A Deep Convolutional Neural Network Architecture for Image Classification

W.L. Pretorius

Thesis presented in the partial fulfilment
of the requirement for the degree of
Master of Commerce (Mathematical Statistics)
in the Faculty of Economic and Management Sciences at the University of Stellenbosch

Supervisor: Dr. M. M. C. Lamont

December 2020

PLAGIARISM DECLARATION

1. Plagiarism is the use of ideas, material and other intellectual property of another's work and to present it as my own.
2. I agree that plagiarism is a punishable offence because it constitutes theft.
3. Accordingly, all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.
4. I also understand that direct translations are plagiarism.
5. I declare that the work contained in this thesis, except otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this thesis or another thesis.

Student number	Signature
W.L. Pretorius	01 September 2020
Initials and surname	Date

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the following people and organisations:

- My loving parents, for all their hard work, emotional and financial support through the years that enabled me to be where I am today;
- To my supervisor, Dr. Morné Lamont, for his guidance and patience in giving me the freedom to explore the ideas in this thesis;
- To my best friend and love, Marilé, for her never-ending support, guidance and understanding;
- To my friends and close family for believing in me and motivating me;
- To Kaggle for hosting a platform to share datasets;
- To the Ackerman family for providing me housing and support in the 'Corona virus(COVID-19)' - pandemic.

ABSTRACT

A Deep Convolutional Neural Network Architecture for Image Classification

W.L. Pretorius

Thesis: MCom (Mathematical Statistics)

December 2020

Convolutional Neural Networks (CNNs), a specialised form of *Neural Networks* (NNs), are well-known for their state-of-the-art results obtained in *Computer Vision* (CV) and *Deep Learning* (DL) tasks throughout the past few years. Some of the exciting application areas of CNNs include image classification, object detection, video processing, natural language processing, and speech recognition.

The powerful learning ability of deep CNNs is primarily owed to the use of multiple feature extraction stages that can automatically learn representations from the data. The availability of a large amount of data and improvement in hardware technology has accelerated the research done in CNNs, and recently interesting deep CNN architectures have been reported. Several inspiring ideas to bring advancements in CNNs have been explored, such as the use of different activation and loss functions, parameter optimisation, regularisation, and architectural innovations by using different layer structures.

Therefore, the objective of this study is based on image classification and object detection tasks, that is, creating custom-designed CNN architectures for deployment on real-world datasets while comparing these custom-designed architectures to those state-of-the-art architectures found in literature while comparing different optimisation procedures and activations functions. All major developments of CNNs are discussed and critically considered, with a view to improve CNNs in the context of the number of parameters used to obtain satisfactory results and additionally to obtain a better understanding of the term known as a 'black box' which is usually associated with CNNs such that they are complex models with little understanding in the way how their classifications are done.

The most promising modern CNN architectures with associated hyperparameters are further explored by means of empirical work. Evaluation is done on the validity of findings reported in the literature and comments are made on the effectiveness of recent proposals through the use of five different real-world datasets. The empirical work done will be complemented by additional coded notebooks that could be used to implement state-of-the-art techniques, as well as for comparative and model assessment experiments.

Key words: Convolutional Neural Networks (CNNs); architecture; activation functions; optimisation.

OPSOMMING

'n Diep Konvolusionele Neurale Netwerk Argitektuur vir Beeldklassifikasie

("A Deep Convolutional Neural Network Architecture for Image Classification")

W.L. Pretorius

Tesis: MCom (Wiskundige Statistiek)

Desember 2020

Konvolusionele Neurale Netwerke (KNNe), 'n gespesialiseerde vorm van neurale netwerke, is bekend vir hul nuutste resultate wat gedurende die afgelope paar jaar in rekenaarvisie en diepleer take behaal is. Sommige van die opwindende toepassingsgebiede van KNNe sluit beeldklassifikasie, voorwerpopsporing, videoverwerking, natuurlike taalverwerking en spraakherkenning in.

Die kragtige leervermoë van diep KNNe is hoofsaaklik te danke aan die gebruik van verskeie funksie-ekstraksie-fases wat outomaties voorstellings uit die data kan leer. Die beskikbaarheid van 'n groot hoeveelheid data en verbetering in hardeware-tegnologie het die navorsing wat in KNNe gedoen is, versnel, en daar is onlangs berig oor interessante diep KNN-argitekture. Verskeie inspirerende idees om vooruitgang in KNNe te bring, is ondersoek, soos die gebruik van verskillende aktiverings- en verliesfunksies, parameter optimalisering, regulering en argitektoniese innovasies waardeur verskillende laagstrukture gebruik word.

Daarom is die doelstelling van hierdie studie gebaseer op beeldklassifikasie en opsporingstake, dit wil sê die opstel van KNN-argitekture wat ontwerp is vir die implementering op werklike datastelle, terwyl hierdie ontwerpte argitekture vergelyk word met 'state-of-the-art' argitekture wat in die literatuur voorkom, terwyl verskillende optimaliseringsprosedures en aktiveringsfunksies vergelyk word. Alle belangrike ontwikkelings van KNNe word bespreek en krities oorweeg, met die oog op die verbetering van KNNe in die konteks van die aantal parameters wat gebruik word om bevredigende resultate te verkry, en om 'n beter begrip te kry van die term wat bekend staan as 'swart boks', wat gewoonlik geassosieer met KNNe is, waar dit komplekse modelle is met min begrip van die manier waarop hul klassifikasies gedoen word.

Die mees moderne KNN-argitekture met gepaardgaande hiperparameters word verder ondersoek deur middel van praktiese werk. Evaluering word gedoen oor die geldigheid van die bevindings wat in die literatuur gerapporteer word en kommentaar word gelewer op die doeltreffendheid van onlangse voorstelle deur die gebruik van vyf verskillende werklike datastelle. Die empiriese werk wat verrig word, sal aangevul word deur addisionele gekodeerde notaboeke wat gebruik kan word om moderne tegnieke te implementeer, asook vir vergelykende en model assesserings eksperimente.
Sleutelwoorde: Konvolusionele Neurale Netwerke (KNNe); argitekture; aktiverings funksies; optimalisering.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

LIST OF ABBREVIATIONS AND/OR ACRONYMS

AI	Artificial Intelligence
ML	Machine Learning
CV	Computer Vision
DL	Deep Learning
ANN	Artificial Neural Network
NN	Neural Network
CNN	Convolutional Neural Network
GPU	Graphical Processing Unit
TL	Transfer Learning
CPU	Central Processing Unit
MSE	Mean Squared Error
SGD	Stochastic Gradient Descent
RMS	Root Mean Squared
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
MNIST	Modified National Institute of Standards and Technology

CHAPTER 1
INTRODUCTION

1.1 ARTIFICIAL INTELLIGENCE

The progress of *Artificial Intelligence* (AI) has paved the path of how human knowledge expanded in the last few decades with the rise of new technology and the abundance of data. The field of AI ranges across numerous disciplines, for example, computer science, mathematics, economics, philosophy, and even ethics which suggests that this field has several wide-ranging applications and can be viewed from different perspectives. A thorough explanation of these applications can be found in Russell and Norvig (2003).

A common definition in literature is that AI represents any technique that enables machines or computers to mimic human intelligence by learning to imitate and perform it (Poole *et al.*, 1998). Equivalently, AI is the effort of the machine to represent cognitive functions that humans possess such as seeing, speaking, thinking, learning, and problem-solving. This definition was brought forward when Alan Turing, who cracked the Nazi encryption machine, Enigma, and helped Allied Forces win World War II, raised the question "can machines think?" which established the fundamental goals and visions of AI (Turing, 1950). This question escalated a revolution in technology and research which formally started at the Dartmouth Summer Research Project on AI led by John McCarthy in 1955 (McCarthy *et al.*, 2006).

Thereafter, AI gained remarkable attention (see Figure 1.1) especially when International Business Machine's supercomputer, Deep Blue, beat world chess champion, Garry Kasparov, in 1997 (McPhee *et al.*, 2015). Today, AI is being used in every part of our lives from healthcare, manufacturing, and entertainment to retail and banking services. This corresponds to the many subfields currently encountered in AI such as *Machine Learning* (ML), natural language processing, speech, robotics, expert systems, and vision. It is almost impossible to predict the speed at which AI literature is growing which is largely due to the exponential growth of data that is freely available and the rapid development of computers and mobile phones through the decline of hardware prices, making it easier to access technology that did not exist a few years ago.

Unfortunately, the growth of technology not only has positive potential such as decreasing everyday human labour, but it could have negative impacts on society such as the loss of jobs owed to being automated, creating a massive socio-economic or ethical challenge for governments and corporations.

This thesis will only consider the ML and Vision subfields whereas the researcher encourages the reader to consult Russell and Norvig (2003) for a brief overview of the other subfields.

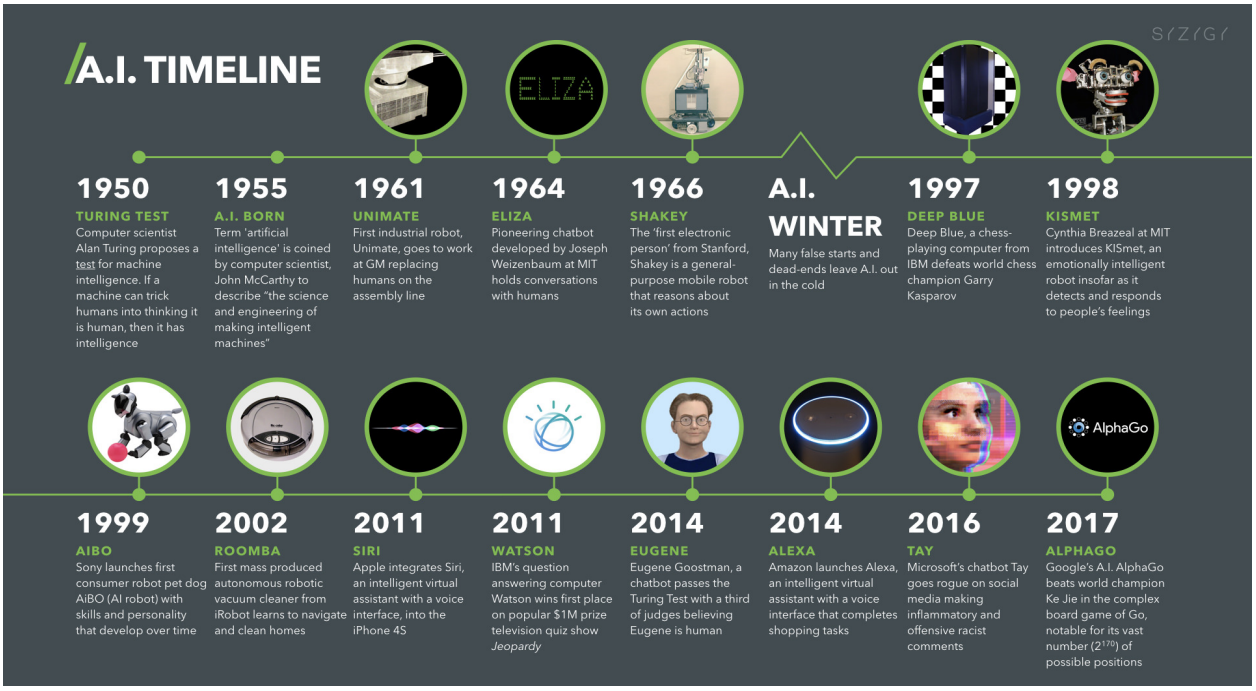


Figure 1.1: A brief history of AI.
Source: Marsden (2017) [Online].

1.2 MACHINE LEARNING AND VISION

A machine or computer is said to be 'learning' if its performance of a specific task improves with experience and time (Thrun and Pratt, 1998), meaning that a performance metric and example or input is needed to train the machine with little or no human interference. The specific task can be anything ranging from analysing company profit over time to classifying whether an image contains a dog or a cat. These tasks are usually referred to as regression tasks or classification tasks, respectively. For the machine to learn accurately, it requires a large number of input examples or data. The accuracy is defined over some metric called the error or cost which should be minimised to ensure that the machine improves over time. Fundamentally, the machine is expected to adapt based on the situation it is given through the use of specific algorithms that are dedicated to improving the performance and prediction of the task. This approach is generally referred to as model building, and there exist different approaches as to how a model is designed which depends on the data it is fed. The main approaches are called supervised learning and unsupervised learning.

If the inputs, data, or examples fed to the model are labelled as well as the desired output, for example, images of objects with corresponding labels, the approach is called supervised. These labels provide supervision for the algorithm at hand while the model uses these labels to provide a prediction or output. The prediction is then compared to the corresponding input data or often

called training data to estimate model error or cost. Using this error, the initial algorithm adjusts the model parameters to reduce the loss (loss is the value of the objective function that is being minimised and error involves an interpretable metric of the model's performance). In essence, the model tries to find a general rule, pattern, or function that maps the input to the desired output. The variables contained in the function are referred to as parameters or features. The goal is to train the model in such a general way that it could accurately map an input to an output when it is fed unseen data. This process in its entirety is known as learning by example (Hastie *et al.*, 2009). An in-depth mathematical study on this topic will be done in Chapter 2.

Contrary to supervised learning, unsupervised learning is when the inputs contain no labels and only examples. The goal of the algorithm is to find a specific structure of the data, for example, using clustering methods to calculate which examples are related and which are not. This thesis is particularly interested in the supervised approach, more specifically in the classification of objects in image data. Therefore, unsupervised learning is beyond the scope of this research. The reader could consult (Hastie *et al.*, 2009) for a deeper understanding of unsupervised learning.

Similarly, to AI, ML also consists of many subfields. However, the Vision subfield of AI can be combined with ML to obtain a topic called *Computer Vision* (CV). This phenomenon is what enables machines such as computers and mobile phones to observe their surroundings. This is where the machine or model uses digital images or videos to perform and surpass tasks similar to the capability of the human visual system. This field started to evolve together with AI from the 1960s according to Huang (1996) when "Larry Roberts, who in his PhD thesis at MIT discussed the possibilities of extracting three-dimensional geometrical information from two-dimensional perspective views of blocks (polyhedra)". Since then, CV has become an important aspect of image processing, especially from the fact that computers view images differently to the way humans do. Machines or computers count the number of pixels and dissect borders from objects by measuring shades of colours through pixel weights. Today, these algorithms are being used in countless applications ranging from facial recognition[1] to medical image segmentation (Hesamian *et al.*, 2019), and more.

1.3 THE IMAGE AND THE IMAGE PIPELINE

An image is a two-dimensional representation of the visible light spectrum (see Figure 1.2). A computer stores an image in a variety of formats, however, the most commonly used in CV is RGB — Red, Green, and Blue. Every pixel is a combination of the brightness (see Figure 1.3) in the RGB

[1] https://www.nist.gov/programs-projects/face-recognition-vendor-test-frvt-ongoing

channel, ranging from 0 (darkest) to 255 (brightest), for example, yellow is represented as (255, 255, 0). Equivalently, computers store images as multi-dimensional arrays, matrices, or tensors (see Figure 1.4) where colour images are three-dimensional (e.g. 1920 x 1080 x 3), and where greyscale (black and white) images are two-dimensional (e.g. 1920 x 1080 x 1).

Image processing and pipelines (see Figure 1.5) utilise the form of images and are important subjects of CV (Szeliski, 2010), usually summarised in the following intuitive steps:

- Image capture: Image is captured through a specific device and transformed by a computer to numerical form.
- Pre-processing: Numerical image is modified to extract and emphasise certain important features, for example, noise reduction using special filters on the image.
- Segmentation: The selection of important features, for example, edges.
- Classification: The use of a model to classify the object in the image.

These pipelines guide how computers read and understand images for image classification. In other words, images are simplified by reducing the visual information contained within the image to simple numerical values or matrices which can be fed into ML models. Refer to Chapter 4 and Chapter 6 for a detailed discussion on how the image pipeline is used for a specific classification task.

Throughout the years with the exponential increase in the availability of more complex data and data types, this pipeline opened up numerous difficulties for traditional ML models such as the Support Vector Machine, Logistic regression model, and so forth which perform well on small datasets but struggle with large datasets. For image classification, this struggle is due to an increase in the complexity of certain abstract image features, such as the change of individual pixels in an image (due to lens limitations regarding the change of lightning condition) depending on the time of day which can only be identified using a human-like understanding of the data (Goodfellow *et al.*, 2016). Therefore, traditional ML models can have problems when these types of images are scaled, rotated, or denoised. However, there exists a solution called *Deep Learning* (DL) which is another type of ML subfield closely related to CV. In general, DL uses supervised and/or unsupervised approaches to automatically learn representations hierarchically for classification and regression architectures amounting to a greater number of parameters used compared to traditional models learning shallow architectures (Ranzato *et al.*, 2008).

1.4 DEEP LEARNING

The field of DL has gone through several meanings and definitions in the last few years. Defined as the easing of complex concepts into simpler ones and the reconstruction of the simpler concepts into new complex ones directs that an algorithm has to establish an order of concepts to be considered as 'deep' (Goodfellow *et al.*, 2016). In simple graph theory, this relates to a multi-layered visualisation. However, a more common definition suggests that DL is known as *Artificial Neural Networks* (ANNs) or in short *Neural Networks* (NNs) (Skansi, 2018). These networks are theoretically inspired by the biological nervous system of the human brain (see Figure 1.6).

The brain uses an infinitely large inter-connected network of neurons for information processing to model the observations it experiences. This happens when a neuron gathers observations from other neurons using dendrites. These neurons then sum together all the observations and if the resulting value is greater than a specific threshold, it fires or activates a signal to the other connected neurons through the axon (Phillips, 2015). Inspired by the workings of the brain, NNs were indirectly introduced in 1958 by Frank Rosenblatt who proposed the Perceptron learning model (see Figure 1.7) (Rosenblatt and Papert, 1957).

This model (which will be expanded on in Chapter 3) consists of five nodes and a single-layer that are interconnected which form the basis of most existing NN architectures. These are:

- Inputs: Numerical values (neurons), which can be any real number.
- Connections: These are the associated weights of every input.
- Weighted sum: The sum of all the inputs and connections, passed to the activation function.
- Activation function: The threshold.
- Output: A prediction.

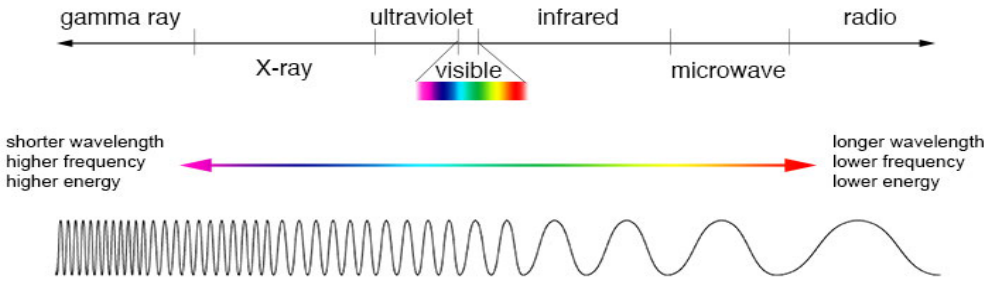


Figure 1.2: The light spectrum.
Source: NASA's Imagine The Universe (2013) [Online].

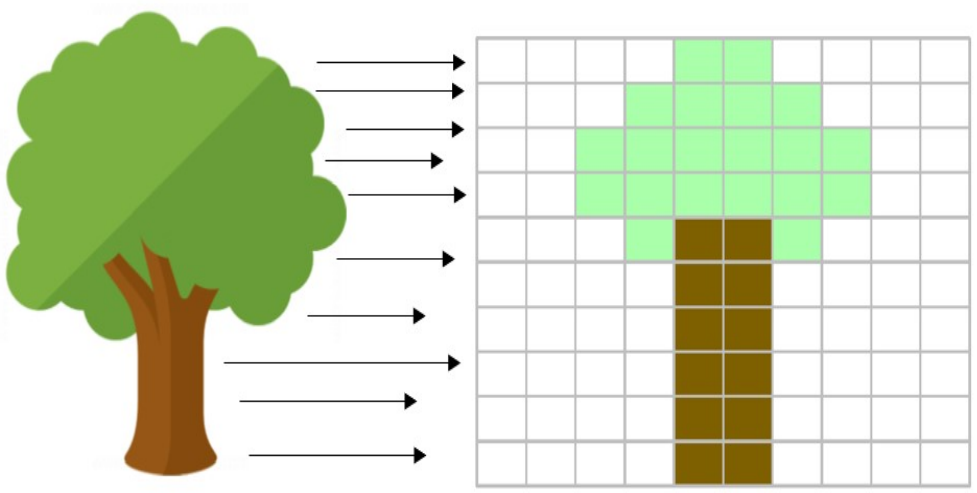


Figure 1.3: RGB channel example of how a computer perceives an image.

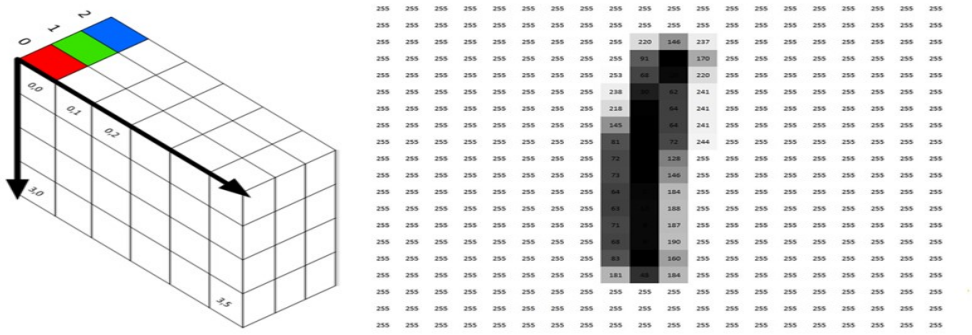


Figure 1.4: Image matrices and tensors example of 3-dimensions on the left and 2-dimensions on the right.

NN models (refer to Chapter 3 for a complete definition) then originated when researchers started using more layers to implement more complex predictions, which formed the notion of feedforward NNs. These added layers often described as multi-hidden-layers (see Figure 1.8), allows for the capturing of non-linear relationships which enable the training of more features and parameters contributing to the massive success of DL research since 2006 where the term 'deep learning' was first introduced by Hinton *et al.* (2006). This structure complemented the notion that NNs have the ability to model almost any input-output relationship it is being fed. Currently, DL models are receiving extensive attention not only in research but also in practice. Large companies such as Facebook, Google, Twitter, Instagram, and WhatsApp are deploying deep NN models for numerous purposes which include image recognition[2], speech recognition[3], and recommendation systems which have stemmed from a variety of available DL models.

[2]https://waymo.com/
[3]https://www.apple.com/siri/

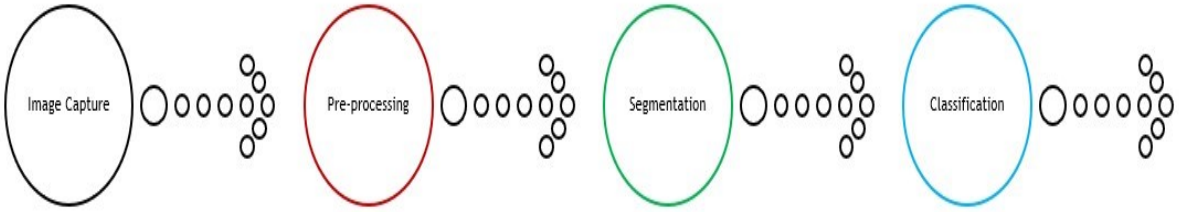


Figure 1.5: Image pipeline utilising the form of images.

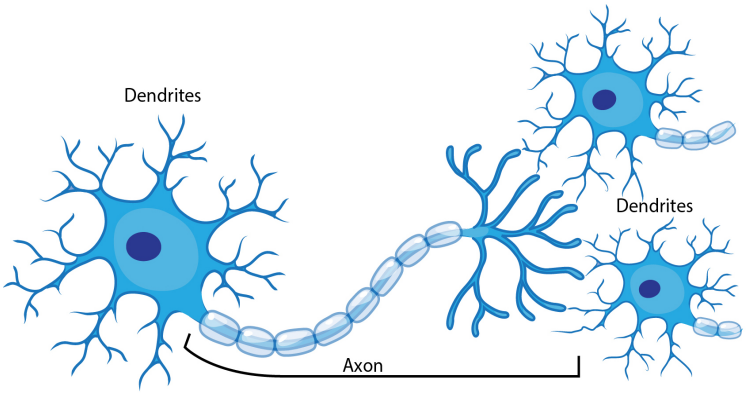


Figure 1.6: Illustration of the biological nervous system of the human brain.
Source: Phillips (2015) [Online].

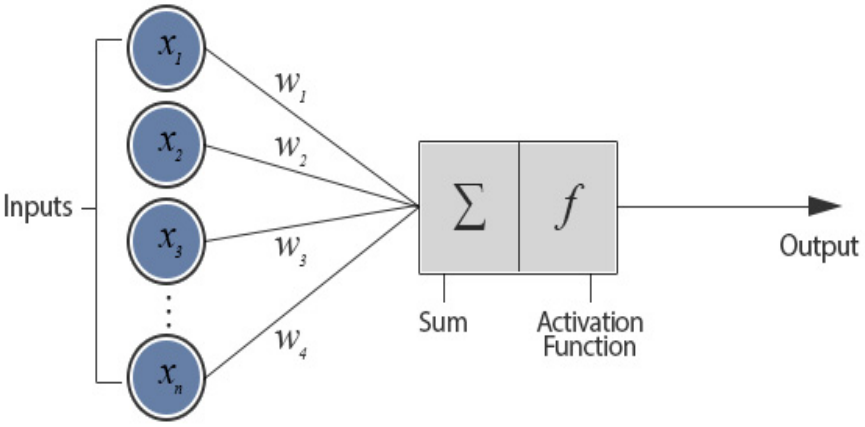


Figure 1.7: Illustration of the simple Perceptron model.
Source: Jacobson (2013) [Online].

These models can be categorised into several forms (see Figure 1.9). One of which is of particular interest for this thesis, known as *Convolutional Neural Networks* (CNNs). These networks changed the CV and image processing world and can be regarded as one of the most famous existing

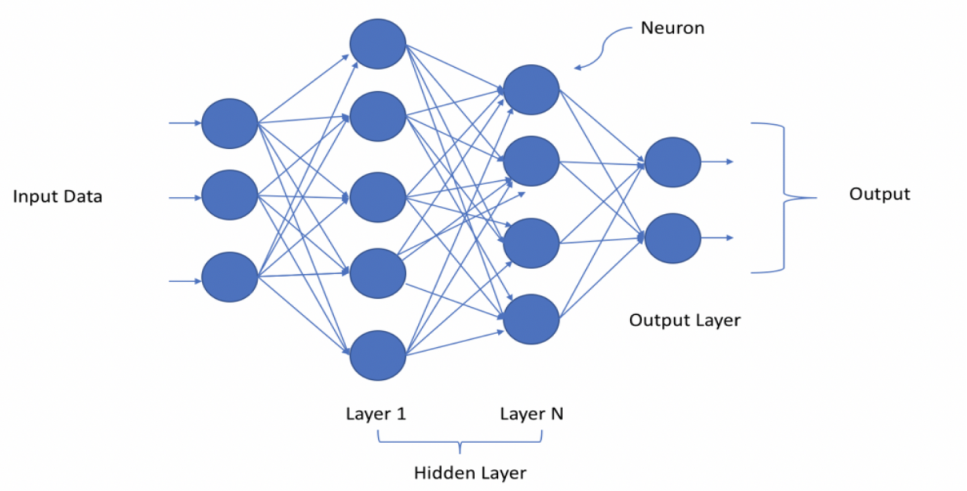


Figure 1.8: General illustration of a multi-hidden-layer NN.
Source: Moolayil (2019) [Online].

architectures for image classification and object detection (Girshick *et al.*, 2014). This network became particularly useful since its introduction through the work of LeCun in 1989 who processed handwritten zip code data (LeCun *et al.*, 1989) as part of an image classification project for the U.S Postal Service and is currently considered as the state-of-the-art architecture for these tasks (Krizhevsky *et al.*, 2012), due to its ability to exploit spatial correlation of image data.

The structure of this architecture is divided into multistage non-linear layers that use a large amount of data for processing. However, no exact rule exists on how the individual layers should be structured but the typical manner of how CNNs are organised consists of two parts (see Figure 1.10) namely the feature extraction portion and the classification portion. The feature extraction part consists of a combination of convolutional (filtering) and pooling (subsampling) layers whereby each layer learns an abstract representation of its input (neuron). These portions can be seen as hyperparameters (parameters set by the researcher or user) since each portion can contain multiple layers within them. The convolutional layer employs a feature detector using what is called a filter or kernel to identify different aspects of an input (usually an image in this case) such as edges, corners, and colours. Every convolutional layer in the architecture learns more complex patterns outputting many different transformations to what is called feature maps. These transformations are then outputted to an activation function, just as the case with ANNs, which not only improves the learning of abstractions from the filters but also embeds non-linearity in the feature space. This is then followed by a subsampling layer which not only reduces the input dimension by preserving the information of the image but also summarises the network such that it makes the inputs invariant to geometrical distortions (LeCun *et al.*, 2010). This may seem like a

loss of data, but subsampled layers only attain the most important features of the image, reducing model calculations. This also indicates that the pre-procession of image data is not required since this is done automatically by the successive convolutional and pooling layers, meaning that the CNN can learn accurate representations from the given inputs. Lastly, these layers are submitted to an output layer that uses a specific function to perform classification operations. Refer to Chapter 4 for a detailed discussion on this architecture.

In traditional ML for image classification, extraction of features from the image is carried out manually by the researcher beforehand. These features are then used in models such as the Support Vector Machine (Forsyth and Ponce, 2012). If the features were incorrectly chosen, it would directly lead to a bad classification. This problem, however, does not exist with CNNs since it calculates the features automatically which were emphasised earlier.

As mentioned earlier, images have certain colour depth channels. The convolutional operator uses these channels to operate in a certain dimension. This thesis will only cover applications from 2-dimensional images that will either be coloured or greyscale. These 2-dimensional input images can be seen as image matrices where:

$$\text{input} = \text{image height} \times \text{image width} \times \text{number of channels(depth)},$$

$$\text{number of channels} = \begin{cases} 1, \text{ if the input image is greyscaled.} \\ 3, \text{ if the input image is coloured.} \end{cases}$$

It is evident that the power of this DL model increases as the number of layers increases. However, with great power comes great responsibility when training these models for practical use, since they can rapidly become very deep. Very deep models undergo heavy processing which demands highly powerful machines, due to the large amount of data that is needed to efficiently train them. This was a massive obstacle for DL models for about two decades up until 2006 when Hinton *et al.* (2006) introduced a greedy layer-wise pre-training algorithm that contributed to the success of training these models faster and more efficiently. Nonetheless, this would not have been possible without the availability of two key components of DL. Firstly, with the availability of large, open, and labelled datasets such ImageNet. Secondly, with the availability of cheap and massively parallel computing power in the form of *Graphical Processing Units* (GPUs).

