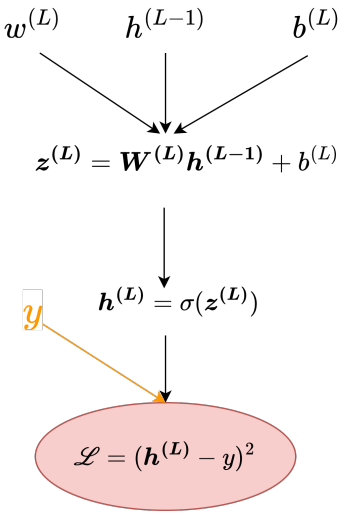


Figure 2.8: Computational graph of the last layer of a neural network.

Here, $z^{(L)}$, $h^{(L)}$, $w^{(L)}$ are calculated using equations 2.13, 2.14, and 2.15. To compute $\frac{\delta \mathscr{L}}{\delta w^{(L)}}$ we need to apply the chain rule, because $\mathscr{L}$ is a function of $h^{(L)}$, which in respect is a function of $z^{(L)}$, which in turn is a function of $w^{(L)}$, which is the desired variable that we want to differentiate in order to. Looking at Fig. 2.8 and using the chain rule we get:

$$\frac{\delta \mathscr{L}}{\delta w^{(L)}} = \frac{\delta \mathscr{L}}{\delta h^{(L)}} \frac{\delta h^{(L)}}{\delta z^{(L)}} \frac{\delta z^{(L)}}{\delta w^{(L)}} \tag{2.16}$$

We now have a derivative we couldn't compute straight away, $\frac{\partial \mathscr{L}}{\partial w^{L}}$, written as a multiplication of 3 derivatives we can easily compute just by looking at their analytical expressions in eqs. 2.13, 2.14, and 2.15. In the end, we get the following equation:

$$\frac{\delta \mathscr{L}}{\delta w^{(L)}} = 2(h^{(L)} - y) \times \sigma'(z^{(L)}) \times h^{(L-1)} \tag{2.17}$$

Now we have the derivative of the loss with respect to the weights of the last layer, we still miss the derivative of the loss with respect to the weights of the penultimate layer, $w^{(L-1)}$. Looking at the computational graph of the 2 layers represented in Fig. 2.9, we can see that applying the chain rule just as before, starting from the loss and ending at $w^{(L-1)}$ gives us $\frac{\partial \mathscr{L}}{\partial w^{L-1}}$ as:

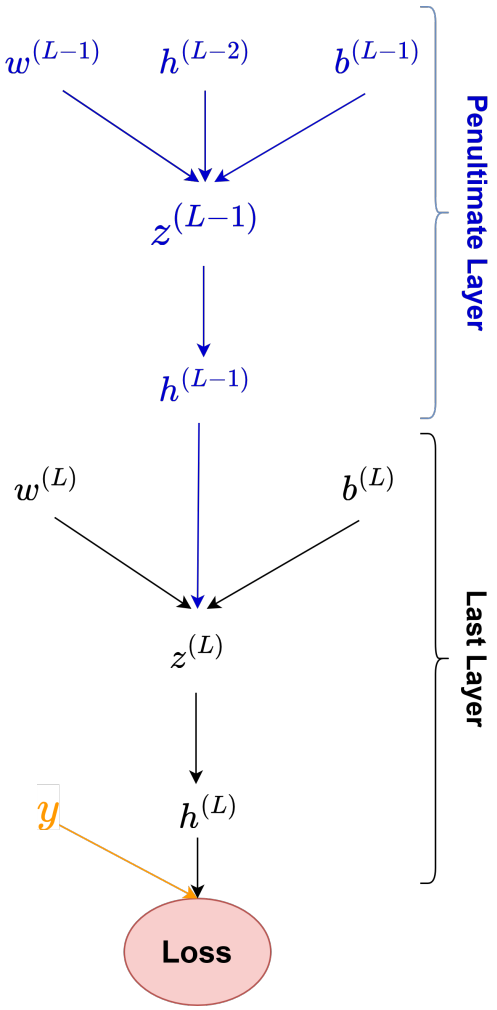


FIGURE 2.9: Computational graph of the last two layers of a neural network (operations were removed to allow an easier read).

$$\frac{\delta \mathscr{L}}{\delta w^{(L-1)}} = \frac{\delta \mathscr{L}}{\delta h^{(L)}} \frac{\delta h^{(L)}}{\delta z^{(L)}} \frac{\delta z^{(L)}}{\delta h^{(L-1)}} \frac{h^{(L-1)}}{z^{(L-1)}} \frac{z^{(L-1)}}{w^{(L-1)}} \tag{2.18}$$

And again we are able to compute the derivatives of the loss with respect to the weights, this time the weights of the penultimate layer. Is this idea of starting from the loss and using the chain rule backwards until we reach the desired weights/biases that gives the name to backpropagation because we are indeed back-propagating the gradients until the beginning of our neural network in order to have all the derivatives of the loss in respect to the parameters of the network. In the end, we should be able to have all the terms in 2.11 that are needed to do step t of the gradient descent algorithm. And by successively doing gradient descent steps, we optimise our neural network parameters and get successively better outputs.

Note that so far we have simplified things and worked only with one training point,

a two-layer model, and each layer only had one neuron. If our network has many layers, we just have to backpropagate the gradient further, there is no problem at all, the thinking is exactly the same. If we work with multiple training points, we simply average all the derivatives for each training point, for example:

$$\frac{\delta \mathscr{L}}{\delta w^{(L)}} = \frac{1}{N} \sum_{k=0}^{N-1} \frac{\partial \mathscr{L}_k}{\partial w^{(L)}} \tag{2.19}$$

Where N is the number of training points. If our network has more than one neuron per layer instead of working with scalar variables we work with matrices, the fundamental ideas are exactly the same, we just have to take special care with the dimensions of each matrix. It is out of the scope of this dissertation to go in depth on the formalisation of backpropgation when there are many neurons per layer, but once again the fundamental idea is exactly the same.

2.3.5 Quick Summary of Training a Neural Network

We can summarise the fundamental process of training a Deep Learning model (a neural network) as:

1. Feed one input/output pair $(\vec{x_n}, y_n)$ to the neural network.
2. Apply all the linear and non-linear transformations using eq. 2.7 for all layers until the last layer is reached and an output $\hat{y}$ is computed.
3. Compare the network output $\hat{y}$ with the target value y_n and compute the loss value.
4. Calculate the gradients and update the gradient vector (2.11) by successively propagating the gradients backwards through the neural network until the inputs are reached.
5. Update the network's parameters using eq. 2.10.
6. Repeat the above steps until the loss value is good enough.

