UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

Analysis of Neural Networks. Applications to Interpretability and Uncertainty

Autor: Gabriel Marín Sánchez

Directors: **Dr. Antoni Benseny Ardiaca**
Departament de Matemàtiques
i Informàtica
Alberto Rubio Muñoz
SCRM - Lidl International Hub

Barcelona, 21 de juny de 2020

Acknowledgements

M’agradaria donar les gràcies a totes aquelles persones sense les quals ni aquest treball ni els meus estudis haguessin estat possibles.

Als meus pares per atorgar-me el privilegi d’estudiar un grau universitari que, per desgràcia, no tothom pot tenir.

A la Laia, per estar sempre allà i ensenyar-me tantes coses sobre balenes. De ben segur que sense tu, ni jo ni aquest treball seríem el mateix.

A en Martí, per acompanyar-me en aquesta aventura des del primer dia i per ajudar-me sempre. També per animar-me a començar els estudis d’Informàtica, sense els quals aquest treball no hagués estat possible.

A Elias y a Pablo, por darme la energía necesaria para completar mi último curso.

To Alex, for all your support and for always believing in me.

Als companys i companyes de SCRM, i en particular al Chapter de Data Science, per acompanyar-me durant aquest últim any i ensenyar-me tantes coses.

Gràcies, Antoni, per acompanyar-me durant aquest treball i per tots els consells.

Gracias, Alberto, por todo el tiempo que me has dedicado y por introducirme a este maravilloso mundo de las redes neuronales y la inteligencia artificial. Es y ha sido un placer escucharte y aprender de ti.

Abstract

From image creation and pattern recognition to speech and text processing, the outstanding performance of neural networks in a wide variety of fields has made them a popular tool among researchers. However, the fact that we do not fully understand why their performance is so successful or how they operate converts this technology into a black-box model based on trial and error. In this work, we attempt to give deep neural networks a mathematical representation and present different examples and applications that bring light to the understanding of neural networks' behaviour and usage.

2020 Mathematics Subject Classification. 68T01, 68T07

Contents

Introduction **iii**

1 Feed-Forward Neural Networks **1**
1.1 Mathematical Description . . . 1
1.2 Learning Process . . . 2
1.2.1 Loss Function . . . 2
1.2.2 Gradient Descent . . . 4
1.2.3 Training Step Algorithm . . . 7
1.3 Activation Function . . . 8

2 Supervised Learning **11**
2.1 Case Study: Two Concentric Circles . . . 11
2.2 1D Segment . . . 14
2.2.1 Injective activation function . . . 15
2.2.2 Non-injective activation function . . . 16

3 Unsupervised Learning **19**
3.1 Autoencoder . . . 19
3.2 Case Study: Time Series . . . 19

4 Interpretability **27**
4.1 Convolutional Neural Networks . . . 28
4.2 Interpretability Algorithms . . . 29
4.2.1 Backpropagation: Integrated Gradients . . . 30
4.2.2 Input Modification: Occlusion . . . 31
4.2.3 Attention: Class Activation Mapping . . . 31
4.3 Algorithm Comparison . . . 33

5 Uncertainty **35**
5.1 NormalLoss . . . 35
5.2 Regression Data . . . 36

5.3 Regressor Neural Network with Uncertainty 37
5.4 Uncertainty Neural Network for Black-Box Regressor 38
5.5 Conclusions 41

Conclusions **43**

Bibliography **45**

Introduction

Even though the term *Deep Learning* was first coined at the beginning of 2006 [1], its concept has been around since the first half of the last century. With the goal of imitating the functioning of the human brain, the first artificial neural models were the McCulloch-Pitts neuron [2] and the perceptron [3], introduced in 1943 and 1958, respectively. Due to the lack of computational power and mathematical theory that explained their performance, the usage of such models was not extended.

It was not until the 1990s, with the introduction of the *universal approximation theorems* [4, 5], that neural networks lived a second significant wave. Those theorems stated and proved that feed-forward neural networks with a hidden layer can approximate any continuous function. The only concern, though, is that for these theorems to hold, the number of neurons in the hidden layer must need to be arbitrarily large.

With the proven success of neural networks in tasks such as computer vision, text processing and data classification, it is clear that this field is revolutionising data-driven application. In addition, scientists are starting to apply this technology to areas like Physics and Applied Mathematics. A clear example is [6], where researchers showed that a deep neural network can learn to solve the three-body problem.

The counterpart, though, is that with many new papers released every day, scientists tend to focus on the computational part and avoid the mathematical treatment behind the development. This causes that users end up employing neural networks just because they know they will perform well, but not understand why.

This work aims to analyse the behaviour of neural networks from a mathematical perspective. In chapter 1, we start by presenting a mathematical description of a feed-forward neural network as well as the learning process.

We then explore, in chapters 2 and 3, the supervised and unsupervised types of learnings. We present examples of classification and representation problems.

In chapter 4, we address the interpretability problem in deep neural networks and describe three existing algorithms. Also, we test these algorithms in an image to evaluate their functionalities. In particular, we will attempt to find the parts of the picture that the neural network considers to be more important when classifying it.

Finally, in chapter 5, we present a method to add uncertainty to a black-box regressor model with the use of a neural network, as well as predicting probability distributions.

The code developed in this work has been written in `Python` language and can be found on my `github` page[2], divided in different `Jupyter Notebooks`. The neural networks have been implemented using the `PyTorch`[3] library. Also, to create the plots, we used the `matplotlib`[4] package.

[2]https://github.com/gms12/TFG_Matematiques_Primavera_2020
[3]https://github.com/pytorch/pytorch
[4]https://github.com/matplotlib/matplotlib

Chapter 1

Feed-Forward Neural Networks

1.1 Mathematical Description

Let us consider a function f'. The goal of a feed-forward neural network is to find a function f that approximates f' [7, Ch. 6]. We will proceed to define the necessary concepts to give feed-forward neural networks a mathematical description.

Let n and m be positive integers. Consider the parameters $w \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^m$. We define the *linear function* $\phi : \mathbb{R}^n \to \mathbb{R}^m$ as

$$\phi(x) = w^T x + b, \tag{1.1}$$

where $x \in \mathbb{R}^n$. The parameters w and b will be called weight and bias, respectively.

From now on, n and m will denote the dimensions of the input and output space, i.e. $\mathbb{R}^n$ will be the input space and $\mathbb{R}^m$ the output space. The input and output spaces are also called *feature* or *target* spaces.

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R} \to \mathbb{R}$ be two functions. We define the *component-wise composition* of f and g as

$$(g \bullet f)(x) = ((g \circ f_1)(x), \ldots, (g \circ f_m)(x)), \tag{1.2}$$

where $x \in \mathbb{R}^n$ and f_i corresponds to the i-th component of f, for $i = 1, \ldots, m$.

Definition 1.1. Let L be a positive integer. Also, let $\{k_l\}_{l=0,\ldots,L}$ be a set of positive integers such that $k_0 = n$ and $k_L = m$. Then, a *feed-forward neural network* is a continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$ described as

$$f = h_L \bullet \phi_L \circ \cdots \circ h_1 \bullet \phi_1, \tag{1.3}$$

where $\phi_l : \mathbb{R}^{k_{l-1}} \to \mathbb{R}^{k_l}$ is a liniar function obeying Equation (1.1) and $h_l : \mathbb{R} \to \mathbb{R}$ is a continuous and non-linear function known as activation function, for $l = 1, \ldots, L$. In addition, $L + 1$ denotes the number of layers of the network.

As a remark, the function f depends on a set of parameters θ, which correspond to the weights and biases of each linear function ϕ_l, for $l = 1, \ldots, L$. Then, we will write the feed-forward neural network as f_θ.

As seen in Definition 1.1, a neural network can be divided into layers. There are three types of layers: Input, Output and Hidden.

- The *Input Layer* is the first layer of the network. It takes the input data and does not perform any operation.
- The *Output Layer* is the last layer of the network. It returns the output data and performs the function $h_L \bullet \phi_L$.
- The *Hidden Layer* is any layer between the Input and Output Layers. It performs the function $h_l \bullet \phi_l$ for $l = 1, ..., L-1$, when $L > 1$. The number of hidden layers of a feed-forward neural network is $L-1$.

In addition, each layer can be divided into neurons. The number of neurons per layer corresponds to the dimension of the space where this layer lies in. For instance, if we consider the l-th layer, the number of neurons will be k_l. Also, the j-th neuron of this layer will perform the operation corresponding to the j-th component of $h_l \bullet \phi_l$, which is $(h_l \bullet \phi_l)_j = h_l \circ (\phi_l)_j$.

In the Fig. (1.1) we have a graphical representation of a feed-forward neural network with $L = 2$. The weight of the edges connecting the neurons correspond to the weights of the linear functions, while the bias terms correspond to the weights of the edges connecting imaginary neurons with an output value of 1 and the neurons of the layer.

1.2 Learning Process

Now that we have defined the operation performed by a neural network, we can proceed with the learning process. The goal of this process, also called training, is to find an optimal set of parameters $\hat{\theta}$ for which the function $f_{\hat{\theta}}$ better approximates the original function f'.

1.2.1 Loss Function

To compute the error of the neural network prediction, a *loss* or *cost function* J_θ is used. This function maps the prediction and real value into $\mathbb{R}$ and describes how well is the network performing. The different loss functions can be classified into two families, depending on the problem: regression and classification [7, 8].

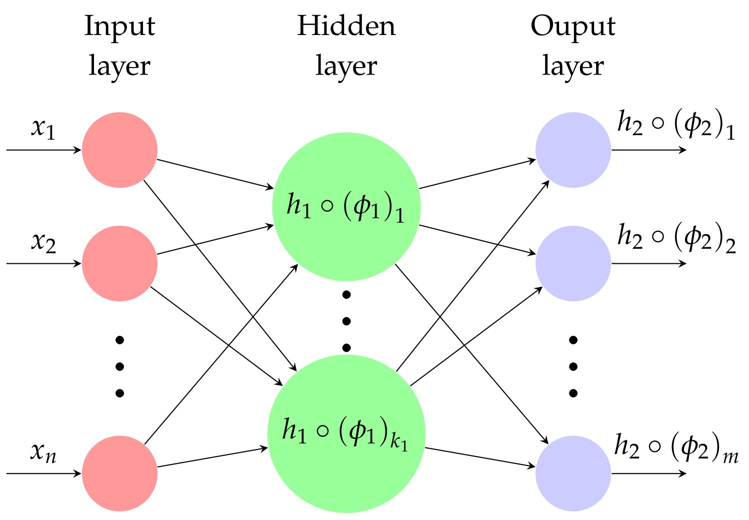


Figure 1.1: Feed-forward neural network with three layers. The parameters n, k_1 and m denote the input, hidden and output dimensions, respectively.

Regression

In regression problems, the target y' can take any value in the output space $\mathbb{R}^m$. In these cases, a common loss function used is the squared error loss. Given an input value $x \in \mathbb{R}^n$ and a target value $y' \in \mathbb{R}^m$, the squared error loss function is

$$J_\theta^{SE}(x, y') = \|y' - f_\theta(x)\|^2 = \|y' - y\|^2, \tag{1.4}$$

where y is the prediction of the neural network.

For a training data with N samples, we can consider the mean squared error loss function (MSELoss),

$$J_\theta^{MSE}(x, y') = \frac{1}{N} \sum_{i=1}^{N} J_\theta^{SE}(x_i, y_i'), \tag{1.5}$$

with $x = \{x_i\}_{i=1,\dots,N}$ and $y' = \{y_i'\}_{i=1,\dots,N}$.

Classification

In classification problems, the target can only take specific values, called classes. In these cases, the model returns a value between 0 and 1 for each class. These values correspond to the probability of the input of being from that class. A common loss function used for classification is the log loss, also called cross-entropy [9]. For an input value $x \in \mathbb{R}^n$, a target $y' \in \mathbb{R}^m$ and a set of classes C, this loss function is

$$J_\theta^{CE}(x,y') = -\sum_{c\in C} y'_c \log(f_{\theta,c}(x)), \tag{1.6}$$

where $f_{\theta,c}(x)$ corresponds to the probability of x being from the class c. Also, y'_c is a binary indicator with value 1 if x is indeed from the class c, 0 otherwise.
In binary cases there is only one output and the previous equation is reduced to

$$J_\theta^{BCE}(x,y') = -\left(y'\log(f_\theta(x)) + (1-y')\log(1-f_\theta(x))\right). \tag{1.7}$$

For a training data with N samples, we can consider the mean of the cross-entropy loss function,

$$J_\theta^{MCE}(x,y') = \frac{1}{N}\sum_{i=1}^{N} J_\theta^{CE}(x_i,y'_i), \tag{1.8}$$

with $x = \{x_i\}_{i=1,\dots,N}$ and $y' = \{y'_i\}_{i=1,\dots,N}$.