Contributing to the use of GPUs in deep CNN models was Ciresan *et al.* (2011) who achieved human-level accuracy on the handwritten digit recognition benchmark dataset modified by the *National Institute of Standards and Technology* (MNIST). Shortly after, another breakthrough was made when Krizhevsky *et al.* (2012) won the 2012 ImageNet challenge, which to date is the largest

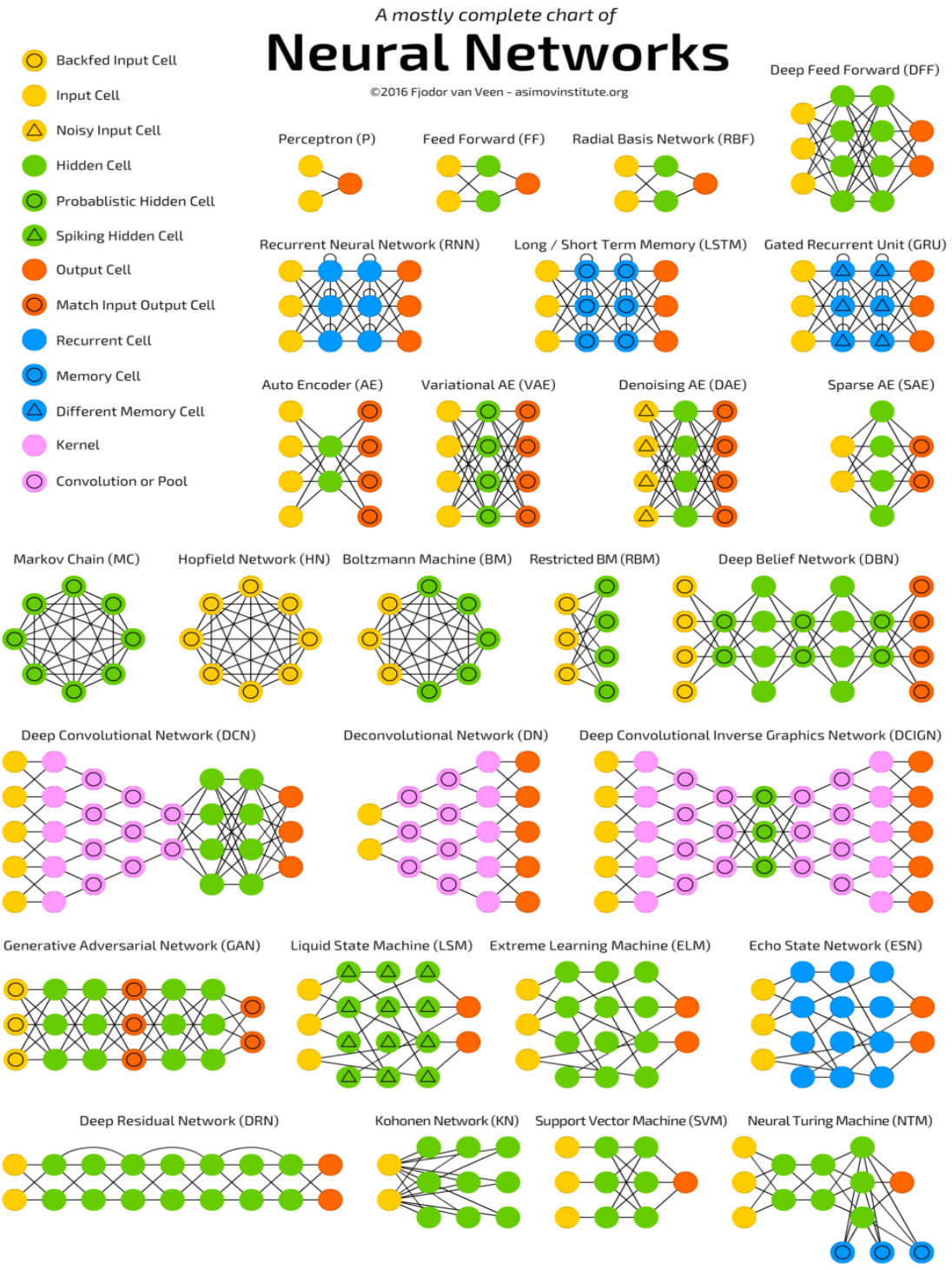


Figure 1.9: Examples of the different type of NNs found in practice and literature. Source: Mehta (2019) [Online].

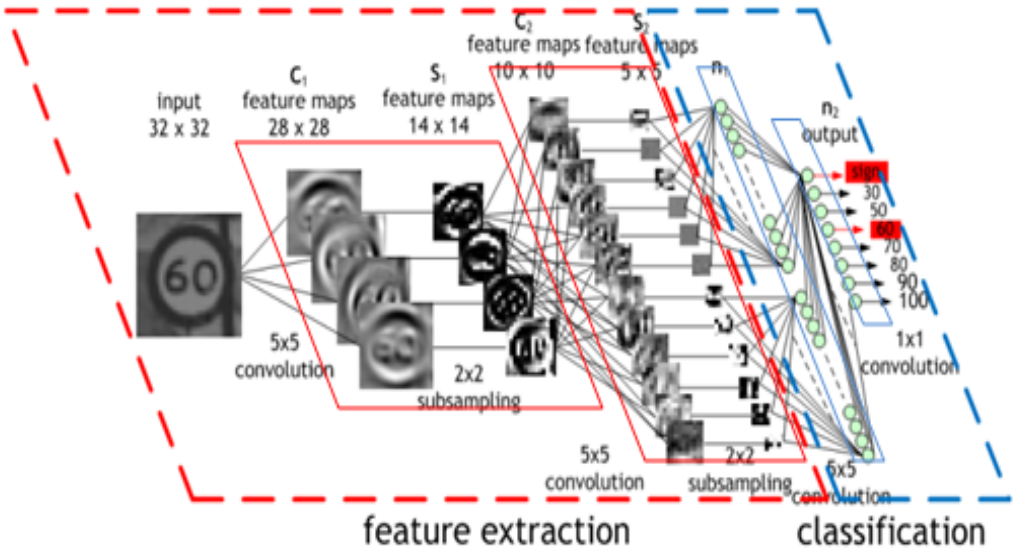


Figure 1.10: Illustration of the general structure of a CNN taking on a feature extraction part and a classification part.
Source: Peemen (2019) [Online].

contest in object recognition, with 1.2 million high-resolution images in 1000 classes or objects. This was done by using a very deep CNN architecture called AlexNet. This architecture took the authors five to six days to train on two GPUs since it consisted of about 60 million parameters. Currently, many deeper architectures (see Table 1.1) exist whereas 60 million parameters might seem small or shallow to some researchers especially considering the speed at which GPU computation has increased.

Table 1.1: Illustration of the size of current state-of-the-art architectures.

Model	Parameters
VGG16	138 357 544
InceptionV3	23 851 784
ResNet50	25 636 712

Source: Keras (2020) [Online].

However, most of these architectures are trained on multiple GPUs, and since powerful GPUs are expensive, the question is how can an individual train these models on their home computer with a maximum amount of having one GPU? The answer is obtained by using a simple phenomenon called *Transfer Learning* (TL). This method allows for the knowledge of one task to be transferred to another related task if some universal pattern in the tasks exists, for example, in image classification, there could be a resemblance between the weights when identifying flower species and the weights

when identifying different fruits. Meaning that patterns learned by the model for flower detection could be useful to detect fruits. In a CNN architectural setting, TL has two common approaches in the form of using a pre-trained model from some dataset, usually the ImageNet set.

The first approach is that the pre-trained model is only used as a feature extractor (Oquab *et al.*, 2014), meaning that specific knowledge is only transferred up to a specific layer for the generation of features keeping the weights fixed. That knowledge can then be used in other learning algorithms. A second approach is that of fine-tuning (Yosinski *et al.*, 2014), meaning that the pre-trained model will be transferred whereby its weights will be adjusted in the new learning algorithm. Both approaches make training faster since the initial weights have already been specified by another model, thereby making it possible to train large architectures on single GPUs. For a detailed discussion on TL and how it is used to mimic certain famous CNN architectures on different datasets, refer to Chapter 5 and Chapter 6.

The challenge of using these learning procedures is to obtain a clear balance between model complexity and model interpretability while selecting the correct hyperparameters, which can be seen as one of the most important trade-off in model building procedures having a direct influence on the accuracy of model prediction. These terms will be expanded on in the next chapter.

1.5 HARDWARE AND SOFTWARE CONFIGURATION

It is no secret that these networks are only as powerful as the computer's hardware capabilities allow them to be. The depth and size of a CNN depend on the hardware's memory capacity as each parameter must be stored in some shape or form. From a low-level introductory perspective, these parameters are mostly to be found in the form of matrix multiplications. These types of calculations give reason to why GPUs are preferred since they were designed for these type of operations (Inference and Learning, 2015) in contrast to *Central Processing Units* (CPUs).

Currently, there exist two major parallel computing platforms known as CUDA[4] and OpenCL[5]. The main difference between them is that OpenCL is open source and CUDA is proprietary. OpenCL is mainly supported by AMD and CUDA by NVIDIA which currently is the leader in the domain of DL. Therefore, it was decided by the researcher to train the CNN models in Python[6] using a GPU from NVIDIA, most notably the GeForce RTX 2080 SUPER[7]. The remaining hardware configurations used can be found in Table 1.2. Many different software libraries for the

[4]https://developer.nvidia.com/cuda-zone
[5]https://www.khronos.org/opencl/
[6]https://www.python.org/about/
[7]https://www.nvidia.com/en-gb/geforce/graphics-cards/rtx-2080-super/

implementation of DL models exist, however, the researcher will only make use of TensorFlow[8] and more specifically Keras[9]. TensorFlow is an open source ML platform that has been developed by Google whereas Keras is an application programming interface built on top of TensorFlow which has a simple design and can be understood with basic knowledge of Python programming without needing years of experience in the deployment of DL models. Related to the simplicity of use and model design, Keras also eases the integration of GPU acceleration when deploying models using the CuDNN[10] library from NVIDIA. Lastly, the researcher will make use of OpenCV[11] which is an open source CV library used to deploy real-time applications such as face detection and object recognition in Chapter 6. The remaining software configurations used can be found in Table 1.3.

Table 1.2: Hardware configurations that will be used in this thesis.

Component	Type
CPU	Intel Core i7-9600k @ 3.60GHz
Memory	G.Skill 3600 MHz 16GB (RAM) + 8GB GDDR6 (GPU)
GPU	GeForce RTX 2080 SUPER

Table 1.3: Software configurations that will be used in this thesis.

Library	Version
Python	3.7.4
TensorFlow	2.2.0
Keras	2.3.1
CUDA	10.1.2
CuDNN	7.6.5
Pandas, Numpy, Matplotlib, Scikit-learn, Seaborn	All up to date, 2020/06/01

1.6 REPRODUCIBILITY AND TRANSPARENCY

In order to ensure accountable and transparent reporting of results, all the code produced in this thesis will be done in the form of Jupyter notebooks. According to the Jupyter website: "The Jupyter notebook[12] is an open source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. Uses include data cleaning and transformation, numerical simulation, statistical modelling, data visualization, machine learning, and much more." These notebooks will then be exported to GitHub[13] which is an open software

[8] https://www.tensorflow.org/
[9] https://keras.io/
[10] https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html
[11] https://opencv.org/
[12] https://jupyter.org/
[13] https://github.com/

development platform. This will guarantee that all the results are reproducible for the reader and easily accessible. Having all results publicly available ensures honesty, integrity, objectivity, and openness which are the pillars of this research. An example of one of the outputs obtained in Chapter 6 can be found in Appendix A, as well as a short illustration on the output obtained by the Jupyter notebooks. The public shared code and documentation formulated in this thesis can be obtained at: `https://github.com/wlpretorius/Thesis.git`

1.7 STRUCTURE AND AIM OF THIS THESIS

1.7.1 Structure

This chapter serves as an introduction to the world of AI, ML, DL, and CV. Since CNNs are a combination of these mentioned fields, a thorough understanding of the theoretical components underlying each topic has to be gained through an in-depth discussion. These discussions will aid the image classifications done by the CNNs in Chapter 6.

Chapter 2 serves as a mathematical introduction to that of ML algorithms which will complement the general discussion on NNs and CNNs. In Section 2.1, a discussion on the statistical learning problem is given with a focus on classification tasks. This section introduces the notion of supervised and unsupervised learning while some mathematical notation is outlined as well as the general structure of ML algorithms, which in turn will complement the way in which CNNs are derived. Section 2.2 discusses the terms of model flexibility and complexity with an in-depth discussion on what it means when a model is underfitting or overfitting. This section also discusses the term that general NN models are associated with, that is, being a black box meaning that it is too complicated for simple tasks, i.e. it fits the given data too well. Section 2.3 provides a remedy to the previously mentioned problem through that of regularisation by penalising some objective function. Section 2.4 introduces accuracy metrics that will be used when CNNs are used for classification, which in turn will be applied in Chapter 6.

Chapter 3 introduces the ANN with an in-depth discussion on the certain structures underlying these models. The ANN is the backbone of a CNN, therefore, a thorough discussion of its inner workings is needed. Section 3.1 discusses the way in which simple statistical learning or ML algorithms can be related to NNs through that of linear regression, the Perceptron and the Logistic model. In this section, an activation function is introduced which could be seen as the most important part of a NN and CNN allowing for non-linear transformations of any given data. Section 3.2 discusses the general structure underlying CNNs while giving examples of different type of activation functions that are currently used in literature. Section 3.3 derives the way in which

ANNs are trained through a mathematical perspective by using backpropagation while introducing some important optimisation procedures needed to successfully train the ANN. Section 3.4 discusses ways to improve and accelerate the training of ANNs. This chapter is complemented by a literature overview in each section, easing the understanding of ANNs and paving the path to the introduction of CNNs.

Chapter 4 discusses general CNNs with an in-depth discussion and literature overview of their inner workings. Section 4.1 serves as a justification on why a CNN should be used for image classification tasks rather than that of ANNs. Section 4.2 describes the structure of this architecture. Section 4.3 derives the way in which CNNs are trained. Section 4.4 discusses regularisation techniques that exist in the literature which were specifically designed for the use in CNNs. Section 4.5 discusses solutions to the black box problem associated with CNNs through that of filter visualisations.

Chapter 5 serves as an introduction and literature review to current state-of-the-art architectures. Section 5.1 discusses the famous ImageNet dataset and the history of CNNs for image classification and object detection. Section 5.2 introduces famous and advanced state-of-the-art CNN architectures with an in-depth discussion on the structure and design of each network. Section 5.3 discusses TL which will complement the way in which the advanced architectures are trained for comparison purposes in Chapter 6. Section 5.4 discusses the different phases of design to build a CNN architecture. Section 5.5 introduces two custom-designed CNN architectures that will be used in the empirical chapter, Chapter 6, for comparison purposes against famous state-of-the-art networks on different datasets.

Chapter 6 is the empirical chapter where the custom-designed architectures are compared against state-of-the-art architectures through the use of TL. Section 6.1 outlines the goal of this chapter. Section 6.2 introduces the five different datasets that will be used to obtain certain comparisons on using different architectures. Section 6.3 discusses the experiments done on each dataset as well as the results obtained.

Chapter 7 is the overall conclusion of the thesis.

1.7.2 Aim

This thesis has several goals or objectives. The first main objective is to give a clear and concise outline of ANNs and more importantly CNNs. This goal will be reached by giving a thorough literature review that relates to current state-of-the-art architectures and DL methods. The understanding of NNs through current literature is a key factor in great research done on this field.

The work done in this thesis focuses mainly on CV methods such as image classification and object detection since these fields are experiencing major breakthroughs but also struggle with long-lasting optimisation procedures and activation functions. For this reason, five datasets will be chosen which are all related to image classification and object detection tasks to complement research done on existing optimisation methods and activation functions used in CNNs.

Additionally, to obtain a clear comparison of these aforementioned hyperparameters, two custom-designed architectures will be built with the aim to indicate the usefulness of general CNNs on datasets from different domains. There is no best single CNN architecture that works well for all data. Therefore, introducing a CNN with a general structure that one could use in different domains could prove beneficial for practical deployment of these models.

The design of these custom architectures introduces another objective, that is, showing that CNNs do not need a large number of parameters (millions) to reach a satisfactory test loss and accuracy. Here, the first architecture will be designed to be much deeper than the second architecture (which can be seen as a simple subset of the first architecture). This goal will be reached when the results, as given in Chapter 6, indicate how powerful the smaller designed architecture is.

To complement these results, an investigation on several claims made in the literature to that of using a certain optimisation method or activation function will be done by comparing state-of-the-art architectures to the two custom-designed architectures. This comparison will be allowed through the use of TL to assess different deep CNNs.

A third objective is to show that CNNs, in general, are not as complicated as they are set out to be. There exist techniques such as filter visualisation to reach this goal. A thorough investigation of these techniques will be done on a specific dataset in Chapter 6, to aid the understanding of the mechanism by which these models identify certain features in images.

Another objective is based on the datasets mentioned, that is, if the designed CNNs can generalise well in different domains. Of the five datasets that will be introduced, two of these datasets can be seen as the core of this thesis which could contribute to research done in these domains, that is, the medical domain and the facial recognition domain. For this reason, a dataset on malaria cases will be used to identify whether a person has indeed been infected or not. The study on this dataset could potentially save lives in rural areas where resources are scarce since the deployment of a CNN is far less resource-intensive than that of manual interpretation of blood samples done by doctors.

The second most important dataset is that of classifying human emotions from a facial emotion recognition dataset. The goal here is to show how a simple CNN can be as powerful as

state-of-the-art facial recognition applications, as well as, showing how simple objects contained in images can be located and identified automatically.

CHAPTER 2
BACKGROUND

2.1 THE IMPORTANCE OF THE STATISTICAL LEARNING PROBLEM: A SUPERVISED APPROACH FOR CLASSIFICATION

Statistical learning theory, often used as a synonym for ML, is the challenge of finding a mathematical predictive function for a given dataset. This concept uses specific functions or algorithms to map complex inputs to simple desired outputs. In this chapter, a mathematical background is provided based on the framework of a supervised classification approach to model building which in turn will aid the understanding of NNs.

An input, predictor, or feature is known as an observation or event from the data that can be measured or written in some numerical form, represented by a vector $\boldsymbol{x} \in R^p$ where each sample or element x_j of the vector is an observed value for the j^{th} feature; $j = 1, ..., p$. In an image classification task, these features are usually the colour intensity values of the pixels (e.g. 0-255).

A data or image matrix used to learn a particular algorithm is denoted by $X : N \times p$ where N is the number of observations represented by each row in the matrix and p is the features of each observation represented by the columns:

$$X = \begin{pmatrix} x_{11} & \dots & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{N1} & \dots & x_{Np} \end{pmatrix}. \tag{2.1}$$

These are the inputs used to map a specific function, denoted as $f(\boldsymbol{x})$, to the desired output, denoted by the vector $\mathbf{y}$. In a supervised classification approach, an output is referred to as a label, category, target, or class associated with a feature, i.e. each element y_i is associated with its corresponding feature in the i^{th} row of X, represented as:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}. \tag{2.2}$$

In a problem where there are multiple labels or targets, K, the matrix representation $Y : N \times K$ is used. These labels can consist of unordered classes for example 'dog' or 'cat' and usually have numerical codes associated with it such as 0 for a dog and 1 for a cat. They can also be ordered,

for example, handwritten digit codes will have the form of $K = 0, ..., 9$.

Generally, any supervised learning problem can be seen as a function approximation problem (Hastie *et al.*, 2013), denoted by:

$$Y = f(X, \boldsymbol{\Theta}) + \boldsymbol{\epsilon}. \tag{2.3}$$

where $f(X, \boldsymbol{\Theta})$ is the underlying model used to predict some outcome Y, $\boldsymbol{\epsilon}$ is a random error vector (having mean 0 and standard deviation 1) containing terms that are irreducible due to the noise present in the labels and $\boldsymbol{\Theta}^T$=$[\theta_1, ..., \theta_p]$ which are the parameters, coefficients, or weights associated with the model. This concept then requires a loss function $L(Y, f(X))$ to penalise the error or misclassifications in its predictions. The simplest model form is known as the linear model. This model is one of the main building blocks of a NN and many other algorithms, whereby a complete understanding of its inner workings is needed. Here, $f(X, \boldsymbol{\Theta})$ is estimated by $\hat{f}$ which is known as the learner or classifier.

In a linear regression setting where $f: R^p \rightarrow R$, the output or response ($\mathbf{y}$) is regarded as a linear function of the input, $\boldsymbol{x}$ (a plane in graph theory), and can be approximated by the function:

$$\hat{\mathbf{y}} \cong \boldsymbol{x}^T \hat{\boldsymbol{\theta}} + \hat{\theta}_0, \tag{2.4}$$

where $\hat{\mathbf{y}}$ contains the predictions; $\hat{\boldsymbol{\theta}}$ is the vector of weights, or coefficients guiding the structure underlying the model; $\boldsymbol{x}^T$ is the vector of inputs which often includes a constant variable 1 such that $\boldsymbol{x}^T$=$[1, \theta_1, ..., \theta_p]$ and $\hat{\theta}_0$ is the intercept or bias which can be included in $\hat{\boldsymbol{\theta}}$ where:

$$\hat{\mathbf{y}} \cong \boldsymbol{x}^T \hat{\boldsymbol{\theta}}. \tag{2.5}$$

To analyse how precise the model was in its predictions a loss function is needed to estimate the errors in its calculations. This function needs to comply with certain properties such that it needs to have a partial derivative. For the linear regression setting, this function is called the *Mean Squared Error* (MSE) given by:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = (\mathbf{y} - X\boldsymbol{\theta})^T (\mathbf{y} - X\boldsymbol{\theta}). \tag{2.6}$$

The advantage of this function is that it has only one global minimum and that it indicates how (dis)similar the predicted output values are to the actual response values. Now, to ensure that the model is improving in prediction, i.e. it reaches an optimal solution, a regular update in its coefficients or weights is needed in order to guarantee that the value of the loss function, MSE, is

as small as possible for the given dataset. For a linear regression model, it is possible to find the exact solution by equating the partial derivative of the MSE function with respect to $\boldsymbol{\theta}$, i.e. taking $\frac{\partial MSE}{\partial \boldsymbol{\theta}} = 0$. Hence, if $X^T X$ is non-singular, the resulting global minimum and optimal solution are given as: $\hat{\boldsymbol{\theta}} = (X^T X)^{-1} X^T \mathbf{y}$ where the weights are optimised respectively.

Unfortunately, this result does not hold for all algorithms in general. It is very seldom that a model can be approximated with some function. The fact is that more complex models require more complex loss functions, some of which cannot guarantee to find a global minimum and usually settles for a local minimum (see Figure 2.1). In this case, the optimisation procedure has to be iterative. Many of the iterative procedures belong to a group called gradient-based optimisation, which will be discussed next with additional methods in the next chapter.

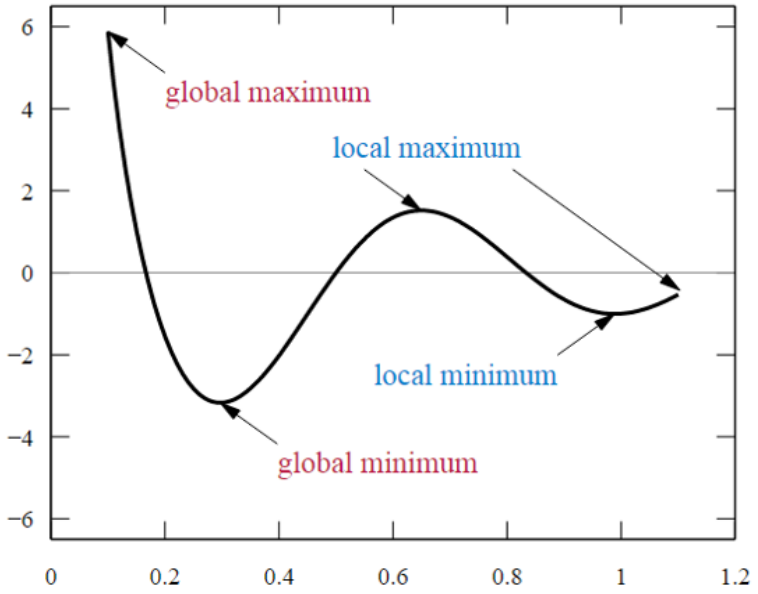


Figure 2.1: Illustrating the difference between the global maximum, global minimum, local maximum, and local minimum.
Source: Statinfer (2017) [Online].

Consider now, a classification model. If the notation is adjusted to where G is a set of categorical discrete outputs, then $|G| = K$ denotes the number of discrete categories in the dataset. The classification of a supervised learning task can be categorised into:

- Binary: The input is only related to one of two classes, $K = 2$.
- Multi-class: There exist more than two classes for each label, but where each sample can only be labelled as one class, for example, fruits can either be an apple, orange, or grape and not all at the same time.

- Multi-label: There exist multiple classes for multiple labels.

This thesis will only consider binary and multi-class classification applications. The learning problem according to Hastie *et al.* (2009) for a multi-class classification problem is then adjusted to predicting G, denoted as $\hat{G}$, where $f\colon R^p \rightarrow (1, ..., K)$ needs to be calculated using the given inputs situated in X. Usually, G is represented as a dummy or indicator variable where $G = \begin{cases} 1, \text{ contains class } K \\ 0, \text{ otherwise} \end{cases}$, implying that $Y_G\colon K \times 1$ have all elements zero except in the G^{th} position which is indicated as 1, hence $Y_k = \begin{cases} 1, \text{ if } k = G \\ 0, \text{ if } k \neq G \end{cases}$, with $k = 1, ..., K$. This notation makes it possible to use the elements of Y_G as quantitative outputs or labels, meaning the values of $\hat{Y}_k = [\hat{Y}_1, ..., \hat{Y}_K]$ can be predicted. The classifier is then reduced to $\hat{G} = \max_{k \in (1,...,K)} \hat{Y}_k$ which relates to the class with the highest predicted value.

In a linear function approximation setting the problem reduces to the estimation of accurate class scores, given by $\hat{Y}_k = \hat{f}_k(X) = \boldsymbol{x}^T\hat{\theta}_k + \hat{\theta_{k0}}$ where $\hat{f}_k$ is an estimate of the true function f_k which contains the actual relationship between the inputs and output of class k. This function partitions the input space into different sub-regions which is formed by the linear decision boundary (see Figure 2.2 for an example of a binary-class problem). This decision boundary is the line between classes k and classes l whereby it is the of points for which $\hat{f}_k(\boldsymbol{x}) = \hat{f}_l(\boldsymbol{x})$. This set of points then form an affine map or hyperplane in the given input space, given by $\{\boldsymbol{x}\colon(\hat{\theta_{k0}} - \hat{\theta_{l0}}) + (\hat{\theta}_k - \hat{\theta}_l)\boldsymbol{x}^T = 0\}$. If the weights are then calculated from the data observations, $\boldsymbol{x}$, can be classified by computing $\hat{f}_k(\boldsymbol{x}) = \boldsymbol{x}^T\hat{\theta}_k$ for $k = 1, ..., K$ and then finding $\hat{G} = \max_{k \in (1,...,K)} \hat{f}_k(\boldsymbol{x})$.

However, a problem can occur when the predicted class scores are considered in terms of posterior probabilities, i.e. when $P(G = k|X = \boldsymbol{x}) \cong \hat{f}_k(\boldsymbol{x})$ since these predicted class score values are not the best estimates of the posterior probabilities due to the values falling outside of the [0,1] interval which violates simple probability laws. To overcome this problem, one can use the Logistic model and the logit transformation to obtain:

$$P(G = k|X = \boldsymbol{x}) \cong \frac{e^{\hat{f}_k(\boldsymbol{x})}}{\sum_{l=1} e^{\hat{f}_l(\boldsymbol{x})}}. \tag{2.7}$$

This function is called the Softmax function and ensures that the estimates of the posterior probabilities lie in the [0,1] interval. Now, finding a loss function for this model is not as straight forward as in the linear model case, since no closed-form solution for the weights exists. This brings forth the instance where a gradient-based optimisation approach is needed, through minimising the negative

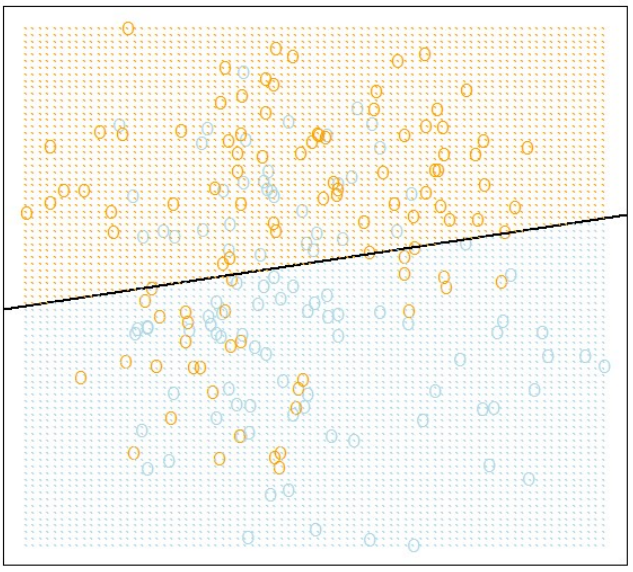


Figure 2.2: A linear decision boundary, $\boldsymbol{x}^T\hat{\theta_k} + \hat{\theta_{k0}} = 0.5$, for two classes, blue = 0 and orange = 1, with their respective input space.
Source: Hastie *et al.* (2009).

log-likelihood function by iteratively searching for these minimum values.

For example, consider the Logistic model for a binary classification task. It follows that as before: $\mathbf{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{\theta}$ with training data (x_i, y_i) where y_i are the labels having $y_i \in (0, 1)$ for each given input x_i. Here: $X = \begin{pmatrix} 1 & x_{11} & ... & x_{1p} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_{N1} & ... & x_{Np} \end{pmatrix}$, where x_{ij} denotes the j^{th} feature for the i^{th} observation with $x_{i0} = 1$ as before, with $\boldsymbol{x}^T = \begin{bmatrix} 1 & x_{i1} & ... & x_{ip} \end{bmatrix}$ and $\boldsymbol{\theta}^T = [\theta_1, ..., \theta_p]$. The goal is to predict the probability that a given example belongs to the class 1 or class 0. The functional form of this problem is given as:

$$P(y_i = 1|x_i, \boldsymbol{\theta}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{1 + e^{\boldsymbol{x}^T\boldsymbol{\theta}}} = \sigma(\boldsymbol{x}^T\boldsymbol{\theta}), \tag{2.8}$$

$$P(y_i = 0|x_i, \boldsymbol{\theta}) = 1 - P(y_i = 1|x_i, \boldsymbol{\theta}) = 1 - f_{\boldsymbol{\theta}}(\boldsymbol{x}). \tag{2.9}$$

Here, $\sigma(\boldsymbol{x}^T\boldsymbol{\theta})$ is referred to as the Sigmoid or Logistic function, which has an S-shape (non-linear) taking on values that are squeezed into the [0,1] interval, easing the interpretation of $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ as a probability. A simple plot of $\sigma(\boldsymbol{x}^T\boldsymbol{\theta})$ is given in Figure 2.3. The graph indicates that:

- $\sigma(\boldsymbol{x}^T\boldsymbol{\theta}) \to 1$ as $\boldsymbol{x}^T\boldsymbol{\theta} \to \infty$.
- $\sigma(\boldsymbol{x}^T\boldsymbol{\theta}) \to 0$ as $\boldsymbol{x}^T\boldsymbol{\theta} \to -\infty$.
- $0 \leq \sigma(\boldsymbol{x}^T\boldsymbol{\theta}) \leq 1$.

