Machine Learning Techniques For The Estimation of The Operating Parameters of Solar Cells

Papadomichelakis Georgios

A thesis presented for the
Master of Sciences
degree

Committee
Theodoros Katsaounis(Advisor)
Michael Plexousakis
Vangelis Harmandaris

Department of Mathematics and Applied Mathematics
University Of Crete
Greece

Abstract

We consider the problem of predicting the internal temperature in photovoltaic cells depending on ambient and/or internal factors. In this thesis, we use machine learning techniques, specifically deep learning and neural networks to accurately forecast the temperature, using methods we developed. We present an introduction to the mathematical background of neural networks and build some using the Python3 programming language and TensorFlow. Lastly, we present the numerical results comparing what our neural networks managed to predict to the actual temperatures measured.

Acknowledgements

The completion of this project could not have been possible without the support and encouragement of my professor and mentor Theodoros D. Katsaounis. I deeply feel the need to express my sincere appreciation for guiding me in my knowledge journey and opening up a new path to a very interesting aspect of science.

To my professor, Michael Plexousakis, who was there since my very first day until today in my academic years, offering answers and advice whenever I had questions, I feel sincerely thankful.

I also want to thank Dr. Vangelis Harmandaris for his contribution as part of the committee and for being an excellent professor.

Lastly, I would like to thank also Dr. Konstantinos Kotsovos of Saudi Aramco RD Center - Carbon Management Division at KAUST Saudi Arabia, for providing the solar cell data used in this present thesis. This work was also partially supported by grant KAUST-OSR–2020-4433.02 of KAUST Saudi Arabia.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

During the recent years, there has been an ever-growing research interest in *Artificial Intelligence* (AI). Another term we hear about, more and more often, is *Machine Learning* (ML) which can be seen as a subset to AI. *Neural Networks* (NNs) and *Deep Learning* (DL) are both included in that subset of Machine Learning. This thesis is about using *Artificial Neural Networks* (ANNs) and deep learning methods, in order to deal with a problem coming from the photovoltaic cells' industry. In particular, we are interested in developing supervised machine learning techniques for estimating the operating temperature of photovoltaic cells using various parameters.

In chapter 2 we describe the general problem in detail. The main principals behind how photovoltaic cells work, how ambient or internal factors may interfere with their energy production and how these factors are related with the leading cause of the energy loses we observe in practice -which is the cell's temperature- are discussed. We discuss why cooler cells yield higher energy and vice versa.

In chapter 3, we give a brief introduction to Artificial Neural Networks with their properties and characteristics. We compare them with the biological networks of our brains and seek to understand the motivation behind the creation of the ANNs. We examine the learning process, how someone can represent a neuron or a whole network in a computer and what are the algorithms available we use to "teach" that network.

Chapter 4 is about the three different methods we developed in order to address with our case study, the implementation of the ANNs, the data used in order to train the networks and their representation on a computer. First method is about examining how only external factor related to the climate can affect the temperature of the photovoltaic cells, using different deep neural networks. We hope that they will be able to recognize, to what extent, those ambient features play a role in heating up or cooling down the cells.

Lastly, for this method, we talk about how even if we use only a posteriori data, meaning that we need to gather the data in the first place using sensors, before starting training our network, we can surpass this problem, using typical meteorological years. The next method is about neglecting two ambient factors and replace them with characteristics of the photovoltaic cells themselves. Lastly, for the third method, we use different formulas developed and found in the literature that approximate the operating temperature of the cells. Since those formulas rarely give accurate results, we train different ANNs to find combinations of those formula outputs to better approximate the temperature.

Finally, in chapter 5, we present the numerical results obtained for every different method we developed. We study the performance of the ANNs, and their dependency on some key factors that affect them. We conclude that all methods give excellent results with the right parameters and structure of the DNNs, however, the first method that considers more ambient factors produces the most accurate results out of the three. We discuss how those structures and parameters affect the accuracy of the models.

Chapter 2

Problem Description

Photovoltaic solar cells (PV cells) are semiconductor devices constructed to convert sunlight to electricity. PV cells absorb energy only with wavelengths from a specific window of the solar spectrum, not the sun's heat. Silicon based solar cells absorb light of wavelength $\lambda \in [280, 1100]$ nm. Like all other semiconductors, PV cells are sensitive to temperature. Increased temperature of the solar cells leads to reduction of the energy yield efficiency. It is a misconception that solar cells work better in extremely hot climates and not in the cold ones. Even in cold weather days, solar panels can convert sunlight into electricity as long as solar irradiance hits the panel. In fact, cool climates are optimal for solar panel efficiency. To understand this we need to understand how PV systems work.

The sun emits energy (light) in the form of waves. The active part of a solar cell is a wafer made of a semi-conductive material, typically silicon. A semiconductor is a type of material that normally does not conduct electricity well, but it can be more conductive under certain conditions. When the photons hit the PV cells, electrons in the silicon are put into motion. This creates an electric current, which is sent to the power grid, batteries, etc. Electrons are at rest (low energy) in cooler temperatures. When these electrons are activated by increasing sunlight (high energy), a greater difference in voltage is attained in the solar panel, which creates more energy. That's why solar cells produce electricity more efficiently when it's colder.

PV modules are usually tested at a temperature of 25 degrees Celsius, however, that refers to the temperature of the panel itself, not the ambient temperature. On a sunny day, when the solar panels are the most useful, PV modules are excepting to have a far higher temperature than the ambient one, resulting in a constantly increase in efficiency loss as the temperature rises. Thus, the ideal conditions for PV modules are sunny skies but cool

ambient temperatures, conditions that are not met at most of the earth's locations. However, the modules' temperature depends not only on the ambient temperature or the sun's irradiance but also on other external factors, such as the wind speed. For example, strong winds can help to cool the panels.

Considering this, in some parts of the world, PV cells' energy yield drops during the hottest hours of the day and rises during the cooler ones, while in other areas the exact opposite is observed.

In practice, the importance of approximating the temperature in which solar cells work the best, originate from the photovoltaic industry sector. We can estimate with good enough precision what the energy yield will be given at a specific temperature of the cell, but we do not know, to what extent, external and/or internal factors combined affect their temperature. Hence, we can not predict the temperature a priori and, thus, predict what the energy production will be, given the climate conditions. Since we know that the energy production depends heavily on the cells' temperature, adjusting or interfere, when possible, with the key factors which temperature depends on, can lead to increased performance of the cells. Of course, increased or even peak performance is something which is beneficial and profitable for the owner of the PV systems and for the environment.

In addition, it is extremely important to have a general idea of the approximate energy production a PV system could yield, before installing it on a specific spot. In this way, different and more suitable technologies of PV cells could be used or even use techniques of active cooling. Reducing the cells' exposure to the sunlight during the hottest hours could be another solution. For example, if increased exposure heats the panel so much that the energy output yield highly drops, having a mechanism that could shade part of the system when needed could be more cost effective. Consequently, the practical application of estimating the temperature is an important task.

So the purpose of this work is to examine the correlation between the PV modules' temperature and other variables like solar irradiance, wind speed, air temperature, open circuit voltage and short circuit current, so as to give a fast and reliable way of predicting the expected PV modules' operating temperature.

We will use machine learning techniques to do so, specifically deep neural networks (DNNs). After we give a brief overview of the way the DNNs function, we implement three different methods to approximate the temperature of the cells. The first one, is by using the aforementioned data, the second one, is by using part of those data and two characteristics provided by the module's manufacturer: the module's rated efficiency and Nominal Operating Cell Temperature (NOCT). The third method uses some empirical formulas

that have been developed in the literature to model the cell's operating temperature. However, these models are not very accurate, so the third method is using those models to produce the expected results in temperature and try to find a suitable combination of them in order to obtain an estimation that fits better to the observed data.

Chapter 3

Artificial Neural Networks

Artificial Neural Networks (ANNs) are motivated by the modern concepts of how the human brain functions and learns, where hundreds of billions of interconnected neurons process data. Of course, recreation of human awareness and thinking is still within the realm of science fiction, however, artificial intelligence and as an extent deep learning and ANNs, have been booming over the recent years. In this chapter, we will try to give the reader a brief introduction of what ANNs are and what their correlation to the human brain function is as well as the mathematical background, their representation in a computer and the learning process.

3.1 Biological and Artificial Neural Networks

All the algorithms that are utilized in deep learning are inspired, to a great extent, by the way biological neurons and neural networks function and process data in the brain. Figure 3.1 depicts one of the very first drawings of neurons, drawn by Santiago Ramon y Cajal back in 1899 [7], based on what he saw after examining Purkinje cells (A) (also called Purkinje neurons) and granule cells (B) from pigeon cerebellum with the microscope. Based on Santiago Ramon y Cajal's drawings, who is currently considered the father of modern neuroscience, the neurons, one of them labeled as "A" in the image, have big bodies in the middle and long arms that stretch out and branch off to connect with other neurons. Figure 3.2 [12], depicts a modern model of what brain tissue looks like. This image gives us a sense of how firmly they are packed together and how many of them are in a small piece of brain tissue.

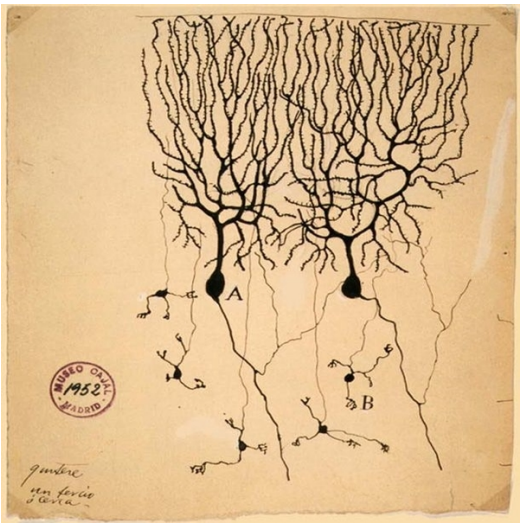


Figure 3.1: Santiago's Drawing

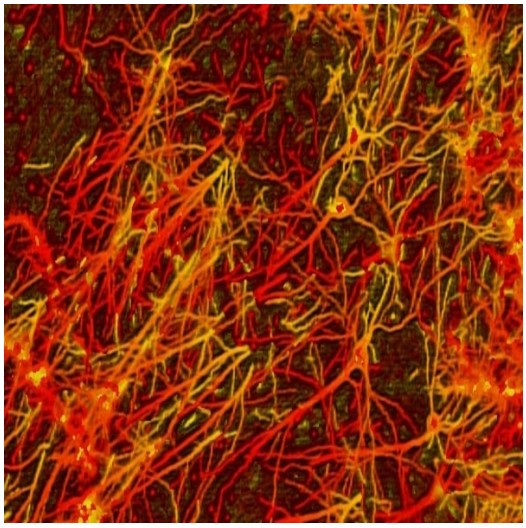
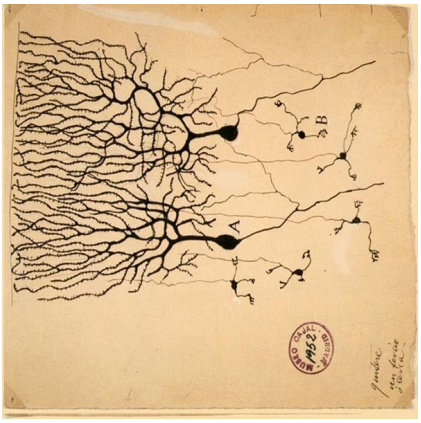
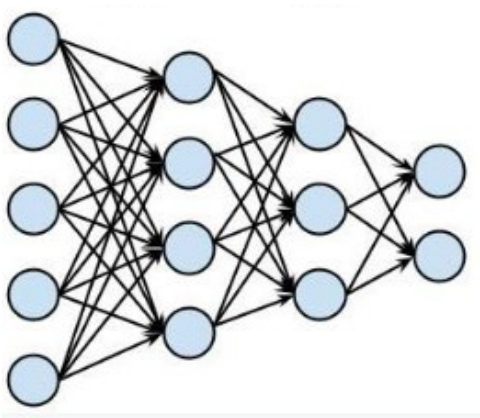


Figure 3.2: A Brain's Network

Rotating Santiago's drawing by 90 degrees counterclockwise (Figure 3.1) we observe that the rotated drawing looks like the ANNs we might have come across (Figure 3.2).



Rotated drawing



Sketched NN

In Figure 3.3 a sketched neuron is shown. The main body of the neuron is called the *soma*, which contains the nucleus of the neuron, while the network of arms sticking out of the body is called the *dendrites*. The arm that sticks out of the soma is called the *axon*. Whiskers at the end of the axon are called the *terminal buttons* or *synapses*. The dendrites receive electrical impulses which carry information or data from sensors or synapses of other adjoining

neurons. The dendrites then carry the impulses or data to the soma. In the nucleus, electrical impulses or data are processed by combining them together, and then they are passed on to the axon. The axon then carries the processed information to the synapse and the output of this neuron becomes the input to thousands of other neurons which are connected to the previous neuron. Learning in the brain, occurs by repeatedly activating certain neural connections over others when sensors are triggered and electrical impulses or data are passed from neuron to neuron. This process reinforces those connections that were activated, making them more likely to produce a desired outcome given a specific input. Once the desired outcome occurs, the neural connections causing that outcome becomes strengthened.