These fundamental steps might seem very simple, but there are important details and issues that make the process of training a neural network complicated and nuanced. We're going to go through some nuances of training a deep learning model in the next section.

2.3.6 Important Details when Training a Deep Learning Model

Problems with Gradient Based Optimization

As we have seen, Gradient Descent is a way to minimize an objective function $\mathscr{L}(\theta)$ parameterised by a model's parameters θ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta L(\theta)$ w.r.t. to the parameters. The learning rate α determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley. This "surface created by the objective function" is commonly called ***error surface*** and, in the case of a neural network with many weights and layers, it has many dimensions (one for each weight), hills, and valleys. An example of such an error surface can be seen in Fig. 2.10:

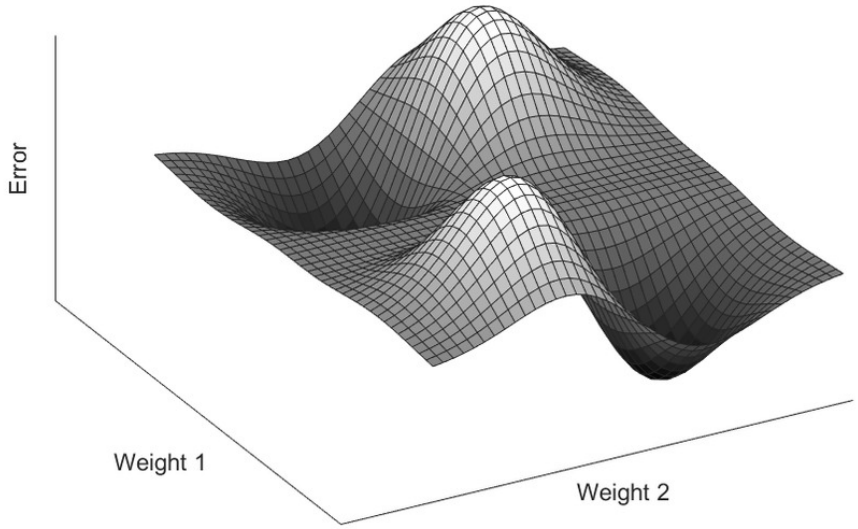


FIGURE 2.10: Error surface as a function of only two weights [6].

.

Problems arise for many reasons, but mostly because the error surface is non-convex, highly dimensional, and has many local minima and flat regions. For example, choosing the learning rate (the size of the steps) is difficult: choosing a small learning rate makes the convergence to a minimum very slow, while a learning rate that is too large could make your loss function "jump" over the minimum or even never converge. One other problem is when we encounter flat regions, where the loss stays constant in the neighbouring regions; here it is ideal to have a large learning rate, otherwise the derivative of

the loss w.r.t. the weights would be zero, and the step of gradient descent would always keep us at the same place.

All these problems can be mitigated using adaptions/improvements of the Gradient Descent algorithm. But note that **there does not exist an algorithm to solve the problem of finding an optimal set of weights and biases for a neural network in polynomial time**. This is equivalent to saying that training a neural network is a ***NP-complete problem***.

Mini-batch Stochastic Gradient Descent (SGD) is one of the most used gradient based optimisation algorithms: it takes n random data points, computes the forward pass for all, average the loss over all data points, and then does one step of gradient descent. The group of n random data points is called a ***Batch*** of your initial dataset. SGD leads to better convergence but is still far from perfect: the learning rate is constant throughout the training, the learning rate is the same for all parameters, and convergence is still very hard. We can mitigate some problems of SGD by adding *momentum*, which is just a way of accelerating convergence by focussing the updates towards directions with greater slope, that is, directions where the derivatives w.r.t. to the weights are higher.

The other optimisation algorithm that is mostly used is ***Adaptive Moment Estimation*** or ***Adam*** which uses something similar to the concept of momentum, but most importantly it uses adaptive learning rate for each parameter. This means that Adam stores the past gradients and uses them to update the learning rate for each parameter at each step of the gradient descent. In this way, the learning rate takes into account how much the gradients have changed previously and makes the avoidance of flat regions and local minima much better.

In general, SGD and Adam are recommended optimisers to use, but there are many more [7], each with its strengths and weaknesses that make them suitable for particular Deep Learning problems.

2.3.6.1 Training, Validation and Test Sets

While we want to model the patterns in the data, we would usually want to use our models in real-world scenarios where the data would not be exactly the same as the data we used to train our network. For this exact reason, normally we separate our training data (or just dataset) in 3 parts: ***training dataset***, ***validation dataset*** and ***test dataset***. With the training set we train our model, the neural network will try to model this data

and will be optimised to minimise the loss on this data only. While training, in parallel, we evaluate our model's performance on the validation set. If the model is focusing too much on the training data patterns, we will see that while the loss on the training data is low, the loss on never seen points while training (the loss on the validation set) is not decreasing. This means our model is ***overfitting*** the training data. There are techniques to minimise overfitting, we will see them further ahead.

Even if our model does not have access to the validation set, we do, and we will try to decrease the loss on the validation set as much as possible, so we steer our model into performing well on the validation set. We *use the test set to make a final unbiased measure of how our model is performing on never seen data*.

Usually, the initial dataset is divided into 80% training set, 10% validation set, 10% test set, but other proportions could be used [3], [8].

2.3.6.2 Overfitting and Underfitting

One of the main issues in machine learning is the tension between optimisation and generalisation. I think François Chollet said it perfectly:

> *Optimisation refers to the process of adjusting a model to get the best performance possible on the training data (the learning in machine learning), whereas generalisation refers to how well the trained model performs on data it has never seen before. The goal of the game is to get good generalisation, of course, but you don't control generalisation; you can only adjust the model based on its training data.*
>
> (François Chollet in [3])

The goal is to make the model generalise to never seen data so that it performs at a good level in real-world applications when new data is used.

When we start training, optimisation and generalisation are both poor, but as training starts, both the loss on the training and validation sets begins to decrease. At this point, the losses are correlated, as the loss on the training set goes down the loss on the validation sets also goes down. In this regime, it is usually said that our model is ***Underfitting***:the network has not learnt all the patterns in the training data. But after a certain number of training iterations (steps), generalisation stops improving, and validation loss stays constant or even begins to worsen. Now our model has started to ***overfit***. The network is learning patterns that are specific to the training data but are irrelevant to

new data. The goal is to be in the sweet spot between underfitting and overfitting, where our model has learnt all the important patterns in the data but is still not focussing on specific and detailed patterns.