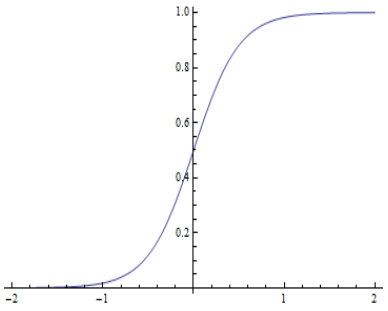


Figure 2.3: Illustration of the Sigmoid function.
Source: Kyurkchiev and Markov (2015).

The logit function or log-odds are such that:

$$\log\left(\frac{P(G=1|X=\boldsymbol{x})}{P(G=2|X=\boldsymbol{x})}\right) = \boldsymbol{x}^T\boldsymbol{\theta}. \tag{2.10}$$

The loss function is then given by taking the log-likelihood as:

$$L(\boldsymbol{\theta}) = -\sum_i \left[y_i \log(f_{\theta_i}(x_i)) + (1-y_i)\log(1-f_{\theta_i}(x_i))\right]. \tag{2.11}$$

This loss function, which is called the binary cross-entropy loss, is used here to train the model to classify the test labels as either class 1 or 0, by examining which of the two classes are most probable, i.e. if an output is such that $P(y_i = 1|x_i, \boldsymbol{\theta}) > P(y_i = 0|x_i, \boldsymbol{\theta})$ it will be labelled as 1. This is the same as checking whether $f_{\boldsymbol{\theta}}(\boldsymbol{x}) \geq 0.5 \to \hat{\mathbf{y}} = 1$ or $f_{\boldsymbol{\theta}}(\boldsymbol{x}) < 0.5 \to \hat{\mathbf{y}} = 0$ on the decision boundary. The classes can be transformed into labels, such as cat or dog, yes or no, and so forth depending on the dataset. The advantage of this function is that it is convex meaning it will have one minimum and no local minimum that it might remain stuck in and that it is differentiable and easy to optimise. However, no easy computable closed-form solution exists as in the case of the linear model so an iterative descent method is needed to find the optimal values. The aim remains to minimise $L(\boldsymbol{\theta})$, so the derivative can be calculated and set equal to 0. The derivative of $L(\boldsymbol{\theta})$ is given by the following:

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \theta_j} = -\sum_i x_{ij}(f_{\theta_i}(x_i) - y_i), \tag{2.12}$$

and in vector form, it follows that:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = -\boldsymbol{x}\left(\mathbf{y} - \sigma(\boldsymbol{x}^T\boldsymbol{\theta})\right). \tag{2.13}$$

To find the optimal parameters while reducing the loss, a standard gradient descent algorithm can be used. This algorithm finds the minimum by using steps proportional to the negative of the gradient of the function at the current point taken (moves down in the direction of the steepest slope) (see Figure 2.4), and it also updates the weights while moving along the gradients. The algorithm is given in equation form as:

$$\theta_{j+1} = \theta_j - \eta \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}). \tag{2.14}$$

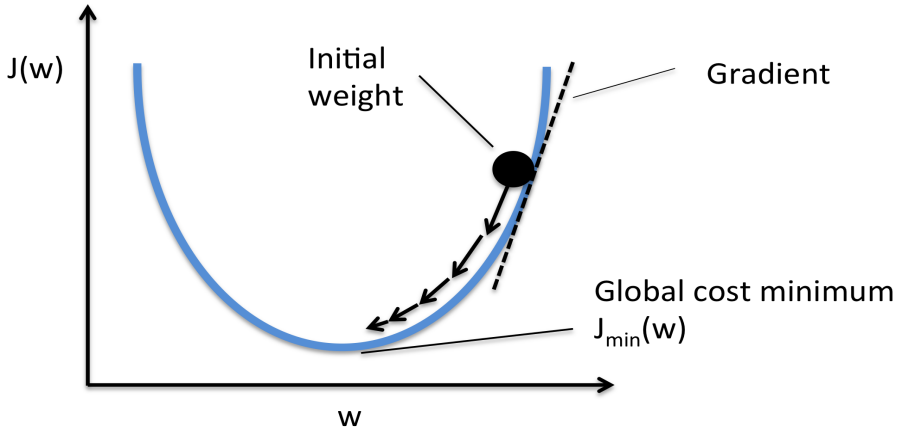


Figure 2.4: Illustrating the way in which vanilla gradient descent finds optimal loss values, here $J(w) = L(\theta)$.
Source: Durán (2019).

The algorithm cycles as follows:

1. While true,
2. $\theta_j = \theta_{j-1} - \eta \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$,
3. if $\|\theta_j - \theta_{j-1}\| < \epsilon$, then break.

Here, $\eta \geq 0$ is the learning rate or step size set by the user with θ_j initiated randomly. There is a trade-off in choosing the learning rate since a large learning rate implies that more weight is put on the derivative such that large steps are made at each iteration. A smaller learning rate implies that less weight is put on a derivate, and if the learning rate is too small the algorithm will take longer to converge whereas if the learning rate is too large the algorithm might miss the

optimal parameter values by remaining stuck in local minimum. This method of gradient descent calculates the gradient for the whole training set to output one update. This can be a very slow process which will be infeasible for large datasets, especially for NNs since the loss of multi-layer NNs are non-convex so the algorithm might remain stuck in many local minimums. The solution to this problem is brought forward by using *Stochastic Gradient Descent* (SGD), which updates the parameters after each training example (x_i, y_i) passes through the algorithm by means of batches of randomly chosen training examples. This method is usually much faster, where the algorithm is given in equation form as follows:

$$\theta_{j+1} = \theta_j - \eta \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}, (x_i, y_i)). \tag{2.15}$$

The algorithm cycles as follows:

1. While true,
2. i = random index,
3. $\theta_j = \theta_{j-1} - \eta \nabla(\theta_{j-1}, (x_i, y_i))$,
4. if $\|\theta_j - \theta_{j-1}\| < \epsilon$, then break.

However, SGD also has its problems due to its stochastic nature as it might have tendencies to overshoot the exact minimum, but it has been shown that if the learning rate is slowly decreased, SGD will almost certainly converge to a local or global minimum for non-convex and convex functions, respectively (Ruder, 2016). There exist many different types of gradient descent optimisations, some of which will be viewed in more detail in the following chapters.

In a multi-class setting, the multinomial Logistic model becomes of interest which is taken as the Logistic model with multiple labels. Here, $y_i \in (1, ..., K)$ with $P(y_i = k|x_i, \boldsymbol{\theta})$ where $k = (1, ..., K)$ are the different classes and the goal is to estimate the probability of each class label occurring. The output will be a K-dimensional vector. As stated earlier, the Softmax function is used as the logit transformation. In this case:

$$f_k(\boldsymbol{x}) = \frac{e^{f_k(\boldsymbol{x})}}{\sum_l e^{f_l(\boldsymbol{x})}}, \tag{2.16}$$

where $f_l(\boldsymbol{x})$ is the scores inferred by the model for each class in k. This loss function is called the categorical cross-entropy loss and is written in terms of the Softmax function as:

$$L(\boldsymbol{\theta}) = -\sum_{i}^{k} y_i \log(f_k(\boldsymbol{x})), \tag{2.17}$$

where $\boldsymbol{\theta}$: $p \times K$ and the loss is minimised using a gradient descent method to reach an optimal solution since it is also a convex function and differentiable. An important note is that in this setting, the output label needs to be one-hot encoded (see Figure 2.5) when training the model using appropriate software, meaning that if $y = 3$, it needs to be converted to $y_{one-hot} = [0, 0, 1, 0, 0]$ for a 5-class classification problem. Software, such as Keras, do this in one line of code.

Label		0	1	2	3	4	5	6	7	8	9
4	Hot Encode ->	0	0	0	0	1	0	0	0	0	0
5		0	0	0	0	0	1	0	0	0	0
2		0	0	1	0	0	0	0	0	0	0
6		0	0	0	0	0		1	0	0	0

Figure 2.5: One-hot Encoding a 10-class classification problem with labels $y = (4, 5, 2, 6)$.

The discussion above can be generalised to where most ML algorithms follow the same structure or building blocks. The general structure can be described as a composition of the following: a dataset, model, loss function, and optimisation procedure. The most important aspect then, for practical purposes, is to ensure that the algorithm can generalise on unseen data indicating that these algorithms are much more than just an optimisation problem. The generalisation performance of the learning algorithm guides the choice of model selection and provides a measure of quality for the final chosen model. The challenge, however, is that researchers or users do not always have access to unseen data. A common trick in the statistical learning process is to then randomly split a part of the dataset into a training set and to keep or hold the other part of the data as a test or validation set which will be independent of the training data.

In data-rich situations, it is usually the case that the dataset is randomly divided into three parts; training, validation, and test. The training set, $\Im = [(x_i, y_k), i = 1, ..., N, k = 1, ..., K]$, is the dataset which is used to build and fit different models outputting the training error, $\overline{err} = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{f}_k(x_i))$. Thereafter, the validation set is used to estimate the prediction error, $L(y_i, \hat{f}_k(x_i)|\Im)$, for model selection. After choosing the final model, the test set is used to assess the test or generalisation error, $Err_{\Im} = E[L(y_i, \hat{f}_k(x_i)|\Im)]$, of the final chosen model. As an example:

- Training set: A set of examples used for learning, that is to fit the parameters of the classifier.
- Validation set: A set of examples used to tune the parameters of a classifier, for example, to choose the number of hidden units in a neural network.

- Test set: A set of examples used only to assess the performance of a fully specified classifier.

The ratio of how the data should be split into two or three parts (if one has access to a large dataset) depends on the signal-to-noise ratio in the data, and training sample size, so no explicit rule on how to split the data exist as long as the model is assessed on some independent dataset related to the training data. The results obtained in this thesis, of Chapter 6, will make use of both training and test sets, as well as training, validation, and test sets with the ratio of the splits made arbitrarily by the researcher. Note, that the difference between test error and test loss is that the error function measures the change of an observable value from a prediction of the model, whereas the loss function operates on this error to quantify the negative effect of an error (the loss is the value of the objective function that is being minimised), meaning that an error is given as a value of less than 1 while loss can take on a value of greater than 1.

The discussion above indicates two separate goals in model learning procedures, namely the model selection part and the model assessment part, which emphasises how important the statistical learning problem is since these concepts directly influence model complexity and model interpretability. Choosing the optimal trade-off between complexity and interpretability is a crucial aspect in model building procedures.

2.2 MODEL COMPLEXITY AND MODEL INTERPRETABILITY

There are two reasons why a model would not perform well on unseen data, i.e. its generalisation error is very high. These reasons depend on the representational capacity (ability to fit a variety of functions) of a model or algorithm (Goodfellow *et al.*, 2016). The first reason is when a model does not have enough capacity and underfits the underlying function, and the second reason is when a model has too much capacity and is overfitting the underlying function (see Figure 2.5).

If a model is underfitting, then it has failed to obtain a low enough error rate on the training data. This means that the model was not complex enough to capture the input-output relationship of the data resulting in a model with high bias and low variance. In contrast to underfitting; if the difference between the training-and-test error is too large, the model is said to be overfitting meaning too much training has been done to find patterns in the training data (performs extremely well on the training data and very poor on the test data), which is the result of picking patterns that are caused by random noise (Hastie *et al.*, 2013). In this case, the model is too complex having a large variance and low bias.

The ideal model (optimal) should provide a clear balance between the bias and variance trade-off to ensure it has minimised the test error while maintaining as much complexity and

interpretability as possible (see Figure 2.6). If a model is too complex (usually used as a synonym for overfitting), it has less interpretability and is referred to as a 'black box'. NN models are often seen as black boxes because of their highly non-linear structures taking on many parameters at once, however, this does not mean that these models are not useful. It only means that the model will not give any insight into the exact form of $f_k(\boldsymbol{x})$, indicating that they are still useful in many prediction tasks since they are usually highly accurate in their estimates. There exist different ways to combat the idea of a black box in a NN setting, especially in CNNs, by visualising the different convolutional layers. This idea will be discussed in more detail, in Chapter 5 and Chapter 6.

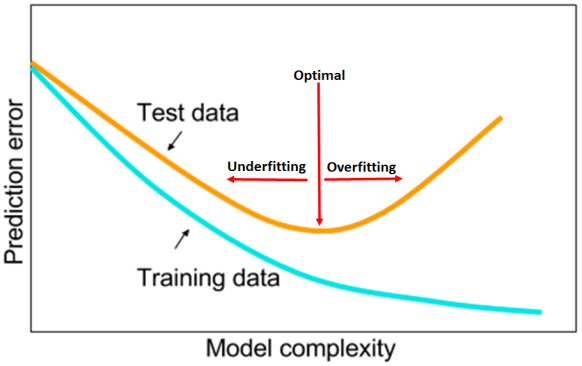


Figure 2.6: The trade-off between underfitting and overfitting. Model complexity (x-axis) refers to the capacity of a model. This figure also shows the typical shape of training-and-test error. Source: Hastie *et al.* (2009).

In an overfitting and underfitting trade-off setting, there exist a way to control model complexity, to improve the model's generalisation ability, called regularisation which introduces another hyperparameter in the model selection task.

2.3 REGULARISATION

Regularisation is referred to as a strategy to reduce the generalisation error and not the training error, through specific modifications made to the learning algorithm (James *et al.*, 2013). According to Hastie *et al.* (2009), the idea is to optimise the loss over a large class of functions where each function is penalised in some way to avoid overfitting, usually through the use of constraints on the model. The optimisation problem becomes:

$$\min_{f \in H} \left[\sum_{i=1}^{N} \underbrace{L(y_i, f(x_i, \theta_i))}_{\text{Loss}} + \underbrace{\lambda J(\boldsymbol{\theta})}_{\text{Penalty}} \right], \tag{2.18}$$

where the first term measures the closeness of the data; the second term is known as the penalty

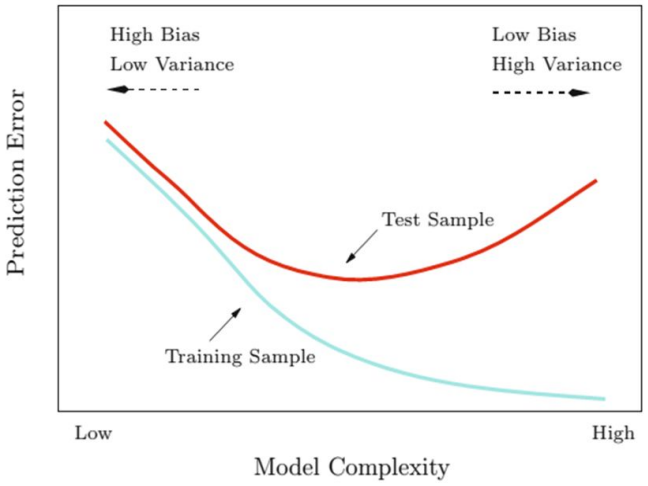


Figure 2.7: Additional representation of the trade-off between underfitting and overfitting. Model complexity (x-axis) refers to the capacity of a model. This figure also shows the typical shape of training-and-test error.
Source: Hastie *et al.* (2013).

or regularisation term and λ is a fixed (non-negative, $\lambda \geq 0$) smoothing or tuning parameter also known as the weight decay establishing a trade-off between the two terms ('Loss + Penalty'), i.e. controlling the complexity. Here, H is the set of functions on which $J(.)$ is defined. λ regularises the function, algorithm, or model if:

- $\lambda = 0$: The original loss function is obtained, and no penalty is introduced, meaning the algorithm has the potential to overfit.
- $0 < \lambda < 1$: A penalty is introduced, model weights are forced closer to zero as λ increases, increasing bias and decreasing variance, resulting in a reduction in complexity.

The challenge is to find the value of λ resulting in an optimal model (reducing variance while not overly increasing bias). Therefore, λ is considered as a hyperparameter, and in this thesis, λ will be manually chosen.

The optimisation problem above leads to one of the most used regularisation techniques in ML known as L2 regularisation. These techniques, together with data augmentation, dropout, and early stopping, which are specifically designed regularisation methods for NNs, will be discussed in-depth in Chapter 3 and Chapter 4.

Now, after the hyperparameters have been adjusted and the appropriate model has been selected, the model will be tested on some independent dataset related to the inputs. After this has been done, researchers usually require some additional information on how the classifier has performed with respect to the target classes in the dataset (not only through the use of a test error).

A way this can be done, to extract additional information on the classification performance of a model, is through the use of a confusion matrix which outputs certain useful classification metrics.

2.4 THE PERFORMANCE OF A CLASSIFICATION TASK THROUGH A CONFUSION MATRIX

A confusion table or matrix is a summary of the prediction results obtained in a classification problem. It showcases the different ways in which a model or classifier was confused in its predictions while giving insights on the type of errors that were made during a classification procedure. A confusion matrix for a binary classification problem has the following form in Table 2.1:

Table 2.1: Illustration of a confusion matrix.

		Predicted ($\hat{y}$)		
		$\hat{y}=0$	$\hat{y}=1$	Total
True (y)	$y=0$	TN	FP	n_0
	$y=1$	FN	TP	n_1
	Total	$\hat{n_0}$	$\hat{n_1}$	n

The following formulas are of importance:

- TN = True Negative; FN = False Negative; FP = False Positive; TP = True Positive.
- $Accuracy = \frac{TN+TP}{n}$; The proportion of times the classifier was correct.
- $Missclassification\ rate = 1 - Accuracy = \frac{FN+FP}{n}$.
- $Precision = \frac{TP}{\hat{n}_1}$; The proportion of the predictions that were truly positive.
- $Recall = \frac{TP}{n_1}$; The proportion of positive classes correctly detected.
- $F\text{-}Score = \frac{2 \times Recall \times Precision}{Recall + Precision}$; A higher proportion indicates higher predictive power.

It follows that an ideal classifier will return a high precision and recall value. However, the importance of a metric depends on the type of data. If the data is for instance related to a medical setting, recall will be more important since a high recall relates to a low false negative rate, for example, if a patient is positively classified for an infectious disease then the occurrence of a false negative is unaccepted since the patient could be discharged, thus mixing with a healthy population. If the data is related to a recommendation setting, a high precision will be of more importance, for example, when classifying if an email is spam or not then a low false positive rate will be the goal, since it will be better to have some spam emails in an inbox than having regular mails classified as spam.

In the multi-class classification case, the confusion matrix is adjusted to many binary problems

separated for each individual class, i.e. the output is binarised, and the same metrics are used. This table, as well as the binary case, is easily obtained through the use of the Python library, Scikit-Learn. Here, the average number of correct predictions, accuracy, is measured using:

$$\frac{1}{N}\sum_{k=1}^{K}\sum_{x:Y_k} I(Y_k = \hat{Y_k}), \tag{2.19}$$

such that the indicator returns 1 if the classes match or 0 if otherwise.

2.5 SUMMARY

In this chapter, a brief overview of the statistical learning problem that will be dealt with in the rest of the thesis was provided. Important keywords such as supervised, unsupervised, training error, test error, test loss, Logistic function, Softmax function, cross-entropy, optimisation, one-hot encoding, model complexity and interpretability, overfitting and underfitting, regularisation, and accuracy metrics were introduced. This chapter serves as a necessary background for the understanding of the mathematical fundamentals needed to construct NN architectures.

CHAPTER 3

ARTIFICIAL NEURAL NETWORKS

3.1 THE PERCEPTRON AND THE LOGISTIC MODEL AS AN INTRODUCTION TO NEURAL NETWORKS

The inspiration of the Perceptron model introduced by Rosenblatt stemmed from the McCulloch-Pitts unit which was designed to mimic the computational abilities of the human nervous system. Considered as the first type of ANN, this model followed the same structure as the Perceptron model (see Figure 3.1), however, the problem was that this unit only considered logical functions with no free parameters that can be adjusted and thus no learning was performed. Nonetheless, this unit still had the capabilities to compute any logical function (McCulloch and Pitts, 1943).

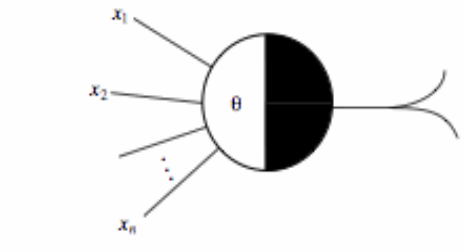


Figure 3.1: Illustrating the McCulloch-Pitts unit with inputs x_i and threshold θ.
Source: Kalita (2014) [Online].

Considered as one of the simplest ANNs with only a single neuron, the Perceptron was designed to introduce weighted connections between units with an algorithm that updates these weights to ensure that the network is learning (Rojas, 2013). The theoretical goal of the Perceptron was to minimise the distance of misclassified points to the decision boundary in order to find a separating hyperplane in the training data (if the data was linearly separable).

A hyperplane or straight line in 2-dimensional space has the form: $f(x_i) = x_i\theta_i + \theta_0$ which complements the general form of a single-layer (inputs are usually not regarded as a layer) ANN (see Figure 3.2) where x_i are the inputs $(i = 1, ..., n)$ with associated weights $\theta_i = \omega_i$, the bias is $\theta_0 = b$ (which is added otherwise the line will be fixed at the origin) and the output (which is binary in this case) is $y_i = f(x_i)$. This notion can be formulated as $y_i = \sigma(\sum_{i=1}^{n} \omega_i x_i + b)$ indicating that y_i is only activated if some threshold is exceeded. The threshold is:

$$y_i = \sigma(.) = \begin{cases} 1, \text{ if } \omega_i x_i + b. \\ 0, \text{ otherwise.} \end{cases} \tag{3.1}$$

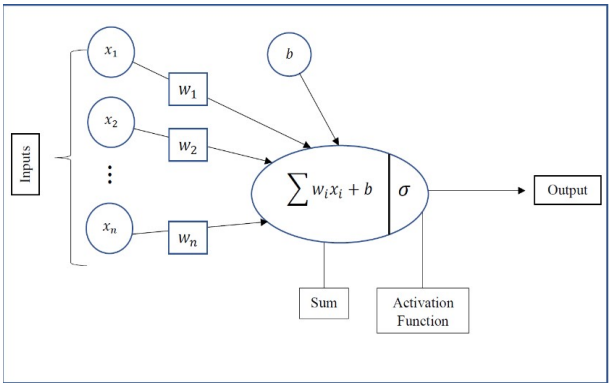


Figure 3.2: Illustration of the components of a simple ANN.

Here the activation function, $\sigma(.)$, provides a binary output using a simple step function, or more formally, the Heaviside step function (see Figure 3.3) named after the mathematician Oliver Heaviside (1850–1925). The activation function is used to convert the weighted sum of the neuron to an output. In NN architectures, this output is served as an input to the next layer (see Figure 1.8). In this case, the activation function linearly splits the hyperspace into two categories; the linear combination that exceeds the threshold and the linear combination that does not exceed the threshold. There however exist many different activation functions, of which some important ones regarding the training of NNs will be studied in this chapter.

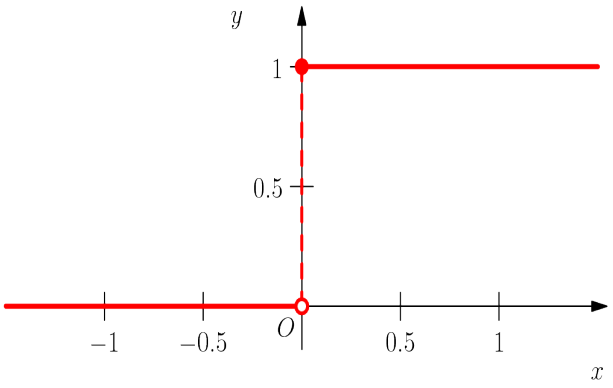


Figure 3.3: Illustrating the Heaviside activation function.
Source: Kyurkchiev and Markov (2016).

The Heaviside step function is particularly important for the Perceptron which allows that during training the weights are learned while a linear combination of the input features is computed. The training of these weights is performed together with the minimisation of the loss function. Adjusting the notation found in Hastie *et al.* (2009), the loss function is given as $L(\omega_i, b) = -\sum_{i \in M} y_i(\omega_i x_i + b)$ where M is the set of misclassified points. Related to the discussion in Chapter 2, this quantity is proportional to the distance of the misclassified points to the decision boundary. Also, SGD is

used to find the optimal separating hyperplane. The gradients are computed as:

$$\frac{\partial L(\omega_i, b)}{\partial \omega_i} = -\sum_{i \in M} y_i x_i. \tag{3.2}$$

$$\frac{\partial L(\omega_i, b)}{\partial b} = -\sum_{i \in M} y_i. \tag{3.3}$$

The updates of the weights while minimising the loss are then in the form:

$$\begin{pmatrix} \omega_i \\ b \end{pmatrix} \leftarrow \begin{pmatrix} \omega_i \\ b \end{pmatrix} + \eta \begin{pmatrix} y_i x_i \\ y_i \end{pmatrix}. \tag{3.4}$$

$$\omega_{i+1} = \omega_i + \eta \nabla L(\omega_i, b), \tag{3.5}$$

where η is again the learning rate which determines how fast the parameters update. If the classes are linearly separable then the algorithm will converge. The practical goal Rosenblatt had in mind was to introduce a complex model for image recognition by using only a single-layer of Perceptron's. This model was however very limited, as analysed by Minsky and Papert (2017) where they found that the Perceptron could not solve the XOR function due to it only being useful in problems where the data is linearly separable, i.e. it is only capable of separating data points with a single line. Additional problems were brought forward such that even when the data was linearly separable the algorithm might output many solutions depending on the starting value chosen (Ripley, 2007). The structure of the algorithm, to date, remains to be the building block of many different NN architectures.

Evidently, for a classification problem, the activation function used in the Perceptron model would not output any meaningful probability prediction values due to the lack of continuity incorporated in the function. An improved choice of activation function would be to use the Sigmoid function for binary classification. Using this function, brought forth the Logistic model, which can be depicted as another 'network' having a single-layer. The Logistic model is composed of the linear model $f(x_i) = \omega_i x_i + b$ and the Sigmoid function $g(f) = \frac{1}{1+e^{-f}}$ transforming the predictions to lie in a [0,1] interval. The network of the model is in Figure 3.4.

The inputs are given as x_i with associated weights ω_i and bias b to obtain an improved decision boundary. This layer is fed into the single neuron which is the function composed as $g(f)$ serving as the activation function, i.e. firing an output if some threshold is exceeded by passing through the linear combination of inputs. The only difference between the Logistic model and the Perceptron is the activation function being used.

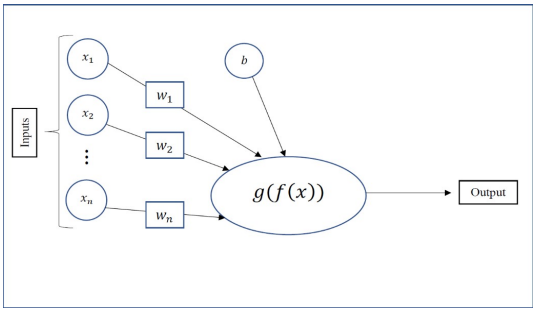


Figure 3.4: Illustrating a single-layer NN where weights and biases passes through an activation function.

Both models have connected their inputs to an output where an activation function is used in the last layer (a single-layer, again emphasising that the inputs are not regarded as a layer) to make some prediction (compare Figures 1.8, 3.1, and 3.2). Both models also have linear decision boundaries represented as a hyperplane. These linear decision boundaries could prove costly for data that is not linearly separable, especially data from real-world applications which intuitively is less likely to be linearly separable. This problem vanishes when a network includes additional layers, known as hidden layers, with non-linear computations from suitable activation functions in each layer which should also be non-linear. This network is called a multi-layer Perceptron although it has many names such as an ANN, a multi-layer NN, and a feedforward NN.

This network is nothing other than a non-linear model and depending on how they are constructed a NN can approximate any continuous linear or non-linear function using a single-hidden-layer, i.e. NNs are universal approximators (Cybenko, 1989). Statistically speaking, other non-linear models exist but in order to use them, a similar structure in the data needs to be observed. This is not the case with a NN as it can learn any structure, complex or not, using subsequent hidden layers depending on how complex the data is. It only needs a single-hidden-layer to approximate any function, but in practice to obtain a better approximation one usually add more hidden layers such that the model can learn more detail regarding the problem at hand.

It follows now, that a NN with no hidden layers and without an activation function (i.e. the identity function as an activation function) is a ordinary linear regression model, and a NN with no hidden layers and the Sigmoid function as activation is a Logistic model (Bishop, 2006). These linear models end up solving convex optimisation problems for which a global optimum can be easily computed, whereas a NN solves non-convex optimisation problems and in turn, outputs non-linear decision boundaries making them highly parallel processes. The selection of a suitable activation function is what allows a NN to efficiently learn non-linear structures in the data.

This chapter aims to provide the reader with a complete overview of the general structure

underlying a NN, as well as its inner workings from how they are trained and the selection of appropriate activation functions, to the selection of different optimisation methods through the use of gradient descent. The following discussion will introduce important terminology necessary to ease the transition into the next chapter. This chapter can be seen as the fundamental building block to all the models used in the empirical Chapter 6.

3.2 THE ARCHITECTURE OF A MULTI-LAYER NEURAL NETWORK

In its most central form, an NN is a weighted graph consisting of an ordered set of layers where every layer is a set of nodes, neurons or units. The Figure 3.5 below is a depiction of the standard multi-layer NN.

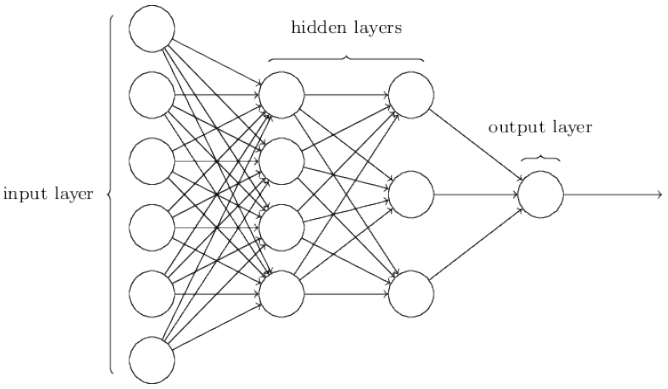


Figure 3.5: Illustration of a simple feedforward NN.
Source: Gupta (2017).

3.2.1 The Layers

Multi-layer NNs have many neurons (circles in Figure 3.5) connected in a series of layers, each processing an input with an associated weight and bias which is sent to the next layer. The input layer contains the data being fed into the first layer of the model. The hidden layer, which is the core of the network, consists of different units where each unit outputs a weighted combination or sum of its inputs combined with a non-linear transformation. Each output of a layer of units is then in turn fed as inputs to each unit in the next layer. The output of a node is calculated by applying a function to the output of the nodes belonging to the previous layer. Therefore, some layers will process the original data fed into the model while other layers further down the series will only process data received from previous neurons. The output layer converts the hidden layer activation to an output suitable for say, classification. The network is fully connected meaning that all nodes in the previous layer are connected to all nodes in the following layer, therefore each hidden unit

sums over all the input units. This gives the network a 'web' graph structure as observed in Figure 3.5. The number of hidden layers in the network is referred to as the depth, and the number of nodes in each hidden layer is referred to as the width.