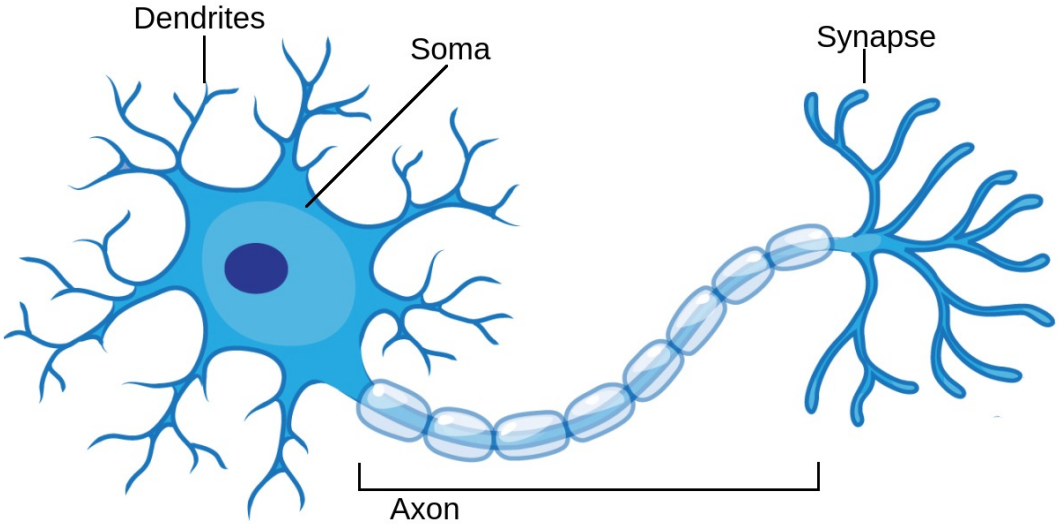


Figure 3.3: A cartoon representation of a human-brain neuron

The key idea we want to keep from this short overview of biological neurons is that after a certain input is passed through the human sensors (for example the eyes) to some neurons, some of them activate and pass the processed data to other neurons, from which some of them in turn will be activated and pass the information to other neurons and so on. For more information we refer the reader to [6].

Learning process in ANNs, very much resembles the way learning occurs in the brain. Our ANN consist of artificial neurons and those artificial neurons behave in the same way as biological ones. They consist of a soma, dendrites and an axon, so they can receive some input data and pass on the output of each specific neuron to other adjoined neurons. Before they pass

the data to adjoined neurons, data are combined and processed inside the soma just like it is done in the biological neurons. The end of the axon can branch off, to connect to many other neurons, but for simplicity we're just showing one branch in Figure 3.4.

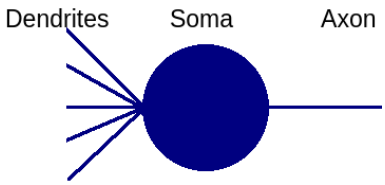


Figure 3.4: An artificial neuron

3.2 Structure of ANNs

3.2.1 Neurons and Layers

A *feedforward* neural network is an ANN which is described by an algorithm working in layers where the connections between the neurons do not form a cycle. The word "feedforward" refers to the idea of passing data from the input neurons to other neurons and so on, as previously described. An ANN consists of an input layer of nodes (or neurons by analogy to the biological brain, or perceptrons), one or more hidden layers of neurons and a final layer of output neurons which can in turn consist also by one or more outputs. Neurons are data-processing units, represented by numbers on a computer, with whom the numeric data from the previous inputs will be processed.

In order to define a neuron we need a set of synapses (or connecting links), each characterized by a weight which is also just a number. Specifically, a piece of data x_j at the input of synapse j, connected to the neuron k, is multiplied by the synaptic weight w_{kj}. The first subscript refers to the neuron currently taking into account while the other refers to the synapse to which the weight refers. Those weights typically lie in a range of numbers, but we will come to this point in the sequel. In the human brain, the neurons either "fire" (activate) or not. In ANNs the neurons are not that limited. Firstly, we need to define what an activation function is, because depending on that activation function, the neurons can be activated or not, or even be activated with different strengths. So an activation function is simply a mathematical function that takes the input data and outputs them after being processed, commonly between a permitted range e.g. $[0, 1]$ or $[-1, 1]$. Typically, there is also another set of numerical parameters, called biases, which are constants

added to the numeric data, before passing them through the activation function. This additional parameter in the Neural Network is used to adjust the output along with the weighted sum of the inputs to the neuron. Bias values allows us to shift the activation function to either right or left as well.

Mathematically speaking, we can describe the neuron k, illustrated in Figure 3.5 as following:

$$z_k = \sum_{j=1}^{m} w_{kj} x_j + b_k, \tag{3.1}$$

where w_{kj} is the corresponding weight for the neuron k and synapse j and b_k is the bias term. The outputs of those neurons, after the data have been processed and the activation functions have fired, are set as

$$y_k = a(z_k), \tag{3.2}$$

where $a(\cdot)$ is the activation function. We will call the terms y_k "predictions". The motivation of this designation, is that the outputs of the activation functions are what we consider as the final prediction of the model at the output layer.

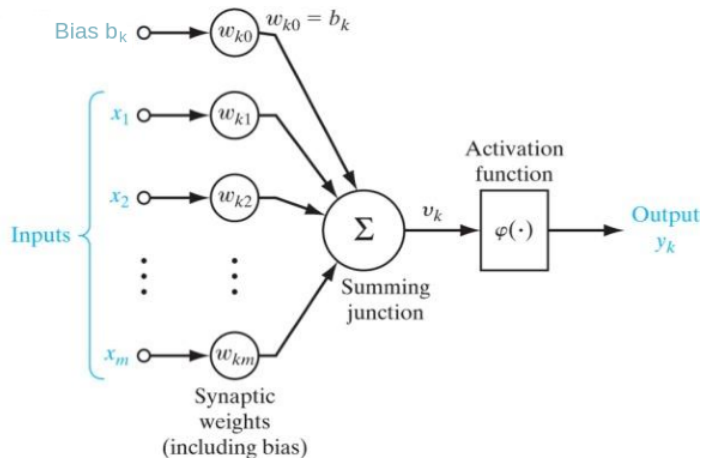


Figure 3.5: one neuron with m inputs

We can imagine layers to be simply columns of neurons stacked together. A fully connected neural network describes a network where every single neuron of a layer L_i is connected to every single neuron in the next layer L_{i+1}. With this notation, we can set up layers of neurons for a multilayer neural network. Conventionally, the input layer is not taken into account

whenever we want to number the layers. So we can think of it as "Layer 0" and the "first" layer to be the layer of neurons that the data are passed right after the input layer, also called the "*first hidden layer*". The term "hidden" has no special meaning. It simply implies that these neurons are performing intermediate calculations. Consequently, the layer after the first one is called the "*second hidden layer*" and so on, until the last one which is labeled as the "*output layer*" in most of the literature.

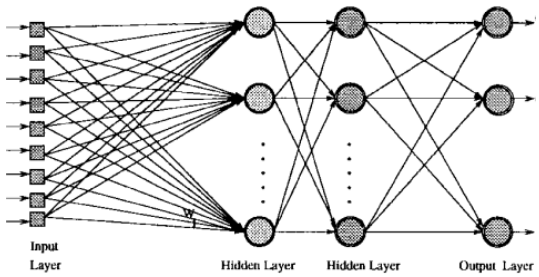


Figure 3.6: An ANN with two hidden layers

The term "depth" is regularly used to describe the number of hidden layers, hence, the designations "*deep neural network*" and "*deep learning*" simply refer to a network with many hidden layers. For example, the ANN in Figure 3.6 has a depth of 2. Following the term "depth", the term "*width*", refers to the number of neurons (or nodes) an ANN has, however, this term is rarely used, due to the fact that this description is not well defined in case the network has not a single fixed number of neurons in every layer, like in Figure 3.7.

3.2.2 Weights, Biases and Activation Functions

Before moving on to the theory of how the ANNs work, a deeper analysis on how the weights and biases are combined and fed into the activation function will give a better intuition on the field of reference. We have already introduced the terms weights, biases and activation functions, but we have not seen how these terms are mathematically represented. As mentioned above, the data from a layer are passed to another one using (3.1) to strengthen the connection between two neurons. This connection strength is actually what we call a weight, practically a number. We can represent all the synaptic weights connecting all neurons of a given layer fully connected to its previous layer, by a matrix of weights. For the nth layer, the element w_{ij}^{n} is the weight that the ith neuron at the nth layer applies to the output from the jth neuron at the $(n-1)^{th}$ layer. This notation is in full agreement with the

notations we have used so far. Since the only layer with no previous layer is the input one, we start counting layers from the first hidden layer up to the output layer, so this representation of the weights is well defined for all layers.

In particular a matrix $W^n = \begin{bmatrix} w^n_{1,1} & \cdots & w^n_{1,k} \\ \vdots & \ddots & \vdots \\ w^n_{m,1} & \cdots & w^n_{m,k} \end{bmatrix} \in \mathbb{R}^{m,k}$ contains all the weights needed to represent the connections between the layers n and $(n-1)$, with m being the number of neurons in the nth layer and k being the number of neurons in the $(n-1)^{th}$ layer.
The notation for the biases is simpler and can be expressed as a vector, $b^n \in R^m$ for the nth layer, with each element b^n_i corresponding to the unique neuron i.

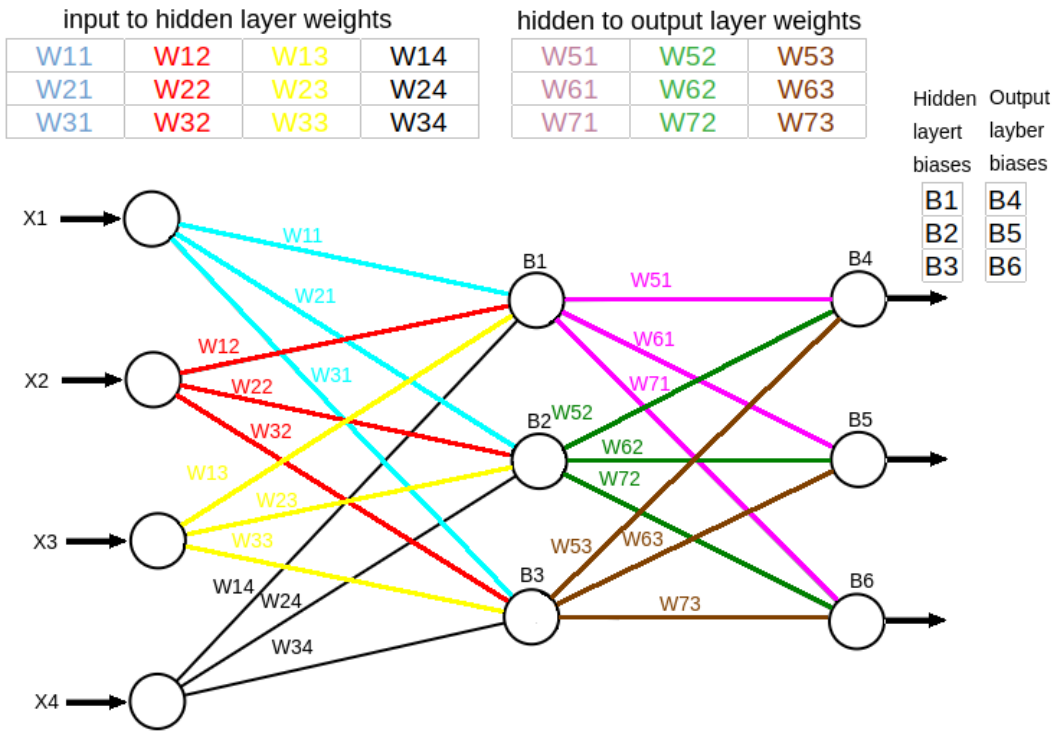


Figure 3.7: Visualization of the relation between matrices and axons

Figure 3.7 is a visualization of the notation we used above. As we can see every connection between two neurons of adjoined layers is just a number and all those connections can be expressed as a single matrix. Every individual neuron, from the first hidden layer to the output layer, has its own bias-number which are represented as one-dimensional arrays in a computer.

At this point, a question arises. How many hidden neurons/layers should one use? Depending on who is asked the answer varies. There is no theory on the number of hidden neurons. Most users rely on experimentation. There are proposed a few rules of thumb, however many of them going again each other. We mention some of them, which we will later follow in the implementation of the ANNs.
• If the data is linearly separable a single layer network (i.e no hidden layers) can do the job.
• If data is less complex and is having fewer dimensions or features then neural networks with 1 or 2 hidden layers would probably work.
• If data has large dimensions or features, then to get a better solution, 3 or even more hidden layers can be utilized. [14]
It is extremely important to note that increasing hidden layers would also increase the complexity of the model and choosing hidden layers such as 8, 9, or more may lead to increased training time and overfitting, thus producing unreliable results for unseen data.

Once the number of hidden layers has been decided the next task is to choose the number of nodes in each hidden layer. For a neural network with N input neurons and M output neurons, T. Masters suggested a number of $\sqrt{MN}$ hidden neurons [8]. The actual optimal number can still vary between one half and two times the geometrical mean value of N and M.
D. Baily and D.M. Thompson (J.O. Katz) suggested the number of hidden neurons be about 75% of the number of input neurons however sometimes with more complex data, as many as 300% of the number of input neurons can be used [8]. A common practice is to decrease the number of hidden neurons in subsequent layers, however this idea can not be always used, depending on the nature of the problem. For example, implementing this on a classification problem would be useful, to get more and more close to pattern and feature extraction and to identify the target class.

Mathematically speaking, an ANN aims to find a mapping $f(x)$ that maps the input(x) to the output(y). The function $f(x)$ can be arbitrarily complex. The Universal Approximation Theorem (UAT) states that no matter what $f(x)$ is, there is a neural network with one single layer that can approximate it, arbitrarily close. This result holds for any number of inputs and outputs and seems to hold for any number of layers as well, however there is no mathematical theory supporting that and it is an open question. Hornik, Stinchombe, and White published a proof of the UAT for a single layer network. The proof we present here, follows [2]. Next we state the

UAT. We begin with the necessary definitions.

Definition 1 Let I_n denote the n-dimensional unit cube $[0,1]^n$ and $M(I_n)$ denote the space of finite, signed regular Borel measures on I_n. We say that σ is *discriminatory* if for a measure $\mu \in M(I_n)$

$$\int_{I_n} \sigma(y^T x + \theta) d\mu(x) = 0, \quad \forall y \in \mathbb{R}^n, \theta \in \mathbb{R},$$

implies that $\mu = 0$.