1.2.2 Gradient Descent

To optimise the loss function J_θ with respect to the set of parameters θ, which correspond to the weights and biases, the conventional approach is to use Gradient Descent. This method was first introduced by Louis Augustin Cauchy [10] and consists on updating the parameters by subtracting the partial derivative of the function with respect to each parameter, multiplied by variable called the *learning rate*. For example, the weight component w_{ij}^l of a layer l will be updated in the following way,

$$w_{ij}^l = w_{ij}^l - \eta\frac{\partial J_\theta}{\partial w_{ij}^l}, \tag{1.9}$$

where η is the learning rate. A proof that this method does indeed converge to a local minimum under specific conditions can be found in [11, 12].

The performance of this optimisation method depends vastly on the value of the learning rate, which has to be chosen taking into consideration the context of the problem the neural network is attempting to solve. If this value is too small, it can lead to slow convergence, whereas a value too large may cause divergence or fluctuations around a minimum.

A limitation of the gradient descent is that there is no guarantee that the minimum reached is global, so the training process can get stuck in local minimum or saddle points, especially for highly non-convex loss functions [13].

Many new optimisation algorithms based on the gradient descent method attempt to face these problems. Some examples are the Adagrad and Adam methods, widely used in Deep Learning models [14].

1.2.2.1 Backpropagation

Another issue found when using gradient-based methods is the calculation of the gradients, which can be computationally expensive. To minimise this cost, the concept of *backpropagation* was introduced and developed [15, 16, 17]. The main idea behind it is the chain rule.

This algorithm consists of, once calculated the loss function, computing the gradients of the output layer and use those to compute the rest going backwards.

For convenience, let us introduce some notation. Following Definition 1.1, let $u^{(l)}, z^{(l)} \in \mathbb{R}^{k_l}$ be such that

$$u^{(l)} = \phi_l(z^{(l-1)}) = (w^{(l)})^T z^{(l-1)} + b^{(l)} \tag{1.10}$$

$$z^{(l)} = h_l \bullet \phi_l(z^{(l-1)}), \tag{1.11}$$

for $l = 1, \ldots, L$. It is clear that $z^{(0)} = x$, the input, and $z^{(L)} = y$, the output.

Lemma 1.2. (Backpropagation) Let $l \in \{1, \ldots, L\}$ be the number of the layer and $j \in \{1, \ldots, k_l\}$ be the index of a certain neuron in that layer. Let $A = \{1, \ldots, k_{l+1}\}$ be the set of indexes corresponding to the neurons of the layer $l+1$. Then, the partial derivatives of the loss function with respect to the weights and bias of this layer are

$$\frac{\partial J_\theta}{\partial w_{ij}^{(l)}} = \beta_j^{(l)} z_i^{(l-1)} \quad ; \quad \frac{\partial J_\theta}{\partial b_j^{(l)}} = \beta_j^{(l)}, \tag{1.12}$$

where

$$\beta_j^{(l)} = \begin{cases} \frac{\partial J_\theta}{\partial z_j^{(l)}} \frac{\partial h_l\left(u_j^{(l)}\right)}{\partial u_j^{(l)}}, & \text{if } l = L \\ \left(\sum_{a \in A} w_{ja}^{(l+1)} \beta_a^{(l+1)}\right) \frac{\partial h_l\left(u_j^{(l)}\right)}{\partial u_j^{(l)}}, & \text{otherwise} \end{cases} \tag{1.13}$$

for $i = 1, \ldots, k_{l-1}$.

Proof. Using the chain rule we obtain the following:

$$\frac{\partial J_\theta}{\partial w_{ij}^{(l)}} = \frac{\partial J_\theta}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial u_j^{(l)}} \frac{\partial u_j^{(l)}}{\partial w_{ij}^{(l)}} \quad ; \quad \frac{\partial J_\theta}{\partial b_j^{(l)}} = \frac{\partial J_\theta}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial u_j^{(l)}} \frac{\partial u_j^{(l)}}{\partial b_j^{(l)}} \tag{1.14}$$

Now we need to compute each component. Using the expressions of $u_j^{(l)}$ and $z_j^{(l)}$, from Equations (1.10) and (1.11), we get:

$$\frac{\partial z_j^{(l)}}{\partial u_j^{(l)}} = \frac{\partial h_l\left(u_j^{(l)}\right)}{\partial u_j^{(l)}} \tag{1.15}$$

$$\frac{\partial u_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial}{\partial w_{ij}^{(l)}}\left(\sum_{k=1}^{k_l} w_{kj} z_k^{(l-1)} + b_j^j\right) = z_i^{(l-1)} \tag{1.16}$$

$$\frac{\partial u_j^{(l)}}{\partial b_j^{(l)}} = \frac{\partial}{\partial b_j^{(l)}}\left(\sum_{k=1}^{k_l} w_{kj} z_k^{(l-1)} + b_j^j\right) = 1 \tag{1.17}$$

Only one term is left to compute. When $l = L$, i.e., the neuron is in the output layer, we have $z_j^{(L)} = y_j$, and the derivative can be calculated directly. For $l \neq L$, we can consider the loss function J_θ as a function of the elements of the set $\{u_a^{(l+1)}\}_{a \in A}$. Then, by taking the total derivative, we obtain the following:

$$\begin{aligned}\frac{\partial J_\theta}{\partial z_j^{(l)}} &= \sum_{a \in A} \frac{\partial J_\theta}{\partial u_a^{(l+1)}} \frac{\partial u_a^{(l+1)}}{\partial z_j^{(l)}} = \sum_{a \in A} \frac{\partial J_\theta}{\partial z_a^{(l+1)}} \frac{\partial z_a^{(l+1)}}{\partial u_a^{(l+1)}} \frac{\partial u_a^{(l+1)}}{\partial z_j^{(l)}} \\ &= \sum_{a \in A} \frac{\partial J_\theta}{\partial z_a^{(l+1)}} \frac{\partial h_{l+1}\left(u_a^{(l+1)}\right)}{\partial u_a^{(l+1)}} w_{ja}^{(l+1)}\end{aligned} \tag{1.18}$$

We have a recursive expression, and by defining the parameter $\beta_j^{(l)}$ as in Equation (1.13), we obtain the Equations (1.12). □

As a remark, we are assuming the neural network is fully-connected, i.e., all neurons of layer l are connected with the neurons of the layer $l + 1$. When this is not the case, we can consider the weights of the non-existing edges to have a value of 0.

Also, the values of $\frac{\partial J_\theta}{\partial z_j^{(L)}}$ and $\frac{\partial h_l\left(u_j^{(l)}\right)}{\partial u_j^{(l)}}$ depend on the selection of the loss and activation functions. For example, if we take the squared loss function and the sigmoid activation function, Equations (1.4) and (1.21), we have:

$$J_\theta^{SE}(y', z^{(L)}) = \|y' - z^{(L)}\|^2 \implies \frac{\partial J_\theta^{SE}}{\partial z_j^{(L)}} = 2\left(z_j^{(L)} - y_j'\right) \tag{1.19}$$

$$h_l(x) = \frac{1}{1+e^{-x}} \implies \frac{\partial h_l\left(u_j^{(l)}\right)}{\partial u_j^{(l)}} = h_l(u_j^{(l)})\left(1 - h_l(u_j^{(l)})\right) \tag{1.20}$$

1.2.3 Training Step Algorithm

In Algorithm 1 we can see the pseudo code representing a step of the training process.

Algorithm 1 One iteration of the training process using gradient descent

```
1: function TRAINSTEP($x$, $y'$, $L$, $d$, $b$, $w$, $\eta$)
2:     $z^{(0)} \leftarrow x$
3:     for $l = 1, \dots, L$ do                                          ▷ Feed-forward
4:         $u^{(l)} \leftarrow \phi_l \left( z^{(l-1)} \right)$
5:         $z^{(l)} \leftarrow h_l \bullet \phi_l \left( z^{(l-1)} \right)$
6:     end for
7:     for $l = L, \dots, 1$ do                                          ▷ Backpropagation
8:         for $j = 1, \dots, k_l$ do
9:             $a \leftarrow \frac{\partial h_l \left( u_j^{(l)} \right)}{\partial u_j^{(l)}}$
10:            if $l = L$ then
11:                $\beta_j^{(l)} \leftarrow \frac{\partial J_\theta}{\partial z_j^{(l)}} \times a$
12:            else
13:                $\beta_j^{(l)} \leftarrow \left( \sum_{a \in A} w_{ja}^{(l+1)} \beta_a^{(l+1)} \right) \times a$
14:            end if
15:            for $i = 1, \dots, k_{l-1}$ do
16:                $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \beta_j^{(l)} z_i^{(l-1)}$        ▷ Update weight value
17:            end for
18:            $b_j^{(l)} \leftarrow w_{ij}^{(l)} - \eta \beta_j^{(l)}$                          ▷ Update bias value
19:        end for
20:    end for
21:    return $w, b$
22: end function
```

During the neural network's learning process, this step will be repeated iteratively for a given number of rounds until, ideally, the value of the loss function converges to a minimum.

1.3 Activation Function

The activation functions are a fundamental part of the architecture of a deep neural network, as seen in the Definition (1.1). They are used to control the output of each neuron and also to decide whether a neuron can be fired or not [20].

One of the main factors to take into account when selecting an activation function is non-linearity. Only non-linear functions satisfy the conditions of the universal approximation theorems [4, 5]. Also, if all activation functions of a neural network are linear, then the model becomes the same as a network with only one layer. However, linear activation functions can indeed be used in some applications.

Differentiability and the range of the function are also essential properties. When the function is not differentiable, errors can arise during the backpropagation process. Besides, for finite ranges, optimisation tends to be more stable.

Now, we proceed to present and describe some of the most commonly used activation functions.

Sigmoid

The sigmoid function can be seen in Fig. (1.2a) and is given by:

$$\mathrm{sigmoid}(x) = \frac{1}{1+e^{-x}}. \tag{1.21}$$

Its range lies between 0 and 1, which makes this function very useful for output neurons when the desired prediction is a probability, especially in binary classification problems.

The downside of using the sigmoid as the activation function is that it becomes almost flat on both sides, which implies that, for large and small values, the gradient will be very small. Hence, the gradient descent process can be slowed down.

Hyperbolic Tangent

The Hyperbolic Tangent function can be written as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{1.22}$$

As seen in Fig. (1.2b), it is a shifted version of the sigmoid but centred at the 0 value with a range that lies between -1 and 1. Even though it has the same downside as the sigmoid function, the fact that it has a wider range and is centred at 0, makes its performance superior in hidden neurons.

ReLU

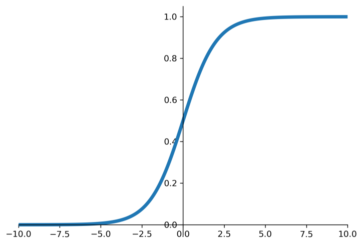
The rectified linear unit (ReLU) function, seen in Fig. (1.2c), was first introduced by Hinton et. al. in 2010 [21] and is given by

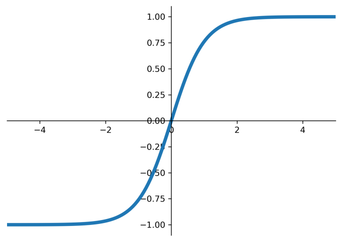
$$\mathrm{ReLU}(x) = \max(0, x). \tag{1.23}$$

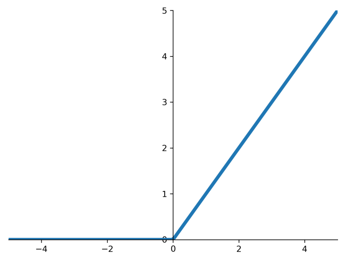
It has been proven that the learning process is faster using this activation function [22]. Besides, it has an improved performance in comparison with the previous functions and can generalise better [20]. For these reasons, the ReLU function is very popular among researchers and used in most of the state-of-the-art models.

An inconvenience of this function is that it is not differentiable at 0. However, it is not very likely to obtain an absolute 0 value during the training process.