It is not required that the hidden layers in the network should have the same width. The number of nodes is allowed to vary across the hidden layers and the output layer may also be arbitrary depending on the task, for example, in the classification of ten image classes, the output layer will consist of ten nodes each corresponding to the associated output class. Most of the calculations in a NN occur in the hidden layers. These hidden layers transform the decision boundary to a non-linear space when appropriate non-linear activation functions are chosen. The activation function is used to convert the weighted sum of the inputs into a non-linear output when some threshold is surpassed ensuring that the network can learn complex tasks.

3.2.2 Activation Functions

To fit non-linear NNs, a non-linear transformation function has to be applied to every output of each input layer. Having an appropriate non-linear activation improves the approximation done by the network and allows that the model learn complex structures. Several different activation functions exist since these functions are technically any simple differentiable non-linear function, each with their own pros and cons. The usage will depend on the type of network and the type of layer the network is operating in.

One of the first choices in the literature to use as an activation function was the Sigmoid function (Rumelhart *et al.*, 1988), introduced in Chapter 2 as:

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

This function, however, lost its popularity in practical applications as an activation function in the hidden layers as it has some major drawbacks in the optimisation procedure of NN models. These problems are owed to the derivative of the function becoming flat on both extremes, therefore slowing down the learning process and causing the gradients to vanish (Krizhevsky *et al.*, 2012). Thus, for large absolute values of x, the gradient could be too small to be useful for learning even if the training data abundantly populate these regions. This is emphasised when the function saturates to 0 when x takes on a large negative value and saturates to 1 when x takes on a large positive value. The Sigmoid outputs are also not zero centred (it is centred at 0.5), meaning when optimisation of the network is done, the gradient updates move in different directions causing

undesired zigzag effects. The use of the Sigmoid function becomes apparent in the output layer of a binary classification task when the desired output of the network should produce outputs between 0 and 1 as estimates of conditional class probabilities. In the context of a multi-class classification task, the output layer uses Softmax as an activation function which was introduced in Chapter 2. This again indicates the close relationship of the Logistic model to a NN. The following activation functions should only be used in the hidden layers of a NN.

An activation function that is zero centred is the hyperbolic tangent or tanh activation function given as:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{3.7}$$

As in the case with the Sigmoid function, the tanh function also leads to gradients that tend to zero causing issues during training where the weights might not update at all resulting in a non-optimal model and vanishing gradients. These vanishing gradients increase the difficulty for parameters to update accordingly while improving the loss function. In both cases, the exponential functions are also expensive to compute causing slower convergence when the network is trained.

Currently, the most used activation function in practice and research, considered as the default for training most convolutional and fully connected layers, is the Rectified Linear Unit, ReLU. The ReLU function (Nair and Hinton, 2010) is given as:

$$f(x) = max(0, x) = \begin{cases} x, \text{ if } x > 0. \\ 0, \text{ otherwise.} \end{cases} \tag{3.8}$$

The function can be read as; if the input is negative the output will be zero and if the input is positive the output will be positive. The vanishing gradient problem from the Sigmoid and tanh functions is eliminated when this function is used since it will always contain a derivative equal to 1 for positive values of x, meaning that the algorithm converges faster (up to 6 times faster than Sigmoid and tanh (Krizhevsky *et al.*, 2012)) since it is cheap to compute and that the function does not saturate in positive regions. The function however still has a caveat, since it is not differentiable when $x = 0$. This is usually not a problem in software implementation with NNs (Goodfellow *et al.*, 2016), but a ReLU layer can fail to activate in such a case. This notion is referred to as the "dead ReLU" problem which happens if a layer is fed an input of 0 causing the trained weights to be zero and thus the gradient also to be zero with output 0, leading to a series of layers that might never activate. The typical occurrence of this problem is when the learning rate is chosen too high or there is too much negative bias. To combat this problem, the gradients should be monitored

during training, or a proper, small, learning rate should be chosen. This will become evident in the following section when optimisation techniques of NNs with activation functions are discussed. Another remedy is to initiate ReLU neurons with slight positive bias (e.g. 0.01). Variations to improve ReLU can also be used if the model contains many dead ReLU layers. These include PReLUs (He *et al.*, 2016), Leaky ReLUs (Maas *et al.*, 2013) and ELU (Clevert *et al.*, 2015).

The Leaky ReLU function is given as:

$$f(x) = max(0.01x, x) = \begin{cases} x, \text{ if } x > 0. \\ 0.01x, \text{ if } x \leq 0. \end{cases} \tag{3.9}$$

Clearly, the function contains a small negative slope which has the purpose of keeping updates alive while preventing the neurons from dying (reduces the chance of a neuron to hit a zero-input value as was the case with ReLU). The function still contains discontinuity at zero, but it is no longer flat below zero, reducing the gradient partially. This results in a somewhat faster convergence rate than that of ReLU.

Parameter Rectified Linear Units, PReLUs, is given as:

$$f(\alpha, x) = max(\alpha x, x) = \begin{cases} x, \text{ if } x > 0. \\ \alpha x, \text{ if } x \leq 0. \end{cases} \tag{3.10}$$

This function is adapted from Leaky ReLU by changing the (pre-determined) negative slope into a parameter of each neuron. This parameter, α, is not learned by the network and is considered as an additional hyperparameter, although some software include small values for α without the user needing to specify it.

Another version of ReLU known as Exponential Linear Unit, ELU, were shown to obtain higher classification accuracy by following the same rule as ReLU for $x > 0$ and decreasing exponentially for $x \leq 0$ (Clevert *et al.*, 2015). Generalisation was also shown to be more accurate than ReLU for networks containing five or more layers since the function attempts to make the mean activations closer to zero. This activation function, also containing an adjustable parameter, α, is given as:

$$f(\alpha, x) = max(\alpha(e^x - 1), x) = \begin{cases} x, \text{ if } x > 0. \\ \alpha(e^x - 1), \text{ if } x \leq 0. \end{cases} \tag{3.11}$$

The fact that the function includes an exponential term can still cause computation time to be

slower. Here, α, is also not learned by the network. This function also does not centre the values at zero, however, this function does not have the dying ReLU problem and it is also more robust to noise as ReLU since the negative part of the function can still saturate to some value (this value is determined by the factor used in α).

Extensive research is still being done on the selection of an activation function and will remain an active topic for the time being as the choice of what function to use is based on a trial-and-error approach. A comprehensive list with a description of the different activation functions can be found in Nwankpa *et al.* (2018). A summary of the structure of the above activation functions can be found in Figure 3.6.

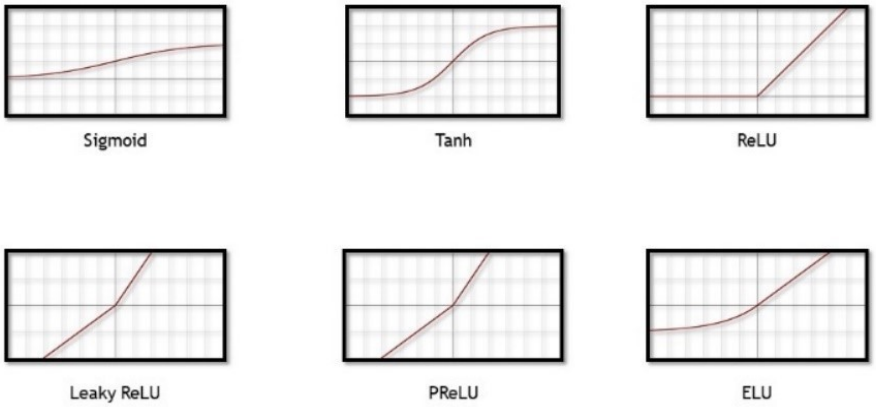


Figure 3.6: Illustration of different activation functions.
Source: Jain (2019).

Interestingly, most of the famous image recognition architectures (that will be discussed in-depth in Chapter 5) made use of ReLU in their hidden layers. This perhaps could be due to the variations of ReLU lacking theoretical justifications to support their state-of-the-art (these architectures will become more clear in Chapter 5) results (Xu *et al.*, 2015). This is evident from Table 3.1 below:

Table 3.1: Activation functions used in the state-of-the-art image recognition architectures.

Application	Hidden Layer	Output Layer	Reference
AlexNet	ReLU	Softmax	Krizhevsky *et al.* (2012).
VGGNet	ReLU	Softmax	Simonyan and Zisserman (2014).
GoogLeNet	ReLU	Softmax	Szegedy *et al.* (2015).
ResNet	ReLU	Softmax	He *et al.* (2016).

Recently, another variation of ReLU surfaced from research namely the Scaled Exponential Linear Unit, SeLU, which is considered as one of the top-performing activation functions for deeper networks (Klambauer *et al.*, 2017).

The function is given as follows, taking on the shape given in Figure 3.7:

$$f(\alpha, x) = \lambda \begin{cases} x, \text{ if } x > 0. \\ \alpha(e^x - 1), \text{ if } x \leq 0. \end{cases} \tag{3.12}$$

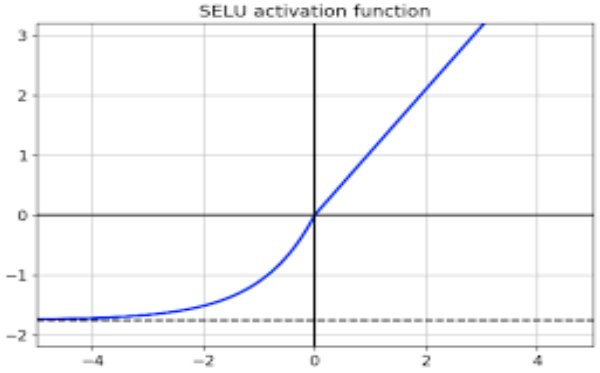


Figure 3.7: Illustration of the shape of the SeLU activation function.
Source: Huang *et al.* (2020).

This function introduced a new class of networks called Self Normalising Networks, SNNs, which not only control the gradients of the learning algorithm through its parameters to ensure that they do not vanish but also introduces the concept of self-internal normalisation in a network.

Normalisation often occurs in three different places when training NNs. The first is known as input normalisation which is when inputs are scaled between 0 and 1 easing model computations resulting in faster training. This has become standard practice for most ML algorithms and happens when data pre-processing is done. Secondly, is batch normalisation, which is when values between each layer of the network are transformed to have mean 0 and variance 1, reducing the chance of extreme values occurring in the algorithm which could negatively affect the model's accuracy on unseen data (Ioffe and Szegedy, 2015). This type of normalisation happens in the hidden layers of the network before the activation function are used to output values to the next layer.

Due to computers not having an infinite amount of memory, training NNs happens in segments referred to as batches usually in data sized in powers of 2, such as 32, 64, 128 or 256 data points depending on CPU and GPU capability. An epoch is then referred to as a single pass of all the training examples through the network, i.e. it is a collection of iterations needed to make a single run through the training data. In this sense, iterations are the number of batches to complete one epoch. Batch size and the number of epochs used in the model is determined by the user and therefore it is considered as an additional hyperparameter. In batch normalisation, the activations

of the previous layer are normalised for each batch. More on batch normalisation will be discussed in the next section.

Thirdly, internal normalisation is when each layer preserves the mean and variance of the previous layer. This reduces the need for batch normalisation in a network since normalisation now occurs inside of the activation function. The output of each activated neuron is therefore already normalised. The values of α and λ are derived by the authors of the function and are to be chosen as $\alpha = 1.67326$ and $\lambda = 1.0507$. For an activation function to be self-normalising it needs both a negative and positive space to shift or transform the mean which is why ReLU is not a good choice in this regard. Now, since $\lambda > 1$, it allows for gradients to be larger than one implying that the function can also shift or transform the variance of each layer. The function also centres the values with a fixed variance indicating that the vanishing gradient problem disappears. However, for the function to perform properly, Klambauer *et al.* (2017) suggested that it needs to be used in conjunction with a weight initialiser and a network regularisation parameter. The methods of weight initialisation will become evident towards the end of this chapter, and regularisation methods will be discussed in-depth in Chapter 4.

The question now remains, how do the weights and biases in the model update to ensure that the gradients do not vanish? Noticeably, NNs are structured in the same manner as previously mentioned ML algorithms, that is, consisting of a dataset specification, a model generating an output, a loss function to assess the output, and an optimisation procedure to find the optimal parameters while minimising the loss function. Hence, training of a NN through optimisation is based on a gradient descent procedure such as SGD. The optimal training (finding the correct weights and biases) of a NN will be discussed next.

3.3 TRAINING A NEURAL NETWORK

Consider a neuron processing a non-linear transformation of its inputs with parameters defined earlier as:

$$y = g(\boldsymbol{\omega}^T \boldsymbol{x} + \boldsymbol{b}). \tag{3.13}$$

Here, $g(.)$ is the activation function. To find the optimal parameters while minimising the loss function of multiple neurons in multiple layers, the NN is trained in consecutive steps. The first step is the same as described in the Logistic network, where inputs are passed through layers and activations are calculated in each hidden layer to produce a final output. This step is initiated when random numbers are assigned to the parameters. There exist formal methods to assign values to the

initial parameters, which will be discussed in the next section. For now, assume that the parameters are initiated randomly. This step, where inputs are passed through the layers to produce an output, is referred to as forward propagating through the network. The predicted output is then measured against the actual output in a functional form to provide the loss, $L(y_i, \hat{y}_i)$, where $\hat{y}_i$ is the predicted output.

To improve the parameters while minimising the loss, the gradient of the loss function needs to be calculated with respect to the weights and biases in the network. The mathematical chain rule provides a way of doing this, one layer at a time, iterating backwards from the last layer updating the parameters accordingly. These steps are repeated using a gradient descent method until the loss function converges to its minimum value. To calculate the gradient of the loss function through the chain rule, the model needs to go through all of its units while estimating their contributions to the overall error. This technique is referred to as backpropagation or reverse differentiation (Rumelhart *et al.*, 1988). This method not only reduced the complexity in the training of NNs, but it also paved the path to how NNs are structured allowing for the exponential growth of research in NNs from the 1990s.

3.3.1 Optimisation Through Propagation

Backpropagation contributes to the calculation of the minimum loss through a gradient descent method. Consider again the structure of a single neuron, with an adjusted notation:

$$y = g(W\boldsymbol{a} + \boldsymbol{b}) = g(\sum_j \omega_{ij} a_j + b_i). \tag{3.14}$$

Here, the real numbers produced by the neurons in one layer are collected into a vector, $\boldsymbol{a}$, with W, a matrix taking on multiple weights with dimension, or the number of neurons, similar to the vector $\boldsymbol{a}$ at the previous layer. The biases are given by the vector, $\boldsymbol{b}$, where the number of components in $\boldsymbol{b}$ also matches the number of neurons at the current layer. The discussion that follows will be based on the feedforward computational graph in Figure 3.8.

The network is composed as follows: It has L layers with 1 and L the input and output layer, respectively. Layer $l = 1, ..., L$ contains n_l neurons of which n_1 is the input data. The network thus maps from $R^{n_1} \to R^{n_L}$. The equations resulting in the feedforward network is:

$$\boldsymbol{z}^1 = W^1\boldsymbol{x} + \boldsymbol{b}^1. \tag{3.15}$$

$$\boldsymbol{a}^1 = g(\boldsymbol{z}^1). \tag{3.16}$$

...

$$\boldsymbol{z}^l = W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l. \tag{3.17}$$

$$\boldsymbol{a}^l = g(\boldsymbol{z}^l). \tag{3.18}$$

...

$$\boldsymbol{z}^L = W^L \boldsymbol{a}^{L-1} + \boldsymbol{b}^L. \tag{3.19}$$

$$\boldsymbol{a}^L = g(\boldsymbol{z}^L) = \hat{\mathbf{y}} = \text{Predicted Output}. \tag{3.20}$$

$$\boldsymbol{J}(.) = \frac{1}{2}\|\mathbf{y} - \hat{\mathbf{y}}\|^2 = \frac{1}{2}\|\mathbf{y} - \boldsymbol{a}^L\|^2 = \text{Loss}. \tag{3.21}$$

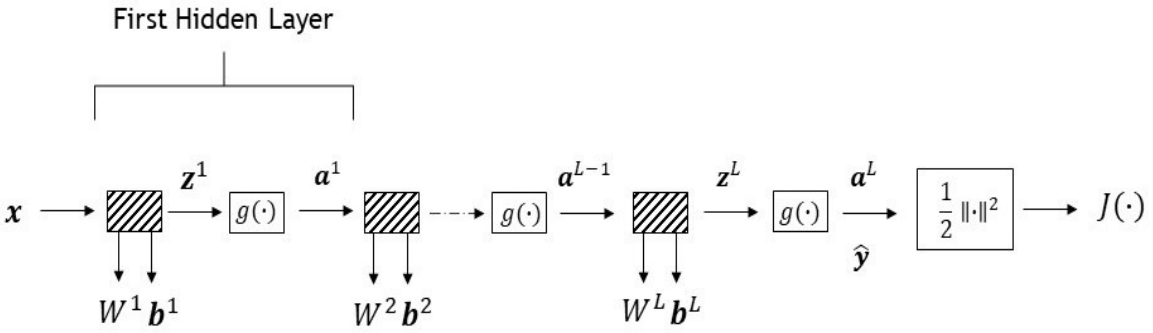


Figure 3.8: Illustration of a feedforward computational graph that will be used to derive the way in which NNs are trained.

Note the use of $J(y_i, \hat{y}_i)$ instead of $L(y_i, \hat{y}_i)$ to avoid confusion since L is now referred to as the last layer. In the above equations, the network can be summarised using a given input $\boldsymbol{x} \in R^{n_1}$:

$$\boldsymbol{a}^l = g(W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l) \in R^{n_l}, \text{for } l = 2, ..., L; \tag{3.22}$$

where a_j^l denotes the output, or activation, from neuron j at layer l and $\boldsymbol{z}^l = W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$ is the weighted input for neuron j at layer l. It follows that $\boldsymbol{a}^l = g(\boldsymbol{z}^l)$ for $l = 2, ..., L$. The matrix of weights at layer l is given as, $W^l \in R^{n_l \times n_{l-1}}$. Here, $\boldsymbol{\omega}_{jk}^l$ is the vector of weights that neuron j at layer l applies to the output from neuron k at layer $l-1$. The vector of biases for layer l is $\boldsymbol{b^l} \in R^{n_L}$ where neuron j at layer l uses the bias b_j^l. The goal of the feedforward algorithm is to produce an output $\boldsymbol{a^L} \in R^{n_L}$.

Suppose that there are now N inputs or training points given as $\{x_i\}_{i=1}^N$ for which there are

target outputs $\{y_i\}_{i=1}^N$ in R^{n_L}. Suppose also that the loss function that needs to be minimised is given as follows:

$$J = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|y_i - \boldsymbol{a}^L(x_i)\|^2. \tag{3.23}$$

This is the quadratic cost function which will be used for simplicity since the $\frac{1}{2}$ cancels out efficiently when matrix differentiation is used. The propagation steps remain the same, regardless of what loss function is used.

Now, backpropagation calculates the gradient of this loss function in order to update the weights and biases through some descent method. The methods of standard gradient descent and SGD have already been introduced, as well as training a NN in batches and epochs due to the limited capacity of memory in a machine. Therefore, the method of gradient descent needs to be adjusted to take batch training into account. Such an approach is called mini-batch gradient descent and is used frequently in the training of deeper NNs. This method can be seen as an amalgamation for SGD and standard gradient descent.

Mini-batch descent performs updates for every batch of training example where batches are usually chosen in powers of 2. To remain consistent with the notation in Chapter 2, the method updates weights (and biases which follows intuitively with the same notation) for every mini-batch of training example, n, as follows:

$$\theta_{j+1} = \theta_j - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, (x_{i:i+n}, y_{i:i+n})). \tag{3.24}$$

That is, for each iteration, z, random indices are chosen between 1 and n to compute the gradient over the training examples of these indices using the four step algorithm:

1. While true,
2. $i_1, ..., i_z$ = random indices between 1 and n,
3. $\theta_j = \theta_{j-1} - \eta\frac{1}{z}\sum_{j=1}^{z}\nabla(\theta_{j-1}, (x_{ij}, y_{ij}))$,
4. if $\|\theta_j - \theta_{j-1}\| < \epsilon$, then break.

This method further reduces the variance of parameter updates giving faster and more stable convergence since the gradient computed at each step is averaged over more training examples, more frequently (Dauphin *et al.*, 2014). This is beneficial when working with large datasets. It also introduces vectorised operations which makes it a preferred method for deep learning applications

that makes use of GPU computing. In SGD, the index i is chosen by sampling with replacement meaning after using a training point, that point is returned to the training set and is equally as likely as any other point to be chosen in the next step. This could introduce additional sampling variation which reduces convergence time. Mini-batch gradient descent cycles each of the N training points in random order. Performing N steps is considered as completing an epoch.

Now, consider the weights and biases stored in a single vector $\boldsymbol{p}$ for simplicity. Suppose that the loss function is such that: $L(\boldsymbol{p})\colon R^n \to R$, then:

$$J_{x_i} = \frac{1}{2}\|y_i - \boldsymbol{a}^L(x_i)\|^2 \Rightarrow \nabla J(\boldsymbol{p}) = \frac{1}{N}\sum_{i=1}^{N} \nabla J_{x_i}(\boldsymbol{p}). \tag{3.25}$$

Hence, for some $m \leq N$ where m is a small number of randomly chosen training points:

$$\boldsymbol{p} \to \boldsymbol{p} - \eta\frac{1}{m}\sum_{i=1}^{m} \nabla J_{x_i}(\boldsymbol{p}). \tag{3.26}$$

Note, there is still some approximation to the gradients of the full loss function when they are calculated in this way, however, in practice it turns out that this is not a problem (Li *et al.*, 2014).

SGD and mini-batch gradient descent are often used interchangeably since SGD is after all equivalent to mini-batch when the batch size is selected to be 1. Both methods are effective in practice, however, as with most of the methods used in NNs, there exist more advanced methods of descent which will be discussed soon. Consider for now the case where SGD is used to train a NN. It follows that:

$$J = \frac{1}{2}\|\mathbf{y} - \boldsymbol{a}^L\|^2. \tag{3.27}$$

The goal now is to compute partial derivatives, using the chain rule, of the loss function with respect to each $\boldsymbol{\omega}^l_{jk}$ and b^l_j. Therefore, let $\boldsymbol{\delta}^l \in R^{n_l}$ be defined as:

$$\delta^l_j = \frac{\partial J}{\partial z^l_j} \text{ for } 1 \leq j \leq n_j \text{ and } 2 \leq l \leq L. \tag{3.28}$$

This term is known as the local gradient or error signal in the j^{th} neuron at layer l where it measures the sensitivity of the loss function to the weighted input for neuron j at layer l. Also, consider the component-wise product of two vectors $\boldsymbol{x}, \mathbf{y} \in R^n$ as the Hadamard product given as the pairwise multiplication of corresponding components $(x \circ y)_i = x_i y_i$. Then, with the notation introduced above together with the vectorised form of the chain rule, the backpropagation sequence can be written formally as follows (Goodfellow *et al.*, 2016):

Lemma 1

$$\boldsymbol{\delta}^L = g'(\boldsymbol{z}^L) \circ (\boldsymbol{a}^L - \mathbf{y}). \tag{3.29}$$

$$\boldsymbol{\delta}^l = g'(\boldsymbol{z}^L) \circ (W^{l+1})^T \boldsymbol{\delta}^{l+1} \text{ for } 2 \leq l \leq L-1. \tag{3.30}$$

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \text{ for } 2 \leq l \leq L. \tag{3.31}$$

$$\frac{\partial J}{\partial \boldsymbol{\omega}_{jk}^l} = \delta_j^l a_k^{l-1} \text{ for } 2 \leq l \leq L. \tag{3.32}$$

Proof

The proof starts by dealing with $\boldsymbol{\delta}^L$. There exist a connection between z_j^L and a_j^L with $l = L$ such that $\boldsymbol{a}^L = g(\boldsymbol{z}^L)$, hence:

$$\frac{\partial a_j^L}{\partial z_j^L} = g'(z_j^L). \tag{3.33}$$

Now:

$$\frac{\partial J}{\partial a_j^L} = \frac{\partial}{\partial a_j^L} \cdot \frac{1}{2} \cdot \sum_{k=1}^{n_L} (y_k - a_k^L)^2 = -(y_j - a_j^L) = (a_j^L - y_j). \tag{3.34}$$

So, using the chain rule:

$$\delta_j^L = \frac{\partial J}{\partial z_j^L} = \frac{\partial J}{\partial a_j^L} \cdot \frac{a_j^L}{z_j^L} = (a_j^L - y_j) \cdot g'(z_j^L), \tag{3.35}$$

which is the component-wise form of (3.29). To show the vectorised derivate, the Hadamard product is introduced using:

$$\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} = \frac{\partial}{\partial \boldsymbol{z}^L} \cdot g(\boldsymbol{z}^L) = diag(g'(\boldsymbol{z}^L)), \tag{3.36}$$

with $g'(\boldsymbol{z}^L)$ the element-wise derivative of the non-linearity with respect to its input vector $\boldsymbol{z}^L$. Since all dimensions now correspond in R^{n_L}, it follows from the Hadamard product that:

$$\boldsymbol{\delta}^L = \frac{\partial J}{\partial \boldsymbol{z}^L} = g'(\boldsymbol{z}^L) \circ (\boldsymbol{a}^L - \mathbf{y}). \tag{3.37}$$

To show (3.30), the chain rule is used to convert z_j^L to $\{z_k^l + 1\}_{k=1}^{n_{l+1}}$ as follows:

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} = \sum_{k=1}^{n_{l+1}} \frac{\partial J}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{z_j^l} = \sum_{k=1}^{n_{l+1}} \delta_k^{l+1} \cdot \frac{\partial z_k^{l+1}}{z_j^l}. \tag{3.38}$$

The connection between z_k^{l+1} and z_j^l is such that:

$$z_k^{l+1} = \sum_{s=1}^{n_l} \boldsymbol{\omega}_{ks}^{l+1} \cdot g(z_s^l) + b_k^{l+1}, \tag{3.39}$$

hence,

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \boldsymbol{\omega}_{kj}^{l+1} \cdot g'(z_j^l). \tag{3.40}$$

Therefore,

$$\delta_j^l = \sum_{k=1}^{n_{l+1}} \delta_k^{l+1} \boldsymbol{\omega}_{kj}^{l+1} \cdot g'(z_j^l) = g'(z_j^l)((W^{l+1})^T \delta^{l+1}), \tag{3.41}$$

which is the component-wise form of (3.30). The vectorised form follows directly as:

$$\boldsymbol{\delta}^l = g'(\boldsymbol{z}^L) \circ (W^{l+1})^T \boldsymbol{\delta}^{l+1}. \tag{3.42}$$

To show (3.31), note that z_j^l and b_j^l is connected through:

$$z_j^l = (W^l g(\boldsymbol{z}^{l-1}))_j + b_j^l. \tag{3.43}$$

Clearly, z_j^{l-1} does not depend on b_j^l, therefore $\frac{\partial z_j^l}{\partial b_j^l} = 1$. Then, from the chain rule, it follows that:

$$\frac{\partial J}{\partial b_j^l} = \frac{\partial J}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial J}{\partial z_j^l} = \delta_j^l. \tag{3.44}$$

Finally, to show (3.32), the component-wise form of $\boldsymbol{z}^l = W^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$ is given as:

$$z_j^l = \sum_{k=1}^{n_{l-1}} \boldsymbol{\omega}_{jk}^l a_k^{l-1} + b_j^l, \tag{3.45}$$

which results in:

$$\frac{\partial z_j^l}{\partial \boldsymbol{\omega}_{jk}^l} = a_k^{l-1}, \text{ independently of } j, \text{ and} \tag{3.46}$$

$$\frac{\partial z_s^l}{\partial \boldsymbol{\omega}_{jk}^l} = 0, \text{ for } s \neq j, \tag{3.47}$$

which follows since the j^{th} neuron at layer l uses the weights from the j^{th} of W^l and applies the weights linearly. Thus, following the chain rule gives:

$$\frac{\partial J}{\partial \boldsymbol{\omega}_{jk}^l} = \sum_{s=1}^{n_l} \frac{\partial J}{\partial z_s^l} \cdot \frac{\partial z_s^l}{\partial \boldsymbol{\omega}_{jk}^l} = \frac{\partial J}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial \boldsymbol{\omega}_{jk}^l} = \frac{\partial J}{\partial z_j^l} \cdot a_k^{l-1} = \delta_j^l a_k^{l-1}, \text{ which completes the proof.} \tag{3.48}$$

If $\frac{\partial J}{\partial \boldsymbol{\omega}_{jk}^{l}}$ is regarded as defining the $(j,k)^{th}$ component in a matrix of partial derivatives at layer l then (3.32) indicates that this matrix is the outer product of $\boldsymbol{\delta}^{l}(a^{l-1})^{T} \in R^{n_l} \times R^{n_{l-1}}$. The full network has its weights and biases calculated by evaluating the output $\boldsymbol{a}^{L}$ through computing $\boldsymbol{a}^{1}, \boldsymbol{z}^{1}, \boldsymbol{a}^{2}, \boldsymbol{z}^{2}, ..., \boldsymbol{a}^{L}$ in a forward pass manner, and after this is done, $\boldsymbol{\delta}^{L}$ is available, then $\boldsymbol{\delta}^{L-1}, \boldsymbol{\delta}^{L-2}, ..., \boldsymbol{\delta}^{2}$ is calculated in a backward pass. To update these gradients for a fixed number of mini-batches, the adjusted SGD algorithm is used and therefore the complete training algorithm can be written as:

1. Choose a random mini-batch sample of size m for training.
2. For each training data point in the batch, do the following:

   a) Forward pass for $l = 2, ..., L$:

$$\boldsymbol{z}^{l} = W^{l}\boldsymbol{a}^{l-1} + \boldsymbol{b}^{l}. \tag{3.49}$$

$$\boldsymbol{a}^{l} = g(\boldsymbol{z}^{l}). \tag{3.50}$$

   b) Compute the gradients and errors:

$$J(y_i, \hat{y}_i) = \frac{1}{2}\|\mathbf{y} - \boldsymbol{a}^{L}\|^{2}. \tag{3.51}$$

   c) Backward pass for $l = L-1, ..., 2$:

$$\boldsymbol{\delta}^{L} = \nabla_{\boldsymbol{a}^{L}} J \circ g'(z^{L}). \tag{3.52}$$

3. Proceed with gradient descent using a learning rate of η for $l = L-1, ..., 2$:

$$\boldsymbol{\omega}^{l} = \boldsymbol{\omega}^{l} - \frac{\eta}{m}\sum_{x}\boldsymbol{\delta}^{l}(\boldsymbol{a}^{l-1})^{T}. \tag{3.53}$$

$$\boldsymbol{b}^{l} = \boldsymbol{b}^{l} - \frac{\eta}{m}\sum_{x}\boldsymbol{\delta}^{l}. \tag{3.54}$$

4. Terminate if $J(y_i, \hat{y}_i)$ has reached its minimum.

The learning process of the above network can be intuitively summarised in Figure 3.9. It should be emphasised that backpropagation is not the full method of training a NN, it is only a way to compute the gradients of all trainable parameters $\boldsymbol{\omega}^{l}$ and $\boldsymbol{b}^{l}$. The gradient descent optimisation

algorithm contributes to the updates of these parameters. Evidently, the choice of optimisation method and learning rate makes a big impact on how the network is trained. It has been mentioned that SGD has some disadvantages, such as slow convergence for large datasets and as argued by Dauphin *et al.* (2014), that the challenge of minimising highly non-convex error functions arises from the function having saddle points (points where one dimension slopes up and another slopes down) which are usually situated around a plateau of the same error making it difficult for SGD to escape these areas since the gradients are close to zero in these dimensions, implying that it would be challenging to use in a practical DL setting. Mini-batch descent also has some disadvantages when used in a DL context, one of which is that it introduces weight updates that are noisy and therefore could prove costly when training a network containing many parameters. The next section will introduce a wide variety of improved optimisation methods as well as the most recent methods found in literature and practice.