Definition 2 We say that σ is sigmoidal if

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty \\ 0 & \text{as} \quad t \to -\infty \end{cases}$$

Theorem 1 *Let σ be any continuous discriminatory function. Given any function $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum G of the form*

$$G(x) = \sum_{j=1}^{N} c_j \sigma(w_j^T x + b_j) \tag{3.3}$$

for which

$$|G(x) - f(x)| < \varepsilon \quad \forall x \in I_n$$

In other words, the finite sums of the above form are dense in $C(I_n)$

Proof. Let $S \subset C(I_n)$ be the set of functions of the form $G(x)$ as in (3.3). S is a linear subspace of $C(I_n)$. We claim that the closure of S is all in of $C(I_n)$.

Assume that the closure of S, say R is not all of $C(I_n)$. Then R is a closed proper subspace of $C(I_n)$. Using the Hahn-Banach theorem [13], there is a bounded linear functional on $C(I_n)$, say L, with the property $L \neq 0$, but $L(R) = L(S) = 0$. By the Riesz Representation Theorem [15], L is of the form

$$L(f) = \int_{I_n} f(x) d\mu(x)$$

for some $\mu \in M(I_n)$, $\forall f \in C(I_n)$. Since $\sigma(y^T x + \theta) \in \mathbb{R}$ for all $y \in \mathbb{R}^n, \theta \in \mathbb{R}$ we have that

$$\int_{I_n} \sigma(y^T x + \theta) d\mu(x) = 0$$

However, we assumed that σ was discriminatory so this implies that $\mu = 0$, contradicting our assumption. Hence, the subspace S must be dense in $C(I_n)$.■

Lemma 1 *Any bounded, measurable sigmoidal function, σ, is discriminatory. In particular, any continuous sigmoidal function is discriminatory.*

The proof of lemma 2 can be found in [2]

Theorem 2 *Let σ be any continuous sigmoidal function. Given any function $f \in C(I_n)$ and $\varepsilon > 0$, there is a sum*

$$G(x) = \sum_{j=1}^{N} c_j \sigma(w_j^T x + b_j)$$

for which

$$|G(x) - f(x)| < \varepsilon \quad \forall x \in I_n$$

In other words, the finite sums of the above form are dense in $C(I_n)$

Proof. Combine Theorem 1 and Lemma 2, noting that continuous sigmoidal functions satisfy the conditions of that lemma.

Remark 1 *The activation functions "sigmoid", "ReLU" and "Tanh" which will be discussed later in this paper, all belong in the class of sigmoidal functions.*

Let's consider this simple network shown in Figure 3.8 with one hidden layer.

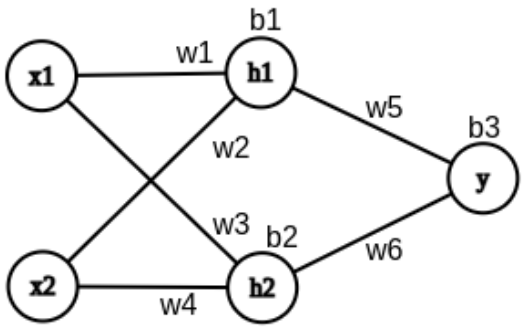


Figure 3.8

Using the previous notation we have

$$h_1 = w_1 x_1 + w_2 x_2 + b_1 \text{ and } h_2 = w_3 x_1 + w_4 x_2 + b_2 \tag{3.4}$$

which are obviously linear functions. In literature, functions like h_1, h_2 are called "hypothesis". Similarly, the prediction

$$y = w_5 h_1 + w_6 h_2 + b_3 \tag{3.5}$$

is also linear in terms of the inputs x_i. Obviously, a linear function cannot approximate any given function which is different to what was stated before by the Universal Approximation Theorem. That is because the UAT, uses a sigmoidal function before propagate the data from a layer to another. So activation functions insert a non-linearity to the system.
As mentioned, activation functions are just mathematical functions, usually restricting the output values in a certain range. Next we present some examples of activation functions commonly used, to get a better understanding of them.

Sigmoid
Sigmoid function is usually the first activation function, someone comes across when studying about deep learning and/or ANNs. Sigmoid outputs values in $(0, 1)$, so it can come pretty handy whenever the model aims to predict a probability. The sigmoid is a differentiable everywhere, monotonically increasing function, with an interesting property regarding it's derivative.

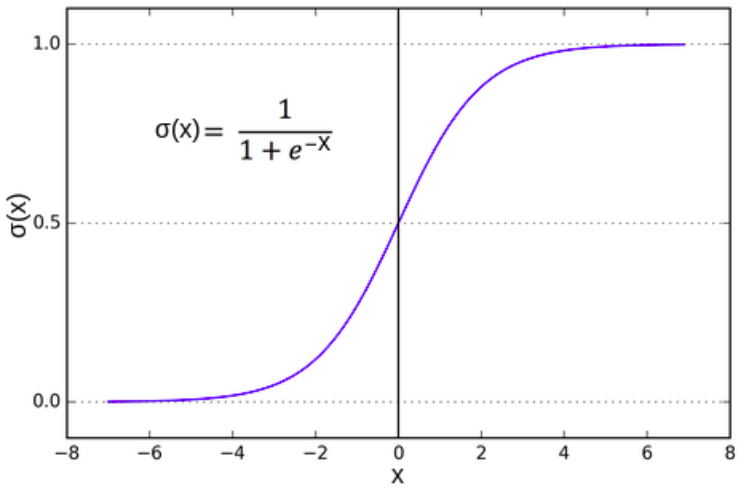


Figure 3.9: Sigmoid function

The sigmoid is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3.6}$$

Figure 3.9 depicts the graph of $\sigma(x)$ and we easily can observe that

$$\lim_{x \to -\infty} \sigma(x) = 0 \text{ and } \lim_{x \to \infty} \sigma(x) = 1, \tag{3.7}$$

while its derivative satisfies

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)). \tag{3.8}$$

Derivatives of activation functions will play an important role at the learning process. We can vary the steepness or the location of the graph with simple algebraic calculations.

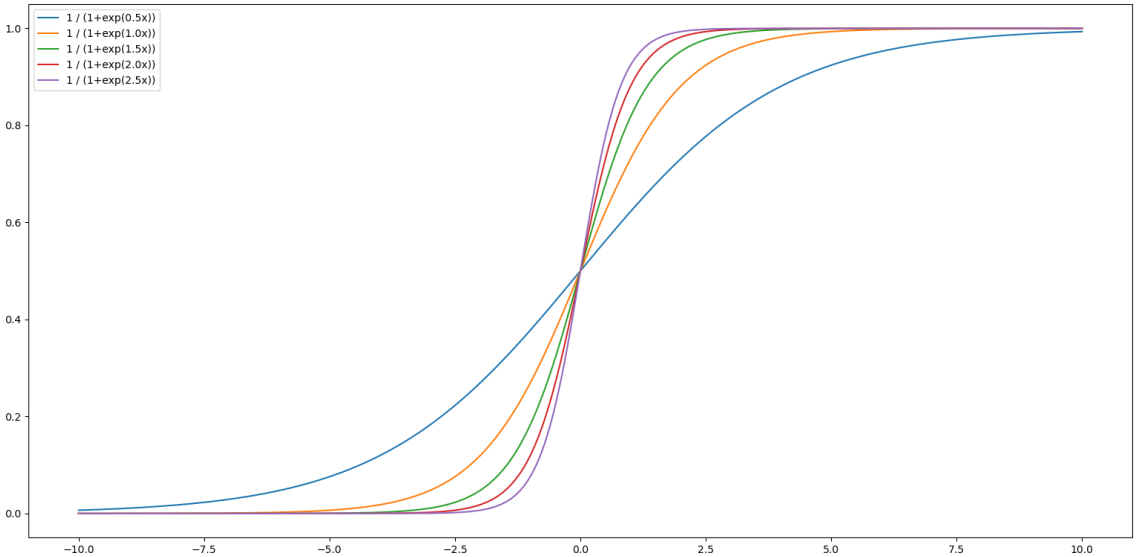


Figure 3.10: Sigmoid functions with various steepnesses

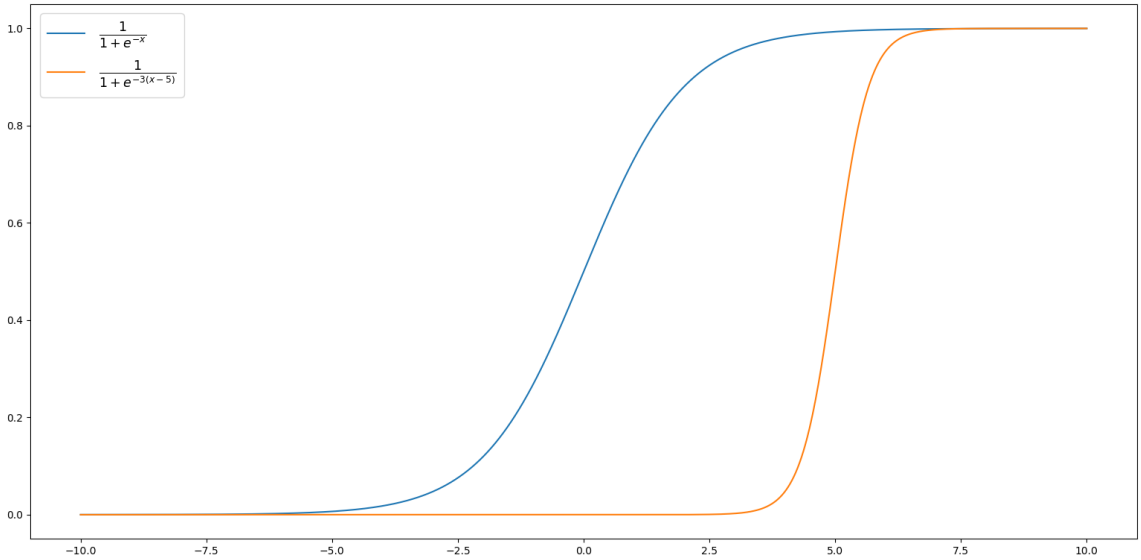


Figure 3.11: Sigmoid functions with different displacement and steepness

For example, Figure 3.10 shows various sigmoid functions with different exponential factors while Figure 3.11 shows the unbiased sigmoid compared to a sigmoid with shifted and scaled inputs.
Lastly, to keep our notation manageable we define the application of the sigmoid function or any other activation function as:
for $x \in \mathbb{R}^m$, $\sigma : \mathbb{R}^m \to \mathbb{R}^m$ is defined by applying the activation function on every element of the vector i.e.

$$(\sigma(x))_i = \sigma(x_i).$$

ReLU (Rectified Linear Unit)
ReLU is another activation function widely used in Convolutional Neural Networks (CNNs) [18] and deep learning. This is an activation function that does not restrict the output in a certain range, since its output can range from 0 to infinity. ReLU is defined as

$$ReLU(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{3.9}$$

or simply put in a compact form, $ReLU(x) = max(0, x)$, with its derivative being a step function

$$ReLU'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{3.10}$$

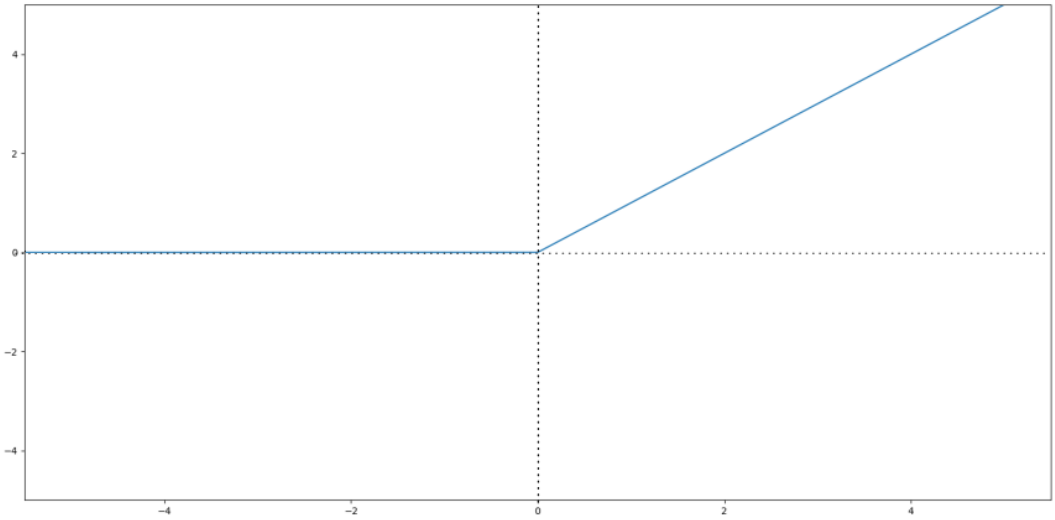


Figure 3.12: ReLU(x)

The derivative is not defined at $x = 0$, however this does not affetct its implementation in practice. One can set the value of $ReLU'(0)$ to be either 1 or 0 without affecting the final results

Hyperbolic Tangent
A similar activation function to the sigmoid is the hyperbolic tangent (tanh) which restricts the output in (-1,1). The advantage over sigmoid is that the negative terms will be mapped to negative terms while positive will be mapped to positive.

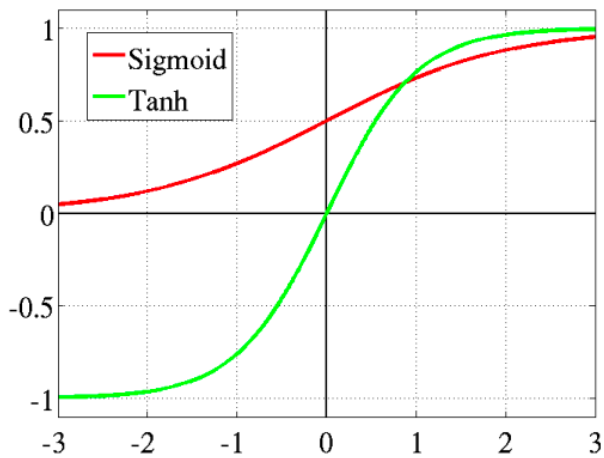


Figure 3.13: tanh vs sigmoid

The derivative of tanh is positive for all $x \in \mathbb{R}$, being

$$(\tanh x)' = 1 - \tanh^2 x, \tag{3.11}$$

so the function is monotonically increasing.

There exists a wide variety of other activation functions, some of them used in different applications. Reader can find more information about activation functions in [10].