Note that this problem of underfitting vs overfitting data is not exclusive to Deep Learning, but has appeared much before. For example, when trying to fit data using polynomial regression, where usually a high-degree polynomial could fit the data almost perfectly, but if we try to apply it to new data, it fails particularly badly. Usually, better performance on never seen data is achieved using a lower polynomial degree than the ones that perfectly fit the training data.

Normally, it is easier to solve underfitting. We can add more complexity to our neural network: add more layers or add more neurons per layer. This makes the network's ability to model data much greater and usually solves underfitting. One important thing to keep in mind when encountering underfitting is the fact that it might be impossible to predict the target with the data we have. This means that the input and output might not be correlated or that the input information we currently have is insufficient. In this case, we might need more input variables (also called features) to predict the output.

Dealing with overfitting is more difficult. There is one obvious solution to overfitting: get more (training) data. A model trained on more data will generalise better. But getting new data is almost always hard or even not possible. In addition, having an efficient deep learning model means that it is able to perform well with the minimum amount of data. When getting new data is not possible, the next-best solution is to apply constraints to the network.This could be reducing the number of parameters (reduce layers and neurons), forcing the weights to have smaller values, and much more. In this way, we are forcing the network to memorise a smaller number of patterns, steering it to only learn the most important patterns and discard the irrelevant ones.

Usually, applying constrains to mitigate overfitting is called ***regularisation***. Let us see some examples of commonly used regularisation techniques.

Weight Regularisation

While training a network, its weights will grow in size in order to handle the specific patterns of the examples in the training data. Large weights make the network unstable. Although the weight will be specialised to the training dataset, minor variation or statistical noise on input data will result in large differences in the output. A simpler

way of regularising a network is to force its weights and biases to take smaller values, thus making them more regular and flexible to different inputs. This is called ***weight regularisation***.

We shrink the values of the weights by adding an extra term to the loss function, which penalises the absolute values of the weights. Therefore, the new loss functions are of the following type:

$$\tilde{L}(\boldsymbol{X}, \boldsymbol{y}, \theta) = L(\boldsymbol{X}, \boldsymbol{y}, \theta) + \lambda\Omega(\theta) \tag{2.20}$$

Here λ is a scalar that measures how much we want to penalise the weights, it is called ***weight decay***. There are two different types of weight regularisation, only differing in which function Ω we use:

1. ***L1*** or ***Lasso*** regularisation - Penalises the *absolute* value of the weights. Defined by $\lambda\Omega(\theta) = \lambda \sum_i |w_i| = \lambda||\boldsymbol{W}||_1$.
2. ***L2*** or ***Ridge*** regularisation - Penalises the *squared* value of the weights. Defined by $\lambda\Omega(\theta) = \frac{\lambda}{2} \sum_i w_i^2 = \frac{\lambda}{2}||\boldsymbol{W}||_2^2$.

The summations serve to show that we sum the values of all weights in the network. Each weight regularisation scheme serves a different purpose, but generally L1 regularisation is used when we know some features are irrelevant, also known as when we have a sparse feature space. L2 regularisation is used more frequently.

Max-Norm Regularisation

Another form of regularising the weights is by scaling them directly in each layer. In ***max-norm regularisation*** we enforce an absolute upper bound on the magnitude of the weight vector of each layer. This means that $||\boldsymbol{W}^{(l)}||_2 < m$ for every layer l. This is an alternative to L2 regularisation, and it is shown to work better most of the time, although it is computationally more expensive.

Dropout

Dropout consists in randomly setting a number of outputs of a layer to zero during training. This could be seen as a random elimination of certain neurons during training. Then during validation/testing, no units are dropped out; instead, the layer's output

values are scaled down by a factor equal to the dropout rate to balance for the fact that more units are active than at training time. The ***dropout rate*** is the fraction of outputs that are zeroed during training. Dropout makes the network not rely too much on certain features and makes it more obvious to the network which features do not correlate with the output.

Early Stopping

One important thing is to always register the value of both training and validation losses during training. This way, we can see if our model is underfitting, overfitting or in the desired sweet spot. Also, when we see that, during training, the validation loss stops improving, we can stop the training process. This way we prevent overfitting by ***early stopping*** training before there is real overfitting.

2.3.6.3 Activation Functions

So far we've only worked with the step function(2.4), but there are others commonly used and present in literature. The choice of activation functions depends on the application. In this dissertation it is important to present 2 activation functions: the hyperbolic tangent and the rectified linear unit (ReLU).

Hyperbolic Tangent

The hyperbolic tangent is defined as:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.21}$$

Which has the following plot:

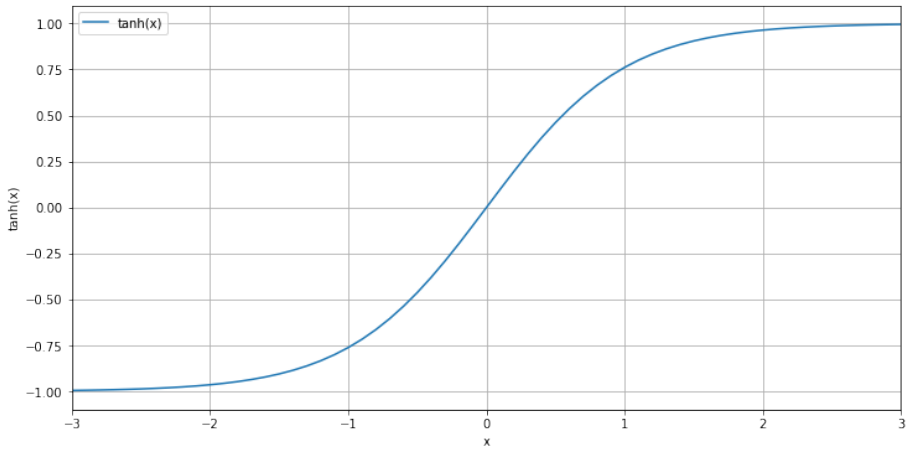


FIGURE 2.11: Plot of tanh(x).

The $tanh(x)$ is used manly because it is differentiable in all its domain, and it is infinitely differentiable. This will be important later. Note that it is not a linear function, as intended to model non-linearities.