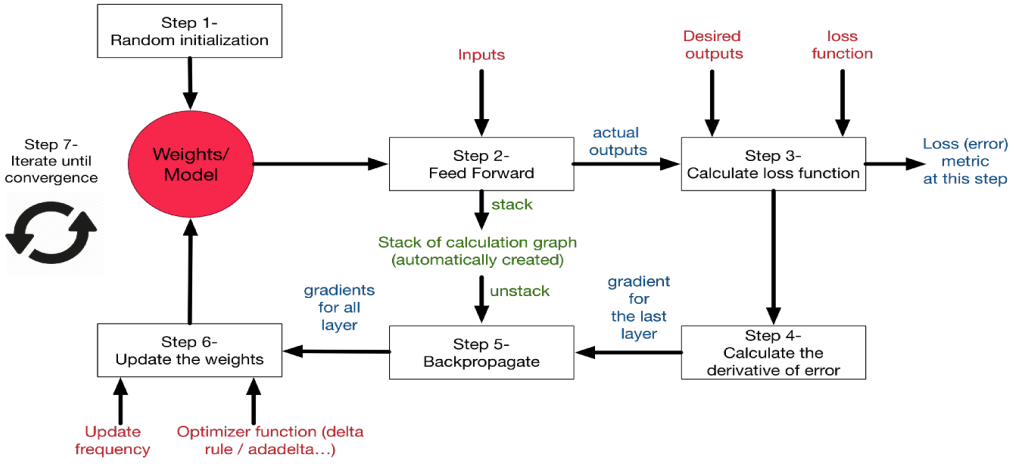


Figure 3.9: Illustration of the steps taken to fully train an ANN.
Source: Moawad (2018).

3.3.2 Gradient Descent Optimisation Techniques

3.3.2.1 Momentum

SGD with Momentum (Polyak, 1964) or Momentum, introduces acceleration when the SGD algorithm is moving through ravines which is areas around the local optima having surface curves with steep dimensions (Sutton, 1986), i.e. long valleys in the optimisation space. Momentum, v, dampens oscillations (see Figure 3.10) by introducing a variable or fraction, μ, in the SGD algorithm to denoise the weight updates using an exponentially-weighted moving average technique. Consider $\partial\omega$ as the gradient of the weights and ∂b as the gradient of the biases, then the algorithm is given

as:

1. Compute $\partial\omega$ and ∂b on the mini-batch, then calculate:

$$\upsilon_{\partial\omega} = \mu\upsilon_{\partial\omega} + (1-\mu)\partial\omega, \tag{3.55}$$

$$\upsilon_{\partial b} = \mu\upsilon_{\partial b} + (1-\mu)\partial b. \tag{3.56}$$

2. Perform weight updates as follows:

$$\omega_{new} = \omega_{old} - \eta\upsilon_{\partial\omega}, \tag{3.57}$$

$$b_{new} = b_{old} - \eta\upsilon_{\partial b}. \tag{3.58}$$

For simplicity, this can be written as ($\boldsymbol{\theta}$ is now a vector containing weights and biases):

$$\upsilon_{new} = \mu\upsilon_{old} + \eta\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta}), \tag{3.59}$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} - \upsilon_{new}. \tag{3.60}$$

The hyperparameter $\mu \in [0, 1)$, which introduces a decaying moving average of the past gradients, is usually chosen as 0.5, 0.90, or 0.99 (Goodfellow *et al.*, 2016). The higher the value of μ, the slower the decay rate. The advantage of this method is it allows for faster convergence with the disadvantage of introducing an additional hyperparameter set by the user.

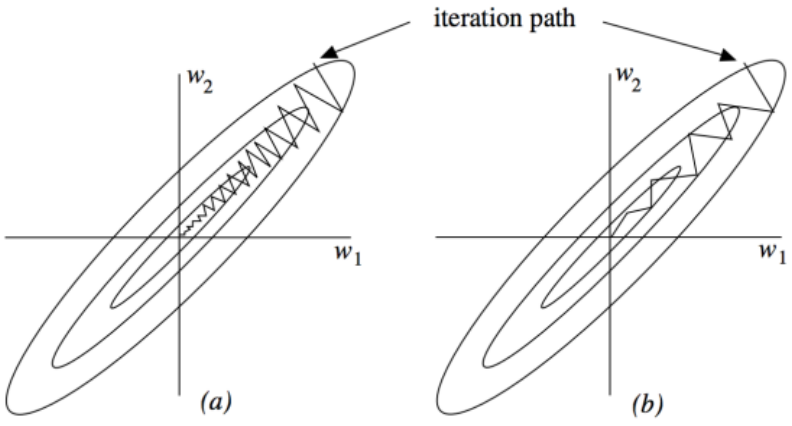


Figure 3.10: Long narrow valley function with the path followed by Momentum given in (a) and SGD in (b).
Source: Sethi and Goel (2015).

3.3.2.2 Nesterov Momentum

Nesterov Accelerated Gradient, NAG, or SGDNesterov was introduced by Nesterov in 1983, where it consists of an additional modification done to Momentum by estimating the position of a gradient one step ahead for each update (Nesterov, 1983). The algorithm does this by calculating:

$$\upsilon_{new} = \mu\upsilon_{old} + \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} - \mu\upsilon_{old}), \tag{3.61}$$

$$\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} - \upsilon_{new}. \tag{3.62}$$

The SGDNesterov method can be seen as a correction factor of Momentum. If the update of Momentum will be too wide then SGDNesterov will correct this update. Both SGDNesterov and SGD with Momentum algorithms work equally well and more or less share the same advantages and disadvantages. For a complete discussion on these methods, refer to Ilya Sutskever's PhD thesis (Sutskever, 2013).

All previous descent methods that were discussed assumed that the learning rate is globally constant for all parameters. There, however, exist additional advance methods to adapt the updates for each individual parameter by using adaptive learning procedures to perform these updates in larger or smaller proportions depending on the importance of each parameter. These adaptive methods, therefore, update the parameters in a separate scheduled manner. These methods are implemented when the range of parameter gradients differ since one global learning rate often struggles to capture the optimal schedule for all dimensions. Thus, it introduces a dynamically set learning rate in each dimension, i.e. every parameter now has its own learning rate allowing the learning rate to be adapted during training. There are multiple ways of manipulating these learning rates at local levels. The most popular methods will be discussed next.

3.3.3 Adaptive Learning Techniques

3.3.3.1 AdaGrad

This adaptive learning procedure performs smaller updates (low learning rates) for parameters associated with frequently occurring features and larger updates (higher learning rates) for parameters with less frequently occurring features (Duchi *et al.*, 2011). The algorithm has been found to greatly improve the robustness of SGD by Abadi *et al.* (2016), when the authors used it for training large-scale NNs at Google.

Consider now a different learning rate for each parameter θ_i at every time step t and let $g_{t,i}$

be the gradient of the objective function with respect to the parameter θ_i at time step t, i.e. the partial derivative of $J(\boldsymbol{\theta})$ wrt θ_i at time step t:

$$g_{t,i} = \nabla_{\boldsymbol{\theta}_t} J(\theta_{t,i}). \tag{3.63}$$

The SGD update for every parameter θ_i at time step t can now be written as:

$$\theta_{t+1,i} = \theta_{t,i} - \eta g_{t,i}. \tag{3.64}$$

AdaGrad then adjusts the learning rate at each time step t for every parameter θ_i based on the past gradients that were computed for θ_i by:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}, \tag{3.65}$$

where $G_{t,ii} \in R^{d \times d}$ is a diagonal matrix containing elements i, i as the sum of squares of the gradients with respect to θ_i while ϵ is a smoothing parameter that avoids division by zero (usually set as $1e$-8). The vectorised notation, using the Hadamard product, is given as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \circ g_t. \tag{3.66}$$

The advantage of this method is that it changes the learning rate adaptively with every iteration implying that there is no need to update the learning rate manually. The default value of η is usually chosen as 0.01 in most software libraries. The disadvantage is that the algorithm accumulates the squared gradients which cause the learning rate to decrease and eventually become too small for any learning to take place. To combat this, AdaDelta was introduced by Zeiler (2012).

3.3.3.2 AdaDelta

AdaGrad is sensitive to initial conditions such that if the optimisation starts with large gradients then the learning rate will become too small. AdaDelta combats this by introducing an exponentially decaying average. This average, $E[g^2]_t$ depends only on the previous average and the current gradient, given as:

$$E[g^2]_t = \mu E[g^2]_{t-1} + (1 - \mu) g_t^2. \tag{3.67}$$

The value of $\mu \in [0, 1)$ is set at 0.90, same as in the case of Momentum. The parameter update

then becomes:

$$\nabla\theta_t = -\eta g_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t. \tag{3.68}$$

The denominator is now the *Root Mean Squared* (RMS) of the gradient:

$$\nabla\theta_t = -\frac{\eta}{RMS[g]_t} \cdot g_t. \tag{3.69}$$

Zeiler (2012), showed that the RMS of parameter updates is then given as:

$$RMS[\nabla\theta]_t = \sqrt{E[\nabla\theta^2]_t + \epsilon}\ , \tag{3.70}$$

where: $E[\nabla\theta^2]_t = \mu E[\nabla\theta^2]_{t-1} + (1-\mu)\nabla\theta_t^2$. Now, since $RMS[\nabla\theta]_t$ is unknown, it needs to be estimated with the RMS of parameter updates until the previous time step. This implies that the AdaDelta update rule becomes:

$$\nabla\theta_t = -\frac{RMS[\nabla\theta]_{t-1}}{RMS[g]_t}. \tag{3.71}$$

$$\nabla\theta_{t+1} = \theta_t + \nabla\theta_t, \tag{3.72}$$

which abolishes the need for a learning rate to be set.

This method and an additional method called RMSProp was independently discovered around the same time in order to solve the shortcomings of AdaGrad's diminishing learning rates.

3.3.3.3 RMSProp

Proposed by Geoffrey Hinton in Lecture 6e of his Coursera Class[1], this method is identical to the first update of AdaDelta:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2. \tag{3.73}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t. \tag{3.74}$$

Hinton suggested that η should be set to a default value of 0.001. This method also reduces the oscillations suffered by Momentum.

3.3.3.4 Adam

Finally, reaching the most popular adaptive learning and parameter update method in current DL architectures named Adaptive Moment Estimation, Adam, stores an exponentially decaying average

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

of past squared gradients, υ_t, like AdaDelta and RMSProp while also storing an exponentially decaying average of past gradients, m_t, similar to Momentum (Kingma and Ba, 2014). The idea was to store both the 1st order moment of g_t and the 2nd order moment of the gradient g_t^2:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t. \tag{3.75}$$

$$\upsilon_t = \beta_2 \upsilon_{t-1} + (1 - \beta_2) g_t^2. \tag{3.76}$$

with m_t and υ_t estimates of the first moment (mean) and the second moment (uncentred variance) of the gradients, respectively. Kingma and Ba (2014) noticed that m_t and υ_t are biased towards zero if they are initialised as vectors of zero's (also when the decay rates are small, i.e. β_1 and β_2 are close to 1) and therefore they introduced bias-corrected first and second moment estimates as:

$$\hat{m}_t = \frac{1}{1 - \beta_1^t}. \tag{3.77}$$

$$\hat{\upsilon_t} = \frac{1}{1 - \beta_2^t}. \tag{3.78}$$

The Adam update rule is then given as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\upsilon_t}} + \epsilon} \cdot \hat{m}_t. \tag{3.79}$$

Kingma and Ba (2014), proposed the hyperparameters $\beta_1, \beta_2 \in [0, 1)$, the exponential decay rates, for moving averages to be set as $\beta_1 = 0.9$ and $\beta_2 = 0.999$ with $\epsilon = 10^{-8}$ and $\eta = 0.01$ or $\eta = 0.001$. Ruder (2016) showed that Adam empirically outperforms all mentioned optimisation procedures in some tasks (see Figure 3.11).

The step size of Adam in each iteration is invariant to the magnitude of the gradient thereby improving the training of models going through areas with small gradients, i.e. ravines or saddle points. In these areas, SGD usually struggles to find a clear path. There are, however, some tasks where Adam does not exactly converge to the optimal solution. Furthermore, Wilson *et al.* (2017) showed that adaptive methods do not always generalise as well as SGD with Momentum when it is tested on a diverse set of DL tasks. Again, this emphasises that there is no clear 'best' method and that DL is the process of trial-and-error. The decision made by the user or researcher on the type of method used is very important and should be stated clearly with reasons. Adam, however, remains the chosen method for optimisation in research and practice due to its rapid convergence in most tasks and will be used in most of the empirical results of this thesis.

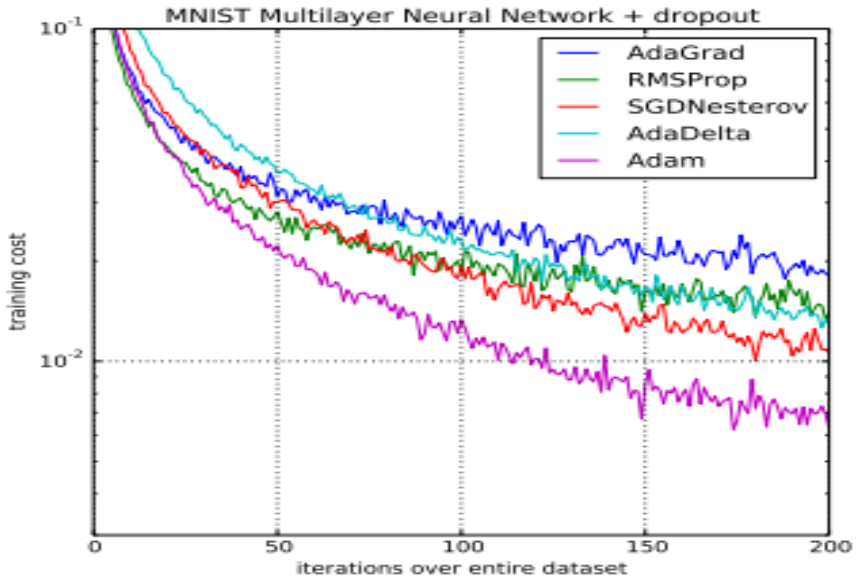


Figure 3.11: Illustrating empirical results of adaptive learning methods on the MNIST dataset. Source: Kingma and Ba (2014).

The vanishing gradient problem should become clear now. It arises from the nature of the back-propagation method for gradient computation. If all weights are small, then all gradients flowing back during propagation will become even smaller after every layer as they are multiplied causing the gradients to become so small after every batch update resulting in them vanishing. This implies that the units in the earlier layers update very slowly compared to the units in the last layers. If the architecture has many hidden layers, the network might never converge. This emphasises the importance of the activation function; to be differentiable and to ensure that gradients remain large enough through the hidden layers; as well as the selection of a proper learning rate (see Figure 3.12) to ensure that the network does not remain trapped at saddle point areas. A properly chosen and small learning rate will force the weights to update smoothly while avoiding large steps which could constitute in disproportional behaviour of the gradients.

The vanishing gradient problem will result in increased training time for the network while reducing the prediction accuracy of the output layer. There are several ways to combat this problem. One might suggest that initiating the weights to be large in order to compensate for vanishing gradients could counter the problem but in practice, it causes overfitting and adds noise to the model, therefore, causing the training procedure to diverge. In the case of a proper activation function, He *et al.* (2015) has shown that ReLU units preserve the size of the error during backpropagation while only saturating in one direction, which escalated the popularity of this activation function in the last decade.

The use of proper optimisation together with appropriate initialisation and normalisation methods can rapidly improve and accelerate the training rate of a network. The next section will discuss these terms for combatting vanishing gradients while obtaining faster and optimal convergence in general NN architectures.

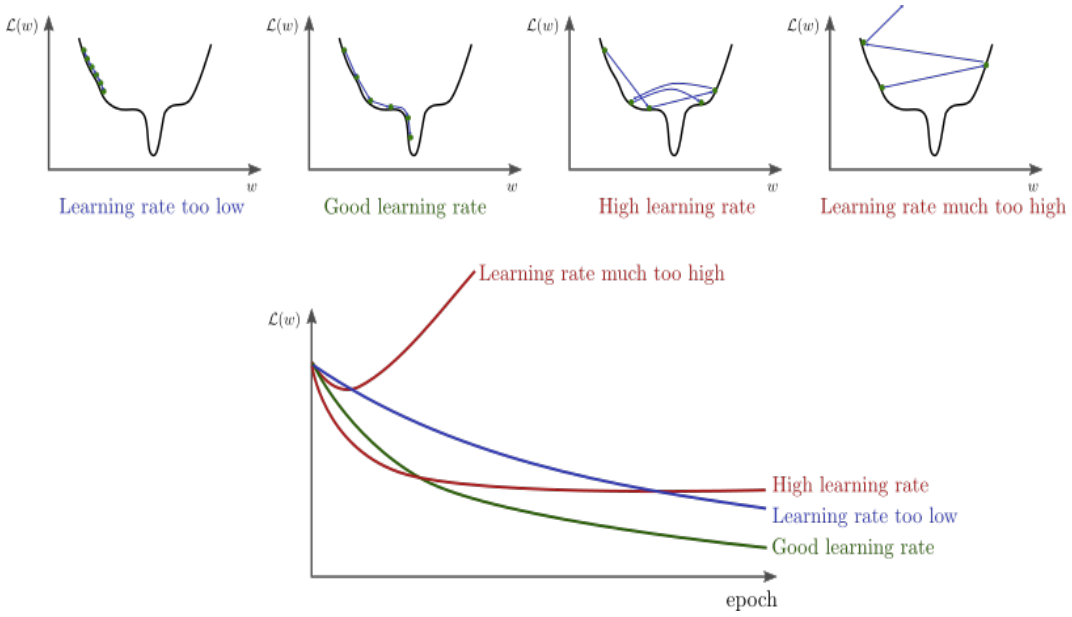


Figure 3.12: Illustration of the different stages of a chosen learning rate.
Source: Hammel (2019).

3.4 ACCELERATING THE TRAINING OF DEEP NEURAL NETWORKS

3.4.1 Weight Initialisation

The aim of weight initialisation is to ensure that the gradients stay alive during the course of forward and backward passes while producing outputs that have a similar distribution across neurons. A poorly initialised network increases the training time of the model and reduces the ability of the model to move through saddle points. A proper model contains more or less the same number of negative and positive weights to ensure that the backward pass is happening efficiently. If the network is initialised with zero weights and biases then every neuron will deliver the same output and consequently producing the same gradient and same update meaning that no learning will occur. The model will run towards the same problem if all weights are initialised the same value, i.e. the symmetry of the weights and biases will never break during training making the network no better than a classical linear model. As a result of this, the model should be initialised in symmetry. Initialising the weights as small or large random values might seem like a better idea

but still produces problems such as vanishing or exploding gradients where both cases decrease model performance and learning.

Glorot and Bengio (2010), showed that random initialisation does not work well for deep networks. Many researchers then suggested that sampling weights and biases from a standard normal (Gaussian) distribution might be a better option, but it turned out that the variance of the outputs, from the initialised neurons, grew as the number of inputs increased. The solution introduced as Xavier initialisation by Glorot and Bengio (2010), was to scale the weights by the square root of its number of inputs. This indirectly preserved the variance when passing through layers. Having constant variance on neuron activations throughout the network without breaking symmetry, i.e. eliminating both vanishing and exploding gradients, was then achieved using this Xavier initialisation. To derive the Xavier method, the following assumptions are needed:

- Input activations should be independent, centred around 0 with the same variance.
- Weights should be independent, centred around 0 with the same variance.
- There should be no correlation between the input and weights.

Given the above:

$$var(z_j^l) = var(\sum_k \boldsymbol{\omega}_{jk}^l a_k^{l-1} + b_j^l) = \sum_k var(\boldsymbol{\omega}_{jk}^l a_k^{l-1}) \tag{3.80}$$

$$= \sum_k var(\boldsymbol{\omega}_{jk}^l) var(a_k^{l-1}) = n_l var(\boldsymbol{\omega}_{j1}^l) var(a_1^{l-1}). \tag{3.81}$$

To ensure that the variance in a forward pass is preserved, initialise every weight with:

$$var(\boldsymbol{\omega}_{jk}^l) = \frac{1}{n_l}, \tag{3.82}$$

and for the backward pass:

$$var(\boldsymbol{\omega}_{jk}^l) = \frac{1}{n_{l+1}}. \tag{3.83}$$

In practice, the average over both is used such that Xavier initialisation is given by:

$$\boldsymbol{\omega}_{jk}^l \sim \mathcal{N}(0, \frac{2}{n_l + n_{l+1}}). \tag{3.84}$$

This method works particularly well for Sigmoid or tanh activation layers but struggles when it is initiated in networks having very deep layers where each layer uses ReLU activations (Simonyan and Zisserman, 2014). This led to the introduction of He initialisation by He *et al.* (2015), which was specifically designed for layers that use the ReLU family of activations. The authors proposed

to double the variance of initialised weights such that:

$$\boldsymbol{\omega}_{jk}^{l} \sim \mathcal{N}(0, \frac{4}{n_l + n_{l+1}}). \tag{3.85}$$

This method increased the rate of convergence (see Figure 3.13) and is widely used in practice.

Weight initialisation in the empirical chapter will be done using the He method. In order to ensure that the units remain Gaussian throughout the network, batch normalisation was introduced whereby it explicitly forces the activation to take on unit Gaussian distributions at the beginning of training. Batch normalisation, therefore, applies pre-processing at every layer of a network which increases the rate of convergence. He *et al.* (2015) showed that this method reduces the number of steps in the optimisation process up to 14 times in training for convergence. The authors also argued that the method acts as a regulariser while allowing for the use of higher learning rates and also reducing the need for strict initialisation methods. In practice, it is regarded as an additional layer in the network that is generally added before every non-linearity layer to avoid saturating nodes.

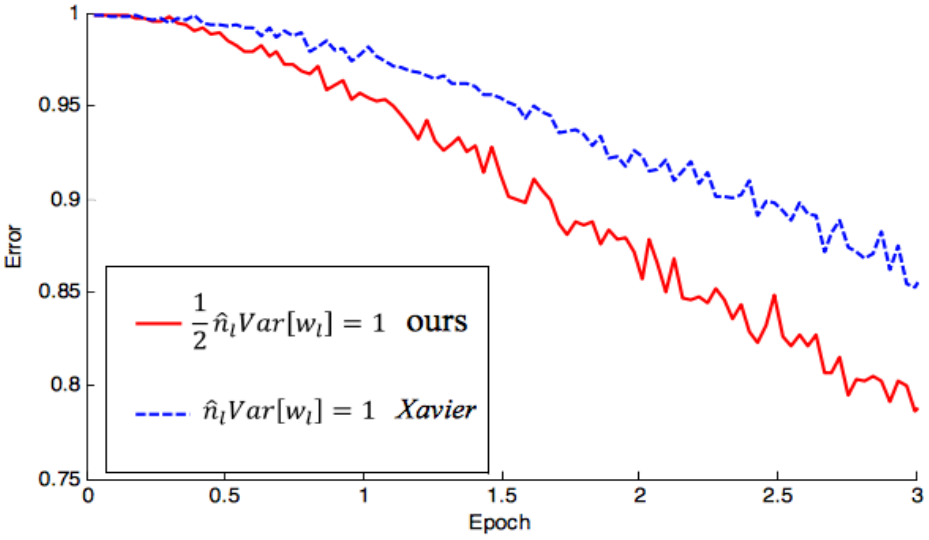


Figure 3.13: Graph from He *et al.* (2015) showing how their refined initialisation strategy (red) reduces the error rate much faster than the Xavier method (blue) for ReLUs.

3.4.2 Batch Normalisation

This method consists of two steps:

- Normalise the mini-batch of size m by subtracting its mean μ from the input and then dividing

the expression by its standard deviation σ.

- Scale the normalisation by a factor γ and shift it by a factor β.

Here, μ and σ are calculated per-batch while γ and β are learned parameters, used across all batches. The procedure is formally given by the following equations:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} z_i^l. \tag{3.86}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(z_i^l - \mu_B)^2. \tag{3.87}$$

$$\hat{z}_i^l = \frac{z_i^l - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}. \tag{3.88}$$

$$y_i^l = \gamma \hat{z}_i^l + \beta. \tag{3.89}$$

Now, ϵ is the smoothing term assuring numerical stability which is chosen to be very small. By normalising each layer, a level of orthogonality is introduced between layers which not only eases learning of deep networks but also reduces the strong dependence of proper initialisation, enabling a shorter time of convergence. In the case of Sigmoid activations, it also resolves the vanishing gradient problem since the inputs of the function are normalised ensuring that gradient does not reach the outer edges of the function where its derivative is extremely small or zero. Batch normalisation also improves ReLU activation performance, however, there is little difference in performance when using ELU activations (Clevert *et al.*, 2015). The method has become very famous in state-of-the-art architectures, which is why it will be used in the empirical work of Chapter 6. This subject will be touched on again when regularisation is discussed in the next chapter.

3.5 SUMMARY

In this chapter, ANNs were explained. The transition from a Perceptron and Logistic model to an ANN was introduced whereas the simple structure of an ANN was outlined followed by a detailed derivation on how the optimal weights and biases are obtained through different optimisation procedures. Additionally, different activation functions were introduced. Lastly, methods to accelerate the training of ANNs was also introduced. This chapter serves as in introduction to CNNs, as CNNs are built and derived using the fundamentals described in this chapter.

CHAPTER 4
CONVOLUTIONAL NEURAL NETWORKS

4.1 INTRODUCTION AND JUSTIFICATION

ANNs run into problems when having to deal with CV tasks such as image pattern recognition. This is due to the fact that a single image contains an enormous amount of information stored as a matrix containing pixel values (see Figure 4.1). For example, a greyscale 128×128 contains 16 384 (128×128×1) pixels. If every pixel intensity of this image is added separately to a fully connected network, every neuron will require 16 384 weights or attributes. A 1920×1080 greyscale image would require 2 073 600 weights and if the image was coloured it would require 6 220 800 (1920×1080×3) weights. If an ANN receives many such images as inputs then the number of computations will become extremely large. Hence, very large models cause overfitting and slow performance (Bishop, 2006).

ANNs, therefore, do not scale well to image data due to its high dimensionality. Furthermore, ANNs do not exploit the natural structure of an image such as high-correlation of neighbouring pixels and repeating patterns, i.e. ANNs do not handle shifts and distortions in images. Many pattern recognition and object recognition tasks require that the output should be translation invariant. Training an ANN with this structural goal would be inefficient (LeCun *et al.*, 1989). Training an ANN also requires that the input data is provided as a vector (a one-dimensional array). In the case where the input data is a matrix image, it will have to be flattened to be transformed into a vector. However, in natural images, there exist very strong links between neighbouring pixels; by flattening the image, this information will be lost, therefore it becomes more difficult for the network to interpret the data during the training phase.

CNNs get around this issue by using additional feature extraction layers to identify abstract representations in images through convolving filters over the matrix inputs. Using these convolutional layers, the network learns to recognise local features such as edges or corners by enforcing spatially shared weights to add invariance. Additionally, subsampling is also done in the form of a pooling layer to reduce network sensitivity to shifts and distortions while reducing the high dimensional inputs. Thus, CNNs have fewer parameters to train compared to ANNs when using the same number of hidden layers.

CNNs were first described in by Fukushima *et al.*, where the authors brought forward a hierarchical NN architecture based on research done by Hubel and Wiesel on the visual cortex system of cats whereby they used different cell layers in their network (Hubel and Wiesel, 1962). Thereafter, Atlas *et al.* (1988) brought forward the idea of using convolutional operators and filters

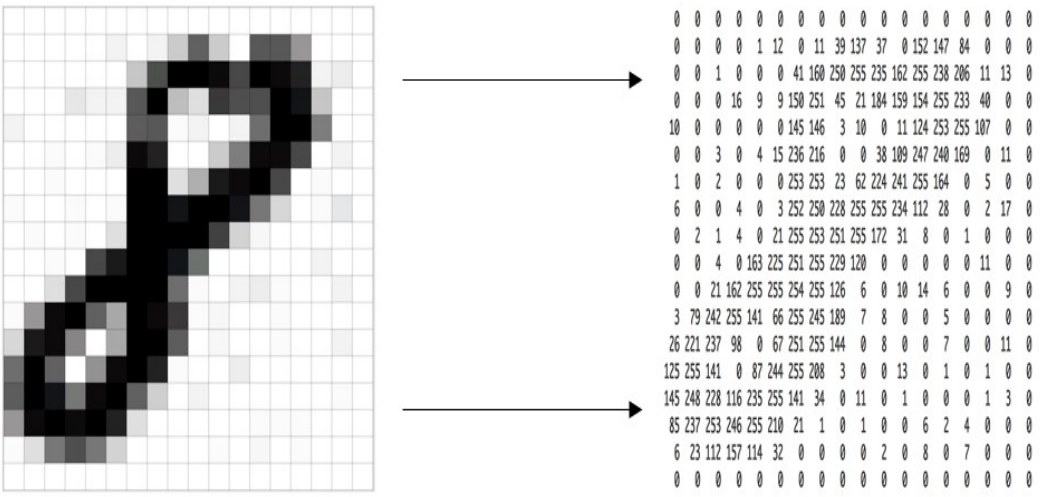


Figure 4.1: Illustration of the pixel values contained in an 18×18 greyscale image of the digit 8. Source: Geitgey (2016) [Online].

(transfer functions) instead of a fully connected system for extracting features from audio signals. It was based on these findings that the so-called 'fathers of CNNs' brought forward the first CNN for image classification (LeCun *et al.*, 1989). The authors showed that CNNs can be trained in the same manner as ANNs using stochastic gradient descent while backpropagating where they stated the different layer types using terms such as convolutional and pooling layers instead of simple and complex cell layers. They were also the first to show how the convolutional layers extracted useful features from their inputs resulting in multiple feature maps while the pooling layers reduced the spatial sizes of these maps, retaining only the important features of each input.

Research and development of these networks slowed down in the 1990s and 2000s due to limited computational power in machines, but ever since the expansion of GPUs from 2010, research in this area started expanding exponentially. Understanding NNs is critical in understanding CNNs as they form the foundation of DL image classification models.

This chapter aims to introduce CNNs by discussing their structure and inner workings. Additionally, regularisation techniques will be discussed in order to ensure that the CNN trained is optimal for predictions on unseen image data. Lastly, this chapter will discuss methods to visualise what is happening when CNNs are trained, to indicate that these models are not as complicated (black box) as they are set out the be.