Another drawback is that the gradient is null for negative values, which implies that there is no effect in the learning process. The leaky ReLU, seen in Fig. (1.2d), attempts to solve this issue by adding a small slope to the negative part. Nevertheless, the usage of the leaky ReLU is not very common.


(a) Sigmoid function.


(b) Tanh function.


(c) ReLU function.

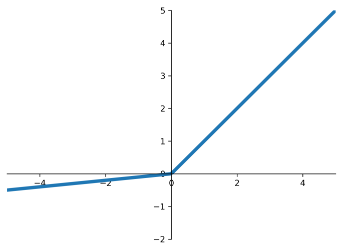

(d) Leaky ReLU function.

Figure 1.2: Examples of activation functions.

In conclusion, all active functions have their pros and cons; the selection depends on the problem one is attempting to solve.

Chapter 2

Supervised Learning

In *supervised learning* the network has a specific target for each input [7, Ch. 5]. Then, the network's goal is to approximate the function that maps the input to the target.

In this chapter, we will explore the performance of neural networks in different classification problems. The network will have to learn how to divide the space given a training dataset with their respective classes, i.e. define the frontiers. In particular, we will study two different cases:

- Binary classification of two concentric circles.
- Binary classification of points in a 1D segment.

We will be using a neural network with one hidden layer and the binary cross-entropy function loss, Equation (1.7), in both problems. The selected optimisation method is the Adam algorithm, which, as we have seen in the previous chapter, is based on the gradient descent technique.

2.1 Case Study: Two Concentric Circles

In Fig. (2.1) we observe the training data, which consists of 200 different points. The network will return a value between 0 and 1, corresponding to the probability of a point of being from the class 1.

The issue now is to decide the number of neurons in the hidden layer. The decision boundary, also called the frontier, is set by a hyperplane in the latent space [18] and formed by the points with an output value of 0.5.

Then, if we consider a hidden layer with 2 neurons, we need a mapping where the transformed points are linearly separable. To achieve this, we can consider the square function as the activation function. In Fig. (2.2a) we see the frontier

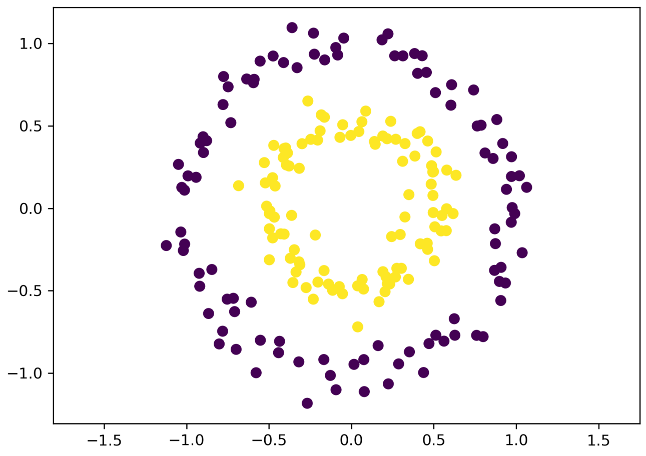
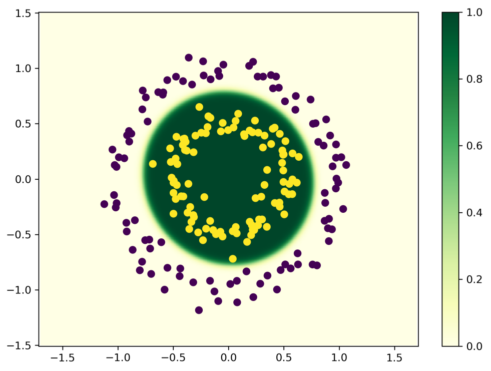
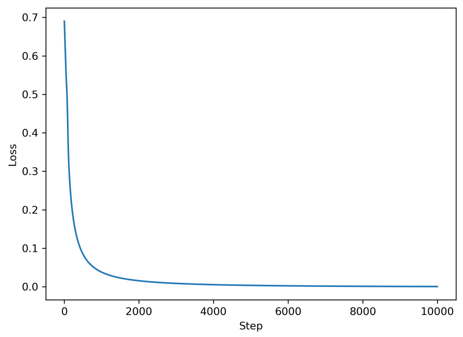


Figure 2.1: Training data for the two concentric circles problem. 200 different points form the two circles, with noise added. Yellow points have label 1 and purple points label 0.

learned by the network after a training process of 10000 steps and a learning rate of 0.01. It is clear that the model has successfully learned to classify the training data and can generalise this classification to the rest of the points of the space. Also, in Fig. (2.2b) we have the evolution of the loss function, which shows that, since it converges to a value of 0, the learning process has been satisfactory.



(a) Learned frontier.



(b) Evolution of the loss function.

Figure 2.2: Result of the learning process of the neural network for a learning rate of 0.01 and 10000 steps with a hidden layer of 2 neurons and the square function as the activation function.

In Fig. (2.3) we have the mapped data in the latent space of the trained network. In this new space, the points are linearly separable, and this is why the model can classify the original space correctly.

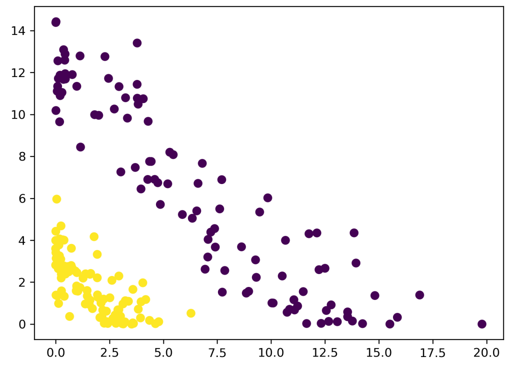
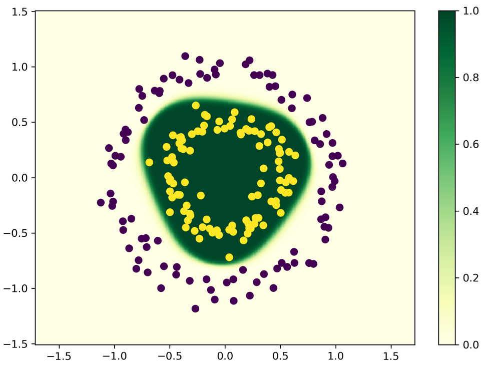


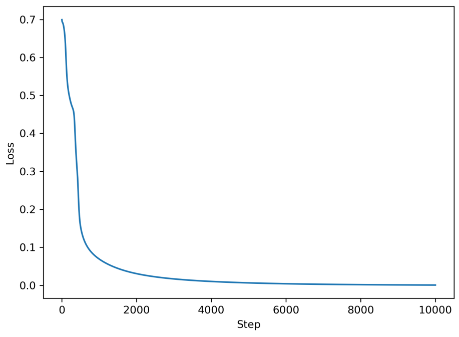
Figure 2.3: Training data mapped into the network's 2D latent space.

In general, though, the square function is not a good option as an active function. The reason behind it is that x^2 is not bounded, and its values increase quickly. Then, large values in the training data would have more weight in the learning process, and the model would not be able to generalise correctly.

Now we will consider tanh as the activation function and 3 neurons in the hidden layer. The same training parameters will be used.



(a) Learned frontier.



(b) Evolution of the loss function.

Figure 2.4: Result of the learning process of the neural network for a learning rate of 0.01 and 10000 steps with a hidden layer of 3 neurons and tanh as the activation function.

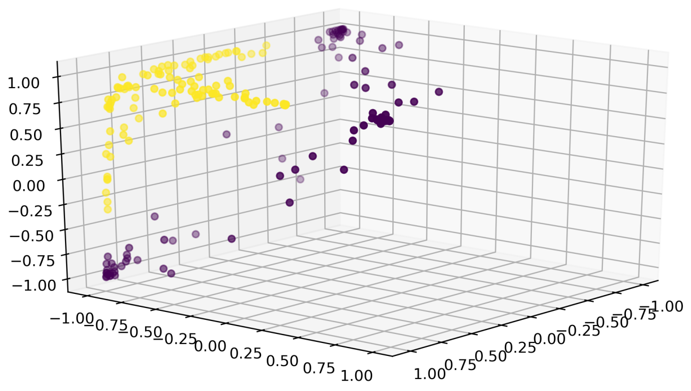


Figure 2.5: Training data mapped into the network's 3D latent space.

In Fig. (2.4a) we see the frontier learned by the network. Also, in Fig. (2.4b) we see the evolution of the loss function, which shows the learning process. The model has successfully learned to divide the space correctly with this training data.

In Fig. (2.5) we have the training data mapped into the 3D latent or hidden space. We see that the network, with the use of an extra dimension, elevates the points. Then, a hyperplane of this space, which is, in this case, a plane, will be able to separate the points of the two different classes. The plane projected to the original space will be the frontier. This process is known as Kernel trick [19].

2.2 1D Segment

Assume we have a set of points that lie in a segment of 1D. Also, assume those points are divided into subsets, distributed one next to each other, and each one is from one of two different classes, respectively. Let $s > 1$ be the number of subsets. In Fig. (2.6) we have the training data.

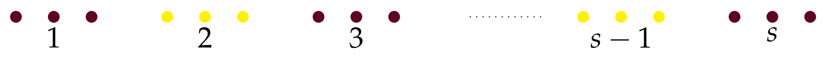
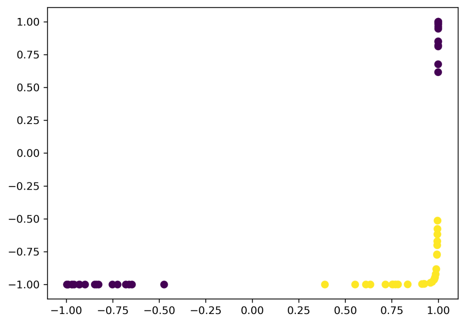


Figure 2.6: Training data for the 1D segment classification problem with s subsets.

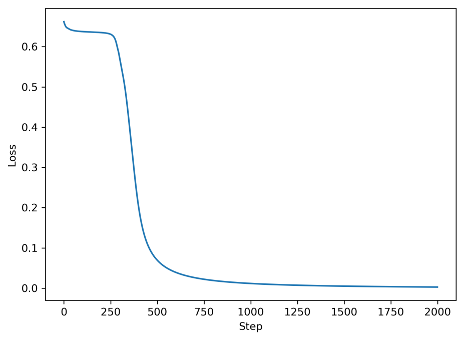
As in the previous case, we need to choose the number of neurons in the hidden layer. We will consider two different situations: injective and non-injective activation functions. In both cases, each subset will contain 30 different points.

2.2.1 Injective activation function

Let us take the function tanh as the injective activation function. First, let us consider the simple case of $s = 3$ and a neural network with two neurons in the hidden layer. As seen in Fig. (2.7b), this network, with a learning rate of 0.01 and 2000 steps, can achieve a loss value of 0, which means that it has learned to classify all the points successfully. Besides, we see that in the latent space in Fig. (2.7a) the subsets are linearly separable and have an "L" shape.



(a) Transformed training data in the latent space.



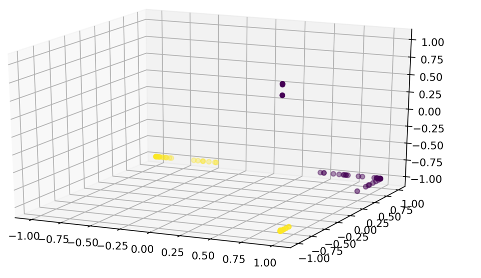
(b) Evolution of the loss function.

Figure 2.7: Result of the learning process of the neural network for a learning rate of 0.01 and 2000 steps with a hidden layer of 2 neurons and tanh as the activation function.

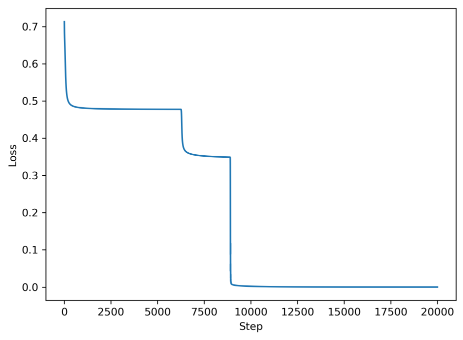
Now, for an $s = 4$, if we try to use the same network as before, it will not be able to learn to classify all points. Due to the injectivity of the transformation, it is not possible to add another subset in the 2D latent space and still make it linearly separable.