The ReLU function

The ReLU function is defined as:

$$f(x) = max(0, x) \tag{2.22}$$

And its plot is the following:

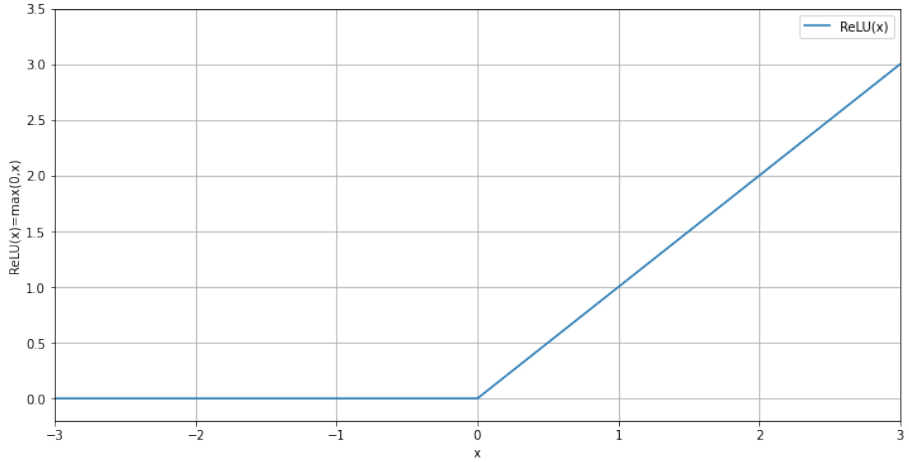


FIGURE 2.12: Plot of $ReLU(x)$.

As you can see, ReLU is almost a linear function, with the exception being the non-linearity in the (0,0) point. This makes ReLU good as an activation function because

they preserve many of the properties that make linear models easy to optimise with gradient-based methods. But ReLUs have the limitation of not being differentiable, not being bounded (so outputs can explode), and if the inputs are negative, the output is always zero. Recently, adaptions of the ReLU have been used, like the Leaky ReLU which is a ReLU with a small slope for negatives values instead of zero slope. Many more adaptations to the ReLU exist, each combats certain drawbacks of the ReLU.

2.3.7 Weights and Biases Initialisation

Another important detail is how we initialise the values of the weights and biases of the network. We can see in Fig. 2.10 that the point where we start the descent is important, but how should we choose it? The short answer is: *it depends on the activation function used*.

For example, if we are using the $\tanh(x)$ function(2.21):

- If we initialise the weights with high values, the term $W^T X + b$ also becomes high, and the step function will only output values close to 1. In this region the gradient of the tanh function is close to 0, so when backpropagating the gradients we get a zero term. The rest of the backward pass will come out all zero due to multiplication in the chain rule. This is something called ***vanishing gradients***.
- If we initialise the weights with low values, the step function will only output 0. And since the derivative at this region is also 0, we encounter the same problem as before.

The goal is to make the gradients flow correctly; for that, you need to correctly initialise the weights of each layer, taking into account the activation function used. But there are two rules of thumb:

1. The mean of the outputs of a layer should be zero.
2. The variance of the outputs of a layer should stay the same across every layer.

Under these two rules, the backpropagated gradient should not be multiplied by values too small or too large in any layer. It should travel to the input layer without exploding or vanishing.[3]

[3] I recommend visiting [8], which gives a perfect explanation (with visuals) of why weight initialisation is so important.

Since we introduced the ReLU and tanh functions, we are going to describe their recommended initialisation schemes. For $\tanh(x)$ we use what is called ***Xavier Initialisation*** [9]. Using this scheme, the weights are initialised with the following Gaussian distribution: $\boldsymbol{W}^{(l)} \sim \mathcal{N}(0, \frac{1}{n^{(l)}})$. Where $n^{(l)}$ is the number of neurons in layer (l). In this way, we keep the variance of each layer always equal to each other. [9]

For the ReLU function, we use what is called ***He*** or ***Kaiming*** [10] initialisation. He initialisation says you should initialise your weights with the following Gaussian distribution: $\boldsymbol{W}^{(l)} \sim \mathcal{N}(0, \frac{2}{n^{(l)}})$.

2.3.8 Different Neural Networks Architectures

So far, we have only talked about feed forward networks, composed by LTUs stacked in multiple layers. And these networks suffice to solve many problems: by changing the number of layers, the number of neurons, the kind of activation functions used, and even the loss function (tanh, step and sigmoid function for classification and MSE for regression) we can tackle vastly different problems. But there are specific situations where perhaps a different type of network works much better.

That is, for example, the case when dealing with image data. Convolutional Neuron Networks (CNNs) were designed to perform much better than feed forward nets using much less resources. It is beyond the scope of this thesis to explain how CNNs work in detail, but the main idea is that by passing a ***filter*** or ***kernel*** through an image, we can extract patterns present in it. Then we can optimise the parameters of a neural network to learn which patterns are important and which are not. All modern image recognition/classification systems use CNNs.

More examples of different neural networks architectures are: Recurrent Neural Networks, Trasnformers, GANs, and Autoencoders. Although they are all useful and interesting to understand, they were not used in this internship, so we will not describe them.

2.3.9 Recap

We now have all the knowledge needed to train Deep Learning Models: we understand what a neuron (LTU) is, what a feed forward neural network does, and what is behind the *learning* part of Deep Learning, namely the Gradient Descent and Backpropagation

algorithms. We have seen the general We also saw some important details about training a neural network effectively: underfitting/overfitting, weight initialisation, and how should we separate the dataset. We can update the workflow of training a neural network given in Section 2.3.5 with information gathered in Section 2.3.6:

1. Choose the appropriate neural network architecture for your problem.
   (a) Choose the number of layers, and number of neurons per layer.
   (b) Choose the activation function of the hidden and output layers.
2. Correctly initialise your network's parameters given the activation functions that the network has.
3. Choose the correct loss function for your problem.
4. Choose one gradient-descent based optimiser.
5. Separate your dataset in: training set, validation set, and test set.
6. Train your model by feeding it a batch of your training set:
   (a) *For each input/output pair present in the batch*: Apply all the linear and non-linear transformations using eq. 2.7 for all layers until the last layer is reached, and an output $\hat{y}$ is computed.
   (b) Compare the network's output $\hat{y}$ with the target value y_n and compute the loss value. Store that loss value.
   (c) Repeat for *all input/output pairs present in that batch*.
   (d) Average the losses of all the pairs for that batch
   (e) Calculate the gradients and update the gradient vector (2.11 by successively propagating the gradients backwards through the neural network until the inputs are reached.
   (f) Update the network's parameters using eq. 2.10.
7. Repeat the steps in (5) for *all the batches*.
8. Repeat (6) until you have a good enough loss value or early stop when overfitting starts happening.