4.2 THE STRUCTURE OF CONVOLUTIONAL NEURAL NETWORKS

The composition of a CNN is somewhat similar to that of an ANN, except for the additional convolutional layer(s) that exists within the hidden layer(s). Also, the hidden nodes are not single neurons anymore, but filters of spatially fixed neurons. The general structure of the layers are given in Figure 4.2 and will be discussed next.

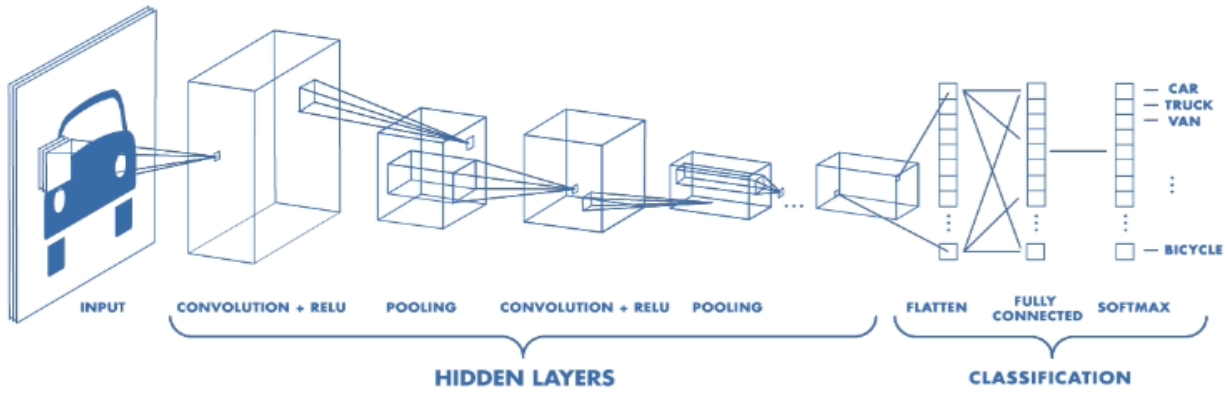


Figure 4.2: Illustrating the structure of a CNN containing an input, hidden layers and classification layer.
Source: Chatterjee (2019) [Online].

4.2.1 A Convolution and the Convolutional Layer

A convolution is a small matrix operator resulting in image filters or kernels that activate when being exposed to specific visual features or structures. This matrix operation introduces fewer weights to train in the network while allowing for significantly faster learning. A convolution is, therefore, the action of using a kernel matrix that is applied to the image by sliding the kernel over the input. The sliding process is a simple matrix multiplication or dot product (which will become clearer in the example below). The convolution can be seen as a specialised linear transformation which transforms data from one domain into another domain (Domínguez, 2015). Depending on the weight values inside the kernel matrix, the output will produce different features or activation maps.

The 3-dimensional volume formed by multiple 2-dimensional matrix colour channels is called a tensor. Given an input image in the form of a tensor, several filters can be used in the form of

convolutions for object detection. The size of each convolutional filter matrix, which is also referred to as the kernel size or neighbourhood size, specifies the receptive field of the filter. The spatial size of this matrix is often set as a square 2×2, 3×3, or, 5×5 matrix and is therefore regarded as a hyperparameter in the network.

The convolutional operation can be visualised through a simple example. Consider therefore the following 3×3 kernel matrix, which will slide across the input feature map:

$$\begin{pmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{pmatrix}. \tag{4.1}$$

At each location, the product between each element of the kernel and the input element (blue grid) it overlaps with is computed and the results are summed to obtain the output at the current location. The result is called the output feature map, which is indicated by the green grid in Figure 4.3.

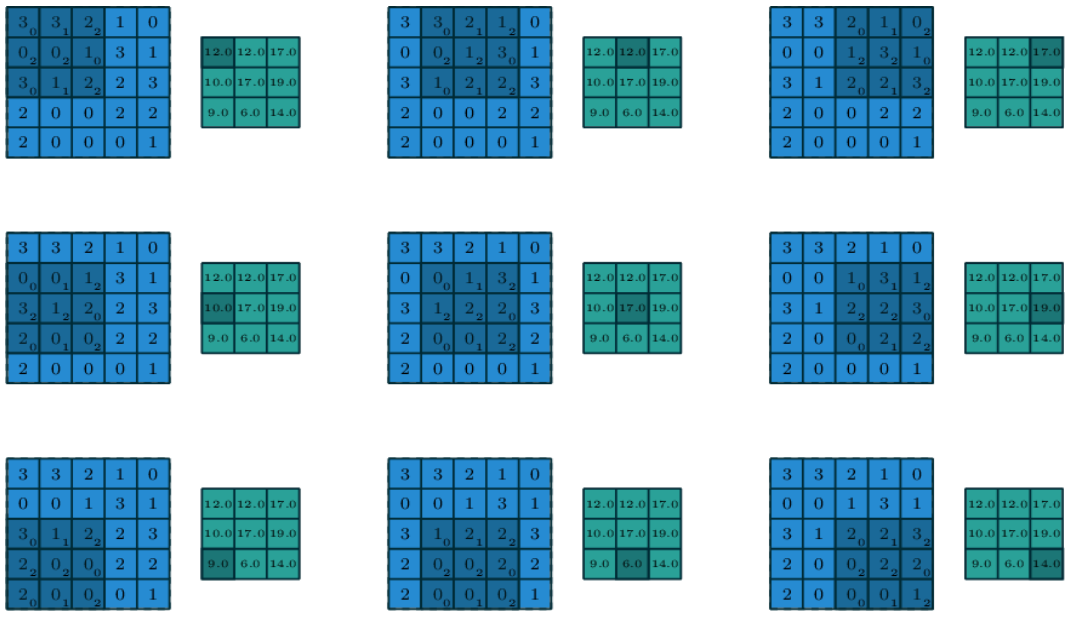


Figure 4.3: Illustrating the different outputs after the kernel in (4.1) slides over the associated matrices.
Source: Dumoulin and Visin (2016) [Online].

Applying these convolutional filter operations successively in a network provides additional benefits such as sparse interactions or sparse weights, parameter sharing, and equivalent representations (Karpathy, 2017).

Sparse weights will occur since fewer connections between input and output values exist. This

means that fewer parameters are needed in the training process which indirectly reduces memory requirements while improving statistical efficiency. The way that this is achieved, is when smaller kernel matrices are applied to the input image or input feature map implying that fewer operations are needed to produce an output. In ANNs, there is no convolutional operation and only matrix operations where outputs are connected to all inputs. The sparsity of convolutions make them very efficient to describe transformations of small receptive fields across inputs.

Parameter sharing throughout the network occurs when the same values for more than one function in a network are used. In ANNs, the elements of each weight matrix are used once for computing an output. In CNNs, each value of the kernel is used at every position of the input. This means that the CNN learns a separate set of parameters for every location.

Equivalent representations occur when convolutions are applied to images where these convolutions create 2-dimensional feature maps indicating important features that appear in the input. The parameter sharing aspect ensures that the convolutions are equivariant to translation meaning that if the input changes then the output changes in the same way or if the object in the input image is moved, its representation will also move by the same amount in the output. These advantages indicate why a CNN is so powerful for image classification tasks.

Now, an important note is that for the CNN to learn effectively, it needs to apply several filters across an input image or feature map. The number of filters is another hyperparameter, together with the step size, also called stride, which is the size of each slide. For example, in Figure 4.3, the step size is 1. The output or feature map has a size which can be controlled by changing these parameters:

- Depth: This is the number of filters used. It does not relate the image depth (1 or 3 channels as mentioned in Chapter 1) neither does it describe the number of hidden layers in a CNN. For example, if the user decided to use 12 filters on an image with input size $28\times28\times3$, the feature map will output as a $28\times28\times12$ map, as given in Figure 4.4. The depth here is 12. Each filter learns different feature maps that are activated in the presence of different image features (edges, patterns, colours, layouts).
- Stride: Stride simply refers to the step size taken when the kernel matrix slides over the input image. Using a larger stride produces less overlap in kernels. Stride is one of the methods used to control the spatial input size i.e. the volume of the inputs into the other layers of the CNN.
- Padding: During a convolution, some borders can be lost depending on the size of the input image and kernel. To compensate for this loss, a solution is to pad the image with more values

to preserve the image size. Often referred to as zero padding (see Figure 4.5), zero's are added around the matrix to ensure that the output is of the same size as the input, i.e. no pixel values are lost after the convolution. Padding can either be valid or same. Same padding is done when the feature map size should be the same as the input size. Valid padding is when the feature map size is less than the input size, meaning that no padding is applied. A simple formula for same padding is given as:

$$\text{padding amount} = \frac{\text{filter size} - 1}{2}. \tag{4.2}$$

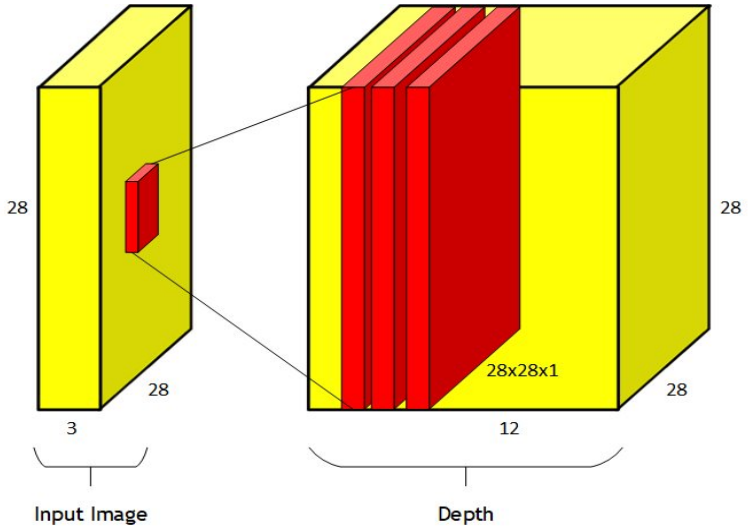


Figure 4.4: Kernel sliding (red small block) example with 12 filters applied to one layer.

To ensure that the input image is covered by the filter in a full symmetrical manner, i.e. no information is lost, the output of the following equation needs to be an integer:

$$\text{output size} = \frac{I - K + 2P}{S} + 1, \tag{4.3}$$

where:

- I = Input image height size.
- K = Kernel or filter size.
- P = Padding size.
- S = Stride size.

After these convolutions are done, the output is sent to an activation function usually ReLU as indicated in Figure 4.2 (see Figure 4.6 for an example on how the ReLU function eases feature

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 4.5: Zero padding example on an input image matrix.
Source: Saxena (2016) [Online].

calculations on an image). The convolutional layer is, therefore, a combination of the filter outputs together with the output of the activation function, i.e. each convolutional layer has non-linearity on its output that is sometimes also called the detector stage. This is equivalent to the activation function of NNs. The activation output is then sent to a pooling layer.

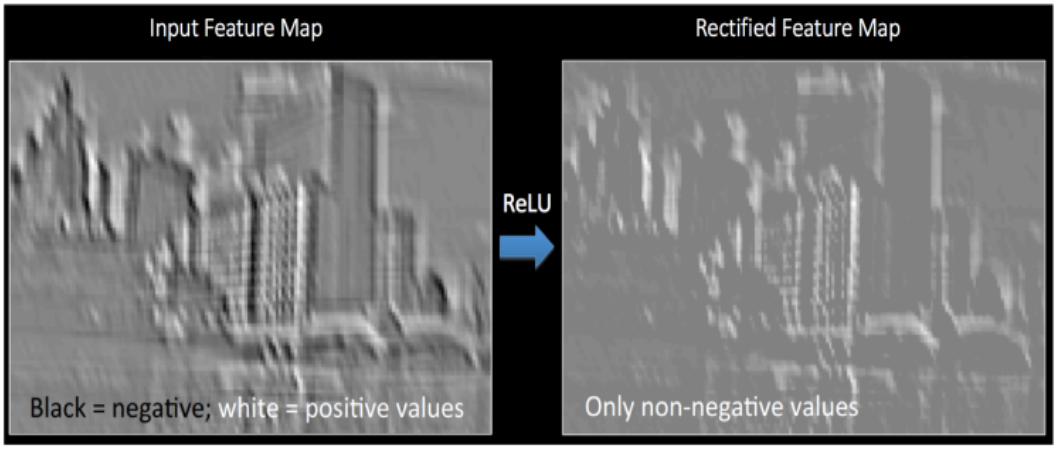


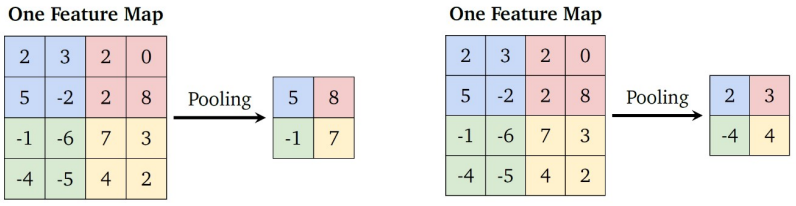
Figure 4.6: Filter visualisation after ReLU has been used on an input image.
Source: Fergus (2015) [Online].

4.2.2 The Pooling Layer

The motivation behind pooling or subsampling layers is to reduce the size of the feature maps produced by the convolutional layers progressively by considering only the features with the highest response in a window of pre-defined size. This is the case for maximum pooling layers, however average pooling has also been used, where the average of the response was used instead of the maximum (see Figure 4.7). This layer does not constitute any learning process, but it is used to downsample the size of the input, while preserving the most important information contained in an input layer i.e. it preserves the depth, but reduces the width and height. Similar to the convolutional layer, the pooling layer can also have different kernel sizes, strides and zero padding.

The biggest advantage of pooling is the reduction of computational cost. Images are data-heavy and thus are slow to convolve through in big networks. Downsampling them inside the network (see Figure 4.8) allows for the addition of more filters while not suffering from slowing down in the network. In terms of a Rectified feature map, refer to Figure 4.9 for an example of a typical output.

The convolutional and pooling layers serve multiple purposes such as extracting the useful features from the images, introducing non-linearity in the network and reducing feature dimensions while aiming to make the features somewhat equivariant to scale and translation (Raschka, 2015). These convolutional and pooling layers are then sent to the final classification layer, i.e. observing Figure 4.2 note that the classification layer has the same structure as the classical ANN. The convolutional and pooling layers reach this layer through a flattened transformation in the flattening layer, to ensure that the data is in the correct form for classification.



(a) Max pooling (b) Average pooling

Figure 4.7: Illustration of the two different subsampling or pooling methods applied on an input matrix, max pooling on the left and average pooling on the right.
Source: Li *et al.* (2015) [Online].

The classification layers, often called the fully connected layers, works in the same way as the standard ANN layers and will always be found at the end of a CNN since they compute arbitrary features and output scores, i.e. these layers are again universal approximators (Krizhevsky *et al.*,

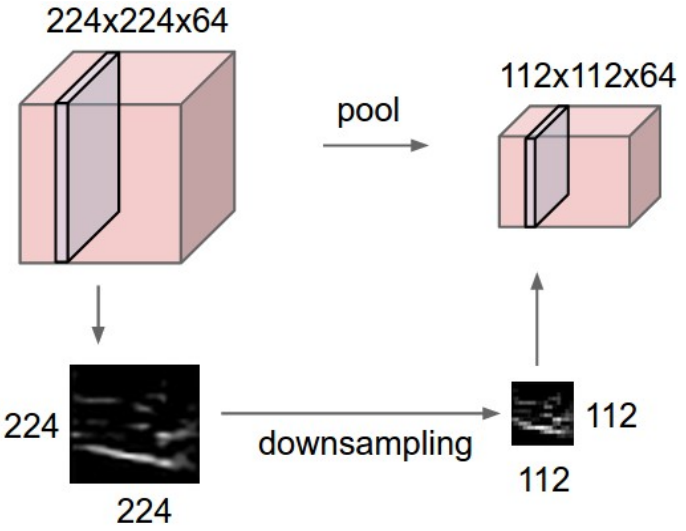


Figure 4.8: The input volume of size 224×224×64 is pooled with filter size 2, stride 2 into an output volume of size 112×112×64. Note that the depth of the image is preserved.
Source: Li *et al.* (2015) [Online].

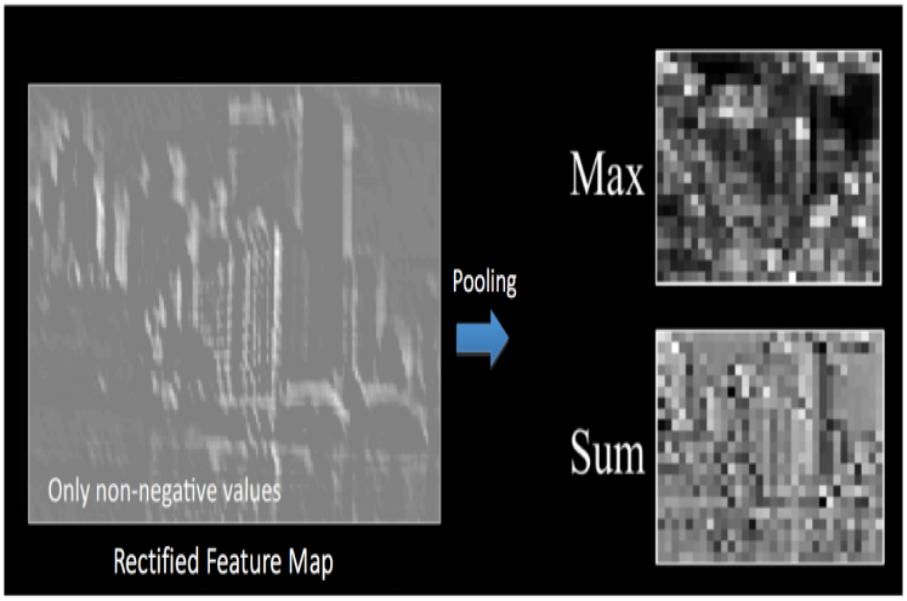


Figure 4.9: This figure indicates the effect of pooling on the Rectified Feature Map that was received after the ReLU operation in Figure 4.6.
Source: Fergus (2015) [Online].

2012). Figure 4.10 can be seen as a summary of the convolutional and pooling layers.

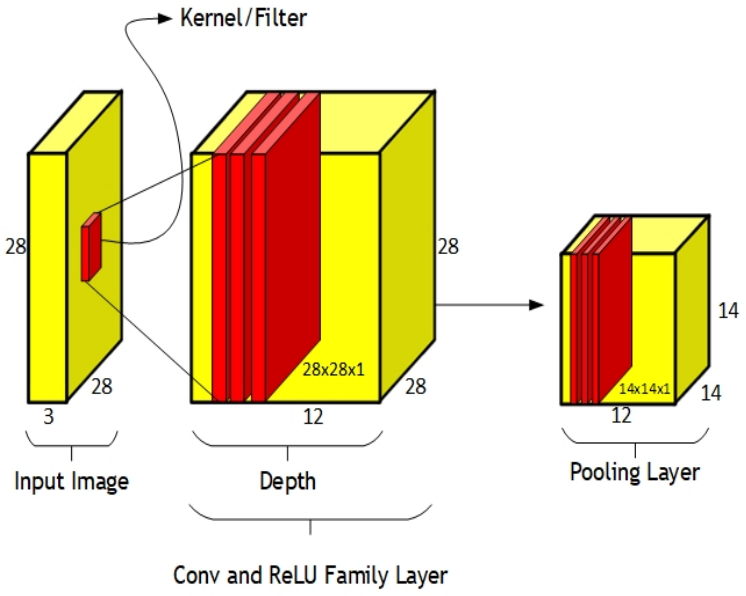


Figure 4.10: Illustration of how an input image is transformed after a convolutional (conv) and pooling layer.

4.2.3 The Flattening Layer and the Fully Connected Layers

No convolutions are done in this layer. This layer transforms the data preserved from all previous layers into a vector, squashing the three-dimensional matrices into a one-dimensional array (see Figure 4.11). The output of this layer can be seen as the input to a standard ANN for prediction, i.e. the input to the fully connected layers. The size of the fully connected layers can be adjusted by the user.

The fully connected layers (also called the dense layers in some software packages) then follow the same structure and workings as the ANN explained in Chapter 3, to produce an output. These layers continue to learn patterns and features of the input images to classify them into classes having probability scores. In this case, if the inputs at the start of the CNN, i.e. the training data, had binary classes then the Sigmoid function will be used for classification purposes otherwise if the training data consisted of multiple classes (more than two) the Softmax function would be used to make predictions. These two functions require one-dimensional inputs which is why flattening was necessary.

A convolutional network that does not include any fully connected layers, is called a fully convolutional network (Ren *et al.*, 2015). However, these will not be of interest in this thesis as they serve purposes beyond image classification.

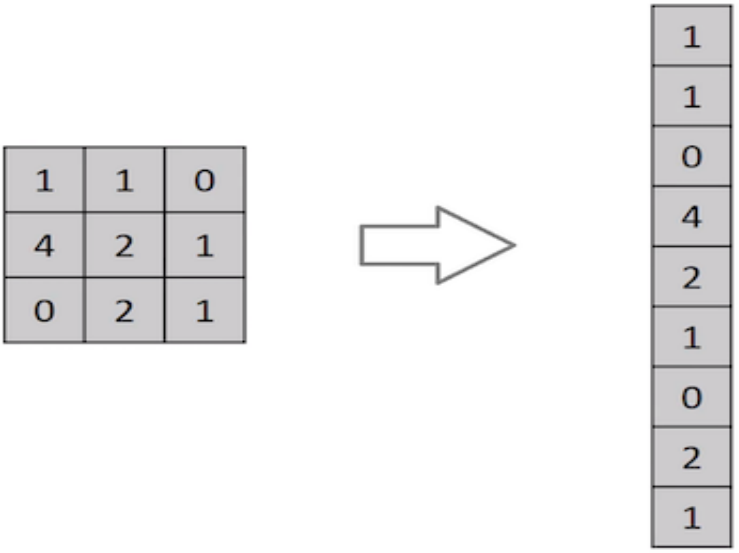


Figure 4.11: Flattened matrix to a vector.
Source: Saha (2018) [Online].

A CNN can constitute of many convolutional and pooling layers with many filters in each convolutional layer with all filters having the same size but with different weight values. The network can also have many fully connected layers. The number of filters used and layers in the network are determined by the user by taking into account that the more layers the network consist of, the higher the chances of overfitting. The key idea behind CNNs with many layers is that the filters in the lower layers (closer to the input image) will learn simple characteristics like colour blobs or where the edges are, while the filters in higher layers (more abstract) will learn complicated characteristics (see Figure 4.12), for example, faces in human facial recognition data.

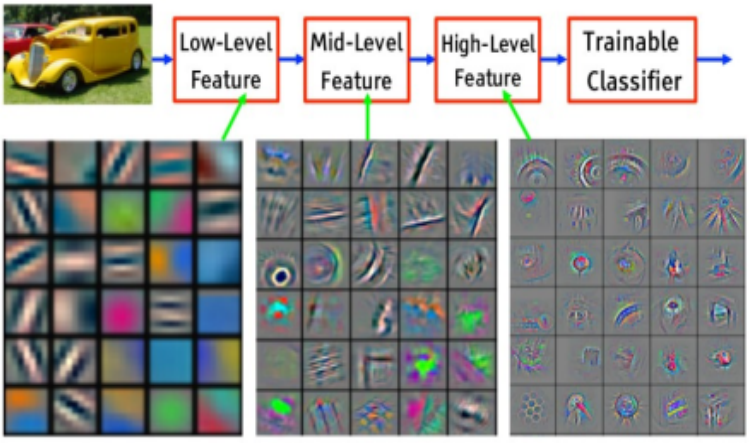


Figure 4.12: Illustrating how filters learn features in different parts of a CNN.
Source: Algobeans (2016) [Online].

In the empirical work of Chapter 6, a thorough investigation of the effect of network depth and layer width on the generalisation performance of CNNs will be done. It is not necessarily the case that very deep (many layers) CNNs will overfit. Regularisation techniques can prevent this. The simplest CNN architecture can be seen in Figure 4.13, which is regarded as a visual summary of the above discussion.

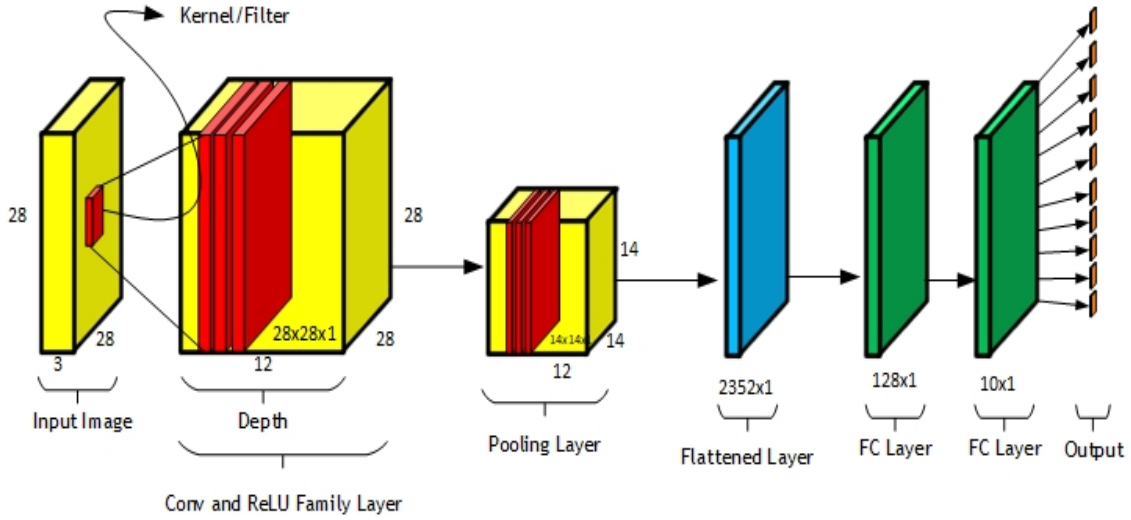


Figure 4.13: Illustration of a CNN passing an input image through the respective layers to reach an output.

4.3 TRAINING CONVOLUTIONAL NEURAL NETWORKS

The overall training of a CNN is similar to that of an ANN with the addition of the convolutional and pooling layers. The forward and backward propagation procedures remain the same with slight tweaks. It is important to note that when a CNN is training, it is learning the optimal filter weight values to minimise the loss function through any of the mentioned optimisation techniques in the earlier chapters.

Mathematically, the features at the location (i, j) in the k^{th} feature map of the l^{th} layer, $z^l_{i,j,k}$, is determined by: $z^l_{i,j,k} = {\boldsymbol{\omega}^l_k}^T x^l_{i,j} + b^l_k$, where $\boldsymbol{\omega}^l_k$ and b^l_k are the weight vector and bias term, respectively, of the k^{th} filter in the l^{th} layer. The kernel $\boldsymbol{\omega}^l_k$ that generates the feature map $\boldsymbol{z}^l_k$ is shared. Now, the non-linearity, $g^l_{i,j,k}$, of a convolutional feature, $z^l_{i,j,k}$, is computed as:

$$a^l_{i,j,k} = g(z^l_{i,j,k}). \tag{4.4}$$

The pooling layers then aim to produce shift-invariance by reducing the dimension of the feature maps. Every feature map of a pooling layer is connected to its corresponding feature map in the

previous convolutional layer. Given a pooling function, $pool(.)$, for each feature map, $\boldsymbol{a}_k^l$, after activation, is written as:

$$y_{i,j,k}^l = pool(a_{m,n,k}^l) \forall (m,n) \in R_{i,j}, \tag{4.5}$$

where $R_{i,j}$ is the local neighbourhood around (i,j).

After several of these convolutional and pooling layers, the output of these is sent to fully connected layers which aim to perform high-level reasoning (Simonyan and Zisserman, 2014). The training method for optimal weights then uses the above-defined layers by propagating through the network to minimise the loss, $J(.)$. The process is started with proper weight initialisation and is done until all epochs are completed such that maximal accuracy of the classification procedure is obtained together with minimal loss. The use of appropriate software automates this process. This will be done in the empirical chapter in a sequential manner by using TensorFlow and Keras.

It is important to remember that in the pooling layers, while propagating, nothing is learnt. Only a transformation of the data occurs. This layer has the ability to speed up the training process depending on the type of pooling method used, as well as the size of pooling used. Larger pooling sizes are usually used in the case of larger input images. The next section will discuss important regularisation techniques that can be used while the network is optimising, to ensure that the model generalises well on unseen data.

4.4 REGULARISATION OF CONVOLUTIONAL NEURAL NETWORKS

The basic structure of convolutional and pooling layers have remained the same throughout the years, however, the trend of increasing the number of convolutional and pooling layers has become more evident as the computational power of hardware components in computers increased. The use of more layers means that the architecture increased in depth which leads to the flexibility of the model increasing. Increasing the flexibility of a model allows the model to learn more complex features. This increase in flexibility will only be beneficial, in terms of model test error on unseen data, up to a certain point as indicated in Chapter 2.

The trick is to design a CNN architecture which will be large enough to capture the complex features in the input images it is being fed while ensuring that the model does not overfit. The purpose of regularisation methods introduced in this chapter is to control the amount of overfitting, by allowing for deeper networks. Goodfellow *et al.* (2016), argues that large architectures with proper regularisation will find the optimal fit for new data. The following section presents regularisation

techniques that will be used in the empirical chapter of this thesis. Some of these methods are also commonly used in ML frameworks, while the others were specifically designed for NNs and CNNs. These popular techniques consist of modification on the loss function, as described in Chapter 2, and/or modification of the network during the training phase.

4.4.1 L2 Regularisation

This method adds a penalty term to the loss function, as in equation (2.18). It penalises the weights $\boldsymbol{\omega}$ from W, but not the bias. This regularisation term can be written as:

$$\text{penalty} = \lambda \cdot \frac{1}{2}\|\boldsymbol{\omega}\|^2. \tag{4.6}$$

This is a traditional regularisation technique that can be found in most ML frameworks, which has also shown positive results on NNs.

The penalty function brings the weights closer to zero, i.e. it penalises larger weights to prevent abrupt changes (from gradients) in a model's decision boundary, for a given λ value as defined in Chapter 2. The method is also known as the L2-norm or weight decay. In backpropagation, the penalty will be applied to the weight updates where λ usually ranges between 0 and 0.01 (Kuhn *et al.*, 2013).

4.4.2 Early Stopping

As discussed in Chapter 2, the test or validation error has the tendency to start increasing as model complexity increases. At that point, the model is starting to overfit on the test or validation data. To prevent this increase in the error from happening, one could stop the training as soon as the particular error stops decreasing (see Figure 4.14).

To ensure that the increase of the specific error was not caused by random noise, the training should not stop after the first increase, but rather after a certain number of epochs (Bishop, 2006) i.e. if for 5 epochs there were no improvement in the error then training should be terminated and the error before the additional 5 epochs should be returned. This is easily implemented in Keras using a callback function. The threshold on the number of epochs is determined by the user.