3.3 Learning Process

Generally, the aim of the any ANN is to minimize the error between predictions and the observations, by adjusting the weights and biases between the connected neurons. In order to do that, the neural network needs to be trained so it will be able to minimize that metric. This process, will normally ensure that the network has "learned" and hopefully, will give "good enough" predictions. This may be better explained by an analogy. As children, we typically learn what is "good" behaviour by being was is right while "bad" behaviour leads to being punished for acting the wrong way. Suppose that a young kid is sitting by the fireplace for the first time. Not knowing what fire is and its effects the kid attempts to touch the fire, so he puts his hand on it. This action will of course cause the kid to get burned. This negative feedback in the language of NNs is getting a big error. So the next time that the kid will attempt to approach the fire, he stands too far away, feeling almost no heat from the fire and thus staying cold. This is also a negative feeling or an error. So the kid will start repeatedly taking small steps towards the fire, evaluating the heat/cold experiencing each time, trying to find the perfect distance to stand. Feeling less cold each time is like getting smaller and smaller errors and by the same reasoning if the kid overpass the "perfect spot", feeling less and less overheat while taking small steps away from the fire, is also like getting smaller errors. In other words, through experience and feedback the kid learns the optimal distance to sit from the fire. The heat from the fire in this example acts as a metric to evaluate the position he is standing — it helps the learner to correct / change behaviour to minimize mistakes.

ANNs "learn" using this iterative process of trying and evaluate, but in order to evaluate how "bad" their attempt was, they need a mechanism to do so. This mechanism is called *cost function*.
In order to understand what "learned" means for an ANN we need to discuss further what the cost function is and what that "cost" of each error is.

3.3.1 Cost Function

A cost function is a mechanism utilized in supervised machine learning [19] which returns the error between predicted outcomes compared with the actual observations. The aim of supervised machine learning is to minimize the overall cost, thus optimizing the correlation of the model to the system that it is attempting to represent. Simply put, cost function informs us of how "bad" our predictions are, so we can make them a bit better every time. This is typically considered as a difference or distance between the predicted value and the actual value. Cost functions vary, depending on the nature of the problem most of the time. However, sometimes choosing one cost function over another is just a matter of taste. Mean error, mean absolute error, mean squared error, multi-class classification error or cross entropy loss are just some of the many alternatives someone has. At this point it is important to stress that there's no "best" choice for every problem and that some of those cost functions cannot even apply at specific types of problems.
We present now some of them, the most commonly used.

Mean Absolute Error (MAE)
This is a simple, self-explained cost function. It measures the average magnitude of errors in a set of predictions,

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|, \tag{3.12}$$

where i is the index of a given sample from the data set, y_i is the prediction for that given sample that the network outputs, $\hat{y}_i$ is the corresponding actual value and n is the number of all the data samples in the data set.

Mean Squared Error (MSE)
The name of the cost function here is also self-explanatory. This one measures the average of the squared magnitude of the errors. MSE is the go-to cost function most of the time in regression problems and the default metric for evaluating the performance of most regression algorithms in R, Python and other programming languages. The reason this specific cost function is so popular is because it has all the good mathematical properties we need. It is differentiable everywhere (while MEA is not for example) and it is a convex function. Under loose restriction, requiring only continuity, convex functions have one and only one minimum. While this is not an exclusive attribute of MSE, it is one the reasons that MSE is used frequently. Of course, every convex cost function, under the same restrictions, share the same property.

MSE is defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2, \tag{3.13}$$

with the notations being the same as before. Often in literature, MSE is shown as

$$MSE = \frac{1}{2n}\sum_{i=1}^{n}|y_i - \hat{y}_i|^2. \tag{3.14}$$

The factor $\frac{1}{2}$ multiplying (3.13) is only used to make the calculations easier at the gradient of the cost and it does not make any difference in our models behavior.

Binary Cross Entropy
Binary cross-entropy (also known as Log Loss or Logistic Loss) is a special case of categorical cross-entropy when there is only one output that just assumes a binary value, for example 0 or 1 (A or B , YES or NO, and so on) to denote "negative" and "positive" class respectively. It measures how far away from the true value the prediction is for each of the classes and then averages these class-wise errors to obtain the final loss. This cost function is defined as

$$Log\,Loss = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log p(y_i) + (1 - y_i)\log(1 - p(y_i))], \tag{3.15}$$

where $p(y_i)$ is the probability of one of the classes and $(1 - p(y_i))$ is the probability of the other class. We can observe that this function is actually a cohesive way of writing a two-branch function. When the output belongs to the first class only the first part of the cost function is activated while the other part becomes 0 and vice versa.
The same idea can be expanded for multiclass classification using the Softmax cost function [4].

3.3.2 Minimizing the Cost Function

As stated before, the goal of any algorithm in deep learning (and machine learning in general) is to minimize a cost function. We can view this cost function as a mountain where high altitude means high cost and low altitude means low cost. Starting at any random point on its surface, we want to finally arrive at the deepest valley (the lowest point of the surface) taking iteratively steps towards that point. Low cost or even better minimization

of the cost means low or minimum errors between our predictions and actual data.

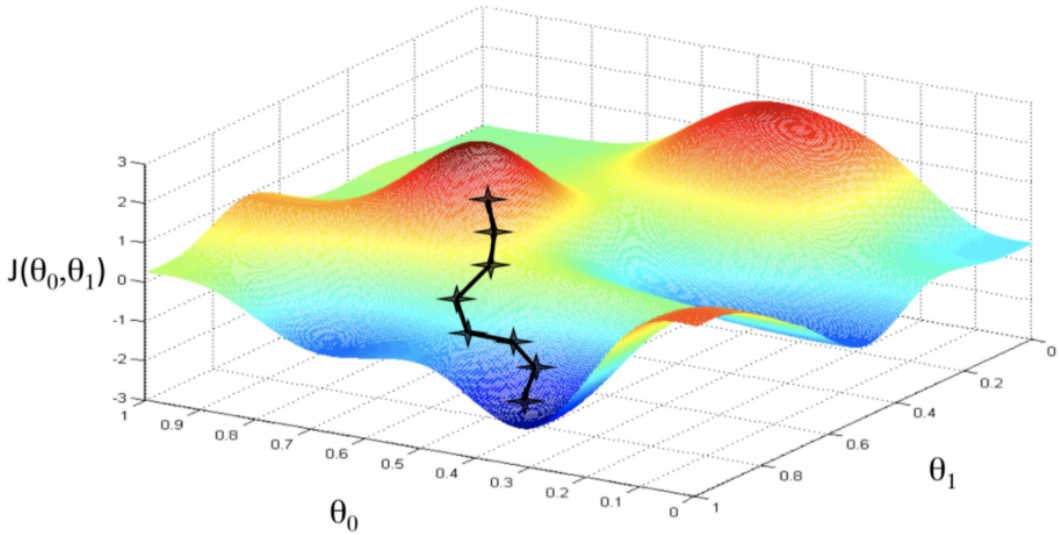


Figure 3.14: Visualization of the cost function

In literature J is commonly used to represent the *cost function* while θ_i are the parameters of the variables in the cost function i.e. weights and biases. In Figure 3.14 θ_0 could be the weight and θ_1 the bias of linear function $y(x) = \theta_0 x + \theta_1$. In order to minimize the cost, machine learning algorithms most often use the Gradient Descent (often referred as Steepest Descent) algorithm, which is a standard method used in optimization problems, emerging from multivariable calculus, [5].
If the dimension of the problem becomes greater than 2 the geometrical representation loses its meaning, however the idea remains the same at arbitrary n dimensions. Therefore, it comes in handy to store the parameters as a single vector $p \in \mathbb{R}^n$.

Definition. The gradient of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ at a point $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is a vector in $\mathbb{R}^n$ of the form

$$\nabla f := \left(\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right). \tag{3.16}$$

Given a function f and a point $x \in \mathbb{R}^n$, the gradient at that point indicates the direction of the steepest ascent of the function, therefore, $-\nabla f(x)$ indicates the direction of the steepest descent at that same point.
In order to find a minimum (local or global), gradient descent, indicates

where our next step should head for. We could summarize the process in the following steps:

- Start with a random point/vector p of parameters θ_i.
- Repeatedly calculate the gradient $\nabla J(p)$, take small steps in that direction and update the parameters using $p := p - \alpha \nabla J(p)$ until we (hopefully) converge to a minimum. The parameter α is called the *learning rate* i.e. the length of the step.

Converging somewhere doesn't always mean that we found the best or even a "good" solution to our problem. For a function with multiple minima, converging somewhere could simply mean that our algorithm got stuck at a local minimum which outputs predictions with big errors and failed to locate the global minimum which would produce the minimum error. This idea is easily understood geometrically in one dimension.

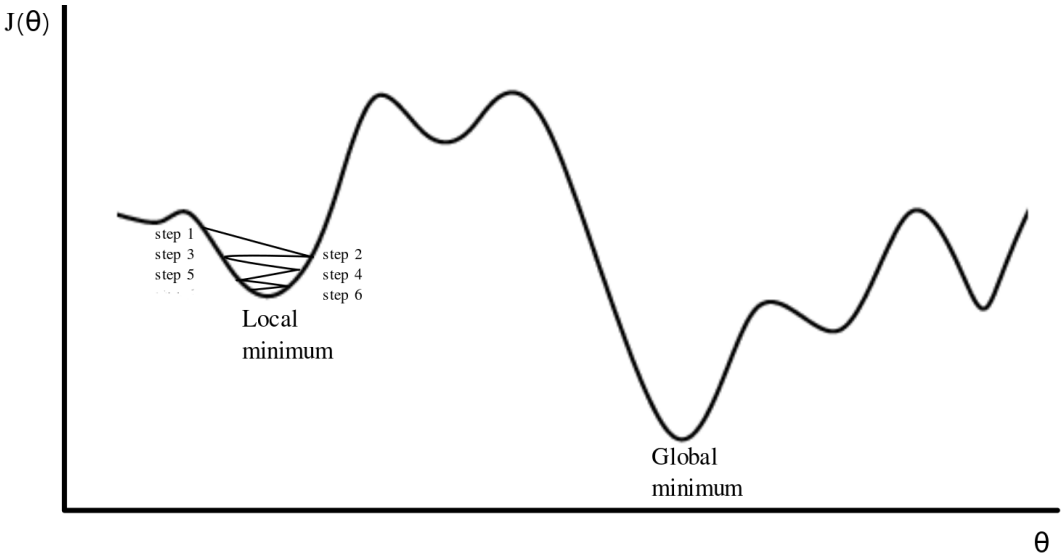


Figure 3.15: Illustration of GD algorithm's step converging to a local minimum

As shown in Figure 3.15, gradient descent algorithm could repeatedly take steps towards the local minimum and get trapped there, never managing to escape, resulting in "bad" predictions. Furthermore, since we perform calculations on an unknown function, we must take discrete steps, meaning this does not even guarantee that the algorithm will converge to local minimum, but it may oscillate near one, or even diverge if the learning rate is too large. This problem occurs frequently, however we could overcome it, simply by

taking a different step size or using some modified method of gradient descent. We will discuss them later in this thesis.
Despite that sometimes problems may occur, the gradient descent algorithm works surprisingly well most of the time. For the record, it might be useful to mention that the method of the discrete steps is motivated by the Taylor series expansion of the cost function calculated at a point $p + \Delta p$ near p.

Batch Gradient Descent (BGD)
Batch Gradient Descent refers to a different way of calculating the gradient for the whole dataset and taking the mean of those to perform one step of the algorithm. This is commonly named as *epoch* as well. This method generally converges in fewer iterations and steadily to the minimum without many oscillations however since we need to pass through the whole dataset in each iteration and calculate all the gradients, this method becomes computationally very expensive.
As many methods in computational mathematics, BGD has a trade-off between accuracy and time complexity of the algorithms. It's not very practical to have a big increase of the time complexity, with rather a small gain in accuracy. That's why in practice, the whole data set is rarely taken in every iteration, especially when the number of examples is large. This problem can be solved by the Stochastic Gradient Descent.

Stochastic Gradient Descent (SGD)
The word "stochastic" means a system or a process that is linked with a random process. SGD takes a random training sample of our dataset, computes the gradient only for that sample and performs an iteration. This idea is used to speed up the process of performing gradient descent, since calculating one instead of thousands or millions of gradients, significantly reduces the time complexity. Of course this will lead to much more noisy or random way to the minima but speed is often more important than the path. After all, SGD tends to work great, despite the randomness acting up in the process.

Mini-Batch Gradient Descent
This variation combines the best of both worlds. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. That is why it is the most common implementation of gradient descent used in the field of deep learning. Instead of taking the whole dataset or single piece of it, mini-batch takes just a "batch", randomly and uniformly selected from the dataset and evaluates the cost based on those examples. Common batch sizes are 32,64,128 and so on, even though, technically someone could use any number of samples. This

method improves the accuracy a lot compared to the stochastic variation and reduces the computational time considerably, compared to the Batch Gradient Descent. The reason behind the speed improvement is the same as in SGD. Fewer samples to evaluate the cost function, means less mathematical operations. The accuracy improvement comes though the randomness the batch is chosen.

Proposition: One can show that the expected value of a mini-batch in SGD is equal to the true empirical gradient in the same way one can show that the mean of any simple random sample is an unbiased estimator of the population mean: linearity of expectation.

The following proof is suggested by the data scientist, Conner Davis, Microsoft [9].

Proof
Linearity of expectation simply means that $E[X+Y] = E[X] + E[Y]$.
We have that

$$J(X) = \frac{1}{n}\sum_{i=1}^{n} Loss(f(x_i), y_i).$$

Differentiate both sides, with regard to x and use the linearity of differentiation to move the ∇ inside the summation $\nabla J(X) = \frac{1}{n}\sum_{i=1}^{n} \nabla Loss(f(x_i), y_i)$.
We want to evaluate

$$E_A[\frac{1}{m}\sum_{i=1}^{m} \nabla Loss(f(x_i), y_i)].$$

Applying linearity of expectation, this yields,

$$E_A[\frac{1}{m}\sum_{i=1}^{m} \nabla Loss(f(x_i), y_i)] = \frac{1}{m}\sum_{I=1}^{m} E_A[\nabla Loss(f(x_i), y_i)].$$

$E_A[\nabla Loss(f(x_i), y_i)]$ means $E[X] = \sum_x x * P(X = x)$.