Instead, if we use 3 neurons in the hidden layer, the model can solve the problem. With a learning rate of 0.01 and 20000 steps, we see in Fig. (2.8b) that the loss function converges to a value of 0. The evolution, though, is not as smooth as in the previous case. The reason behind it is the fact that this problem is more complex, and the search for the global minimum is more difficult.

In Fig. (2.8a) we have the points mapped into the 3D latent space. We see that, as expected, the network uses the extra dimension to add the new subset. Then, we need to increase the number of neurons in the layer for every subset that we add.



(a) Transformed training data in the latent space.



(b) Evolution of the loss function.

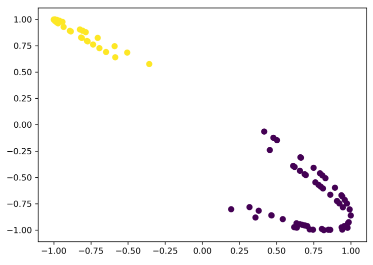
Figure 2.8: Result of the learning process of the neural network for a learning rate of 0.01 and 20000 steps with a hidden layer of 3 neurons and tanh as the activation function.

2.2.2 Non-injective activation function

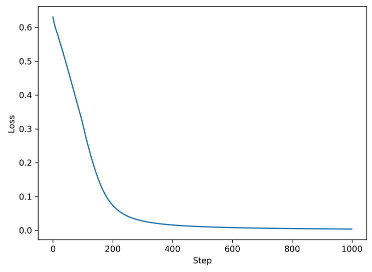
Let us now consider the sine function as the non-injective activation function. If the points from the training data are transformed into the shape of the sine function, i.e., each subset goes to the corresponding maximum or minimum, the system will be linearly separable. In this case, only two neurons in the hidden layer would be sufficient to solve this classification problem for any value of s.

In Fig. (2.9) we have the learning process results of three neural networks for the cases of $s = 3, 5, 7$, respectively. The learning rate used for the three models is 0.01, while the number of steps is $1000, 4000$ and 6000, respectively.

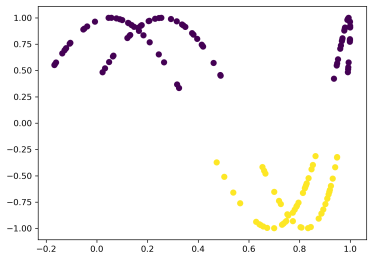
It is clear that, in all cases, the network can classify the subsets with 2 neurons in the hidden space if the activation function is the sine. Even though we could replicate these results for larger values of s, we would need to improve our models to do so, since every time the loss function's global minimum is harder to find.



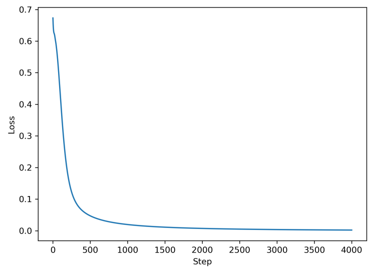
(a) Transformed training data in the 2D latent space for $s = 3$.



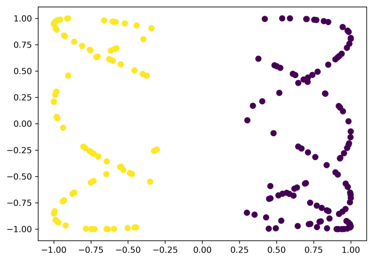
(b) Evolution of the loss function for $s = 3$.



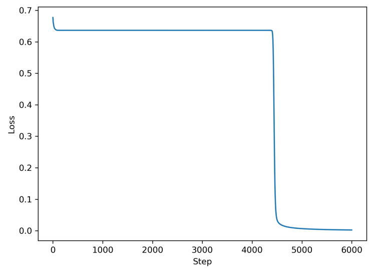
(c) Transformed training data in the 2D latent space for $s = 5$.



(d) Evolution of the loss function for $s = 5$.



(e) Transformed training data in the 2D latent space for $s = 7$.



(f) Evolution of the loss function for $s = 7$.

Figure 2.9: Result of the learning process of the neural network for a learning rate of 0.01 and 20000 steps with a hidden layer of 3 neurons and tanh as the activation function.

Chapter 3

Unsupervised Learning

In *unsupervised learning* the network does not have a specific target for each input [7, Ch. 5]. Then, after experiencing the input data, the network's goal is to learn properties from it.

In this chapter, we will explore the learning of a good simple representation of the data, i.e., a representation that preserves as much information as possible but with some constraints applied. To do so, we will consider a time series dataset and an Autoencoder architecture for the neural network.

3.1 Autoencoder

An Autoencoder is a neural network model that returns an approximation of the input data [7, Ch. 14], so the input and output spaces are the same. Assume the network has only one hidden layer. Then, following Definition 1.1, it consists of two parts: an encoder, with $e(x) = (h_1 \bullet \phi_1)(x)$, and a decoder, with $d(x') = (h_2 \bullet \phi_2)(x')$.

Besides, the dimension of the hidden layer is assumed to be lower than the dimension of the input space. Under this construction, the encoding function projects the input data into a lower dimension space, and the decoder reconstructs this projection. Then, this network acts as an information bottleneck and can capture the most significant properties from the training data.

3.2 Case Study: Time Series

The training dataset for this case study consists of the daily historic market data from 500 random companies, obtained from *Yahoo! finance* with the help of the library `yfinance`[1]. Also, we will divide this data into time series of length 365, corresponding to the number of days in a year. In total, the dataset contains 3071 time series.

The neural network used is an Autoencoder with a single hidden layer, the hyperbolic tangent as the activation function, and the MSELoss function, seen in Equation (1.5). Besides, the selected optimisation method is the Adam algorithm. We will explore the results for different numbers of neurons in the hidden layer: $1, 2, 3, 5, 10$, and 30. To simplify notation, instead of writing k_1, we will use k to denote the hidden layer's dimension. The input and output dimensions are 365. As for the learning rate and the number of training steps, the chosen values are 0.001 and 15000, respectively, for all cases.

In Fig. (3.1) we have the architecture of the Autoencoder neural network.

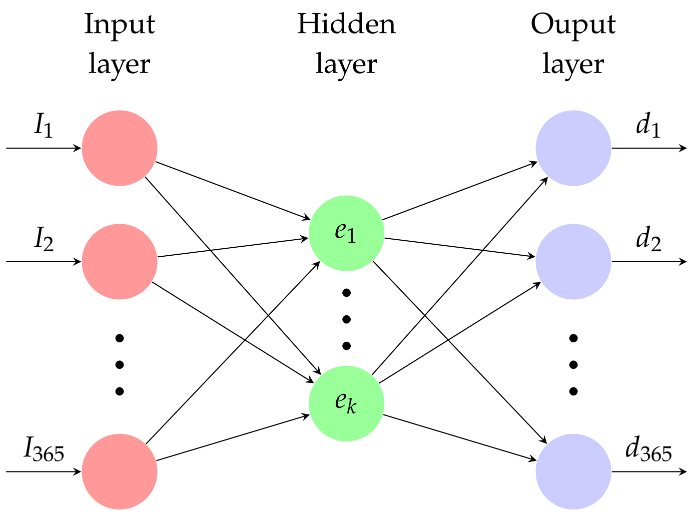


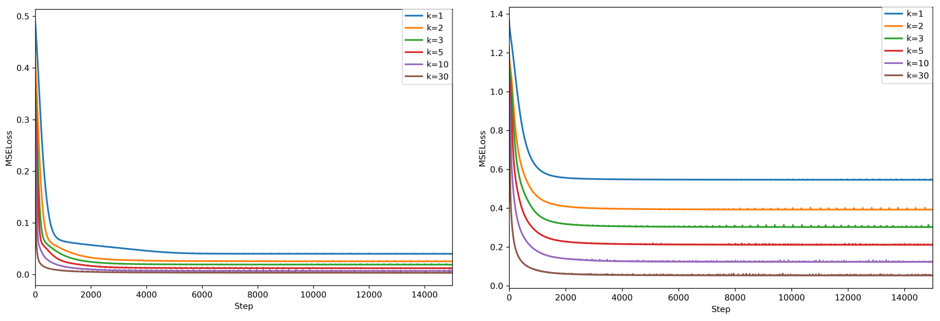
Figure 3.1: Neural Network corresponding to the Autoencoder for the Time Series, where k denotes the dimension of the hidden or latent space.

Before starting with the training process, though, we first need to transform and rescale the data features. The reason behind it is that the range of values in the original data might be too wide. In this case, larger values may have more weight, so the network will not be able to generalise correctly.

To rescale the data, we will consider the standardisation and normalisation transformations. In the first case, the resulting data has 0 mean and a variance of 1. On the other hand, the normalisation transformation takes the minimum value to 0 and the maximum to 1.

[1]https://github.com/ranaroussi/yfinance

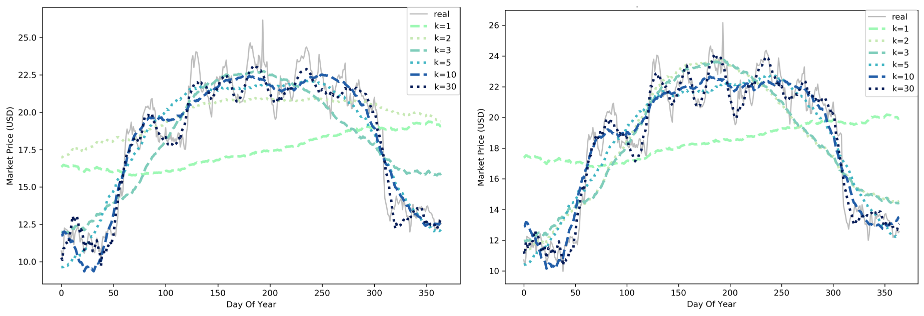
Now we can proceed with the learning process. In Fig. (3.2) we have the evolution of the loss function for all the experiments. We observe that the behaviour is similar for all cases, and they all quickly reach a converging value. Even though the final errors for the normalised data are lower than the standardised ones, this is not a sign that the first networks will perform better than the latter ones. The reason behind it is that since the range of values for the normalised data is narrower than the one for the standardised features, the network's output will be closer to the original data and hence the error will be lower.



(a) Normalised case. (b) Standardised case.

Figure 3.2: Evolution of the loss function for the training process using normalised and standardised data and $1, 2, 3, 5, 10$ and 30 neurons in the hidden space.

To evaluate the results, we will take a random time series. In Fig. (3.3) we have the network's predictions for the normalised and standardised time series and the different numbers of neurons. As expected, the higher the number of neurons in the hidden layer, the better the prediction is. Also, we see that with only 30 neurons, the prediction is very accurate. Then, in Fig. (3.4), we see a comparison



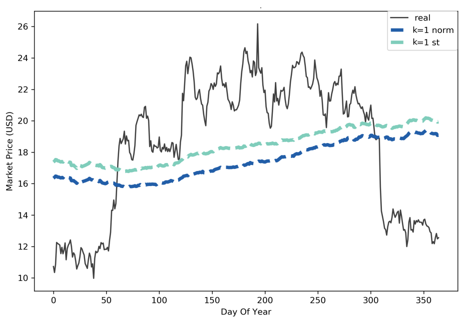
(a) Normalised case. (b) Standardised case.

Figure 3.3: Prediction of a time series for the training process using normalised and standardised data and $1, 2, 3, 5, 10$ and 30 neurons in the hidden space.

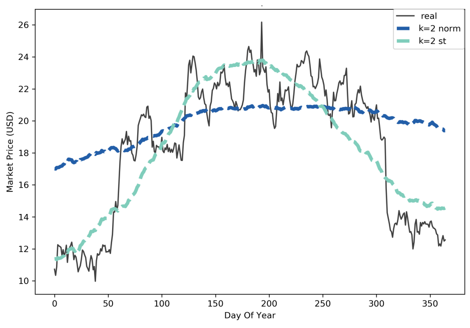
between the predictions of the networks trained with the two types of data and the same number of neurons for the given time series. For $k = 1, 10$, in Fig. (3.4a, 3.4c), we see that the predictions are very close, but for $k = 2, 30$, in Fig. (3.4b, 3.4d), we have some differences.

In the case of two neurons we see that, while the normalised network's prediction is similar to the mean, the standardised one has learned the general trend of the original time series, which is to first grow and then decrease. For thirty neurons, we observe that the neural network trained with standardised data is able to approximate the sudden peaks and lows better.