4.4.3 Dropout

This method was introduced by Srivastava *et al.* (2014) and has been used in many state-of-the-art NN and CNN architectures. The method consists of randomly removing sets of neurons and

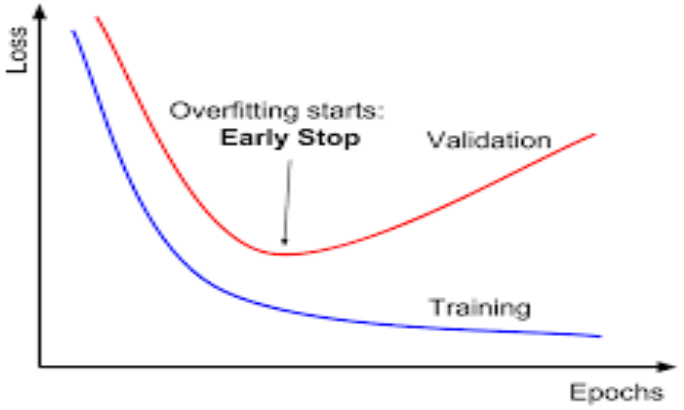


Figure 4.14: Illustrating the early stopping phenomena and how it should be implemented. Source: Nuric (2019) [Online].

their connections when a network is trained, therefore, the neurons that are dropped out do not contribute to the forward and backward propagation of the network (see Figure 4.15). This not only reduces the correlation between units but it also reduces the complex co-adaption since each neuron cannot solely rely on the presence of other neurons anymore. It forces the surviving neurons to learn more robust features which could result in a more accurate network even in the absence of certain information.

To understand how the method works, Srivastava *et al.* (2014) considered a NN with r nodes. This network can then be viewed as 2^r thinned NNs where every thinned network contains the neurons that were not dropped out. The weights of each thinned network are shared by the other thinned networks meaning that there are at most r^2 weights. When the network is trained, each neuron will be removed from the network with probability p where p can be different for every layer. The value of p is set by the user when the network is trained and the optimal value of p is obtained by trial-and-error such that the model does not overfit. However, Srivastava *et al.* (2014) suggested a value of $p = 0.5$ for hidden layers.

Mathematically, consider a NN with notation introduced in Chapter 3. Recall therefore that the feedforward operation for any unit i was given as:

$$z_i^{l+1} = g(\boldsymbol{\omega}_i^{l+1}\boldsymbol{z}^l + b_i^{l+1}). \tag{4.7}$$

Applying dropout, this reduces to the following:

$$r_i^l \sim Bernoulli(p), \tag{4.8}$$

$$\widetilde{\boldsymbol{z}}^l = \boldsymbol{r}^l \circ \boldsymbol{z}^l, \tag{4.9}$$

$$z_i^{l+1} = g(\boldsymbol{\omega}_i^{l+1}\widetilde{\boldsymbol{z}}^l + b_i^{l+1}), \tag{4.10}$$

where $\boldsymbol{r}^l$ is a vector of independent Bernoulli random variables each of which has probability p of being 1 and where this vector is multiplied in an element-wise fashion with the outputs of that layer to create the thinned outputs $\widetilde{\boldsymbol{z}}^l$. The process is then repeated for every propagation and at test time the weights will be scaled as $W_{test}^l = pW^l$. This method can also be seen as adding random noise to the neurons, preventing them from responding to too much noise in the dataset and thus improving generalisation of the whole network.

Furthermore, Dahl *et al.* (2013) has shown that using dropout with ReLUs improves the performance and accuracy of deep NNs, which is why this regularisation technique will frequently be used in the empirical chapter of this thesis. The use of batch normalisation has the same effect on the noise of the model which is why it is normally also used right before dropout.

Recall that it was mentioned in Chapter 3 that the SeLU activation function for hidden layers should be used in conjunction with a weight initialiser and a regularisation method. The weight initialiser is the He initialisation technique while the regularisation method is a modified version of dropout called AlphaDropout.

This technique was discovered by the authors of SeLU when they found that instead of setting the activations to zero (which happens in regular dropout and works well for ReLU activations), setting them to the negative saturation value of the SeLU activation function produces improved results. This way AlphaDropout preserves the mean and variance of the inputs of a layer with SELU activation to their original values, to ensure that the self-normalising property holds even after dropout. Using AlphaDropout is easily done in Keras when building CNN architectures. Research on this technique is not as plentiful as research based on regular dropout, therefore the empirical chapter of this thesis will compare AlphaDropout and SeLU method with regular dropout and ReLU.

4.4.4 Data Augmentation

It is usually the case that a statistician encounters the problem of having only a finite subset of the full data distribution for training a model. Theoretically, having the full data distribution for training and testing models would ensure that the selected model captures all the information in

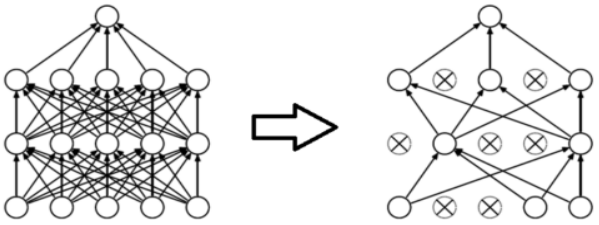


Figure 4.15: Illustration of using dropout with $p = 0.5$ on an ANN.
Source: Srivastava *et al.* (2014).

the data to make optimal future predictions on unseen data. In practice, this is not the case.

To combat this problem, one could artificially expand the input data through augmentation (Goodfellow *et al.*, 2016). In image classification settings, data augmentation consists of transforming the available images into new images without losing any semantic information. This is usually done by rotating the images, flipping, scaling, shifting or various other transformation possibilities (see Figure 4.16). This method will increase the amount of data to train on which would indirectly avoid overfitting. The transformations are easily done in Keras using the ImageDataGenerator function. This technique forms part of data pre-procession and can lead to higher accuracies obtained from the used model. This method will be frequently used in the empirical chapter.

4.5 VISUALISING THE INTERNAL OPERATIONS OF CONVOLUTIONAL NEURAL NETWORKS

The pitfall of most DL models is their model interpretability. Many researchers will comment on the fact that these DL models are completely black box, i.e. as performance increases, the number of parameters increases resulting in the decline of model interpretability (see Figure 4.17). This is a well-known trade-off in statistical learning theory. However, throughout the years, many techniques have been brought forward to deal with this criticism. These techniques give insight to what is happening internally when a NN and CNN, in this case, is trained. These methods are discussed next and will be used in the empirical chapter of this thesis to gain insight on the decisions that the CNN is making. Having such insight could have a number of advantages such as:

- Making the network easier to tune since errors can be identified in the training phase.

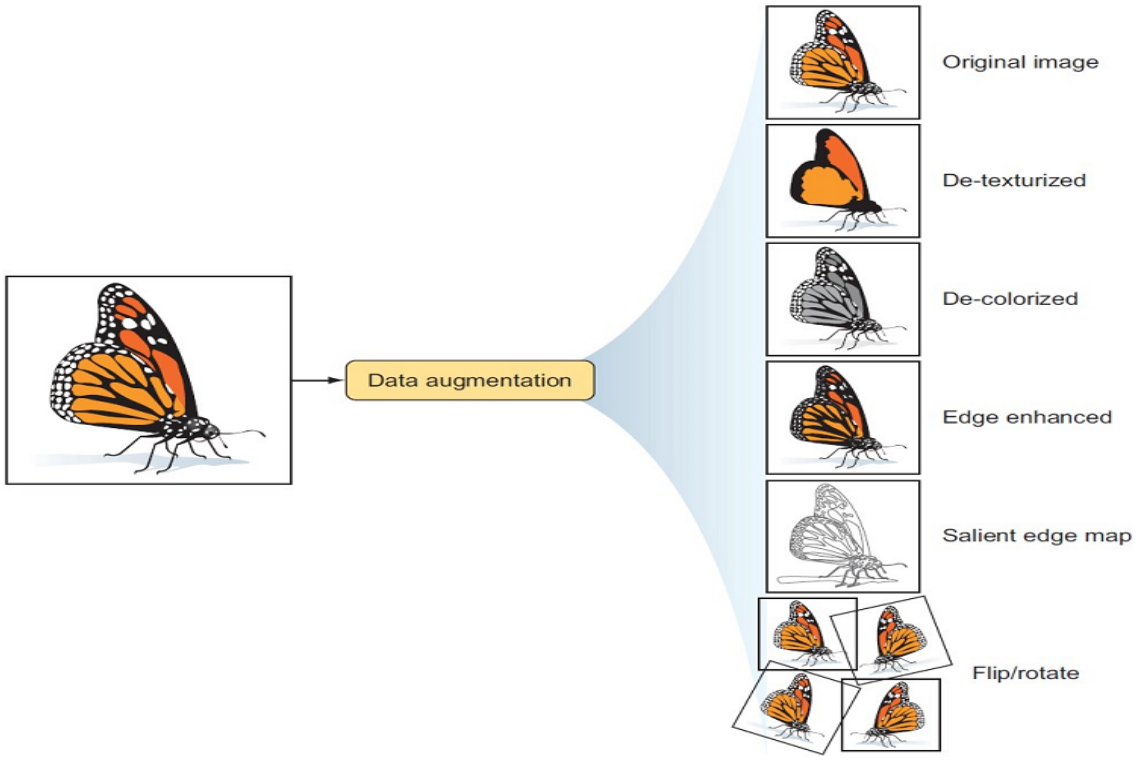


Figure 4.16: Data augmentation example using various methods as shown on the right. Source: Elgendy (2020).

- The functionality and behaviour of the networks can be explained in-depth, easing the deployment of these networks in practice.
- Obtaining the strength and weaknesses of the network, thereby having knowledge and confidence to improve the overall design.

4.5.1 Activation Maximisation

This method indicates how successive convolutional layers transform their input by visualising intermediate activations (Erhan *et al.*, 2009). These intermediate activations display the feature maps that are output by the different convolutional and pooling layers in a network where the inputs are ranked by the magnitude of their activations. This allows the user to understand what sort of patterns activate a particular feature and what the network's perception of a given input class will be (see Figure 4.18). This method is easily implemented in Keras using their built-in visualisation package.

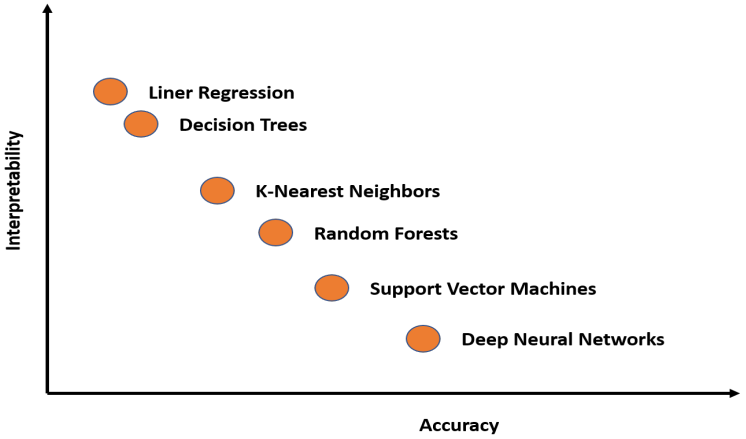


Figure 4.17: Illustration of the interpratibility and accuracy trade-off encountered when using different ML algorithms.
Source: Yang and Bang (2019).

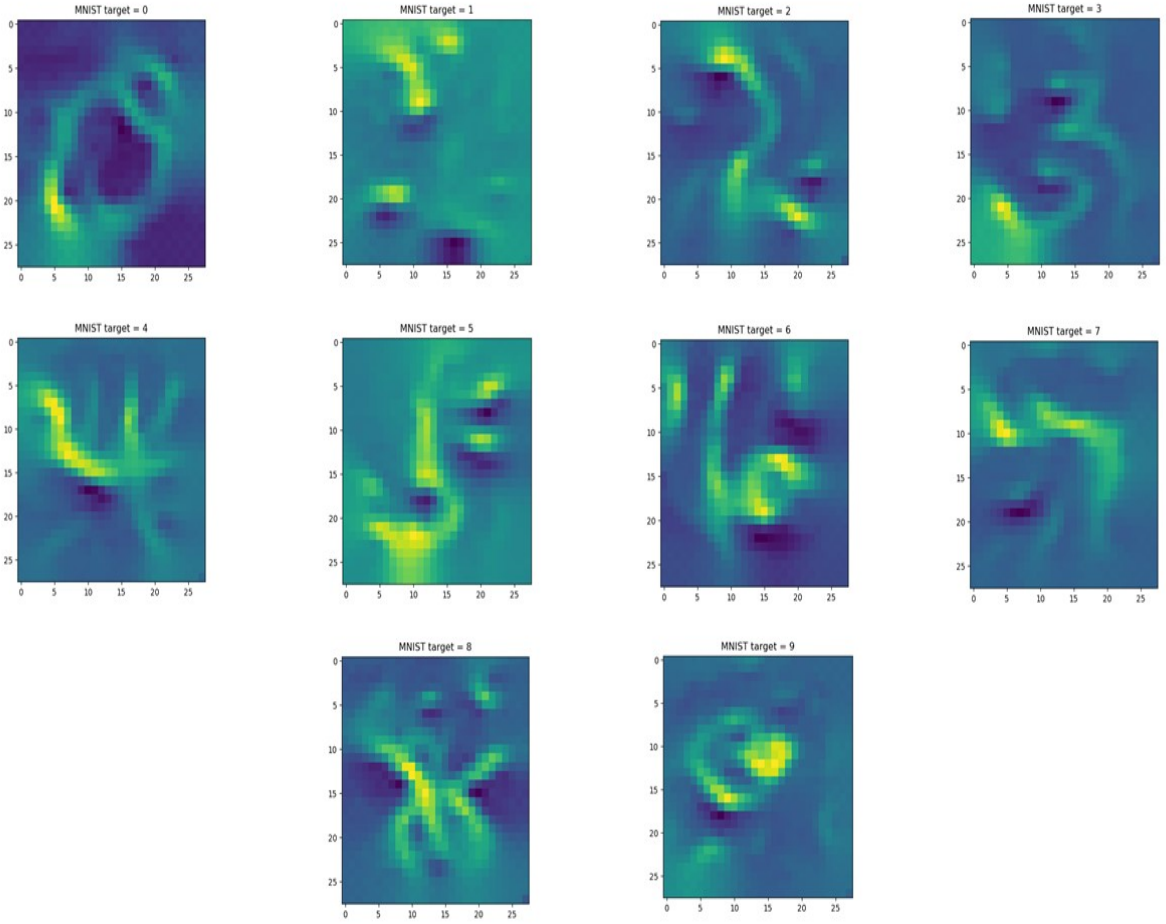


Figure 4.18: Activation maximisation example on handwritten digits.
Source: Stewart (2019).

4.5.2 Saliency Maps

This visualisation technique distinguishes unique features in an image such as edges, resolutions, and more to show which parts of the image is considered as important (see Figure 4.19) when the model is classifying the respective image into a certain class (Simonyan *et al.*, 2013).

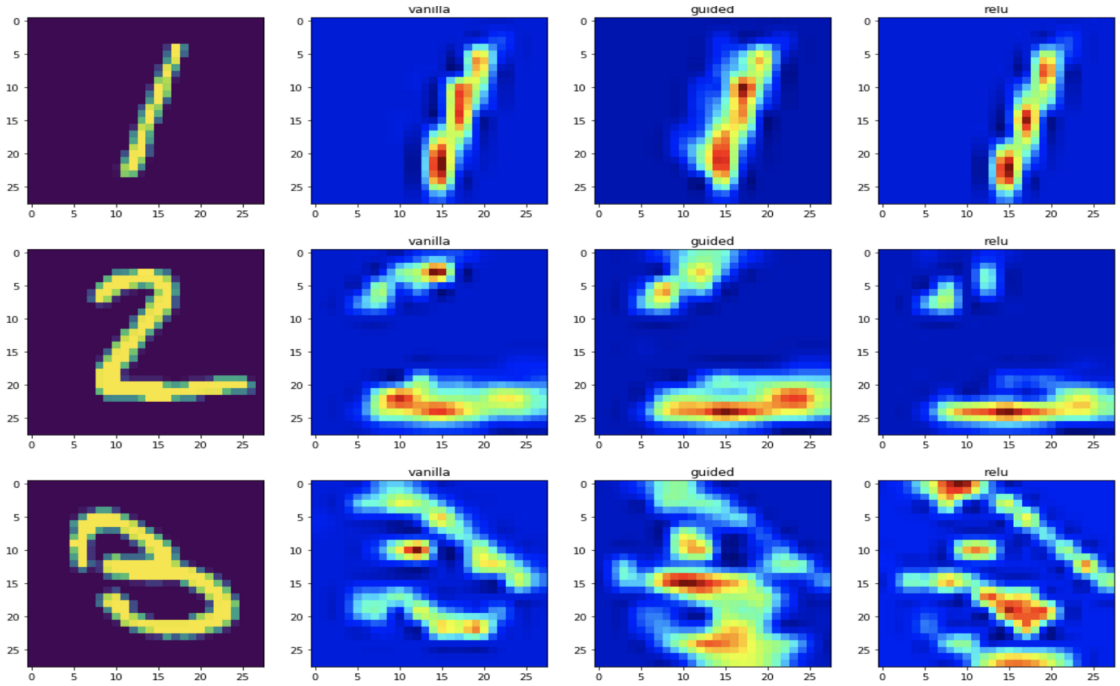


Figure 4.19: Saliency map example on handwritten digits.
Source: Stewart (2019).

4.5.3 Filter and Feature Visualisation

This method allows for close examination of the hierarchical nature of features within a CNN. By visualising the filters, the user can conceptualise what and how the CNN has learnt from its inputs. For example, Figure 4.20 indicates the activations for five layers with corresponding input images from a model used by Zeiler and Fergus (2014). The figure indicates how the filters tend to become more complex and abstract in higher layers when a complex model is being used.

Figure 4.21 illustrates another example taken from Lee *et al.* (2009), on how features evolve through different layers of facial images. The research on how CNNs learn have become plentiful throughout the years. Wang *et al.* (2020) shows another method where users can interactively use their website dashboard to visualise and inspect the data transformation done by the CNN as well as the features learnt in the intermediate layers. Additional research on using interactive dashboards have also become plentiful such as the case where Harley (2015) illustrates how CNNs

learn to recognise handwritten digits from the MNIST dataset (more on this dataset in Chapter 6). This illustration is evident in Figure 4.22 where all the CNN layers that the author defined is visualised. This method is easily implemented in Keras through their visualisation packages.

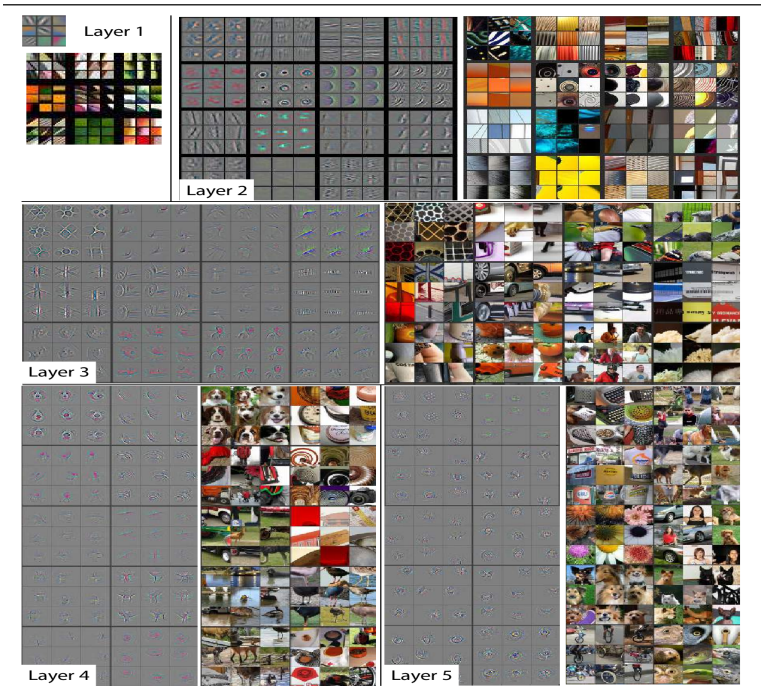


Figure 4.20: Filter visualisation example used by Zeiler and Fergus (2014).

4.5.4 Heat Map

These maps are specifically useful when the user wants to identify which part of an image led to the final classification. This method becomes particularly important when analysing misclassified data. Through the use of the gradients with respect to the final convolutional layer, one can find the importance of a certain class by weighing the gradients against the output of that final layer. Figure 4.23 shows an example of this method through the classification of a flower input image, the daisy.

4.6 SUMMARY

This chapter discussed important structures underlying CNNs. The justification of why CNNs should be used rather than ANNs for image analyses was stated. The training of CNNs has been discussed as well as the regularisation methods needed to appropriately train these models for practical deployment, was outlined. The chapter ended with a discussion on the black box

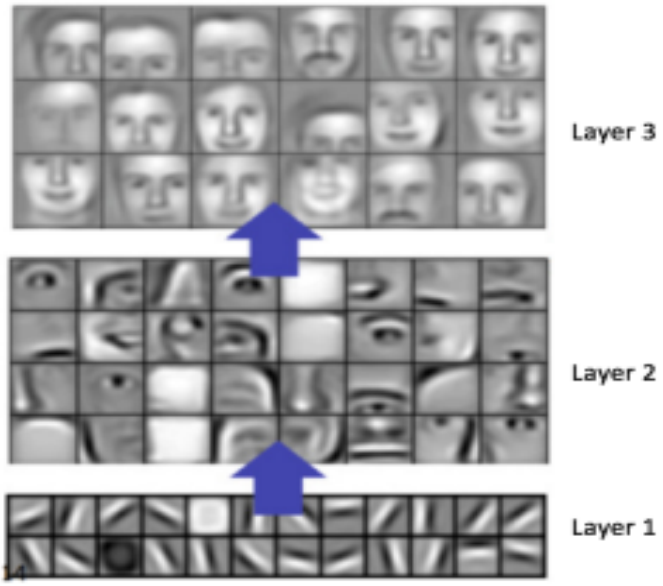


Figure 4.21: Filter visualisation example based on facial feature data.
Source: Lee *et al.* (2009).

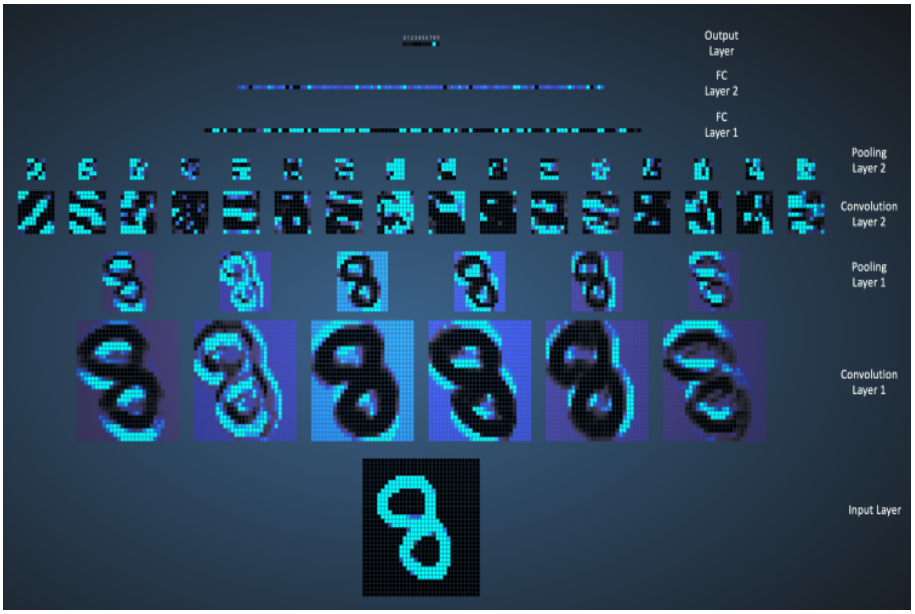


Figure 4.22: Filter visualisation based on the MNIST dataset showing how each layer interprets its output.
Source: Harley (2015).

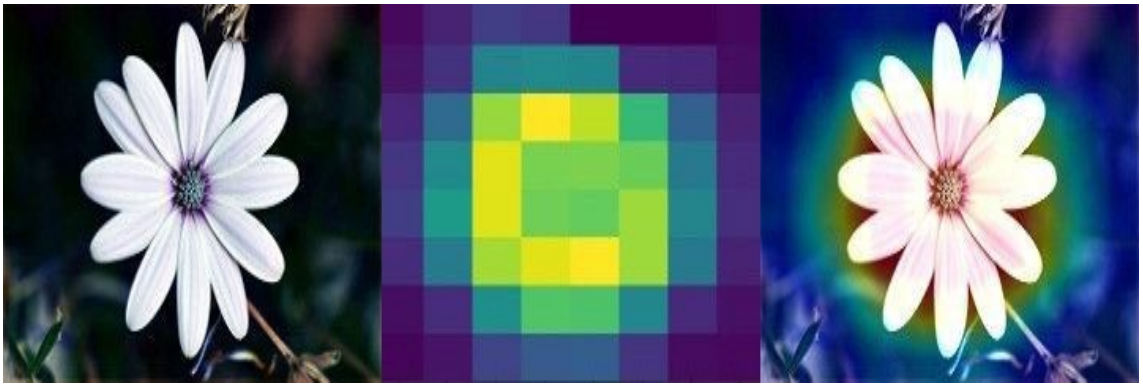


Figure 4.23: Heat map containing an input image of a daisy showing how the CNN perceives its input.
Source: Paliwal (2018).

terminology that researchers often label CNNs with. Methods to visualise filters and layers were also discussed to deal with the criticism that CNNs receive by being too complicated.

CHAPTER 5
STATE-OF-THE-ART ARCHITECTURES

5.1 INTRODUCTION

The introduction of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) from 2010 has brought forward many state-of-the-art CNN architectures. ImageNet is a project that was started by the Stanford Vision Lab at Stanford University. It is one of the largest databases of annotated images for visual recognition research. The rules of the competition undergo minor changes every year, however, the task remains to classify 1.2 million images into 1000 categories.

The breakthrough in CNN performance was introduced by an architecture designed by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton in 2012, called AlexNet (Krizhevsky *et al.*, 2012). This architecture achieved a significant test error rate of 16.4% on the ImageNet set which showed exemplary performance compared to conventional CV techniques. Before this breakthrough in architectural design, LeNet designed by LeCun *et al.* (1998), was the method of choice for solving image recognition problems.

From 2013, the competition saw a large increase in the number of submitted architectures. In 2014, two famous architectures were brought forward. The first one was the winner of the competition, namely GoogLeNet or InceptionNet, which achieved an error rate of 6.7%. This architecture was significantly different from traditional CNN structures since it made use of so-called inception modules (Szegedy *et al.*, 2015). The other architecture was the runner-up in the challenge and was called VGG as designed by Karen Simonyan and Andrew Zisserman (Simonyan and Zisserman, 2014). This architecture has a similar structure to that of traditional CNNs, however, it was designed to be very deep and used over 130 million parameters. This architecture did manage to win the object localisation part of the ILSVRC and is still extensively used today for different recognition tasks.

In 2015, ILSVRC was won by the Microsoft team where they brought forward an architecture called ResNet whereby it achieved an error rate of 3.6%. This is an astounding improvement from AlexNet not only in the reduction of the error rate with improved accuracy but also in the depth used in the model. This model broke the record by having 152 layers which was an enormous task to train.

Results of ILSVRC after 2015 did not bring any major or revolutionary improvements. Several attempts are still brought forward to improve the performance of CNNs, especially that of the reduction in computational cost since the deployment of deep and wide architectures requires a significant amount of internal hardware memory. This brought forward new research opportunities

and trends to design more shallow architectures without compromising the performance of the model.

The above-mentioned architectures contain millions of parameters, but they can be deployed using software such as TensorFlow and Keras from a home computer using pre-trained weights from the ImageNet set. TL allows for this exercise since it introduces the ability to allocate knowledge from one domain to another without loss of generality.

This chapter aims to discuss the above-mentioned architectures (see Figure 5.1) to obtain insight into the design of different CNN models. This insight will then be used to design our architecture for visual recognition that will be used in Chapter 6 to gain empirical knowledge on how a custom-designed architecture will compare, in terms of accuracy and computational cost on different real-world datasets, to the above-mentioned architectures. This comparison will be allowed since TL will be used, which will also be discussed in more detail.

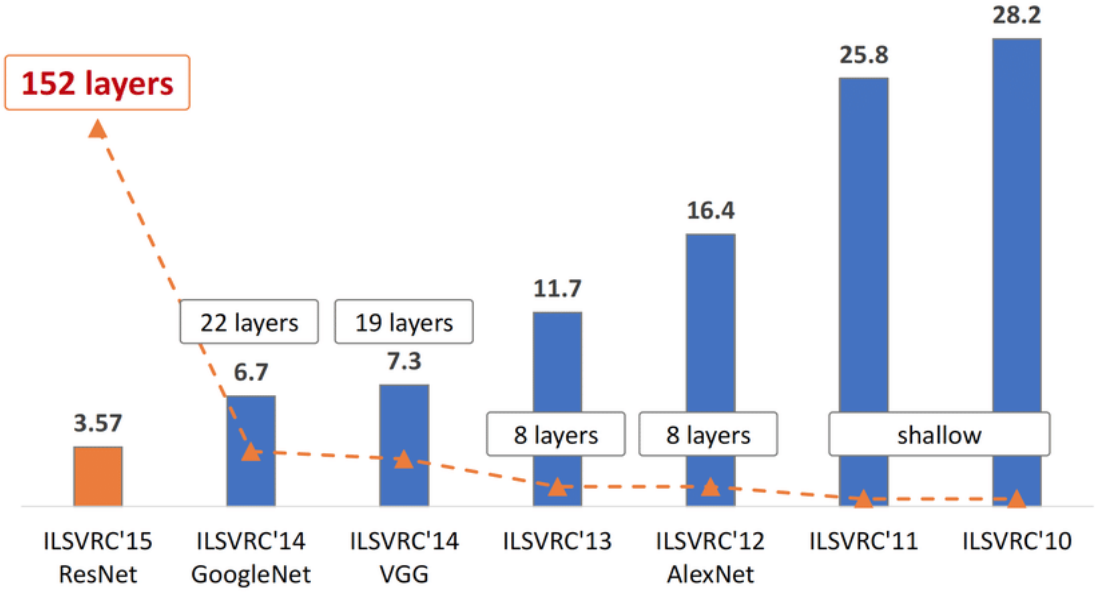


Figure 5.1: History of the test errors achieved by the state-of-the-art architectures based on ImageNet.
Source: Zhou (2018).

5.2 ADVANCED ARCHITECTURES

5.2.1 LeNet

This architecture was already mentioned indirectly in Chapter 1 and Chapter 4. It is considered as the baseline for CNNs and has paved the path for more advanced architectural structures. This architecture follows the usual method of design as described in Chapter 4. It was the first success-

ful CNN and is perhaps the most widely known CNN, where it was mostly used for recognising handwritten digits from images.

The usefulness of this architecture came from the fact that it is able to classify digits (from 0 to 9) in an automatic way using raw pixels, without being affected by small distortions, rotations, or variation of position and scale. This architecture can be summarised in Figure 5.2.