Since the samples are chosen uniformly at random, all their probabilities ($P(X = x)$) are equal to $\frac{1}{n}$, so it's just the average value of the gradient over all samples. That is:

$$E_A[\nabla Loss(f(x_i), y_i)] = \sum_{j=1}^{n} P(i = j) * \nabla Loss(f(x_j), y_j),$$

where $P(i = j) = \frac{1}{n}$, so

$$E_A[\nabla Loss(f(x_i), y_i)] = \frac{1}{n}\sum_{j=1}^{n} \nabla Loss(f(x_j), y_j) = \nabla J(X).$$

Plugging that back in to,

$$E_A[\frac{1}{m}\sum_{I=1}^{m} \nabla Loss(f(x_i), y_i)] = \frac{1}{m}\sum_{i=1}^{m} E_A[\nabla Loss(f(x_i), y_i)],$$

we get

$$E_A[\frac{1}{m}\sum_{i=1}^{m} \nabla Loss(f(x_i), y_i)] = \frac{1}{m}\sum_{i=1}^{m} \nabla J(X) = \nabla J(X),$$

which completes the proof. ■

3.3.3 Backpropagation

Backpropagation is one of the most important steps in the Gradient Descent algorithm. We already discussed how gradient descent works to minimize the cost function by updating the parameter vector p in each iteration (also called epoch). This, as a matter of fact, comes downs to calculating the derivative of the loss function with respect to the each parameter i.e. weights and biases. We give a deeper intuition about that here, since that action of updating the parameters occurs through the process called *Backpropagation*. For simplicity, we suppose that the cost function is taken over a single data point. This will make the idea easier to understand while in the general case of a set with multiple data points is not much more complicated than this.
We recall that in order to minimize the loss function we first need to compute the cost. This was achieved by passing the data from the input layer all the way to the output layer and then compare the prediction of our model with the actual known value using a metric. This is called *forward pass*, also known as *forward propagation*.
Now that we know how inaccurate our prediction is, we need to go all the way from the final level, inductively to the beginning of our network, modifying the weights and biases since those are the ones that determine the prediction. This process of transferring the information backwards in the network coins the term *backpropagation*.
From now on, our aim is to compute the partial derivatives of the cost func-

tion with respect to weights w^l_{jk} and biases b^l_j. Consider the MSE cost function,

$$C = \frac{1}{N}\sum_{i=1}^{N} ||f(x_i) - F(x_i)||^2, \tag{3.17}$$

where $||\cdot||$ is the Euclidean norm in $\mathbb{R}^m$, $f(x_i)$ is our model's prediction at the data point x_i and $F(x_i)$ is the actual value of the function we want our model to approximate at the same data point. Since MSE is differentiable we have

$$\nabla C = \nabla(\sum_{i=1}^{N} C_{x_i}) = \sum_{i=1}^{N} \nabla C_{x_i}, \tag{3.18}$$

where C_{x_i} is the cost of a single data point x_i. This property of calculating the gradient of the cost first and summing them up later is extremely useful, since gradients can be computationally demanding. SGD and mini-batch GD take advantage of that and thus making the code much more efficient.
We now define $A_\ell = \{a^l_i\}$ to be a vector containing all the outputs of the activation functions of nodes in the lth layer. We recall that

$$z^l := \sum_{k}^{n_{l-1}} w^l_{j,k} a^{l-1}_k + b^l_j \in \mathbb{R}^n, \quad \text{for } l = 2, 3, \ldots, L. \tag{3.19}$$

From the relation that propagates the information through the NN passed through the activation function a, we obtain

$$a^l = a(z^l) \in \mathbb{R}^n, \quad \text{for } l = 2, 3, \ldots, L. \tag{3.20}$$

Consider the quantity $\delta^l \in \mathbb{R}^n_l$ defined as,

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}, \quad 1 \leq j \leq n_l \,,\, 2 \leq l \leq L. \tag{3.21}$$

This quantity is called *error* in the jth neuron of layer ℓ. It is an intermediate quantity, useful for analysis and computation, it is used for backprogation of the network and is related to the derivatives of the weights and biases $\frac{\partial C}{\partial w_{j,k}}, \frac{\partial C}{\partial b^l_j}$ by the chain rule.

Before showing some practical relations between cost and weights, we need to define the Hadamard, i.e componentwise, product of two vectors. For $x, y \in \mathbb{R}^n$ we define $(x \odot y)_i = x_i * y_i,\ i = 1, \ldots, n$

Lemma 2 *Let y be an actual value. Then the following relations hold true,*

$$\delta^L = a'(z^L) \odot (a^L - y), \tag{3.22}$$

$$\delta^l = a'(z^L) \odot (W^{l+1})^T \delta^{l+1}, \quad 2 \leq l \leq L-1, \tag{3.23}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l, \quad 2 \leq l \leq L, \tag{3.24}$$

$$\frac{\partial C}{\partial w_{j,k}^l} = \delta_j^l a_k^{l-1}, \quad 2 \leq l \leq L. \tag{3.25}$$

Proof
To prove (3.22) we recall that $\delta_j^L = \frac{\partial C}{\partial z_j^L}$ and $a^L = a(z^L)$. Further we consider the cost function to be the MSE $C = \frac{1}{2}||y - a^L||_2^2$. Then, we have

$$\frac{\partial a_j^L}{\partial z_j^L} = a'(z^L) \quad \text{and} \quad \frac{\partial C}{\partial a_j^L} = \frac{\partial}{\partial a_j^L}\left(\frac{1}{2}\sum_k^{n_L}(y_k - a_k^L)^2\right) = \frac{\partial}{\partial a_j^L}\left(\frac{1}{2}(y_j - a_j^L)^2\right).$$

The second equality is derived from the fact that

$$\frac{\partial}{\partial a_j^L}\left(\frac{1}{2}\sum_k^{n_L}(y_k - a_k^L)^2\right) = \frac{1}{2}\frac{\partial(y_1 - a_1^L)^2}{\partial a_j^L} + \frac{1}{2}\frac{\partial(y_2 - a_2^L)^2}{\partial a_j^L} + \cdots\cdots + \frac{1}{2}\frac{\partial(y_{n_L} - a_{n_L}^L)^2}{\partial a_j^L}$$

so when taking the derivative with respect to j only the jth term remains. Hence

$$\frac{\partial C}{\partial a_j^L} = \frac{\partial}{\partial a_j^L}\left(\frac{1}{2}(y_j - a_j^L)^2\right) = -(y_j - a_j^L) = (a_j^L - y_j).$$

Thus

$$\delta_j^L = \frac{\partial C}{\partial Z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j} = (a_j^L - y_j)a'(z_j^L) = a'(z^L) \odot (a_j^L - y_j)$$

proving (3.22).

To prove (3.23) we recall that from (3.19) we have $z_k^{l+1} = \sum_{s=1}^{n_l} w_{ks}^{l+1} a(z_s^l) + b_k^{l+1}$.
Hence

$$\delta_j^l = \frac{\partial C}{\partial z_j} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_{k=1}^{n_{l+1}} \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l}.$$

By the definition of z_k^{l+1} and by differentiating it with respect to z_j^l only the jth term remains, hence, we have $\dfrac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} a'(z_j^L)$. Substituting this to the previous equality for δ_j^l we have

$$\delta_j^l = \sum_{k=1}^{n_{l+1}} \delta_k^{l+1} w_{kj}^{l+1} a'(z_j^l) = a'(z_j^l)\Big((W^{l+1})^T \delta^{l+1}\Big)_j,$$

which is the componentwise form of (3.23).

For (3.24), since $z_j^l = (W^l a(z^{l-1}))_j + b_j^l$ we obtain $\dfrac{\partial z_j^l}{\partial b_j^l} = 1$. Notice that z^{l-1} does not depend on b_j^l. Using the chain rule

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l,$$

by definition, proving (3.24).

Finally, for (3.25) we will once more use the definition of z_j^l (3.19) and taking the derivative with respect to w_{jk}^l we have

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1}, \quad \forall j, \qquad \text{while} \qquad \frac{\partial z_s^l}{\partial w_{jk}^l} = 0 \quad \text{for} \quad s \neq j.$$

Therefore by the chain rule, we obtain

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a_k^{l-1} = \delta_j^l a_k^{l-1},$$

which completes the proof. ■

3.4 Gradient Descent Optimization Algorithms

Beside the three variants of the gradient descent we studied earlier, which differ on how much data used to compute the gradient of the cost function J, there are other gradient descent-type optimization algorithms that aim to overcome problems such as: a) choosing a proper learning rate, b) avoid being trapped in a local minimum, c) accelerating the convergence of the algorithm. Next we discuss some of the variations. The presentation follows, [1]

3.4.1 Gradient Descent with Momentum

Gradient descent, with momentum can be though as a ball rolling downhill, where the ball will still roll to direction of the previous descent for a while, even if the hill's slope changes.
In areas where the surface of the cost function curves by a large amount in one direction than on others or noisy gradients, something that may happen near local optima, SGD will bounce around in the process of navigating trough that space, resulting in making limited progress or even get stuck in flat spots.
Momentum helps to accelerate SGD by an attempt to maintain a consistent direction. We take a linear combination of the previous heading vector, and the newly-computed gradient vector, and adjust in that direction.

$$\begin{aligned} u_t &= \gamma u_{t-1} + a\nabla_p J(p), \\ p &= p - u_t. \end{aligned} \tag{3.26}$$

Here, γ is just a parameter, called *momentum term*, suggesting the fraction of the previous time step we want to add to the current vector. A usual value of γ is 0.9.

3.4.2 Adagrad (Adaptive Gradient Algorithm)

Adagrad is an adaptive-learning rate algorithm. This means that the learning rate used is not fixed but it changes. Adagrad adapt the learning rate for the parameters depending on their history. The method chooses large learning rates for rare and small updates, while small rates are used for more frequent parameters.
We set

$$g_{t,i} = \nabla_{p_t} J(p_{t,i}),$$

to be a vector of the gradient of the cost function with respect to the step t and parameter p_i. The Adagrad componentwise parameter update becomes

$$p_{i+1,j} = p_{i,j} - \frac{a}{\sqrt{G_{i,jj} + \varepsilon}} g_{i,j}, \tag{3.27}$$

where a is a fixed value of the learning rate and ε is a small quantity which is added to prevent division by zero. In practice, ε takes small values ($\varepsilon \sim 10^{-8}$) and does not affect the effectiveness of the algorithm.
G_i is a diagonal matrix with its (j, j) element given by the sum of the squares of the gradients with respect to p_j in the ith step,

$$G_{i,jj} = \sum_{k=1}^{i} g_{k,j}^2.$$

However, this process has a major drawback. Due to the accumulation of the squared gradients, the denominator grows during training, resulting into small values of the learning rate, to the point where the algorithm is unable to learn anymore.
The issue is resolved by algorithms such as Adadelta or RMSprop. For a deeper understanding of those algorithms we refer to [1].

3.4.3 Adam (Adaptive Moment Estimation)

The *Adam* optimization algorithm is an extension of the stochastic gradient descent we have presented. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients and keeps an exponentially decaying average of past gradients m_t similar to momentum:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ u_t &= \beta_2 u_{t-1} + (1-\beta_2)|g_t|^2, \end{aligned} \tag{3.28}$$

where m_t and u_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of gradients respectively. The authors of Adam propose β_1 to be 0.9 and β_2 to be 0.999.
The vectors m_t and u_t are initialized as zeros, hence, they are biased towards it, especially when the learning rates are small, for example the case for the proposed values of β_1 and β_2 and during the initial steps.
This bias can be overcomed by first calculating the biased estimates, and then calculating bias-corrected estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\ \hat{u}_t &= \frac{u_t}{1-\beta_2^t}. \end{aligned} \tag{3.29}$$

Finally, the update rule yields:

$$p_{t+1} = p_t - \frac{a}{\sqrt{\hat{u}_t} + \varepsilon} \hat{m}_t. \tag{3.30}$$

The proposed value for ε is 10^{-8}, which serves the same purpose as we have seen before.

3.4.4 AdaMax

There are two small variations on Adam, the first one called *AdaMax* and the second one *Nadam*[1]. The formula $u_t = \beta_2 u_{t-1} + (1-\beta_2)|g_t|^2$ from Adam will

scale the gradient inversely proportionally to the ℓ_2 norm of past gradient (u_{t-1} term) and current gradient $|g_t|^2$ meaning as the scale factor goes up the gradient magnitude will go down. The AdaMax uses the ℓ_∞-norm instead of the ℓ_2-norm and reduces to

$$\begin{aligned} u_t &= \beta_2^\infty u_{t-1} + (1-\beta_2^\infty)|g_t|^\infty \\ &= max(\beta_2 \cdot u_{t-1}, |g_t|) \end{aligned} \tag{3.31}$$

Finally, we can plug this into the Adam update rule, replacing the $\sqrt{\hat{u}_t} + \varepsilon$ with the u_t from (3.31), which yields the final update rule for AdaMax:

$$p_{t+1} = p_t - \frac{n}{u_t}\hat{m}_t. \tag{3.32}$$

3.4.5 RMSProp (Root Mean Square Propagation)

RMSProp is another adaptive learning rate optimization technique that choses a different learning rate for each parameter. It was proposed by the father of back-propagation, Geoffrey Hinton in Lecture 6e of his Coursera Class [11]. Gradients of very complex functions like neural networks usually vanish or explode as the data propagates through the function.
RMSprop addresses this issue by dividing the learning rate by an exponentially decaying average of squared gradients. This normalization, results in balancing the step size (momentum), decreasing the step for large gradients and increasing the step for small gradients. RMSprop's update rule is given by:

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2, \\ P_{t+1} &= P_t - \frac{n}{\sqrt{E[g^2]_t + \varepsilon}} g_t. \end{aligned} \tag{3.33}$$

Hinton suggests γ to be set to 0.9, while a good default value for the learning rate n is 0.001.