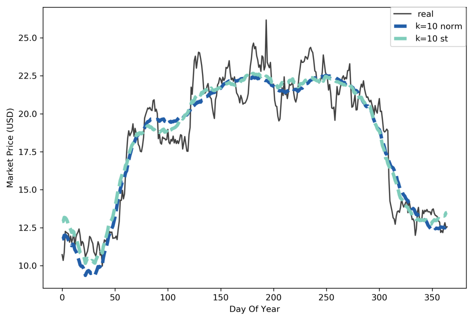
An explanation of this behaviour relies on the differences between the two data transformation methods. First of all, by using the standardised data, the neural network does not need to learn the mean and variance, since they are always the same, 0 and 1. Then, these networks can learn other useful features with fewer neurons. Besides, the normalised transformation does not work well with outliers since all data is located between 0 and 1. For this reason, the networks trained with normalised data are not able to capture the outliers behaviours, sudden peaks and lows, as well as the ones trained with standardised data.



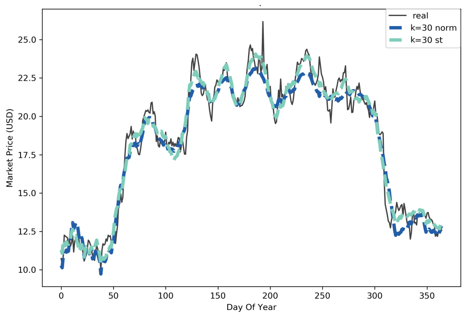
(a) $k = 1$.



(b) $k = 2$.



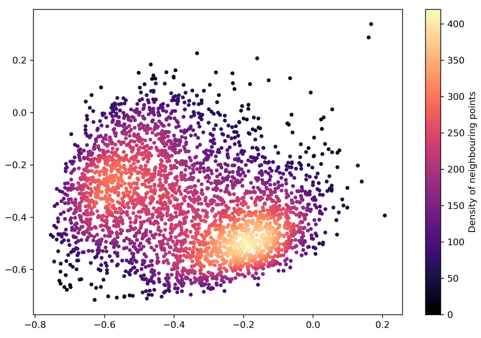
(c) $k = 10$.



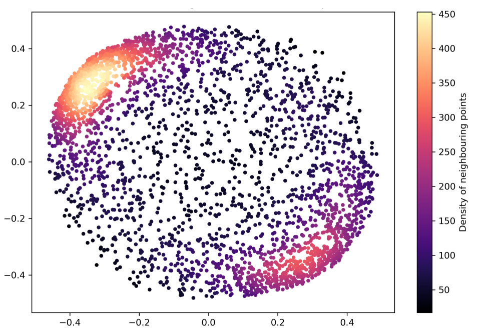
(d) $k = 30$.

Figure 3.4: Normalised vs Standardised predictions for a randomly chosen time series and different numbers of neurons in the hidden layer.

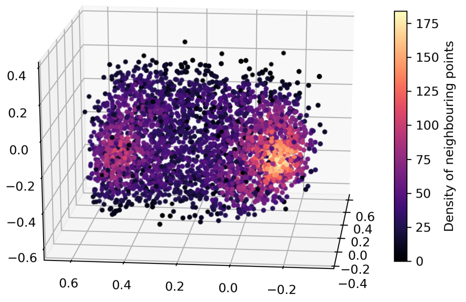
To further explore the results, we will take a look at the latent spaces generated by the hidden layers of each trained neural network. In particular, we will analyse the networks' latent spaces for 2 and 3 neurons in the hidden layer. In Fig. (3.5a, 3.5b) we have the latent spaces generated with normalised and standardised time series, respectively, for $k = 2$.



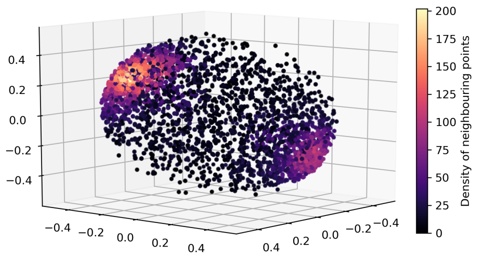
(a) Normalised training data, $k = 2$.



(b) Standardised training data, $k = 2$.



(c) Normalised training data, $k = 3$.



(d) Standardised training data, $k = 3$.

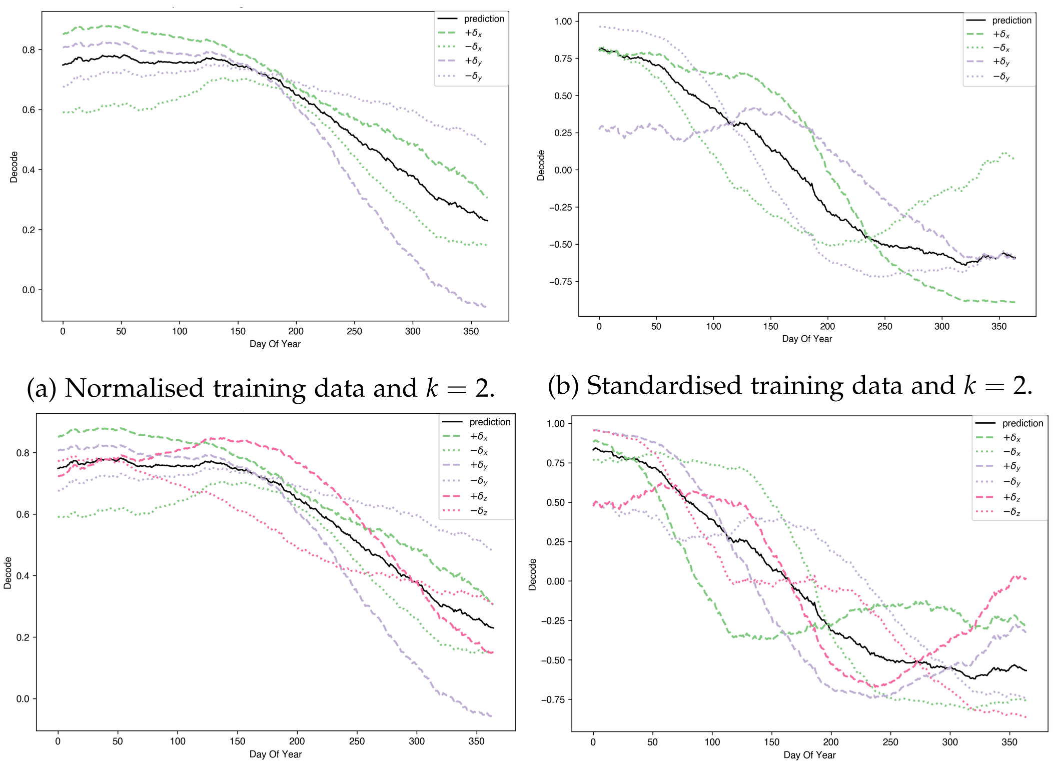
Figure 3.5: Latent spaces generated by the networks trained with normalised and standardised time series and $k = 2, 3$, where the colour of each element represents the number of neighbouring points.

In both cases, we see that there are two zones with larger density of neighbours and, if we inspect the times series that are encoded there, we get that they correspond to the increasing and decreasing time series. Specifically, the highest density zone is formed by increasing time series, which makes sense since most of the time series have that shape. The rest of the points correspond to the transitions between these two cases.

A difference between the two latent spaces is the shape the points form. For the normalised data, the encoded time series seem to concentrate around the two dense zones. In contrast, the standardised data encoded points form an elliptical shape, and the two dense zones are pushed to opposite extremes.

For $k = 3$, in Fig. (3.5c, 3.5d), we observe a similar behaviour as before. Again, the dense zones correspond to increasing and decreasing time series, and the shapes are the same ones but with an extra dimension.

Finally, we will inspect the contribution of each coordinate of the latent space in the network's prediction. To do so, we will consider a random point located in each of the hidden spaces and compare its decoding to the prediction of the same point adding and subtracting a δ value to each component. In Fig. (3.6) we have the results for the four different cases.



(a) Normalised training data and $k = 2$. (b) Standardised training data and $k = 2$.

(c) Normalised training data and $k = 3$. (d) Standardised training data and $k = 3$.

Figure 3.6: Comparison between the decoding of a random point in the latent spaces and the decoding of the same point adding and substracting a value of $\delta = 0.25$ to each component, for $k = 2, 3$ and both types of training data.

For the network with two neurons in the hidden layer and normalised training data, seen in Fig. (3.6a), we have that the x component is more sensitive to the first half of the time series' prediction while the y one is more sensitive to the second half. Also, when adding a third neuron, seen in Fig. (3.6c), the new component z modifies the middle part of the prediction.

From these results, we can tell that, under these conditions, each coordinate attempts to learn how to approximate one specific section of the time series.

On the other hand, for the networks trained with standardised data, we have that the behaviour of each component is more complex than in the previous cases, as seen in Fig. (3.6b, 3.6d), and this is why their predictions tend to be closer to the original time series.

Chapter 4

Interpretability

Deep neural networks are widely used in areas such as image classification, text processing, medical research, product recommendation, and a large etc.. Still, their black-box nature prevents its users to fully understand the reason behind their predictions. As defined by O. Biran and C. Cotton in [23], *interpretability* is the degree to which a human can comprehend the cause of a decision.

The lack of interpretability is one of the reasons why, in areas like medical diagnosis and therapy, the usage of artificial intelligence, and neural networks, in particular, is not more extended. Also, since the European Union approved the General Data Protection Regulation, users have the right to demand explanations to companies on how deep learning algorithms are making decisions about them [25]. Then, understanding why a recommender system is suggesting a specific product to a user, for instance, has become vital for businesses.

Besides, by not clearly understanding how and why the neural networks work, it is very complicated to improve the existing models, since the only approach is trial and error.

In this chapter, we present and describe different existing interpretability algorithms and implement three of them on a deep neural network trained to perform image classification. The reason to focus on the effect on image classification is that the output of this type of problem is more visual and natural to predict. In particular, for each of the interpretability algorithms, we will select a picture and evaluate which parts of it are more important to the model to make its prediction.

Before, though, we introduce the Convolutional Neural Networks, a deep learning model applied to image analysis, among others, that we will use to compare the interpretability algorithms.

4.1 Convolutional Neural Networks

Convolutional Neural Networks, or CNNs, are a type of neural network model for processing data that has a grid-like topology [7, Ch. 9]. Digital images, for instance, can be represented as a two-dimensional grid of pixels. These models are very good at detecting patterns and, in the case of images, edges.

At the end of a CNN, we usually have a fully-connected network that takes the processed data and attempts to solve the original problem, which could be image classification, for example.

Its architecture is based on the convolution operation. The convolution of two given functions f and g, denoted as $f * g$, is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(t)g(t-\tau)d\tau, \tag{4.1}$$

which is commutative. For discretised data, it can be written with a sum instead of an integral. The first and second arguments of the operation are named input and kernel, respectively, while the output is referred to as the feature map.

In machine learning applications, the input and kernel are usually tensors, i.e. multidimensional arrays, where the first one contains the data, and the latter the parameters learned during the training process.

In addition, the convolution is often performed over more than one axis at the same time. So if we consider a two-dimensional input I and a two-dimensional kernel K, we can write the operation as

$$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n), \tag{4.2}$$

where the values of m and n range between the valid values of the indices of the kernel tensor.

Another fundamental concept is the stride, which is the distance between two successive kernel positions. It defines how the values m and n increase and usually has a value of 1 [33]. In Fig. (4.1) we have an example of a convolution operation between a 5×5 input, 3×3 kernel and stride 1.

A typical CNN layer is formed by two blocks. The first one corresponds to the convolution operation, as seen before, plus a nonlinear activation function. The second one is the pooling layer. It reduces the dimensionality of the feature map in order to make the output invariant to small translations of the input [7, Ch. 9][33].

Some of the most used pooling layer operations are the max-pooling [35], which takes the maximum value of the window, and global average-pooling [36],

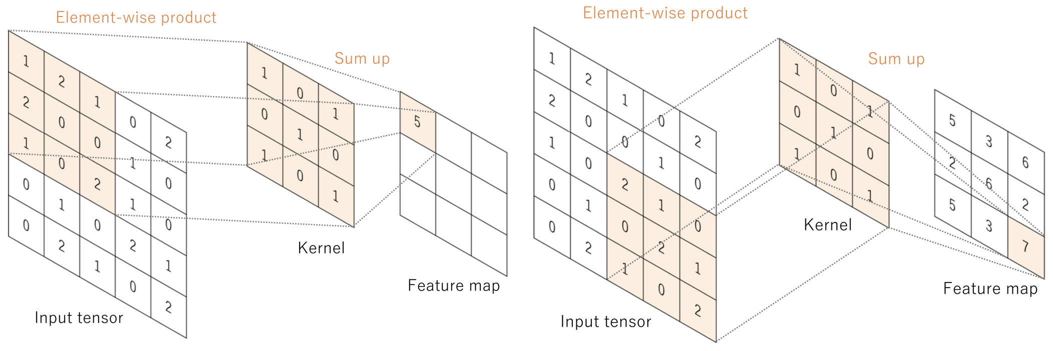


Figure 4.1: Convolution operation between an input of size 5×5 and a kernel of size 3×3 and stride 1, from [33].

which computes the average of the elements in the tensor. In Fig. (4.2) we have an example of a max-pooling operation.