After all this, you can still do ***hyperparameter tuning*** by performing multiple experiments of different parameters of your model; for example, change the number of layers, the number of neurons per layer, the activation functions used, or the learning rate and see how your models perform. Note, in order to not confuse the real parameters of a

model, the weights and biases, with these new parameters (size, depth, learning rate, etc.), we call these new parameters ***hyperparameters***. When training a complex model, the number of hyperparameters can easily become large, say 10. If each has just 2 possible values (which does not happen at all, normally we have an interval of real values that each hyperparameter can take), we have $2^10 = 1024$ possible training configurations! Finding the right configuration is hard and done only by experimenting, using a randomised search algorithm or even a Bayesian search algorithm.

This is why Deep Learning gets its fame of being too empirical and random: small changes of parameters can sometimes lead to very different results, and most of the time it is impossible to understand what led to it. Understanding what is happening with weights and biases distributions or even optimisation convergence is very hard, so understanding what happens in each hyperparameter configuration becomes hugely difficult. For that reason, it is impossible to explain why a certain configuration fails and another gives the best results.

This was just a small introduction to Machine Learning and Deep Learning, there is so much more than what I have said here. But in order to understand what Physics Informed Machine Learning is, and what was done throughout this internship, this introduction will be enough.

Chapter 3

Physics Informed Machine Learning

Recently, with the boom of Deep Learning, people from other areas of research started taking an interest in it. For example: tumour detection systems, drug discovery systems or even the advanced AlphaFold [11] that "can accurately predict 3D models of protein structures".

These breakthroughs are not the product of blindly using Deep Learning on a problem from a different subject. They are, rather, the improvement of Deep Learning systems using domain-specific knowledge from different subjects. In this vein, Deep Learning has much to learn from physics. Centuries and centuries of research in physics has been done, and many ideas, theorems, formulas are transversal to many other areas. Research in probability, statistical physics, Partial Derivative Equations(PDEs) and much more is applicable to Deep Learning. Only recently has there been communication between the two areas, and ***Physics Informed Machine Learning*** is the result of such communication. Other examples are: ***Energy Based Models*** [12] or ***Geometric Deep Learning*** [13]. In this work, we will focus only on Physics Informed Machine Learning.

Today, data collection is very easy in many areas. The problem is: how can we understand the data, and what can we extract from it? Imagine the following: an experiment when you have data of an object's position inside a turbulent flow. Could you predict the object's future position only from this experimental data? In theory, if you have an infinite amount of data detailing every possible trajectory, yes. But this is not possible in real-world experiments. Instead, you end up with a moderate amount of data. Now, even if you use a Deep Learning model, your model will not generalise properly. Here enters your physics knowledge: you know the Navier-Stokes and the continuity equations behind turbulent flow, you might also know the density of the fluid, the mass, and

density of the object in case, why not use all this knowledge? ***This is the basic idea behind Physics Informed Machine Learning, use physics to guide/steer the learning process of existing machine learning models***. This way we can have solutions that adhere to the physics of the system; at the same time, we can accelerate the training process; our model will not blindly try to find patterns in the data, now it has extra information on how the solutions should look like, making generalisation and convergence much better.

The formal definition of physics informed machine learning is:

> *Making a learning algorithm physics-informed amounts to introducing appropriate (. . .) biases that can steer the learning process towards identifying physically consistent solutions.*
>
> (Karniadakis et al. in [14],2019)

There are three types of physics bias that can be applied to our model.

1. ***Observational biases*** - By using data that embodies the underlying physics of the system, we allow our model to learn patterns that reflect the physical structure of the data. This is the most obvious type of bias and, if the data is correctly gathered, is almost always used correctly. It is also the weakest type of bias.

2. ***Inductive biases*** - Our prior physics knowledge can be directly applied to modify our model's architecture, so that predictions are ***guaranteed*** to implicitly satisfy a set of physical laws. For this exact reason, inductive biases are called ***hard constraints***. A good example are CNNs, as they are invariant to translations (meaning that they can detect patterns on an image independently of its location) and symmetries. We apply these invariances directly to the architecture so that they cannot be violated.

3. ***Learning biases*** - Learning biases are introduced by choosing loss functions that take into account the underlying physics of the system. This way, they make the model favour solutions that adhere to the underlying physics of the system. These are called ***Soft constraints***, they only steer the process of learning and can be violated, unlike the previous hard constraints.

Inductive biases would be the ones to use if one could easily build architectures that suited each physical law (conservation law, or even symmetries). Moreover, their extension to more complex tasks is challenging, as the underlying invariances or conservation

laws that characterise many physical systems are hard to implicitly encode in a neural architecture.

Because of this, learning biases are more commonly used; they allow for greater flexibility and can be used in many problems. Using simple feed forward networks and add a new physical loss term, we straight away steer the network to output physical adhering solutions. There are examples of models having inductive biases, but the two main ones are the ***Deep Garleking method*** [15] and ***Physics Informed Neural Networks***(PINNs) [16]. In this work we focused only on PINNs, so we are going to be detailing them.

3.0.1 Physics Informed Neural Networks

PINNs were the object of study in this internship; we wanted to understand how they perform, where they perform better and worse, what their strengths and weaknesses are, and more. So it is important to get a good idea of what they are and how they work.

PINNs are an adaptation of feed forward neural networks: they are feed forward networks with added physical learning biases. They originated in this 2018 paper [16] by Raissi et al. titled *"Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations"*. They described PINNs as:

> *PINNs are neural networks that are trained to solve supervised learning tasks while respecting any given laws of physics described by general nonlinear partial differential equations.*
>
> (Raissi et al. in [16], 2018)

PINNs are neural networks (deep learning models) that integrate information from data plus information from partial derivative equations (PDEs) of the system by adding the PDEs into the loss function. The underlying physics of a system are enforced by adding extra terms in the loss function that pertain to the PDEs of the system. The first PINNs ([16]) were built to solve complex non-linear PDE equations of the form $\frac{\partial u}{\partial t} + u\mathcal{F}(\frac{\partial u}{\partial x})$, where $\boldsymbol{x}$ are space coordinates, $\boldsymbol{t}$ is a time coordinate, and $\mathcal{F}$ is a non-linear function of the time derivative of u. We will use the example given in [14] to explain how a PINN works. Consider the viscous Burger's PDE, with ν as the diffusion coefficient:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{3.1}$$