In the first equation, we compute an exponential average of the square of the gradient. The name "exponential" comes from the fact that the weight of previous terms decays exponentially. Since it's performed separately for each parameter, gradient g_t corresponds to the component of the gradient along the direction represented by the parameter we are updating.

We multiply the hyperparameter γ with the exponential average $E[g^2]$ computed till the last update. We then multiply the square of the current gradient with $(1-\gamma)$ add those two terms together to get the exponential average till the current step.

Chapter 4

Implementation/Methodology

We now move on to the case study scenario, where we implement deep neural networks to a realistic problem. That is the prediction of the temperature developed inside the PV cells using three different methods, that differ on the inputs provided to the neural network. The DNNs implemented, vary in the number of hidden layers and number of neurons used, as well as, the learning rates and the optimization algorithms used. The key difference between the first and the last two methods we examine, is that the first one is based on *a posteriori* data, (latin for "from the latter"), meaning that the data were based upon actual observation, they were experimental data. Those are data collected during the PV cells were operating.
The seconds and third method, in contrast with the first one, use *a priori* data (latin for "from the former"), meaning that those data were gathered without examination or analysis and they are supported by past experience.

Since some of the variables used in methods 2 and 3 might seem that they need to be collected during the specific time and date we examine and that is of course, contradictory to the a priori definition, it is important to mention that variables like solar irradiance, air temperature and wind speed can be estimated using typical meteorological years (TMY). These are collections of data, already existing, for a specific location, listing various meteorological elements for one year period and predetermined time points during the day, typically every hour. Those data are, of course, just estimations using previous years (normally 10 previous years or more) to estimate the next following year's meteorological conditions. With that in mind, all data in need to implement method 2 and 3 can be found through those TMY.

The neural networks we used were mostly built using TensorFlow 2.5.0[16] running on a laptop with a low-end AMD APU (A12-9720P) without a dedicated cuda core GPU and 6GBs of DDR4 ram. This is extremely important

to mention since hardware makes a huge difference in time needed to complete a run. A second setup was also used, building the neural network from scratch using Python 3.9.5[17], both giving similar results.

4.1 Train Set, Validation Set and Test Set

A common practice before proceed to the learning process is splitting the available data into 2 or 3 **distinct** data sets. These data sets will consist of a *training* set, a *test set* and sometimes a *validation set*. For example one can randomly choose 70% of the data to make up the train set and 30% to make up the test set, or 60% for the train set, 10% for the validation set and 30% for the test set. Percentages can vary of course and there is no rule of thumb for choosing them. Like many other things in machine learning, the train-test-validation split ratio is quite specific to each individual use case and it gets easier to make judgement with experience.

The **training set** is what it sounds like. It is the part of the data we will use to train our model. During each epoch, our model will be trained over and over again on those same data points and it will continue to learn about the features of this data. We hope that this will eventually make our model able to predict accurately on new previously unseen data. Learning during the training process does not guarantee that our model has actually learned to recognize patterns among features. Very small errors during the training phase is sometimes a bad indicator, as this can mean that our model has "overfit" and it will perform purely on the new unseen data. We will not discuss in detail about overfitting but the idea of it is that our model becomes really good at being able to classify the data in the training set but it's unable to generalize and make accurate classifications on data that it wasn't trained on, [20].

The **validation set** (also called dev set or development set) is another sample of the total data that is used to validate our model **during** the training process. This validation process helps us by giving information that may assist us with adjusting the hyperparameters. We know that during training, the model will be classifying the output for each input in the training set and after computing the loss, it will adjust the weights/biases. The data in the validation set do not take part in the training process, but they are only used to compute the loss function on unseen data, providing an unbiased evaluation of the model. So when a validation set is used, the model will provide us with two different losses. One of the training set and one of the validation set.

The loss on the validation set is expected to be higher than the corresponding loss on the train set but this is expected. Recall the samples of the validation set is separate from those in the training set so when our model is validating on the validation set, that part of data does not consists of samples the model is already familiar with from training. What we do not want to see is huge differences between the two losses as this is commonly caused when overfitting.

Another use of the validation set is *early stopping*. If our model's error on the validation set is smaller than a threshold we set for a certain number of consecutive epochs, there might be no need to continue training. The model might have learned to predict the outputs at an accepted level so the training process can stop earlier than expected.

The **test set** is also self-explanatory. It is the sample of data used to provide an unbiased evaluation of the final model fit on the training dataset. This set is only used once, after a model is completely trained. Note that many times there is no validation set used and validating of the model occurs only after the training is complete. The loss on the test set is also expected to be a bit higher than the training set, but again, big differences indicate there is something wrong.

4.2 Method 1

The first method we implemented was based on data gathered by sensors placed at panels. Samples were collected from around 6:50 to 18:20, every five minutes for a 10 day period, from 28-Mar-2018 to 06-Apr-2018. The sample data make up a total of 1379 samples for each one of the 4 parameters: *solar irradiance* $G_{irr}(w/m^2)$, *air temperature* $T_{air}(C)$, *open circuit voltage* $V_{oc}(V)$ and *short circuit current* $I_{sc}(A)$. We can represent those data as a 1379×4 matrix.

An extra sensor was also placed to record the wind speed, represented by $W(m/s)$, from 28-Mar-2018 00:00:00 (UTC) to 06-Apr-2018 23:00:00 (UTC), however those samples were taken once every hour. Due to lack of information of the interior time nodes we used linear interpolation to estimate the wind speed in those points. We can think of those data as an extra column, expanding the matrix dimensions to 1379×5, see Table 4.1.

Of course, date and time play no role in the cell's temperature so we will not feed them in the NN, but we present them here for a complete understanding of the data form. The useful variables used in the networks will be the five ones mentioned above.

Lastly since our aim is to estimate the PV modules' temperature T_{mod}, we also tracked it in the same five-minute nodes, so the neural network can compare the predictions with those values and "learn" to forecast the temperature. This gives us an array of 1379×1.

Date	**Time**	$\boldsymbol{T_{air}}$	$\boldsymbol{G_{irr}}$	$\boldsymbol{V_{oc}}$	$\boldsymbol{I_{sc}}$	$\boldsymbol{W}$	$\boldsymbol{T_{mod}}$
28-03-2018	06:50:00	24.3	38.41	31.31	0.306	4.22	23.32
28-03-2018	06:55:00	24.6	55.53	31.64	0.386	4.25	23.51
28-03-2018	07:00:00	24.8	69.14	32.01	0.492	4.29	23.92
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
06-04-2018	18:20:00	28.7	33.42	30.32	0.227	6.53	27.15

Table 4.1: Representation of data used in Method 1 as matrices

We can think every row of the matrix as a single vector of $\mathbb{R}^5$. Those five variables will be the data we feed to the neural network at the input layer. Before feeding the data to the network, we normalize them. There are various normalization strategics, depending on the data themselves while sometimes it is just a matter of preference, for example changing the variance of the data, or scaling to a range, or a combination of those. Normalization is one the most important actions one can perform to speed up the algorithm and help the convergence. Of course, normalization is a whole other chapter in machine learning science which is beyond the scope of this thesis, so we will not dive into details. However, normalizing the data hardly ever does any harm so it is a very standard practice to normalize the dataset before proceed to feed them in the network.

The normalization method we performed was simply scaling our data, since the raw data gathered from the sensors had very different ranges. For example short circuit voltage was between 0.22 to 8.61, while solar irradiance values were ranging from 29.72 to 957.0.
T_{air} was divided by 40, G_{irr} was divided by 1000, V_{oc} by 35, while I_{sc} and W were both divided by 10. Lastly T_{mod} was divided by 65. All those numbers were chosen to be slightly higher than the maximum value in the dataset for each variable and since all the values were positive all data were scaled to reside in $[0, 1]$

So our network begins with five nodes and propagates the processed data to first hidden layer. Various combinations of hidden layers and number of neurons have been implemented, with hidden layers starting from just 2, up

to 5 and with number of neurons in every hidden layer being 5, 10, 20, 30, 40. All networks output a single value which is the temperature forecast. This gives us a total of 20 different structures tested. In every case the neural networks were fully connected.

4.3 Method 2

The second method we implemented was a lot like the first one with two differences. This method is focused more on the PV modules characteristics per se, rather than the external factors that may affect the internal temperature. Open-circuit voltage and short-circuit current are electrical characteristics of the PV module that are not available under operating conditions. Thus instead of using the open-circuit voltage V_{oc} we used the module's efficiency n_{ref} in percentage which is 14.5% for the monofacial solar panels we studied. So just like in the first method we have a the same 1379×5 matrix but now the column of V_{oc} has been replaced by 0.145 (14.5%) for every element of that column. Since the panels we examine are all the same this number does not change.

Secondly, we replaced the short-circuit current I_{sc} by the Normal Operating Cell Temperature (NOCT, symb: T_{NOCT}) which for the modules we examined is 45.7 $°C$. These are the PV module specifications based on manufacturer's datasheets. Thus the forth column of the matrix will be changed to 45.7 for every element instead of the V_{oc}.

The normalization for every variable remains the same as before except the two new variables we inserted. Since the module's efficiency is between 0 and 1 there is no need to make any change to that so we leave it as it is. However since T_{NOCT} is way bigger than 1 it needs to be normalized. We divide the column of T_{NOCT} by 65 which is the number we used to normalize the T_{mod} as well.

Date	**Time**	T_{air}	G_{irr}	n_{ref}	T_{NOCT}	W	T_{mod}
28-03-2018	06:50:00	24.3	38.41	0.145	45.7	4.22	23.32
28-03-2018	06:55:00	24.6	55.53	0.145	45.7	4.25	23.51
28-03-2018	07:00:00	24.8	69.14	0.145	45.7	4.29	23.92
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
06-04-2018	18:20:00	28.7	33.42	0.145	45.7	6.53	27.15

Table 4.2: Representation of data used in Method 2 as matrices

All setups deployed were also fully connected and various runs for each of 2 to 5 hidden layers and 5, 10, 20, 30, 40 number of neurons in the hidden layers were tested giving a total of 20 structures as before.

4.4 Method 3

In this method we use some of the models we find in literature to forecast the module's temperature [21]. However most of those models assume that the solar cells operates under nominal (STC) conditions ($25°C$) which is hardly ever happening in reality. We use five of those models, some of them using electrical characteristics of the module such as I_{sc} and V_{oc} while others use parameters like the air temperature, wind velocity, solar irrandiance and module's efficiency. Those parameters have the advantage that they can be available a priori.
The five models we use provide us with an estimation of the module operating temperature in Celsius degrees:

$$T_c^t = T_{air} + \frac{G_{irr}}{800}(T_{NOCT} - 20)(1 - n_{ref})\left(\frac{9.5}{5.7 + 3.8W}\right) \tag{4.1}$$

$$T_c^S = T_{air} + 0.0138G_{irr}(1 + 0.031T_{air})(1 - 0.042W)(1 - 1.053n_{ref}) \tag{4.2}$$

$$T_c^C = 0.943T_{air} + 0.028G_{irr} - 1.528W + 4.3 \tag{4.3}$$

$$T_c^L = 30.006 + 0.0175(G_{irr} - 300) + 1.14(T_{air} - 25) \tag{4.4}$$

$$T_c^K = T_{air} + G_{irr}e^{-3.473-0.0594W} \tag{4.5}$$

In the beginning of this thesis (2), we mentioned that those formulas are not very accurate. In Figure4.1 we can see all those formulas compared to the experimental temperature measured.

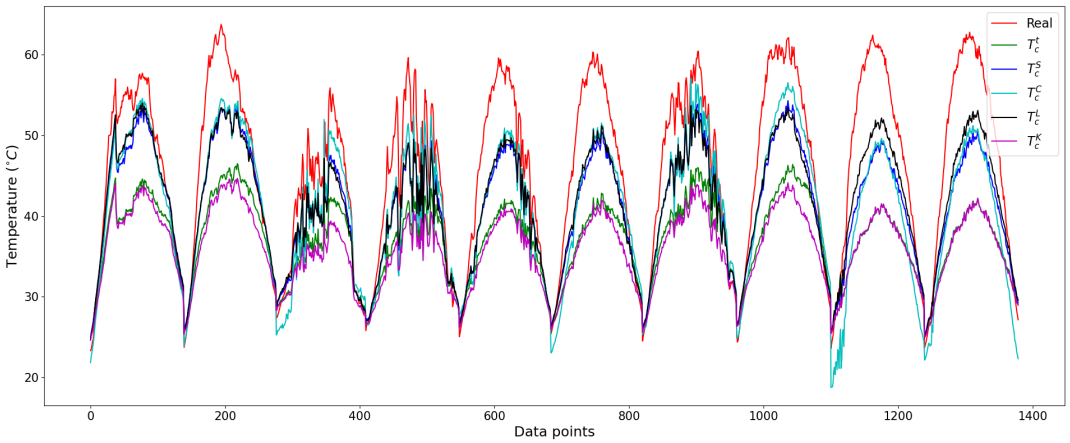


Figure 4.1: Different formulas compared to real data

We feed those equations with the data available and they provide us with estimations of the operating temperature of the solar cells. This leads to a new 1379×5 matrix with every row having 5 different temperature estimations.

Date	**Time**	$\boldsymbol{T_c^t}$	$\boldsymbol{T_c^S}$	$\boldsymbol{T_c^C}$	$\boldsymbol{T_c^L}$	$\boldsymbol{T_c^K}$	$\boldsymbol{T_{mod}}$
28-03-2018	06:50:00	24.76	24.94	21.83	24.63	24.64	23.32
28-03-2018	06:55:00	25.26	25.53	22.54	25.27	25.09	23.51
28-03-2018	07:00:00	25.61	25.97	23.06	25.73	25.41	23.92
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
06-04-2018	18:20:00	28.98	29.23	22.32	29.55	28.95	27.15

Table 4.3: Representation of data used in Method 3 as matrices

Every row of this matrix consists of a vector in $\mathbb{R}^5$ that will be fed into the neural network. Since every element of the matrix is a temperature estimation we normalize them by dividing with 65 just like the measured temperature. The same structures of ANNs as in the two previous methods were tested.