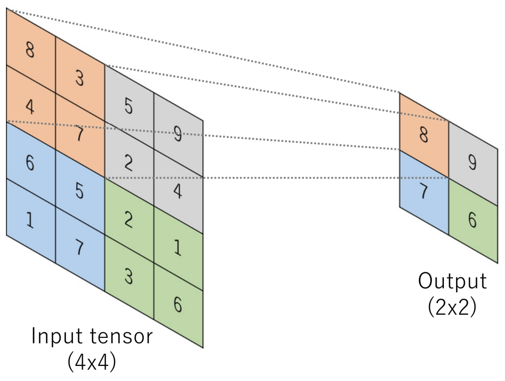


Figure 4.2: Example of max-pooling operation with a filter size of 2×2 and a stride of 2, from [33].

Finally, in Fig. (4.3) we have an example of a CNN model. We can see that multiple kernels can be applied in an input tensor, generating different feature maps in each layer.

4.2 Interpretability Algorithms

When addressing the interpretability of a neural network model, we can evaluate three different contributions to determine an importance value: features and input data, neurons and layers. We will only focus on the first type.

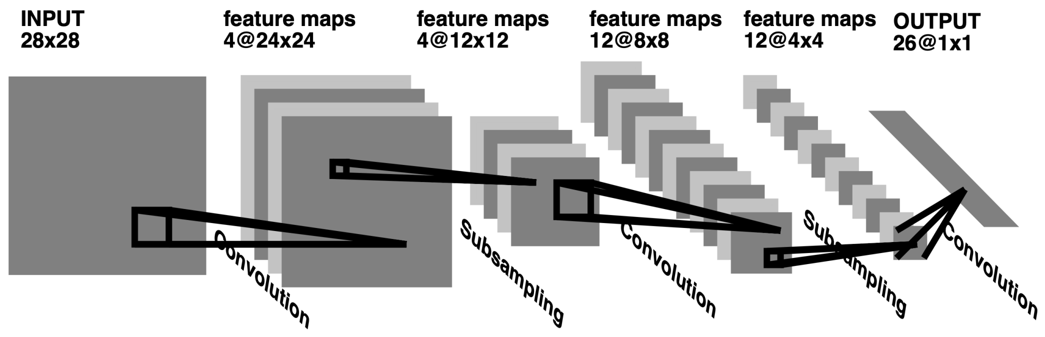


Figure 4.3: Example of Convolutional Neural Network architecture, from [34].

Many of the existing methods require a baseline to compare the predictions and assign importance values to the input elements. The selection of this element depends on the context of the problem and usually represents a lack of features. For example, if the input data consists of images of numbers with a black background, a common baseline would be a black picture.

Since the interpretability methods are hard to evaluate empirically, it is challenging to design such algorithms and compare them. However, we can consider two axioms that can help us: Implementation Invariance and Sensitivity [26].

Given an interpretability algorithm, it holds the *Implementation Invariance* axiom if, for two different networks that return the same output for the same input, it gives the features the same attribution values.

To hold the *Sensitivity* axiom, if the input differs from the baseline and the network returns a different prediction, then the method must give to the distinct feature a non-zero attribution value.

Moreover, a useful method must be able to be applied to an existing neural network model without the need of modifying its architecture.

Finally, we can classify most of the existing algorithms into two categories, those which use the backpropagation process and the ones that modify the input space. We proceed to analyse the two groups and present one algorithm per each. Furthermore, we also explore a different approach called attention.

4.2.1 Backpropagation: Integrated Gradients

This family of methods is formed by the algorithms that make use of the gradients or a similar parameter computed during the backpropagation process. Their goal is to assign a value to each feature corresponding to its contribution to the network's prediction.

Some examples are the Gradient X Input, DeepLIFT [29], GradientSHAP [30] and Integrated Gradients [26].

Due to the nature of the chain rule, the usage of gradients guarantees that the Implementation Invariance axiom holds. However, the Sensitivity axiom does not in many of these implementations [31]. On the other hand, methods such as DeepLIFT, that replaces the gradients with differences, do hold Sensitivity but fail to hold the first axiom; they depend on the network's architecture.

A backpropagation based method that does hold both axioms is Integrated Gradients. This algorithm consists of computing the integral of the gradient of the path between input and the baseline. Given the input x, the baseline x' and the network function $F(x)$, the i^{th} element of the integrated gradient is defined as

$$\left(x_i - x_i'\right) \times \int_0^1 \frac{\partial F\left(x' + \alpha(x - x')\right)}{\partial x_i} d\alpha. \tag{4.3}$$

A drawback of this method, though, is that the calculation of the integrals can be computationally expensive.

4.2.2 Input Modification: Occlusion

The interpretability methods that are based on input or feature modification change the value of each feature of the input data and compare its prediction to the original one so they can assign an importance value to it. The more significant the difference between the prediction, the more relevant this feature is.

The problem with this type of methods is that, since they have to compute the network's prediction for each modification, it can be computationally expensive in comparison with other methods.

Some examples are Feature Permutation [24, Ch. 5.5], Shapley Value [32] and Occlusion [27]. In particular, the Occlusion method is focused on image data and, instead of going pixel by pixel, it divides the image in rectangles of a given size and replaces one of them for a baseline each time. By using a window of pixels rather than modifying one at each step, the computational cost decreases substantially.

4.2.3 Attention: Class Activation Mapping

Attention-based methods take a different approach, and instead of computing new parameters or modifying the input data to grant importance values to features, they focus on analysing the existing weights given to each feature. In particular, we will explore the Class Activation Mapping (CAM), an interpretability algorithm for image classification using CNNs [28].

CAM attempts to detect in which parts of the image the network is looking at in order to make its prediction by making use of the architecture of most of the image classification models.

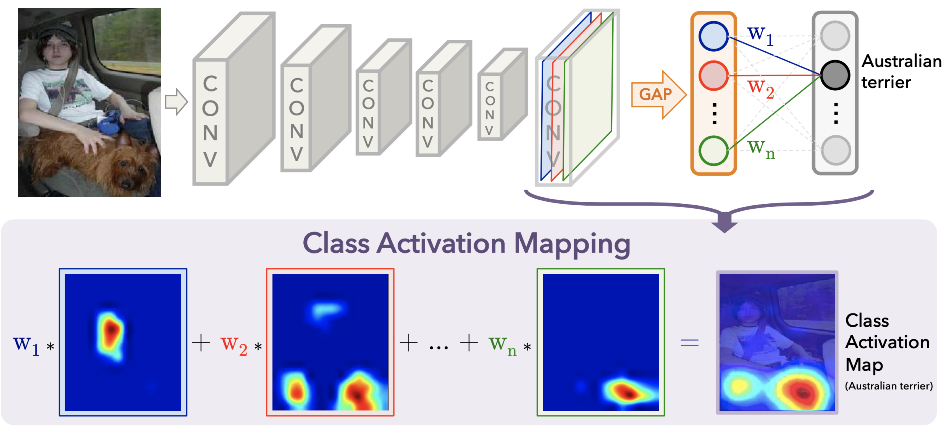


Figure 4.4: Illustration of the Class Activation Mapping architecture, from [28].

This architecture consists of a CNN connected to a feed-forward neural network with two layers that will perform the classification. The input layer of this second network will receive the result of applying the global average pooling on the feature maps from the last convolutional network. This operation returns a single value for each feature map, so the dimension of the input layer is the same as the number of feature maps in the previous layer. The output returns a value between 0 and 1 for each possible class corresponding to the predicted probability.

Once a prediction is made, the process of creating a class activation map is straightforward. First of all, we consider the class we want to study and select the corresponding output neuron. Then, we take the weights of the connections between this neuron and the previous layer. Finally, we multiply each weight to the associated feature map from the last convolutional layer, which is an image, and sum all the results. This weighted sum will be the class activation map. For clarity, in Fig. (4.4) we have an illustration of the procedure to create these maps.

Let us take the ResNet model [37] as an example. This network takes images with a size of 224×224, and the last convolutional layer contains 2048 7×7 feature maps, i.e. 2048 pictures of size 7×7. After the pooling process, we will have 2048 single values. Then, given a class prediction, we perform a weighted sum of the 2048 feature maps with the corresponding weights. Finally, we rescale the resulting map of size 7×7 to 224×224 to compare it with the original image.

4.3 Algorithm Comparison

In this section, we present a comparison between the results obtained by applying the described interpretability algorithms to an image. We have used the library `captum`[1] for the Integrated Gradients and Occlusion methods implementations. As for the CAM algorithm, we have used the developers' source code[2]. Finally, as for the neural network, we have used the `ResNet-18` convolutional network [37], pretrained with more than a million images from the `ImageNet` dataset[3].

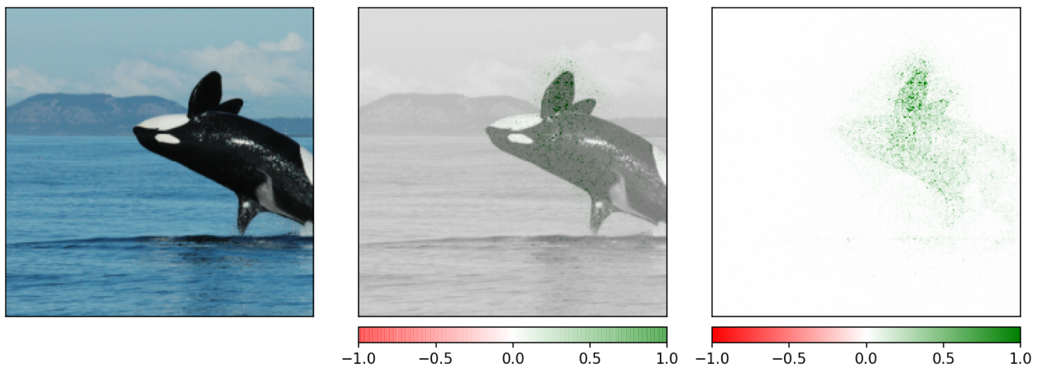
In Fig. (4.5) we have the results of the three algorithms. The input image consists of an orca breaching, and the trained network correctly classifies it as a `killer whale`. We see that all the methods detect that the important part of the image, as for the network, is the orca itself, as expected.

We see that using Integrated Gradients and Occlusion, the most relevant section of the image are the pectoral fins. Besides, we can observe that the network does not pay attention to the land behind the cetacean.

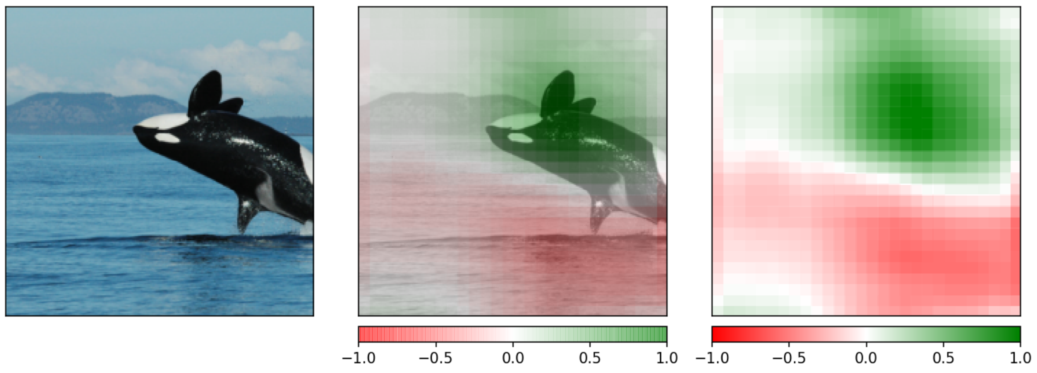
Even though the Integrated Gradients output seems to be more precise, this might not be the case for any other image, as this picture has been cherry-picked to show the behaviour of the three methods.