The goal of our model is to learn the appropriate mapping $(x,t) \mapsto u$ given a number of samples. A normal supervised learning task. But as we have seen before, most of the time we do not dispose of many data points and need extra information in order for the model to properly learn this function and then generalise. Here, the PDE comes to the rescue: we force the model's predictions to adhere to the PDE at the same time as they adhere to the data points. Note, adding this extra PDE Loss term could be seen as Data Augmentation, we are in fact adding more "points" where the loss function can be computed. The PINNs framework can be seen in Fig. 3.1 and is the following:

1. Feed the inputs (x,t) to the neural network.
2. Get a prediction $\hat{u}$.
3. Compute all derivative terms in the PDE:
   (a) To compute derivatives, we use backpropagation, just as when optimising weight values.
   (b) We will assume that ν is known.
4. Compute both sub-loss terms: $\mathscr{L}_{PDE}$ and $\mathscr{L}_{Data} = \mathscr{L}_{MSE}$ using:

$$\mathscr{L}_{PDE} = \frac{1}{N_{PDE}} \sum_{i=1}^{N_{PDE}} \left(\frac{\partial \hat{u}}{\partial t} + \hat{u} \frac{\partial \hat{u}}{\partial x} - \nu \frac{\partial^2 \hat{u}}{\partial x^2} \right)^2 |_{(x_i,t_i)} \tag{3.2}$$

$$\mathscr{L}_{MSE} = \frac{1}{N_{Data}} \sum_{j=1}^{N_{Data}} \left(u_j - \hat{u_j} \right)^2 \tag{3.3}$$

5. Weight each loss and add them together:

$$\mathscr{L}_{Total} = w_{MSE}\mathscr{L}_{MSE} + w_{PDE}\mathscr{L}_{PDE} \tag{3.4}$$

6. Iterate until the loss on the validation set is good enough.

There are some important things to highlight/discuss. Firstly, it is important to stress that the initial part of the model (the blue block in Fig. 3.1) works exactly as a feed forward network. Even the sub-loss term $\mathscr{L}_{MSE}$ (3.3) is the same we've been using. We then add the PDE block (the yellow block), with the goal of computing the new sub-loss $\mathscr{L}_{MSE}$ component. How is the loss on the PDE built? The system's PDE (3.1) is the following:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \nu \frac{\partial^2 u}{\partial x^2} = 0 \tag{3.5}$$

Recalling that the goal is to minimise the loss, we can now try to minimise the left-hand side of this equation (because we know that the solution should be 0). The left-hand side of 3.5 is called the ***residual*** of the original PDE. To overcome problems with negative values cancelling positive ones, we square this residual (just like in the MSE) and obtain eq. 3.2. The next problem is how to compute the derivative terms in the residual? Here enters backpropagation, if *the inputs to the network are the variables that we want to derive in respect to* by backpropagating the gradients from the output of the network $\hat{u}$ until the inputs x and t we can compute $\frac{\partial \hat{u}}{\partial x}$, $\frac{\partial \hat{u}}{\partial t}$, and $\frac{\partial^2 \hat{u}}{\partial x^2}$. [4]

Now we are in position to compute $\mathscr{L}_{PDE}$ using eq. 3.2.

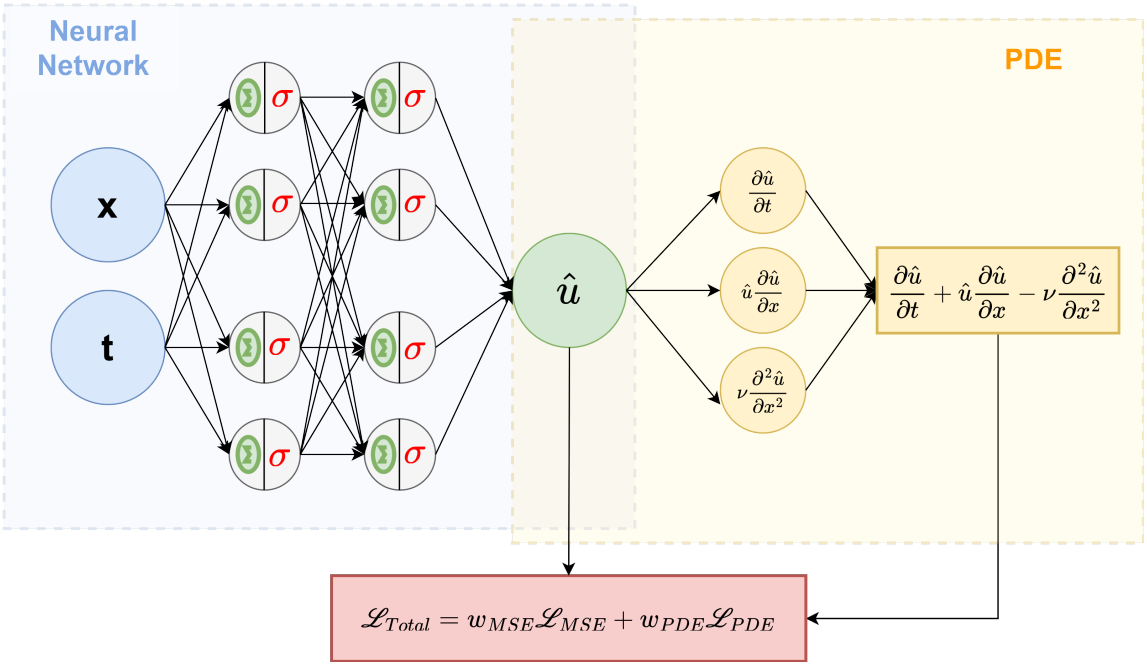


Figure 3.1: PINNs framework diagram.

Like we have seen before, the total loss (eq. 3.4) is a weighted sum of the two sub-losses. But why do we weight the sub-losses? We are going to see clearly in the results section that setting different weights to each sub-loss gives greatly different results. This is because of many reasons, but mostly because the sub-loss can differ in magnitude, and by weighting them we correct that difference so that the optimisation does not favour one or another.

[4] In order to compute high-order derivatives, what is usually done is: a backward pass until the inputs, save that computational graph, and then make another backward pass on top of it. This way we can get second and higher order derivatives.