Chapter 5

Numerical Results

We begin by comparing the results obtained using TensorFlow and the code we build from scratch. It is important to mention that Stochastic Gradient Descent was chosen as the optimizer for the "in house code" due to it's simplicity and the low computational complexity. It is also important to note, that despite the fact that we allowed the "in house" code performed a lot more iterations than the TensorFlow, the time required for the first to finish compiling was still lower. This happens as a result of the difference in computing the updates of the weights and biases. In the "in house" code we used equations (3.22)-(3.25) of Lemma 2 to compute the cost gradients. This way there was no need to compute the cost of the training set in every single iteration, something which is very computational demanding. We only computed the cost every 10.000 iterations -just to have some data for plotting- thus reducing the time the code needed to finish extensively.

As an example of this, below are shown two images, comparing the results in temperature forecasting, one produced using the TensorFlow and one using the code we developed.

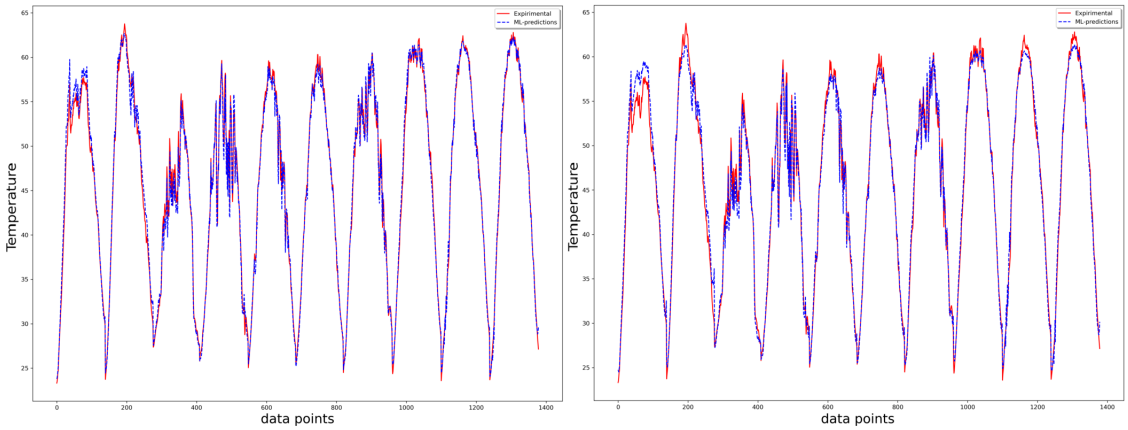


Figure 5.1: Left: "in house" code - Right: TensorFlow

Both of the NNs we used to predict the temperature of the PV module shown in Figure 5.1 used the first method to do so, had 2 hidden layers and used SGD with a learning rate of 0.2. As expected errors are not exactly the same but they are close enough. Our NN from "in house code" had a MAE of $0.697°C$ at 20 million epochs while the TensorFlow's NN produced a MAE of $0.889°C$ at 50 thousand epochs. Despite the huge difference in epochs, our NN took 2320 seconds to complete while the TensorFlow's took 3088 seconds. Note that TensorFlow computes gradients in a different way than the "in house code", so epochs can not be compared as an absolute number. They do not perform the same actions. 20 million epochs on the TensorFlow would likely have a produced a vastly different result in terms of error order.

Since both TensorFlow and "in house code" give similar results, but TensorFlow makes things easier in implementation we proceed to compare some of the key features the DNNs incorporate using TensorFlow only. These are the number of hidden layers, the number of neurons, the learning rate, the number of epochs, the optimization algorithms and last but not least the percentage of samples used in the training sets. Considering that this is a multivariable object of study and examining all those parameteres at once is not very practical, we discuss each of those features individually or in pairs, keeping everything else the same.
Note that the errors presented above and in the upcoming sections are obtained by validating our model on all available examples after training, unless specified otherwise. All neural networks implemented in this thesis, use the **sigmoid** activation function and used **mean squared error** as an error metric of the cost function. However we present the differences between predictions and experimental values in mean absolute error, since this is closer

to the human perception of differences in temperature.

5.1 Number Of Hidden Layers and Neurons

The learning rate was set to 0.02 since this one seemed to be a good choice during the preparation and the test runs for the problem of this thesis. The optimization algorithm we kept through this study was Adamax even though all of the algorithms mentioned in 3.3.2 and 3.4 were tested and produced similar results. The reason for this choice was simply past experience from other problems where Adamax was yielding the best results, plus, no other algorithm seemed to excel comparing to the others at the test runs. The number of epochs used for this section was 50000 for every run.

Table 5.1 shows the mean absolute errors in Celsius degrees obtained using the **first method** for various combinations of number of hidden layers (HL) and number of neurons. The heading of the table refers to the number of hidden layers while the left column refers to the number of neurons in every hidden layer.

Neurons \ HL	2	3	4	5
5	0.711	0.623	0.602	0.609
10	0.589	0.578	0.532	0.447
15	0.630	0.463	0.435	0.375
20	0.619	0.629	0.392	0.396
30	0.684	0.445	0.326	0.299
40	0.567	0.530	0.322	0.229

Table 5.1: MAEs ($^{\circ}C$) Method 1

What we observe is that in general, adding layers and neurons will improve the accuracy of the network. However the improvements will begin fade past a number of neurons and layers, giving smaller and smaller improvements while increasing the complexity of the algorithm.

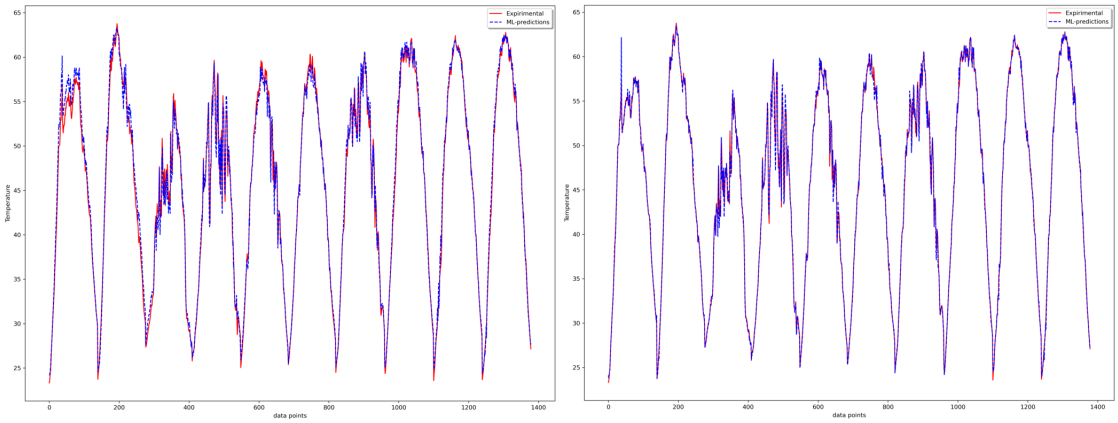


Figure 5.2: Left: 2 hidden-5 neurons - Right: 5 hidden-40 neurons

In Figure5.2 we can see the graphs of two different DNNs from the ones we used to get the results of Table 5.1, comparing the observed temperature (red) with the network's predictions (blue). On the left side of the figure we have the predictions of the network with just 2 hidden layers and 5 neurons in each hidden layer, while on the right side we have the predictions of the network with 5 hidden layers and 40 neurons in each hidden layer. Both of those give excellent results and succeed to approximate the temperature really close to the true value most of the times. However, the network with just 2 hidden layers fails to approach the temperature enough at the extreme temperatures, while the bigger network is practically on the experimental data all of the time.

Another thing that we can observe from the table, is that the descending course of the error is not always guaranteed by adding neurons. While the general pictures implies that result, most of the times while we are approaching a minimum, the cost function starts to fluctuate vigorously up and down. This makes stopping at a lower point of the cost function at the last epoch just a matter of luck, but we will discuss it in 5.2.

Moving on to the **second method**, we obtained the results shown in Table 5.2.

Neurons \ HL	2	3	4	5
5	1.233	1.182	1.074	1.109
10	1.106	0.863	0.945	0.806
15	1.067	0.771	0.663	0.699
20	0.928	0.659	0.564	0.471
30	0.978	0.609	0.563	0.622
40	0.963	0.604	0.475	0.424

Table 5.2: MAEs ($^{\circ}C$) Method 2

It depicts the mean absolute errors in Celsius degrees obtained using the second method for the same combinations of hidden layers and number of neurons as before. The results are just like what was expected. This method gives less accurate results than the first method since less data about the external factors that affect the PV cells were fed into the NN. Moreover, increasing the number of layers and/or neurons will rapidly improve the accuracy of the model's predictions in the beginning and tend to fade past a point.

For the **third method** the results were surprising. Since this method uses temperature estimations as inputs, we expected better results in accuracy after feeding them into a NN rather than the two previous methods. However that was not what we observed. Table 5.3 depicts the mean absolute error in Celsius degrees using the third method for the same combinations of hidden layers and number of neurons.

Neurons \ HL	2	3	4	5
5	1.351	1.396	1.210	1.316
10	1.426	1.085	1.005	1.063
15	1.276	0.958	0.860	0.797
20	1.316	0.937	0.747	0.677
30	1.262	0.866	0.644	0.654
40	1.275	0.786	0.654	0.430

Table 5.3: MAEs ($^{\circ}C$) Method 3

This table makes it clear that using more hidden layers and number of neurons will improve the accuracy, however, just like every other previous method, improvements tend to shrink as the layers/neurons are increased

more and more.

	Method 1	Method 2	Method 3
MAE(5HL, 40N)	0.229	0.424	0.430

Table 5.4: comparison of MAEs ($°C$)

In Table 5.4 we collect the smallest MAE produced by each method. All methods gave the smallest MAE for the same number of hidden layers and neurons, which were the biggest we tested. Method 2,3 produce an error which is twice as large as the error of the Method 1. The main reason for this behaviour is that Method 1 uses as input quantities related to the actual performance of the solar cell such as I_{sc}, V_{oc}. However these measurements might not be available in realistic situations.
Methods 2,3 use as input parameters which are mostly known a priori, thus representing more actual cases at the expenses of increased error.
Nevertheless, all methods produced errors which are below $0.5°C$ which is about 1% of the solar cells' NOCT.

5.2 Dependency on Learning Rates

Recall that learning rate, is simply the size of step the algorithm takes pointing in a direction. Most of the times small learning rates require more epochs, however even if large learning rates can rapidly decrease the error, too large learning rates often cause more trouble, since they can make the model overshoot the minimum or even completely diverge. In this section, we present some of the conclusions we came into. In Table 5.5 we present the observed errors for various compinations of learning rates and epochs. We use the DNN that was developed for the first method, with 5 hidden layers, 40 neurons for each layer, using the Adamax optimizer, i.e. the one that gave the best results overall.
The table rows represent number of epochs while the columns represent learning rates. The same data are graphically represented in Figure 5.3. Note that in order to make the graphs easier to understood and to stress out the fading improvement past a point, we interpolated the observed data using cubic splines.
Lastly we want to mention that for results shown below, we re-run the DNN for every combination of learning rate and number of epochs. That is why the error might be slightly different from the Table 5.1. Even re-running the

exact same DNN, will in general produce slightly different results, since the initialization of the weights and biases is random in TensorFlow, however they should have the same behavior.

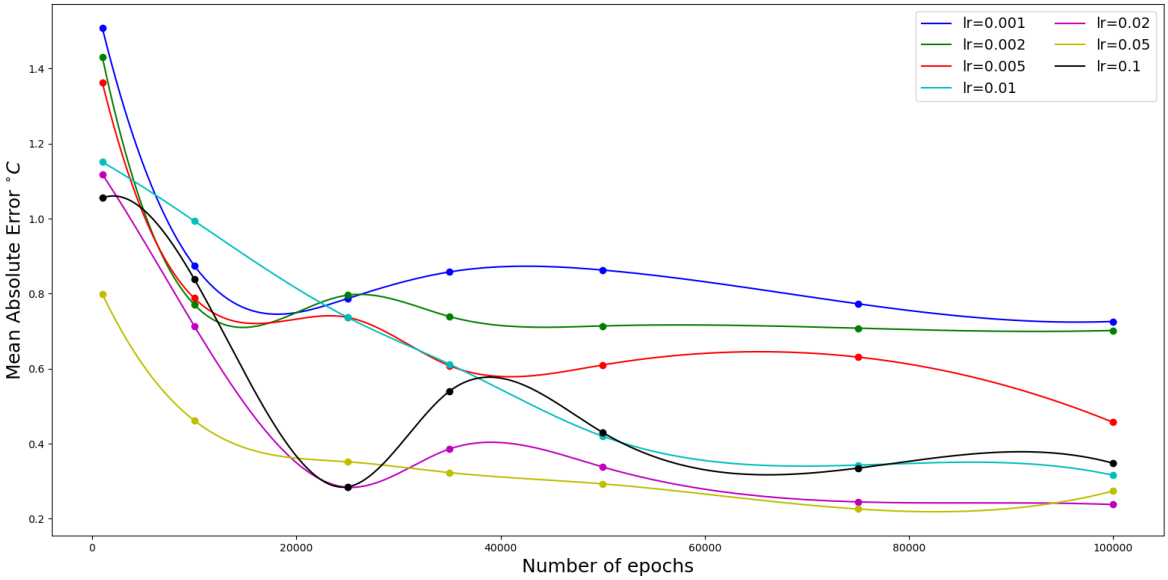


Figure 5.3: Errors for various learning rates and epochs

epochs \ lr	0.001	0.002	0.005	0.01	0.02	0.05	0.1
1000	1.508	1.431	1.363	1.151	1.118	0.799	1.055
10000	0.875	0.771	0.788	0.994	0.714	0.462	0.839
25000	0.787	0.796	0.737	0.737	0.284	0.352	0.285
35000	0.858	0.739	0.608	0.612	0.386	0.323	0.540
50000	0.863	0.714	0.610	0.420	0.338	0.293	0.430
75000	0.773	0.708	0.631	0.343	0.245	0.226	0.335
100000	0.726	0.702	0.457	0.316	0.238	0.274	0.348

Table 5.5: MAEs ($°C$) for various learning rates and epochs

What we can conclude from Table 5.5 is that too small learning rates will reduce the speed the algorithm will converge towards a minimum or even get stuck around a point. On the other hand too large learning rates will quickly reduce the error in the early steps, but they produce a much more unstable learning process, with big oscillations or even converge to sub-optimal solution. We can also observe that no matter the value of the learning rate, the reduction of the cost tends to fade in every case, meaning that our model is unable to learn anymore or fast enough to consider using even more epochs

efficient and productive. An other interesting observation to point out, is that even when the learning rate is "just fine" and the error reduction is more likely with more epochs, it is yet not guaranteed that the algorithm will stop a the best possible solution. This is where the factor "luck" comes into play as we mentioned in 5.1.