Finally, an important aspect to take into account when comparing algorithms is the time of execution. For CAM, since the only operation performed is obtaining the weights, the process is relatively fast and takes less than a second to finish, which makes this algorithm very useful for real-time applications. On the other hand, Occlusion lasts around 20 seconds and the Integrated Gradients over 2 minutes, mainly due to the computation of the integrals.

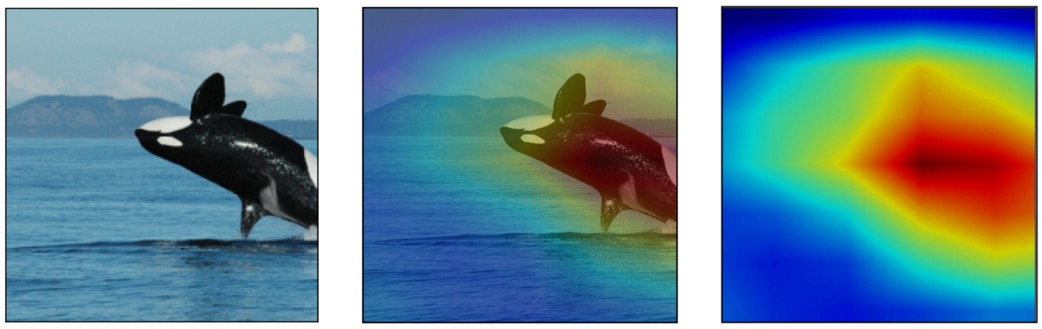
[1]https://github.com/pytorch/captum
[2]https://github.com/zhoubolei/CAM
[3]http://www.image-net.org



(a) Integrated Gradients.



(b) Occlusion.



(c) Class Activation Mapping.

Figure 4.5: Output of the three different algorithms for an image of an orca. Credit for the original image: Kenneth Balcomb, Center for Whale Research.

Chapter 5

Uncertainty

So far, we have seen that neural networks make predictions after a learning process with training data. However, in general, these predictions do not come with a confidence interval that measures how positive the model is about the output. Without the measurement of the uncertainty, the predictions can become useless, no matter how good they are.

Regression and prediction problems are a clear example. Let us consider the case of a model that predicts the number of units that will be sold of a given product. If for a specific day the output is 10, for instance, there is a massive difference if the error is 1 or 100. Then, without error estimation, decisions cannot be taken from the model's predictions.

In this chapter, we will see that neural networks can indeed learn to compute confidence intervals. In particular, we will study a regression problem in two different scenarios. In the first one, we will train a deep neural network to return both the regression prediction and the error. In the second one, we will take a black-box regressor model and use a neural network to add uncertainty to the regression prediction.

Previously, though, we will introduce the NormalLoss, a loss function that will allow us to obtain the confidence intervals based on the normal distribution.

5.1 NormalLoss

Let $\mu \in \mathbb{R}$ be the mean and $\sigma > 0$ the standard deviation. Then, the likelihood of a point x of being in the normal distribution $\mathcal{N}(\mu, \sigma^2)$ is given by the probability density function:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{5.1}$$

Our goal is to find the parameters μ and σ that best fit x in $\mathcal{N}(\mu, \sigma^2)$, i.e. that maximise the likelihood. For computational reasons, it is easier if we take the logarithm of the expression. Then, by changing the sign and ignoring a constant term, we obtain the NormalLoss function, which will have to be minimised:

$$J(x, \mu, \sigma) = log(\sigma) + \frac{(x-\mu)^2}{2\sigma^2}. \tag{5.2}$$

Finally, if we have a set of N points in the training data, we can express the NormalLoss as

$$J(x, \mu, \sigma) = \frac{1}{N}\sum_{i=1}^{N}\left(log(\sigma_i) + \frac{(x_i-\mu_i)^2}{2\sigma_i^2}\right), \tag{5.3}$$

where $x = \{x_i\}_{i=1,...,N}$, $\mu = \{\mu_i\}_{i=1,...,N}$ and $\sigma = \{\sigma_i\}_{i=1,...,N}$, and the goal is to fit x_i in $\mathcal{N}(\mu_i, \sigma_i^2)$ for $i = 1, \ldots, N$.

5.2 Regression Data

The training data for the regression problem, which can be seen in Fig. (5.1), will consist of $N = 300$ different points between 0 and 1, obeying the function $\sin(6x + 0.6)$ and noise added. We observe that points with a value closer to 0 have less noise. The reason behind doing that is to test whether the network will be able to detect areas with different amounts of noise.

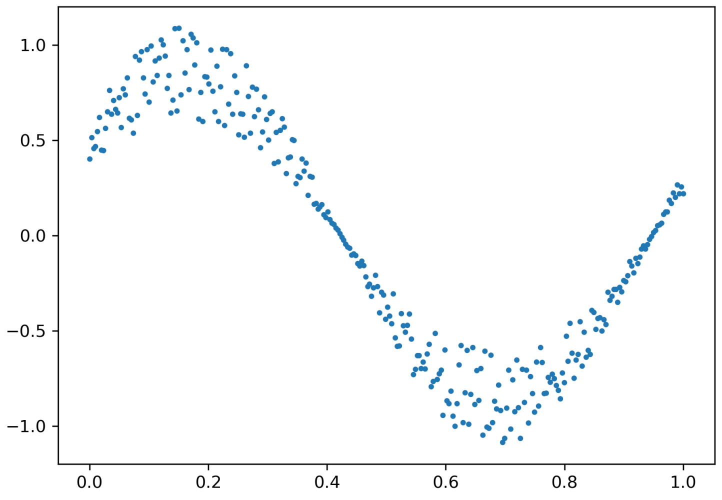


Figure 5.1: Training data for the regression problem consisting of $N = 300$ points.

5.3 Regressor Neural Network with Uncertainty

For the first scenario, we implement a deep neural network with two hidden layers and 128 neurons in each one. The input layer will contain only one neuron, corresponding to the input point, and the output layer two, corresponding to the predicted mean and standard deviation.

The activation functions for the first and second layers is the ReLU, Equation (1.23), and as for the loss function, the NormalLoss. The optimisation method used is the Adam algorithm. Lastly, the selected learning parameters are a learning rate of 0.001 and 5000 steps. In Fig. (5.2) we have the evolution of the NormalLoss function during the training process. We observe that it quickly converges to a value close to 0.

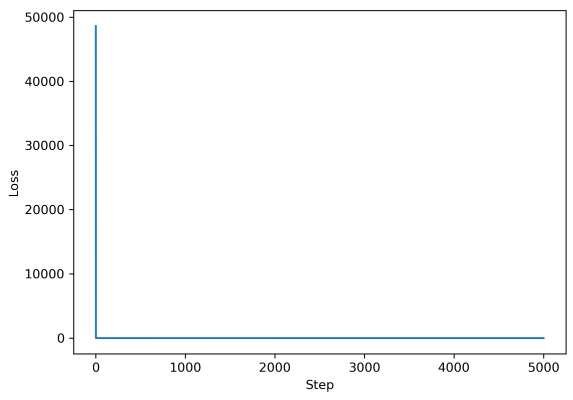


Figure 5.2: Evolution of the NormalLoss loss function for the training process of the regressor neural network with uncertainty.

In Fig. (5.3) we have the prediction of our model. We observe that the network is able to learn to fit all points in confidence intervals. These confidence intervals have been built adding and subtracting multiples of the predicted standard deviation to the mean. As expected, areas with more noise present larger uncertainty, whereas where the noise is null, the uncertainty is almost zero. It is clear then that a neural network, with the use of the NormalLoss, can return a normal distribution for every point instead of simply predicting the regression.

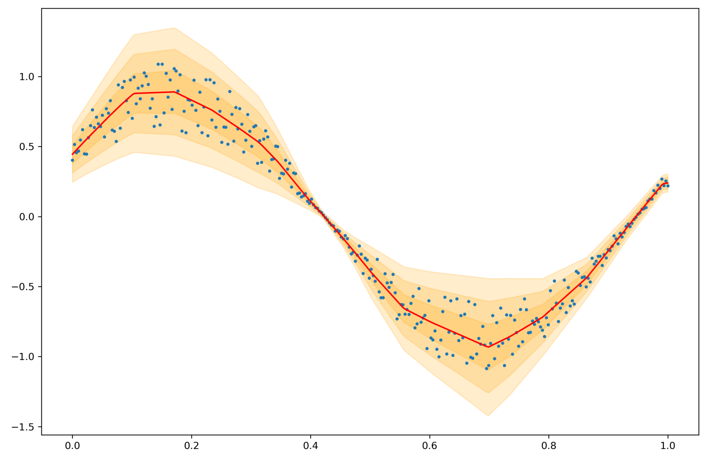


Figure 5.3: Regression and uncertainty predicted by the neural network. The red line corresponds to the mean and the orange area to the confidence interval, built summing the predicted standard deviation to the mean.

5.4 Uncertainty Neural Network for Black-Box Regressor

In this second scenario, the mean is predicted by a trained black-box regressor model, and we will implement a neural network that predicts the standard deviation.

The architecture of the deep neural network is the same as the previous case but with 64 neurons in each hidden layer. Also, the output layer will only contain one neuron, corresponding to the standard deviation.

Again, the loss function used will be the NormalLoss, but this time the mean of each point will not be learned by the network. Instead, it will have a fixed value during the learning process, initially predicted by the regressor.

The selected learning parameters are a learning rate of 0.0001 and 6000 steps.

The first regressor to be evaluated is a Decision Tree [38] with a maximum depth of 5. In particular, we will implement this model using the `scikit-learn`[1] library. Since the max depth of the tree will be 5, the learned regression will have, at most, $2^5 = 32$ different values. In Fig. (5.4) we have the regression learned with our tree model. We see that in noisy areas, the predicted values correspond to the mean of the points.

[1]https://github.com/scikit-learn/scikit-learn

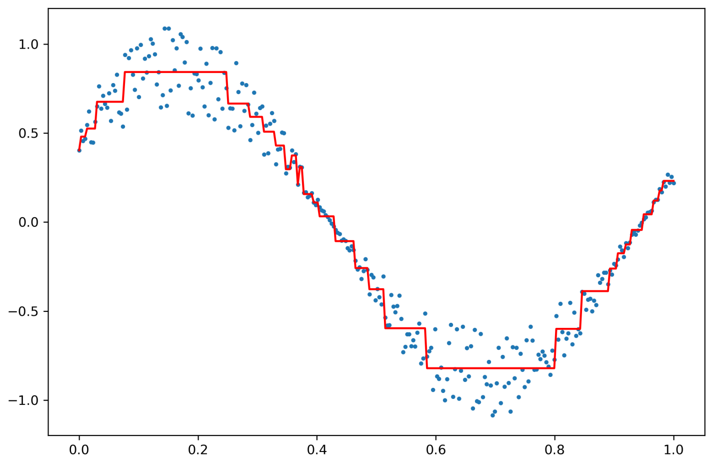


Figure 5.4: Predicted regression by the Decision Tree model with a max depth of 5.

Now, with the prediction from the regressor taken as the mean, we can proceed to train the network. In Fig. (5.5) we have the evolution of the NormalLoss function during the training process. We observe that it quickly converges to a value close to 0.

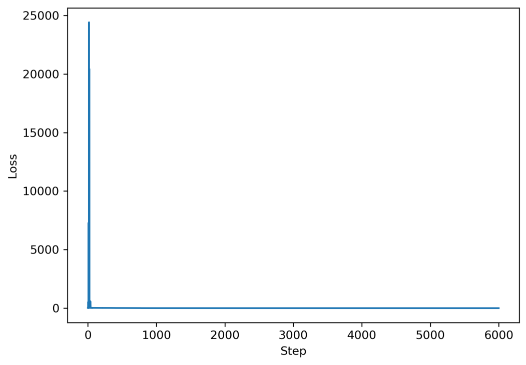


Figure 5.5: Evolution of the NormalLoss loss function for the training process of the uncertainty neural network.

In Fig. (5.6) we have the confidence intervals predicted by the neural network. We observe that the predicted uncertainty successfully fits all points according to their distance to the regression. Besides, if we take a look at the maximum and minimum zones, we see that the network correctly captures the variation of noise at each point. In both cases, the predicted standard deviation first grows, reaches a maximum, and then decreases, as expected.