After computing the total loss, the gradient descent step is just like the one in regular neural networks. The only difference is that the total loss also penalises physical inconsistency, and so the parameters of the neural network will also be optimised to adhere to the underlying PDE. After a few training steps, we should see that the network is outputting both data and physical consistent solutions. When we get to a low enough loss, our model is now capable to output predictions that adhere to the PDE – this means that our model in fact has solved the PDE (up to an error).

3.0.2 Important details

3.0.2.1 Forward and Inverse Problems

So far, we have seen how PINNs can be used to tackle problems where the PDE of the system is known, and we want to use it to improve the model's performance and at the same time solve the PDE. These are commonly called ***forward*** problems, where all the physics of the system are known, and we want to solve the equations for certain input (x, t) coordinates. It is important to note that, in inverse problems, although we say we use Physics Informed Neural Networks, when we start training our neural network is physically agnostic, meaning it has no information of the physics of the system. Then, as training progresses, we gradually steer it towards learning the underlying physics of the system. We still call it a PINN, but it starts out as a regular Neural Network.

However, there are times when we do not have all the physical knowledge. Imagine, for example, that in eq. 3.1 ν is not known. We are unable to proceed as we did before: we cannot compute the residual of the PDE and thus the loss on the PDE. These class of problems, when the PDE is not fully determined, are called ***inverse*** problems, and PINNs offer a solution to them as well.

The goal in inverse problems is to obtain the unknown physical properties from the data, and hence inverse problems are often called ***data-driven model discovery***. The most common approach is to use sparse regression model selection scheme such as PDE-FIND, proposed in [17]. In these approaches, the PDE is discovered by posing the model discovery problem as:

$$u_t = \Theta \xi \tag{3.6}$$

Where Θ is a matrix called ***library of candidate terms (or functions)*** or simply ***library of functions***, that contains all possible derivatives and polynomial terms (e.g. $u, u_x, u_{xx}, uu_x, u^2$) present in the PDE (with the exception of u_t). The candidate terms present in Θ depend on each problem and should reflect some of the physical knowledge we have about each problem. This way, model discovery turns into find the appropriate ξ sparse vector, which will have non-zero entries whenever a corresponding term in Θ is active in the PDE.

Note: The subscript t in u_t represent the derivative of u with respect to t. This notation is introduced to ease representation and make future equations easier to read. For example, the second derivative of u with respect to x becomes u_{xx}, or even the mixed derivative of u is u_{xt}. More formally, u_t is a column vector of size N containing the time derivative of each data sample, $\Theta \in R^{N \times M}$, with M as the number of candidate terms, is a matrix that has all candidate terms evaluated at all sample points, that is:

$$\Theta = \begin{bmatrix} u(\{x,t\})_0 & u^2(\{x,t\})_0 & u_x(\{x,t\})_0 & \dots & u^3 u_{xx}(\{x,t\})_0 \\ u(\{x,t\})_1 & u^2(\{x,t\})_1 & u_x(\{x,t\})_1 & \dots & u^3 u_{xx}(\{x,t\})_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u(\{x,t\})_N & u^2(\{x,t\})_N & u_x(\{x,t\})_N & \dots & u^3 u_{xx}(\{x,t\})_N \end{bmatrix} \tag{3.7}$$

And $\xi \in \mathbb{R}^M$ is:

$$\xi = \begin{bmatrix} \xi_0 \\ \xi_1 \\ \dots \\ \xi_M \end{bmatrix} \tag{3.8}$$

Since Θ will have more terms than the ones in the PDE, ξ will have zero in most entries, because each term ξ_i corresponds to the scalar coefficient of each candidate term in Θ. If once again we were using Burger's equation (3.1) the only activate candidate terms would be uu_x and u_{xx} and in the end we would obtain 1 and ν coefficients for the corresponding terms of ξ and all the other terms would be zero.

The problem of finding ξ can be written as:

$$\arg\min_{\xi} |u_t(\theta^*) - \Theta(\theta^*)\xi|^2 \tag{3.9}$$

Where θ^* are the parameters of the neural network used to compute Θ.

That is to say, the solution to eq. 3.6 is the ξ that minimises the above subtraction. Eq. 3.9 can be solved analytically using the least squares method. We will see the formal definition of the least squares method in the next section.

Note that we determine ξ for each training step, which is why we represent the model's parameters at each step as θ^*. After having ξ we train a PINN just like before with a slightly different PDE loss term, give by:

$$\mathscr{L}_{PDE} = \frac{1}{N_{PDE}} \sum_{i=1}^{N_{PDE}} \left(\Theta_i \xi - \frac{\hat{u}_i}{\partial t} \right)^2 \tag{3.10}$$

Which, if we remember that $\Theta\xi = u_t$, is the regular PINN loss written differently.

The rest of the training is exactly the same as for a normal PINN. Note that the coefficient vector ξ is updated each training step just like the weights, so it will gradually converge to the optimal solution. In the end, if the candidate library of terms has the correct terms, we should have an accurate PDE of the system.

3.0.2.2 Least Squares and QR Decomposition

Suppose we have a system of equations $A\boldsymbol{x} = \boldsymbol{b}$, where $A \in \mathbb{R}^{m \times n}$, and $m \geqslant n$, meaning A is a long and thin matrix and $\boldsymbol{b} \in \mathbb{R}^{m1}$. We wish to find $\boldsymbol{x}$ such that $A\boldsymbol{x} = \boldsymbol{b}$. In general, we can never expect such equality to hold if $m < n$. We can only expect to find a solution x such that $A\boldsymbol{x} \approx \boldsymbol{b}$. Formally, the Least Squares problem can be defined as:

$$\arg\min_x ||Ax - b|| \tag{3.11}$$

This is because we compute the error squared of the solution x. This expression can be simplified by introducing the concept of ***QR decomposition*** [18]. QR decomposition (or factorisation) is the decomposition of matrix A into the product of one ***orthogonal matrix Q*** and one ***upper-triangular matrix R***, giving $A = QR$. There are two important properties of orthogonal matrices: $Q^TQ = I$, and when multiplying a vector by an orthogonal matrix its norm is unchanged:

$$||Qy||_2^2 = (Qy)^T(Qy) = y^TQ^TQy = y^Ty = ||y||_2^2 \tag{3.12}$$

Using these two properties, we can write eq. 3.11 as:

$$\begin{aligned}
&\min_x ||Ax - b||_2 = \\
&= \min_x ||Q^T(Ax - b)||_2 = \\
&= \min_x ||Q^T(QRx - b)||_2 = \\
&= \min_x ||Rx - Q^Tb||_2
\end{aligned} \tag{3.13}$$

And solving $Rx = Q^Tb$ is straightforward: R is an upper-triangular matrix, which makes solving for each x_i entry of x easy as shown in [19]. In the end, we have:

$$\begin{aligned}
&Rx = Q^Tb \\
&\Leftrightarrow x = R^{-1}Q^Tb
\end{aligned} \tag{3.14}$$

The only question is *how do we decompose A into QR?* The most used QR decomposition algorithms are the ***Householder*** and ***Givens*** methods [18]. We are not going to detail them here, but they not hard to understand and implement.