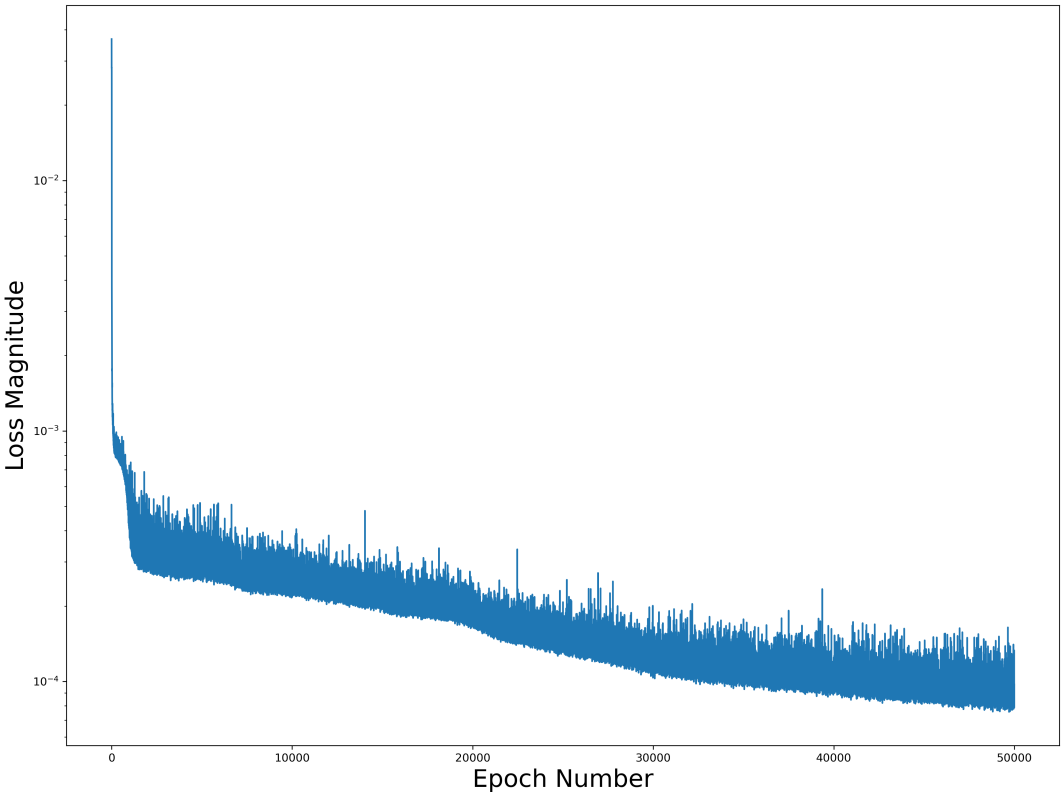


Figure 5.4: A loss function

We can observe this phenomenon in Figure5.4 for example. It is the MSE loss function of a DNN we used for our study problem. The DNN had 5 hidden layers, 10 neurons and a learning rate of 0.02 for 50.000 epochs. The function highly fluctuates, which is expected, however as it is shown here, while the epochs progress, the loss function fluctuates more and more, making it a possible scenario, when the algorithm stop, we could either land at a high point or low point. That is why, stopping at a good point is sometimes a matter of luck.

An other observation we can make, is that there was a rapid reduction of the cost in the beginning but the function tends to decrease at a slower rate as the error is decreased. This behavior of the loss function is what a typical loss function will look like. Fluctuations may be a lot higher or lot lower depending on the step size i.e the learning rate.

As a matter of fact we can see in Figure 5.5 the loss function of 2 DNNs, differing only in the learning rate. Both have 5 hidden layers with 40 neurons each, for 25.000 epochs, however the left cost function corresponds to the DNN with a learning rate of 0.001 and the right cost function corresponds to the DNN with a learning rate of 0.05.

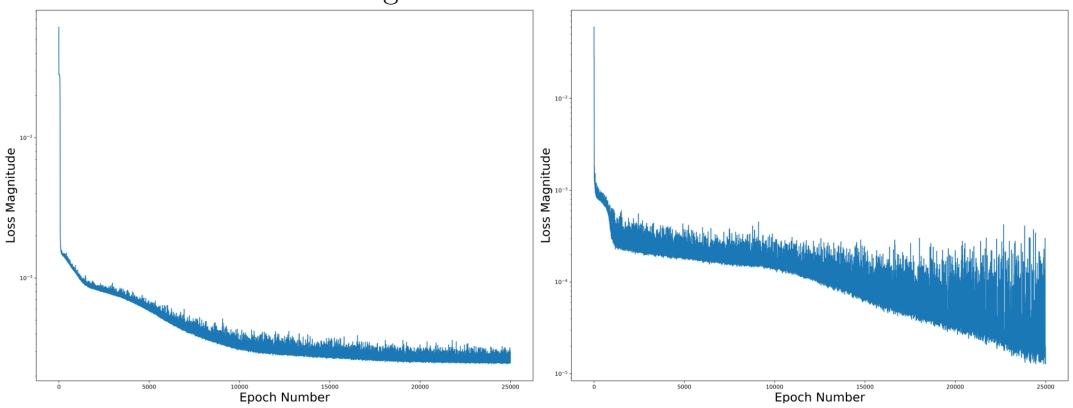


Figure 5.5: Loss function with learning rates 0.001(left) and 0.05(right)

The fluctuations on the right picture are so immense that can change the accuracy by more than on order of magnitude.

5.3 Sampling Percentage

Every result obtained so far, has come from a 70% train - 30% test split, which is the most commonly used in practice. In this section we proceed to examine the effect, the percentage of examples used in the training set, has on our model's accuracy.
The motivation for such a test is time complexity. Fewer training examples result in less time needed to compile the model. If we are satisfied with the accuracy, there is no need wait any longer and keep on training the network. As a bonus to that, if the results are good enough on the test set, training with fewer examples normally means better generalization of the model. Accurate results on many previously unseen data translates to lower chance of overfitting.
Of course, training with fewer examples can not always guarantee that the model will find all the features and learn as good as we would like.

We try out five different splitting ratios, starting from 30% train-70% test sets, exchanging 10% at every test, all the way up to 70% train-30% test

set split. The two DNNs presented here, have 2 hidden and 5 hidden layers respectively, 40 neurons at each hidden layer, trained for 50.000 epochs with a learning rate of 0.02, using the Adamax optimization algorithm for both, using the first method.
For the smaller network with 2 hidden layers the errors for both the whole dataset and test set solely are shown in Table 5.6, while the results from the network with 5 hidden layers are shown in Table 5.7. The percentages on the side of the table represent the percentage of the dataset used for training, while the heading shows the mean absolute errors obtained for the whole data and for the test set only.

	MAE	MAE test set
30%	0.673	0.686
40%	0.713	0.733
50%	0.648	0.671
60%	0.633	0.672
70%	0.567	0.679

Table 5.6: MAEs 2 hidden layers

	MAE	MAE test set
30%	0.656	0.827
40%	0.486	0.702
50%	0.380	0.705
60%	0.316	0.675
70%	0.229	0.599

Table 5.7: MAEs 5 hidden layers

From Table 5.6 we see that the MAE on the test set remains unchanged after the 50% mark for this DNN.
However the situation is different for the DNN with a bigger number of layers where the increased percentage of the training set decreases the MAE on the test set.

While this is beyond the scope of this thesis, we want to close this chapter by stressing out a few things about the Gradient Descent variations we discussed in 3.4 when implemented on different problems. The DNN used was the one 5 hidden layers, 40 neurons for Method 1 build with TensorFlow. While Adamax (3.4.4) seems to produce the best results in terms of accuracy overall in this specific problem, it was not the one giving the fastest results all the time, when we used a given desired error threshold. For example, for a mean squared error threshold of 5×10^{-4}, RMSProp (3.4.5) managed to achieve that, in 2 minutes and 27 seconds, Adamax manged to achieve that in 59 seconds while Adam (3.4.3) reached that threshold in just 36 seconds.

	Adam	Adamax	RMSProp
time	0m 36s	0m 59s	2m 27s

Table 5.8: Execution times of code for threshold MSE of 5×10^{-4}

That time difference was unexpected, since a MSE of 5×10^{-4} is not a very demanding error threshold to reach in that specific problem and from past experience RMSProp was expected to run faster than Adamax. However this was far from the actual outcome. We mention that, just stress out that there is no more accurate algorithm for everything nor a faster algorithm altogether. Every real world problem is different and most of times, one can not predict the behavior the algorithms will have. In practice, experience and trials are what will determine the algorithms used in order to train a network efficiently.

Chapter 6

Conclusions

In this thesis we have developed machine learning techniques for estimating the operating temperature of solar cells. We have used DNN with various number of layers and neurons. Three different approaches were used in terms of the kind of input data for these DNN.

All three methods produced excellent results in predicting solar cell's operating temperature. The first method that took into account all ambient factors and electrical characteristics products the best results out the three. However, errors were small for all methods that have practically no difference.

For the specific problem we work with, errors less than half a Celsius degree, pose no problem. Actually, even bigger errors than that, would be absolutely acceptable and someone could use those predictions. For problems where such errors are tolerable, there is no need to let the network train for more than 50 minutes, time that was needed in order to complete all 50000 epochs. For example, a mean squared error of 5×10^{-4} we mentioned in Table 5.8 translates to about 1 Celsius degree difference in our problem. However, to achieve that error we only needed to wait for less than 1 minute which is extremely lower than the time it took us for 50000 epochs.

Nonetheless, since ANNs are offered for practically an infinite range of problems, one might find smaller errors much more useful. For that, larger and more accurate set of data could probably help improve the accuracy. It is reminded that part of our ambient data such as wind velocity and air temperature, were simply estimations coming from interpolation or typical meteorological years and not the actual measurements.

Bigger networks in terms of width and/or depth could also be a better solution for more precise results, especially when data are more complex and the problem has a higher dimension. Of course this means bigger computational complexity. More epochs seemed to help improving the accuracy, even if the improvement tends to slow down as the epochs progress and even if

more epochs means more time training. However, we need to stress out one more time that this is not guaranteed. For both those cases, a more powerful computer, with a dedicated cuda-core GPU and/or a better CPU could drastically reduce the time needed to finish the training even with more complex network structures, since TensorFlow is designed to run extremely fast and efficient on cuda-cores.

Bibliography

[1] Sebastian Ruder. An overview of gradient descent optimization algorithms. Insight Centre for Data Analytics, NUI Galway Aylien Ltd., Dublin

[2] G. Cybenko, Approximation by superpositions of a sigmoidal function,Math. Control Signals Systems,2 (1989), 303–314

[3] Kurt Hornik, "Approximation capabilities of multilayer feedforward networks". Neural Networks, Vol. 4 (1991), 251–257

[4] Activation Functions: Comparison of Trends in Practice and Research for Deep Learning Chigozie Enyinna Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall

[5] Leonardo Ferreira Guilhoto, An Overview Of Artificial Neural Networks for Mathematicians (2018)

[6] Molecular Cell Biology, 4th edition Harvey Lodish, Arnold Berk, S Lawrence Zipursky, Paul Matsudaira, David Baltimore, and James Darnell. New York: W. H. Freeman; 2000.

[7] Jones EY. 2015 Understanding cell signalling systems: paving the way for new therapies. Phil. Trans. R. Soc. A 373: 20130155. `http://dx.doi.org/10.1098/rsta.2013.0155`

[8] Wang SC. (2003) Artificial Neural Network. In: Interdisciplinary Computing in Java Programming. The Springer International Series in Engineering and Computer Science, vol 743. Springer, Boston, MA. `https://doi.org/10.1007/978-1-4615-0377-4_5`

[9] `https://www.quora.com/How-does-one-show-that-the-expected-value-of-a-mini-batch-in-SGD-is-equal-to-the-true-empirical-gradient/answer/Conner-Davis-2`

[10] Szandala, Tomasz. "Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks." ArXiv abs/2010.09458 (2020)

[11] https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[12] https://cnl.salk.edu/~schraudo/teach/NNcourse/brain.html

[13] Zeidler, E. (1995). Applied Functional Analysis: Main Principles and Their Applications.

[14] https://www.linkedin.com/pulse/choosing-number-hidden-layers-neurons-neural-networks-sachdev

[15] https://mcneela.github.io/machine_learning/2017/03/21/Universal-Approximation-Theorem.html

[16] https://www.tensorflow.org/

[17] https://www.python.org/

[18] S. Albawi, T. A. Mohammed and S. Al-Zawi, "Understanding of a convolutional neural network," 2017 International Conference on Engineering and Technology (ICET), 2017, pp. 1-6, doi: 10.1109/ICEngTechnol.2017.8308186.

[19] Tammy Jiang, Jaimie L. Gradus, Anthony J. Rosellini, Supervised Machine Learning: A Brief Primer, Behavior Therapy, Volume 51, Issue 5, 2020, Pages 675-687, ISSN 0005-7894, https://www.sciencedirect.com/science/article/pii/S0005789420300678

[20] Caruana, R. et al. "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping." NIPS (2000).

[21] Th. Katsaounis, K. Kotsovos, I. Gereige, A. Basaheeh, M. Abdullah, A. Khayat, E. Al-Habshi, A. Al-Saggaf, A.E. Tzavaras, Performance assessment of bifacial c-Si PV modules through device simulations and outdoor measurements, Renewable Energy, Volume 143, 2019, Pages 1285-1298, ISSN 0960-1481, https://www.sciencedirect.com/science/article/pii/S0960148119307177