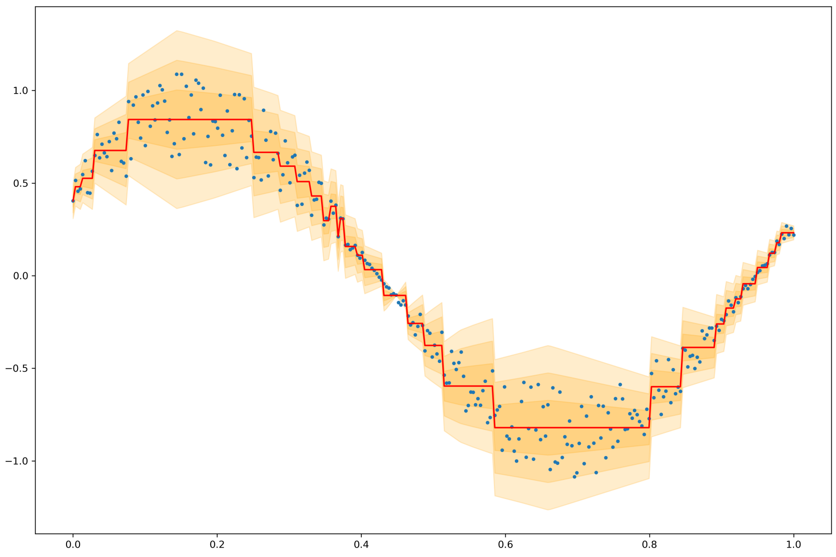
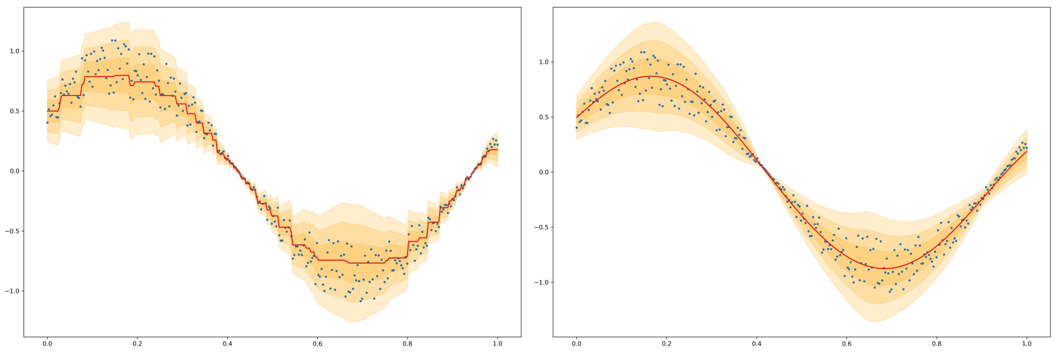


Figure 5.6: Uncertainty predicted by the neural network. The red line corresponds to the mean predicted by the regressor model and the orange area to the confidence interval, built summing the predicted standard deviation to the mean.

Now, we will consider two more regressor models, both implemented using the `scikit-learn` library as well. The first one will be a Gradient Boosting [39] with 25 estimators and the second one a Support Vector Regression (SVR) with a radial basis function (RBF) kernel [40, 41].



(a) Gradient Boosting with 25 estimators.

(b) SVR with a RBF kernel.

Figure 5.7: Uncertainty predictions for two different regressor models.

In Fig. (5.7) we have the results of the neural network trained with the two different regression predictions. We observe that, given a more complex regression, the network is still able to learn to predict the standard deviation for each point.

5.5 Conclusions

After exploring the two different scenarios, we can conclude that with these simple neural network architectures, one can obtain probability distributions instead of simple regression predictions.

It is also clear that, by wrapping any black-box regressor model with this network, we can add uncertainty to the original regression prediction.

Even though here we have used the normal distribution, which is symmetric, we could also use asymmetric distributions to obtain more complex and accurate confidence intervals, as seen in [42]. Moreover, this method has also been proved to perform well with black-box classifier models [43].

Conclusions

We proceed to evaluate the work done in this project and the achieved goals.

First of all, we have been able to give feed-forward neural networks a mathematical description. We have also presented an analysis of the learning process.

Then, after introducing the two most important types of learning approaches, we have implemented different neural networks to solve classification problems. We have also explored a representation problem using time series. We have seen that by visualising the latent space, we can understand further how the network is making the predictions.

Once learned the fundamentals of deep learning, we have presented two different applications. First, we have introduced the interpretability problem and described three distinct algorithms that attempt to evaluate which features are important for a neural network to make their prediction.

We have also seen how neural networks can learn to predict probability distributions instead of simple predictions. In addition, we have proved this model can be used to add uncertainty to existing predictions.

Moreover, we have introduced two of the main deep learning architectures, which are the Autoencoders and the Convolutional Neural Networks.

To conclude, we have not only successfully constructed a mathematical framework to analyse deep neural networks but also learned how to implement them and understand how such models behave in classification and representation problems. Furthermore, we have seen their applications in the fields of interpretability and uncertainty.

The next step from here is to study the remaining type of learning task that we did not explore, which is reinforcement learning[44], and keep investigating other neural networks' architectures that are revolutionising this field such as generative models[7, Ch. 20].

Due to the enormous potential that deep learning offers, this technology is expanding to many research areas. From Astronomy [45] to Marine Biology[46], scientists are starting to see the utility of this tool. As for the author, this work has been a step towards a research career related to quantum artificial intelligence and quantum information.

Bibliography

[1] G. Hinton, S. Osindero, and Y. Teh, *A fast learning algorithm for deep belief nets*, Neural computation, 18(7):1527-1554, 2006.

[2] W. S. McCulloch and W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, The bulletin of mathematical biophysics, 5(4):115-133, 1943.

[3] F. Rosenblatt, *The perceptron: a probabilistic model for information storage and organization in the brain*, Psychological review, 65(6):386, 1958.

[4] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Math. Control Signal Systems 2, 303-314, https://doi.org/10.1007/BF02551274, 1989.

[5] K. Hornik, *Approximation capabilities of multilayer feedforward networks*, Neural networks, 4(2):251-257, 1991.

[6] P. G. Breen, C. N. Foley, T. Boekholt and S. P. Zwart, *Newton versus the machine: solving the chaotic three-body problem using deep neural networks*, Monthly Notices of the Royal Astronomical Society, 494:1365-2966, 2020.

[7] I. J. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, The MIT Press, 2016.

[8] A. Caterini, *A Novel Mathematical Framework for the Analysis of Neural Networks*, UWSpace, http://hdl.handle.net/10012/12173, 2017.

[9] K. Janocha and W. M. Czarnecki, *On Loss Functions for Deep Neural Networks in Classification*, arXiv:1702.05659, 2017.

[10] A. Cauchy, *Méthode générale pour la résolution des systèmes d'équations simultanées*, C. R. Acad. Sci. Paris, 25:536-538, 1847.

[11] L. Armijo, *Minimization of functions having Lipschitz continuous first partial derivatives*, Pacific J. Math. 16, no. 1, 1-3, 1966.

[12] T. T. Truong and T. H. Nguyen, *Backtracking gradient descent method for general C^1 functions, with applications to Deep Learning*, arXiv:1808.05160, 2018.

[13] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli and Y. Bengio, *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, arXiv:1406.2572, 2014.

[14] S. Ruder, *An overview of gradient descent optimization algorithms*, arXiv:1609.04747, 2016.

[15] D. Rumelhart, G. Hinton and R. Williams, *Learning representations by back-propagating errors*, Nature 323, 533-536, https://doi.org/10.1038/323533a0, 1986.

[16] D. Rumelhart, G. Hinton, and R. Williams, *Learning Internal Representations by Error Propagation*, Parallel Distributed Processing, chapter 8, The MIT Press, 1986.

[17] M. Alber, I. Bello, B. Zoph, P.J. Kindermans, P. Ramachandran and Q. Le, *Backprop Evolution*, arXiv:1808.02822, 2018.

[18] E. van den Berg, *Some Insights into the Geometry and Training of Neural Networks*, arXiv:1605.00329, 2016.

[19] C. R. Souza, *Kernel Functions for Machine Learning Applications*, 17 Mar. 2010. Web http://crsouza.blogspot.com/2010/03/kernel-functions-for-machine-learning.html.

[20] C. E. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*, arXiv:1811.03378.

[21] V. Nair and G. Hinton, *Rectified Linear Units Improve Restricted Boltzmann Machines*, Proceedings of ICML, 27, 807-814, 2010.

[22] Y. LeCun, Y. Bengio and G. Hinton, *Deep learning*, Nature 521, 436-444, https://doi.org/10.1038/nature14539, 2015.

[23] O. Biran and C. Cotton, *Explanation and justification in machine learning: A survey*, IJCAI 2017 Workshop on Explainable Artificial Intelligence (XAI), 8–13, 2017.

[24] C. Molnar, *Interpretable machine learning. A Guide for Making Black Box Models Explainable*, Web https://christophm.github.io/interpretable-ml-book/, 2019.

[25] F. Fan, J. Xiong and G. Wang, *On Interpretability of Artificial Neural Networks*, arXiv:2001.02522, 2020.

[26] M. Sundararajan, A. Taly and Q. Yan, *Axiomatic Attribution for Deep Networks*, arXiv:1703.01365, 2017.

[27] M. D. Zeiler and R. Fergus, *Visualizing and Understanding Convolutional Networks*, arXiv:1311.2901, 2013.

[28] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva and A. Torralba, *Learning Deep Features for Discriminative Localization*, arXiv:1512.04150, 2015.

[29] A. Shrikumar, P. Greenside and A. Kundaje, *Learning Important Features Through Propagating Activation Differences*, arXiv:1704.02685, 2017.

[30] S. Lundberg and S. Lee, *A Unified Approach to Interpreting Model Predictions*, arXiv:1705.07874, 2017.

[31] A. Shrikumar, P. Greenside, A. Shcherbina and A. Kundaje, *Not Just a Black Box: Learning Important Features Through Propagating Activation Differences*, arXiv:1605.01713, 2016.

[32] J. Castro, D. Gómez and J. A. Tejada Cazorla, *Polynomial calculation of the Shapley value based on sampling*, Computers and Operations Research, 36 (5). pp. 1726-1730. ISSN 0305-0548, 2009.

[33] R. Yamashita, M. Nishio1, R. K. G. Do and K. Togashi, *Convolutional neural networks: an overview and application in radiology*, Insights Imaging 9, 611-629, https://doi.org/10.1007/s13244-018-0639-9, 2018.

[34] Y. LeCun and Y. Bengio, *Convolutional networks for images, speech, and time series*, The handbook of brain theory and neural networks, The MIT Press, Cambridge, MA, USA, pp. 255-258, 1998.

[35] Y.T. Zhou and R. Chellappa, *Computation of optical flow using a neural network*, IEEE 1988 International Conference on Neural Networks, 71-78 vol.2., 1988.

[36] M. Lin, Q. Chen and S. Yan, *Network in network*, arXiv:1312.4400, 2013.

[37] K. He, X. Zhang, S. Ren and J. Sun, *Deep Residual Learning for Image Recognition*, arXiv:1512.03385, 2015.

[38] W.Y. Loh, *Classification and Regression Trees*, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 1, 14-23, 2011.

[39] J. Friedman, *Greedy function approximation: a gradient boosting machine*, Annals of Statistics, 29(5):1189-1232, 2001.

[40] H. Drucker, C. C, L. Kaufman, A. Smola, V. Vapnik, *Support Vector Regression Machines*, Advances in Neural Information Processing Systems, 9, 2003.

[41] J.P. Vert, K. Tsuda, and B. Schölkopf, *A primer on kernel methods*, Kernel Methods in Computational Biology, 35-70, 2004.

[42] A. Brando, J. A. Rodríguez-Serrano, J. Vitrià and A. Rubio, *Modelling heterogeneous distributions with an Uncountable Mixture of Asymmetric Laplacians*, arXiv:1910.12288, 2019.

[43] J. Mena, A. Brando, O. Pujol and J. Vitrià, *Uncertainty Estimation for Black-Box Classification Models: A Use Case for Sentiment Analysis*, Pattern Recognition and Image Analysis, Springer International Publishing, pp. 29-40, 2019.

[44] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 2018.

[45] D. Baron, *Machine Learning in Astronomy: a practical overview*, arXiv:1904.07248, 2019.

[46] Y. Shiu, K.J. Palmer, M.A. Roch, E. Fleishman, X. Liu, E.M. Nosal, T. Helble, D. Cholewiak, D. Gillespie and H. Klinck, *Deep neural networks for automated detection of marine mammal species*, Sci Rep 10, 607, https://doi.org/10.1038/s41598-020-57549-y, 2020.