Using the correspondence between $A\boldsymbol{x} = \boldsymbol{b}$ and, in our case, $u_{tt} = \Theta\xi$,

We can modify eq. 3.14 to give:

$$\begin{aligned}
&R\xi = Q^Tu_{tt} \\
&\Leftrightarrow \xi = R^{-1}Q^Tu_{tt}
\end{aligned} \tag{3.15}$$

We can now easily solve for ξ and get a vector of coefficients of the PDE.

3.0.2.3 Activation Functions

One key detail when using PINNs are the activation functions used. Because we use Backpropagation to compute gradients, the activation functions multiple time differentiable if we want to compute high-order derivatives. The hyperbolic tangent (2.21) is predominant in PINNs because it is infinitely differentiable, which means that we can compute as high-order derivatives as we wish.

3.1 PINNs as PDE Solvers vs Numerical Methods

PINNs started as a different alternative to solve PDEs, but why use them instead of long-established numerical methods such as finite difference or finite element methods? The main advantages of using PINNs compared to classical numerical methods are the following:

1. **Lower inference time** - After finishing training getting predictions is straightforward and fast, it only takes the time needed to compute the forward pass. Numerical methods, on the other hand, are very slow when inferring; for example, mesh methods and finite element methods are very computationally intensive and slow.

2. **Greater flexibility** - PINNs allow for more use cases. For example, when boundary conditions are more complex or even for very complex PDEs. Also, most numerical methods approaches are designed to tackle a group of PDEs or even a single PDE, while PINNs are a general approach that could work for most of the PDEs at the same time.

3. **Capability of learning non-linear behaviour** - Numerical methods do not deal well with non-linear PDEs (most linearly approximate a derivative). PINNs, on the other hand, due to the use of non-linear activation functions, can easily model non-linear patterns.

4. **Solving Inverse problems** - Numerical methods are not capable of solving inverse problems, unlike PINNs as we have seen.

In contrast, PINNs still have some disadvantages compared to numerical methods:

1. **Stability/Convergence** - While stability in numerical methods is well-defined and studied, in PINNs there is still a lack of understanding of why and how PINNs fail to converge. Optimisation landscapes in PINNs can become highly complex and impossible to reach a local minimum.

2. **Training Time** - While getting predictions after training is done is fast, training the model up to the point takes much longer and, most of the time, longer than what numerical methods do.

3. **Hyperparameter tuning** - Training is most of the time slow, but adding to that, knowing which hyperparameters work best is empirical and takes too much time. Training time plus hyperparameter tuning together make PINNs slower than numerical methods.

Currently, PINNs seem like a good alternative to numerical methods when speed in real-time predictions is needed: for example, monitoring and sensing applications. Another good use case for PINNs is solving high-dimensional and non-linear PDEs,

here they also outperform numerical methods. On the other hand, PINNs still lack understanding in theoretical issues like convergence, stability, architecture design and optimisation aspects.

3.2 Current Research on PINNs

PINNs originated in 2017/2018 in a paper by M.Raissi et al. [16] and were designed to solve PDEs and inverse problems. Since then, PINNs have gained traction in the scientific Machine Learning community (SciML) and have been employed in many different problems and areas such as: fluid dynamics, geoscience, weather forecasting, and optics([20],[21]). With this increased interest in PINNs, many adaptations and improvements have been made. In this section, we will go through some examples of research on PINNs and their adaptations.

On a more theoretical side, there have been improvements in how PINNs should be trained. In [22], the authors propose an adaptive and relative loss weighting scheme. In this new scheme, at each training step firstly the two sub-losses are compared to each other and weighted accordingly to prevent huge mismatches in values, then each loss is compared to the previous step loss and scaled accordingly to prevent huge steps in loss values. They showed improvements over conventional weighting schemes. Another example of improving the training of PINNs is [23], in which it is shown that when training PINNs to solve the wave equation, it is recommended to start training with only $\mathscr{L}_{Data}$ (making it a regular feed forward neural network) and only add the extra $\mathscr{L}_{PDE}$ term when training as stabilised on a low loss value. This enables the training process to be smoother and to converge faster. We will see why this works better later on.

In [24] the authors propose a novel PINN framework, extended-PINNs (xPINNs), which basically divides a complex and big space/time domain in multiple ones. This way, training each sub-domain becomes a simpler optimisation task and at the same time training each sub-domain can be done in parallel, speeding the training process hugely.

In addition to this focus on improving the convergence and stability of PINNs, many authors focused on applying PINNs to many different types of PDEs. For example, very specific advections problems like the advection-dispersion and Darcy flow equations with space-dependent hydraulic conductivity solved in [25], or for specific cases of the

Navier-Stokes equations like in [26]. Other examples are using PINNs to solve integro-differential equations [26] or stochastic differential equations [27] (which use PINNs combined with generative adversarial networks).

On research focused on real-world and industrial scenarios, there are also interesting PINNs applications. For example, this paper by Mathews et al. [28] used PINNs to infer 3D turbulent plasma fields from 2D data. They showed that their PINN approach could predict the behaviour of plasma near the boundaries of the confinement vessel (e.g. a tokamak), whereas conventional numerical approaches struggle.

Another good example of the use of PINNs in industrial problems is [29]. In this paper the authors, use PINNs to determine rotor angles and frequencies, damping coefficients and more variables of a power system.

All in all, PINNs show great promise on many applications, and it is understandable why they are gathering so much interest in both academia and industry worlds. Nevertheless, most results and applications of PINNs have been on more theoretical cases, with applications to real world data still lacking. The growing interest in PINNs and their use in many diverse areas will only help to increase understanding on theoretical aspects of PINNs (convergence, when to use PINNs and when to use numerical methods, how to better train PINNs, and much more). It is expected that the interest in PINNs will only grow when these theoretical issues have been solved, meaning that PINNs will become mainstream when applying machine learning to scientific problems. Plus, when convergence and stability issues have been understood, PINNs will gain traction in real-world applications where one cannot afford to have such issues.