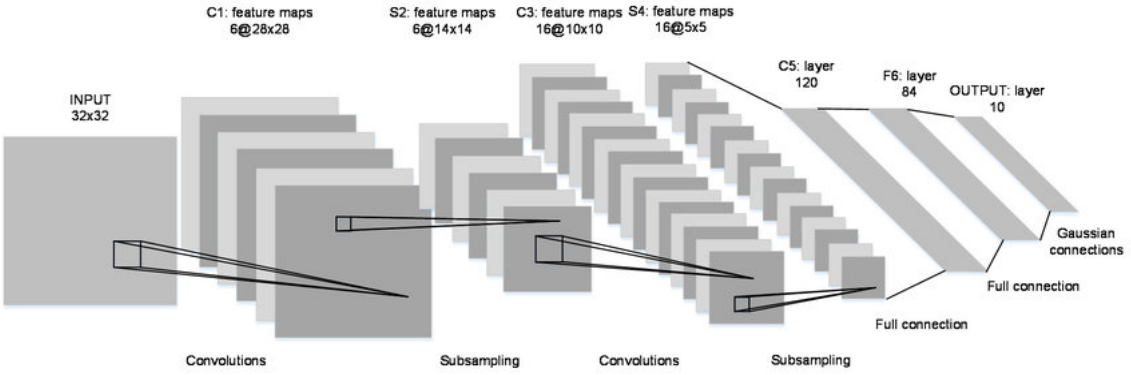


Figure 5.2: Structure of LeNet.
Source: LeCun *et al.* (1998).

The architecture operates on a greyscale input image of size 32×32×1 pixels or 28×28×1 surrounded with padded pixels. The input images were pre-processed to be normalised and centred. The architecture then follows with a convolutional layer with a kernel of size 5×5 which outputs a volume of 28×28 in the first layer. The authors used 6 filters or feature maps, i.e. a depth of 6 in the first layer with a stride of 1 followed by an average pooling layer, which produced an output of 14×14×6. This process is then continued and the whole architecture consisted of 7 layers, 5 of which have trainable weights and 2 that have no trainable weights. The activation function that was used is tanh in the intermediate layers and Softmax in the final layer for classification.

The architecture might seem simple having a shallow depth, but it still had about 60 000 parameters to train and about half a million matrix multiplication operations. The exact number of parameters to be trained in each layer is depicted in Table 5.1. The most weights to be trained is usually in the first fully connected layer, which in this case consisted of about 78% of all the weights to be trained in the model.

5.2.2 AlexNet

AlexNet was the breakthrough for CNNs in visual recognition tasks. This architecture is a deeper and wider version of LeNet. This was the first architecture to stack convolutional layers on top of each other, instead of stacking a pooling layer on top of each convolutional layer. The architecture

Table 5.1: Details of LeNet layers.

Layer	Input Volume (W×H×D)	Kernel Volume (W×H×D/S)	Output (W×H×D)	Parameters
C1	32×32×1	5×5×6	28×28×6	1×5×5×6+6=156
S2	28×28×6	2×2/2	14×14×6	0
C3	14×14×6	5×5×16	10×10×16	6×5×5×16+16=2 416
S4	10×10×16	2×2/2	5×5×16	0
C5	5×5×16	5×5×120	1×1×120	16×5×5×120+120=48 120
F6	1×1×120	1×1×84	1×1×84	120×1×1×84+84=10 164
F7	1×1×84	1×1×10	1×1×10	84×1×1×10+10=850
Total				61 706

is given in Figure 5.3.

The authors split the network into two parts to ease training on different GPUs, for example, the network contained 8 layers and started with an input image of size 224×224×3 with a kernel of size 11×11, with a depth 48 and a stride of 4 on one GPU and a copy on the other GPU, resulting in 96 filters that needed to be trained. The authors also used data augmentation and dropout to improve model generalisation while making use of the ReLU activation function (Krizhevsky *et al.*, 2012). The two GPUs were used not to speed up training but to increase memory needed to train the architecture since it contained more than 60 million parameters. Since at that time, they only had access to NVIDIA GTX 580 GPUs which only had 3GB of memory on each.

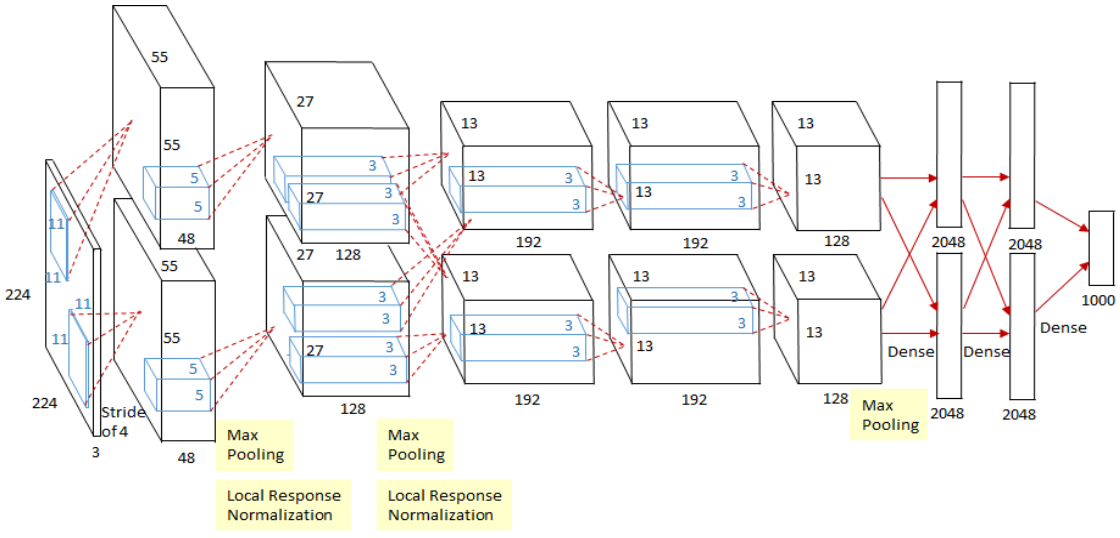


Figure 5.3: Structure of LeNet.
Source: Tsang (2018).

Instead of batch normalisation, which is currently used in most CNN architectures, they made use of local response normalisation. This type of normalisation locally normalises neuron activation

across the different dimensions of the layers. The way it was done is according to the following formula:

$$b^i_{x,y} = \frac{a^i_{x,y}}{(k + \alpha \sum_{j=max(0,i-\frac{n}{2})}^{min(N-1,i+\frac{n}{2})} (a^j_{x,y})^2)^\beta}, \tag{5.1}$$

where:

- a = Activation of neuron.
- $i = i^{th} Kernel$.
- N = Total number of kernels.
- k, n, α, β = Hyperparameters.

The authors made use of the following values for the hyperparameters: $k = 2$, $n = 5$, $\alpha = 10^{-4}$, $\beta = 0.75$. They also made use of a batch size of 128, a weight decay (L2 regularisation) of 0.0005, a learning rate of 0.01, SGD with Momentum at a rate of 0.9 and a weight initialisation from a Gaussian distribution, $\mathcal{N}(0, 0.0001)$, to achieve the winning error rate in the ILSVRC. Table 5.2 indicates the number of parameters needed to successfully train the architecture, in each layer. Note again how the first fully connected layer contains most of the trainable parameters, equating to roughly 62% of all the parameters to be trained is found in this layer.

Some of the design choices used in this architecture became standard in later CNNs, for example, the use of the ReLU activation function is one of the most used activations found in existing architectures after AlexNet. This breakthrough renewed interest in DL in general, allowing for CNNs to become the state-of-the-art visual recognition tools.

Table 5.2: Details of AlexNet with input size of each layer and parameters.

Layer	Volume	Parameters
Input	224×224×3	(not counting biases)
Convolution	55×55×96	34 944
Max Pooling	3×3/2	0
Convolution	27×27×256	307 456
Max Pooling	3×3/2	0
Convolution	13×13×384	885 120
Convolution	13×13×384	663 936
Convolution	13×13×256	442 624
Max Pooling	3×3/2	0
Fully Connected	4096 neurons	37 752 832
Fully Connected	4096 neurons	16 781 312
Softmax	1000 neurons	4 096 000
Total		60 964 224

5.2.3 VGGNet

The Visual Geometry Group in Oxford, who developed VGGNet, continued the trend to use deeper and wider networks. They tested various layers of their architecture and found that using 16 or more layers proved the most sufficient in terms of reducing the error rate on unseen data. Although VGGNet did not manage to win the ISLVRC in 2014, it still brought forward important design mechanics which later became standard in future architectures. This architecture is also used in many different TL problems since it contains a simple homogeneous, but very deep structure which allows for proper training of spatial features in images.

The fundamental difference between VGGNet and that of AlexNet is that VGGNet reduced filter sizes with increased depth. In general, the use of more depth forces non-linearity and greater generalisation. VGGNet also contains more convolutional layers which allow for the capturing of more complex features. A notable feature in VGGNet is that it only allows for filters to be used that are of size 3×3 and pooling of size 2×2. The authors applied the convolutions with a stride of 1 using padding of 1, or same, to maintain the spatial trail of the output volume. Another interesting characteristic is that the number of filters was increased by a factor 2 after each max pooling layer. The main limitation of this model, and why it is rather used in a TL setting, is that it contains more than 130 million parameters which make it computationally inefficient to deploy on systems with low resources. Figure 5.4 and Table 5.3 summarises and analyses VGGNet with 16 layers, respectively.

Table 5.3 indicates the enormous amount of learnable parameters in this architecture. The authors used zero padding in each convolutional layer to ensure that no size changes take place, and they also used dropout with a probability of 0.5. Additionally, a batch size of 256 was used as well as data augmentation. The learning was set to 0.01, and a weight decay of size 0.0005 was used. Further, SGD with Momentum was used at a rate of 0.9 with Xavier initialisation for the weights. As mentioned before, more than 16 layers can be used where the authors also designed a variant using 19 layers, but this network did not perform as well on the ImageNet set than expected (Simonyan and Zisserman, 2014). For this reason, only the 16 layer variant will be shown in this chapter. Take note that the most parameters (roughly 75%) are again situated in the first fully connected layer, however, when TL is used, it was found that the complete removal of this layer has no negative effect or performance downgrade in the architecture's prediction ability (Lin *et al.*, 2013).

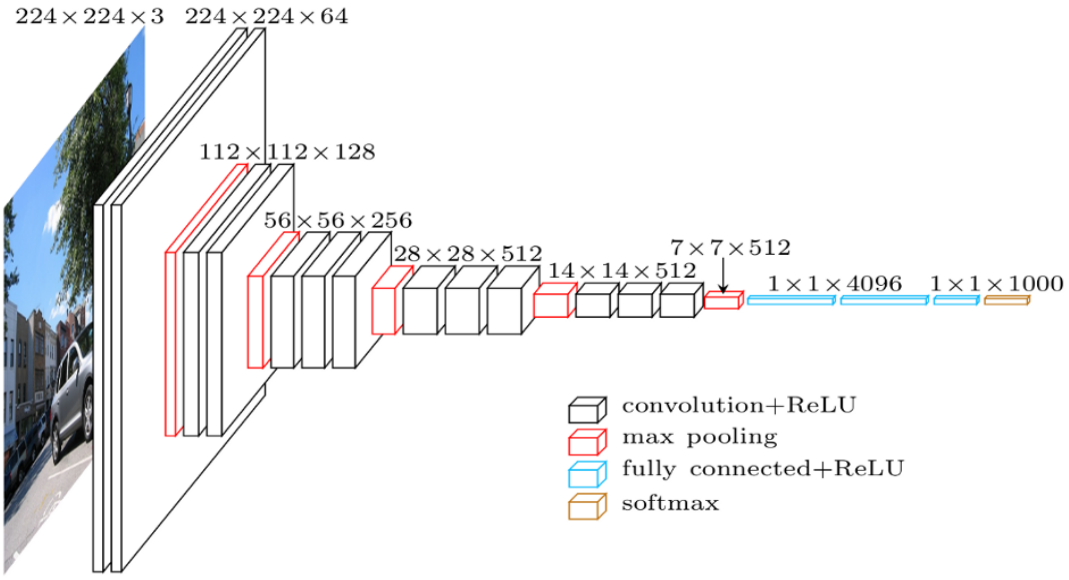


Figure 5.4: Structure of VGGNet.
Source: Hewage (2018).

Table 5.3: Details of VGGNet with input size of each layer and parameters.

Layer	Volume	Parameters
Input	224×224×3	(not counting biases)
Convolution	224×224×64	1 792
Convolution	224×224×64	36 864
Max Pooling	112×112×64	0
Convolution	112×112×128	73 728
Convolution	112×112×128	147 456
Max Pooling	56×56×128	0
Convolution	56×56×256	294 912
Convolution	56×56×256	589 824
Convolution	56×56×256	589 824
Max Pooling	28×28×256	0
Convolution	28×28×512	1 179 648
Convolution	28×28×512	2 359 296
Convolution	28×28×512	2 359 296
Max Pooling	14×14×512	0
Convolution	14×14×512	2 359 296
Convolution	14×14×512	2 359 296
Convolution	14×14×512	2 359 296
Max Pooling	7×7×512	0
Fully Connected	1×1×4096	102 760 448
Fully Connected	1×1×4096	16 777 216
Softmax	1×1×1000	4 096 000
Total		138 344 189

5.2.4 GoogLeNet Inception

The team at Google noticed that applying these architectures, having a large number of parameters, proves costly in practice so they developed the inception module which substantially reduced the number of training parameters in a CNN. The inception modules or blocks were originally inspired by the work of Lin *et al.* (2013), where the authors added a small network between the layers. These small networks were then branded by the Google team as inception blocks.

The inception blocks calculate filters of sizes 1×1, 3×3, and 5×5 in a parallel way with an applied bottleneck layer of a 1×1 convolutional filter to regulate computations (Szegedy *et al.*, 2015). After each convolutional layer, the team used ReLU as activation for the neurons. Each inception block (not layers) has four branches where it is two layers deep. These flows are then joined by a concatenation which collects the output of different branches. The full inception module is shown in Figure 5.5.

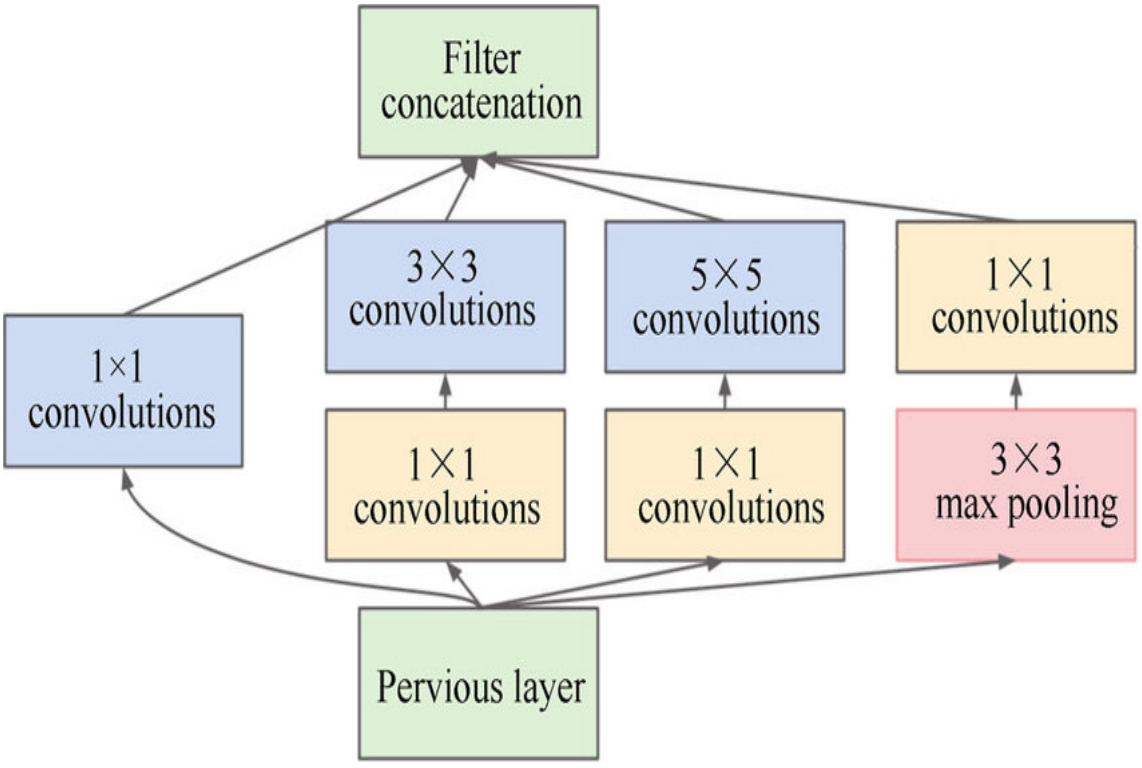


Figure 5.5: Illustration of an inception module and how concatenation works. Source: Szegedy *et al.* (2015).

Additional to the design of the architecture, the team introduced two auxiliary classifier networks that are connected to the intermediate layers. These auxiliary networks are used to increase training time such that the network can discriminate faster through the layers while providing some

regularisation. Figure 5.6 illustrates an example of such an auxiliary network. The full architecture is shown in Figure 5.7.

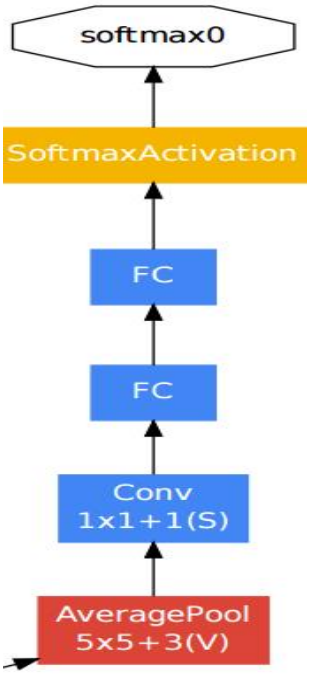


Figure 5.6: An auxiliary network used in InceptionNet to decrease training time. Source: Szegedy *et al.* (2015).

The authors of this architecture managed to reduce the number of parameters in the 22 layer network to about 5 million. They used a batch size of 32 to train this architecture while implementing image augmentation. A weight decay value between 0.0001 and 0.0005 was used and the learning rate was set to 0.0015. SGD with Momentum was set at 0.9 and weight initialisation took place by sampling these weights from a Gaussian distribution with zero mean and small variance. They also weighted the auxiliary network classifiers with a factor of 0.3 while using local response normalisation after each convolutional layer in their first version of this architecture, called inception V1.

In a later version, called Inception V3, the team was the first to use batch normalisation to improve the network's training ability. This architecture contained more parameters, roughly 24 million with 42 layers, but it did manage to outperform Inception V1 (Szegedy *et al.*, 2016).

5.2.5 ResNet

Residual networks, long for ResNet, became the state-of-the-art architecture in 2015. This network was designed by the Microsoft team and made use of so-called skip connections or residual blocks. As discussed in Chapter 3, problems such as vanishing gradients when training NNs are caused by

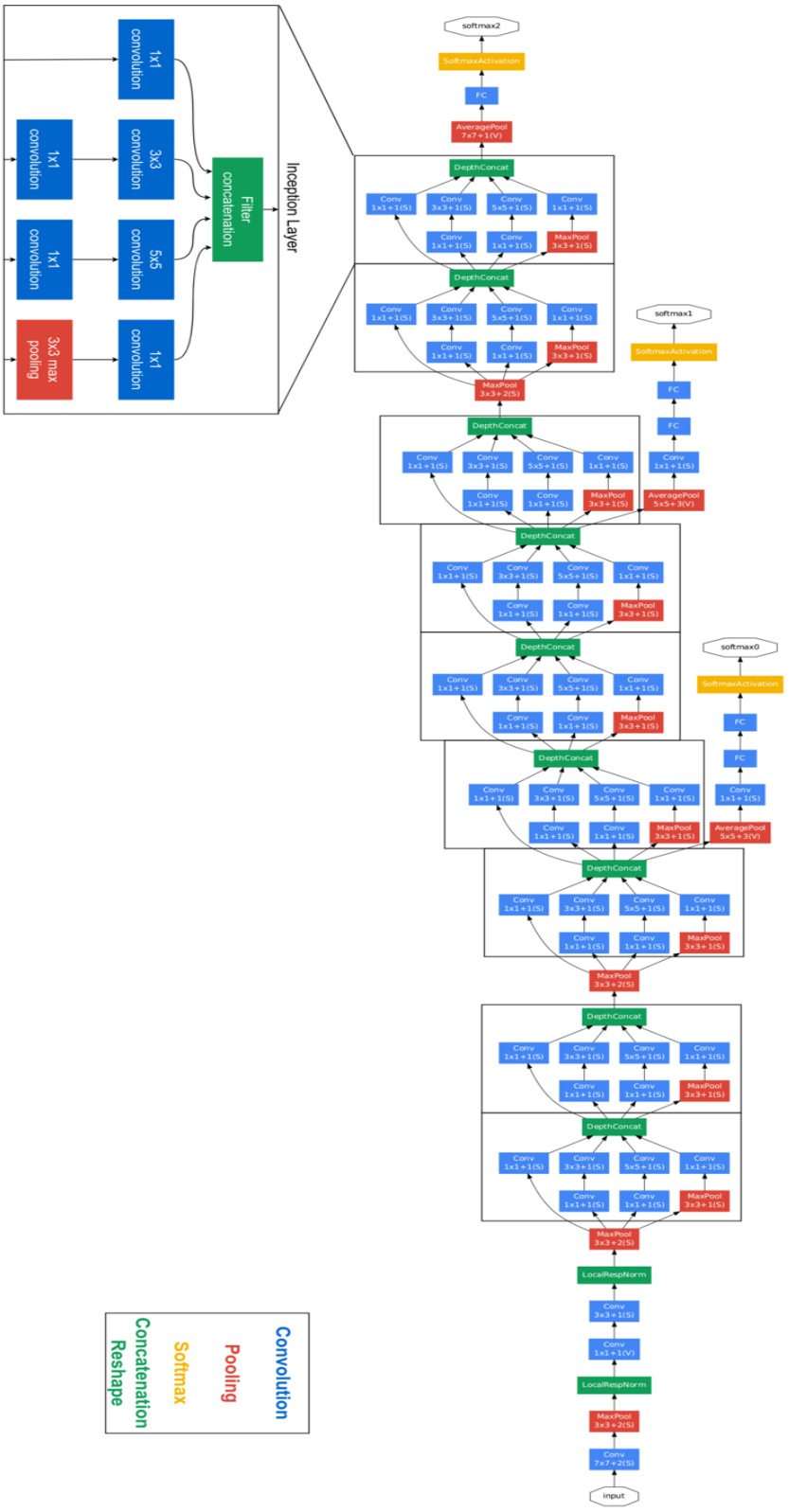


Figure 5.7: The full architecture of InceptionNet with all its layers and auxiliary networks. Source: Karpathy (2017).

increasing the depth of the network due to how the backpropagation algorithm works. However, He *et al.* (2016) noted that this problem is not necessarily caused by increasing the depth of the network, especially if batch normalisation is used, but rather that the main problem is caused by the ability of the model to converge in a sensible amount of time.

These convergence problems are most notable when networks have complex loss surfaces, as is the case in most CNNs. This problem brought forward the idea of using skip connections (or shortcuts) that perform identity mappings allowing a residual function with respect to an identity function to successfully counter the degradiation problem. The degradiation problem is when the addition of hidden layers in deep networks, at a certain point, lead to a decrease in training accuracy.

He *et al.* (2016) suggested that rather than using a group of layers to learn some non-linear mapping of the inputs, say $H(\boldsymbol{x})$, the objective should be to learn a residual function $F(\boldsymbol{x}) = H(\boldsymbol{x}) - \boldsymbol{x}$. The way in which this is done in CNNs is illustrated in Figure 5.8. Here, the figure shows that as the network becomes deeper that it is difficult to learn $H(\boldsymbol{x})$, so the proposed method is to use a skip connection to learn $F(\boldsymbol{x})$. The signal feeding into a layer is therefore also added to the output of a layer located higher up in the network.

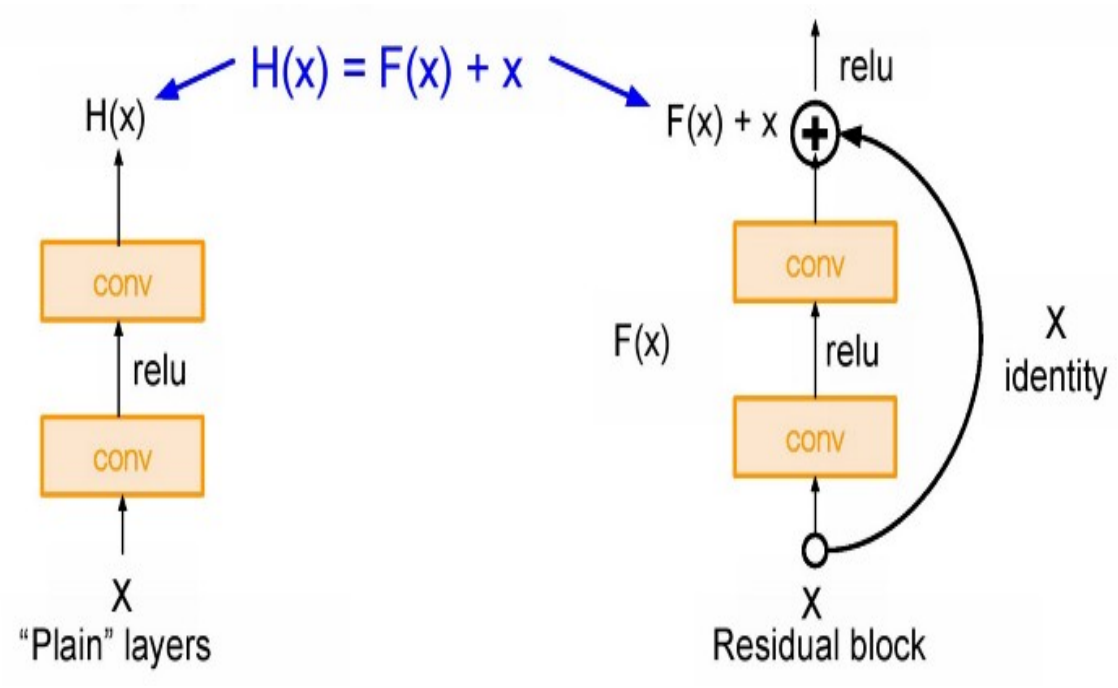


Figure 5.8: The working and structure of a residual block.
Source: Lin *et al.* (2013).

Skip connections used near the beginning of a network will, therefore, output a copy of its inputs, which allows for faster training meaning that initially the model only trains an identity function. Further down the network, rather than waiting for the gradient to propagate back one layer at

a time, skip connections allow gradients to reach the starting nodes with greater magnitude by skipping some layers in between the network.

In ResNet, the Microsoft team made use of 152 layers where they used batch normalisation after every convolutional layer. They used Xavier initialisation and SGD with Momentum at a rate of 0.9 while setting the learning rate at 0.1. They also made use of a batch size of 256 and a weight decay of 0.0005. An interesting note is that they did not use dropout in their network.

5.3 TRANSFER LEARNING

The above-mentioned architectures are rich in parameters and layers. While they attain state-of-the-art performance on ImageNet, training them on a CPU is costly and almost impossible. Training them on arrays of GPUs might also take several days or weeks. TL reduces this training time by using the knowledge of a pre-trained model which is transferred to another task. Either the full architecture is transferred or some parts of it such as the first layers, depending on available hardware power. This does not mean that the whole architecture is transferred, but rather the weights of the pre-trained model. There exist two common approaches to TL.

The first is to use the pre-trained model only as a feature extractor (Oquab *et al.*, 2014). This implies that the pre-trained model is only used up to a certain layer for the purpose of generating features. These features are normally used as the input to another ML algorithm for training another task. It is then possible to build on top of the transferred layers some additional layers which are then learned while keeping the transferred layers 'frozen' or constant meaning that weights of the transferred model will not be updated.

The second approach is that of fine-tuning (Yosinski *et al.*, 2014). In the context of CNNs, this is where a pre-trained network (such as ResNet) is taken by using the already trained architecture's (usually on ImageNet) convolutional layers to aid a new image classification task. The pre-trained model or a part of the model is transferred while additional layers are then added but in this case, the weights of the transferred model can also be updated.

In many CNNs, the first convolutional layers learn low-level features and as the model progresses through the layers, higher-level features or patterns are learned. In fine-tuning, the low-level layers are usually kept frozen and only the high-level layers are trained for new classification purposes. The method of TL has several benefits, especially when the pre-trained models are learned on datasets that contain millions of images. The lower level features learned from a dataset containing millions of examples could prove useful for similar tasks. Training the final classifier is also faster since the weights of the pre-trained model are usually well initialised. Lastly, TL can

also improve the performance of architectures trained on small datasets as indicated by Yosinski *et al.* (2014), where the authors showed the use of transfer learning and fine-tuning improved the generalisation of their tested datasets.

In Chapter 6, the architectures in Section 5.2 will be used from a TL and fine-tuning perspective to obtain insight and comparison on the next custom-designed architecture. The above-mentioned architectures will be pre-trained on ImageNet and are freely available through the Keras software package.

5.4 PHASES OF DESIGN

So far, an in-depth analysis of the structure of CNNs has been discussed as well as the different state-of-the-art architectures that exist in current literature. These architectures mostly follow the same underlying phases of design. These include a dataset, a model, the model hyperparameters and a testing criterion to obtain a measure of model generalisation.

To design a CNN architecture, a proper construction method is required. This construction method usually follows systematic steps to obtain its desired goal. The desired goal for an image classification task is usually to obtain the same accuracy and testing error as that obtained by the state-of-the-art architectures previously discussed. To reach this goal, a custom-designed architecture has to successfully move through different stages or phases to be considered as useful for future implementation in CV problems.

These phases are outlined in the diagram below, Figure 5.9. This diagram will be used to ensure that the designed architecture(s) in this thesis are successfully implemented in order to reach its desired goal, which includes that of having accuracy and testing error close to the TL models that it will be compared against. Figure 5.10 then indicates the full structure of a CNN together with the possible hyperparameters that can be used to successfully train the architecture. These hyperparameters will be tuned for each dataset in Chapter 6 with a full discussion on the selected hyperparameters.

Referring to Figure 5.9, given a specific dataset, the data must be firstly pre-processed meaning that class imbalance and missing data should be adjusted for. It is also important, for CNNs, that all the input images are normalised and adjusted to the same scale of pixels. Inspection should then be done to ensure that the images are in the correct format and that the model has sufficient data to train on. If the model lacks data, data augmentation can be done. The CNN architecture is then created with its required hyperparameters and trained. The model is then tested with appropriate regularisation against unseen data to measure its generalisation ability while a detailed

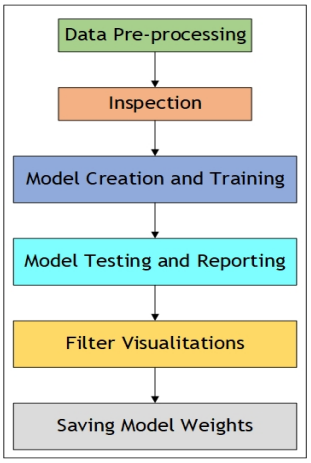


Figure 5.9: Phases of design that will be used to build CNNs for image recognition tasks.

accuracy and testing report is then submitted using the confusion matrix as discussed in Chapter 2. The full architecture is then analysed by using a filter visualisation method to obtain a clear view of how it learned from its specific inputs. Finally, the weights of the architecture are saved for future tasks such as TL or model adjustments.

5.5 A CUSTOM CNN ARCHITECTURE FOR IMAGE CLASSIFICATION

This part of the thesis introduces the final architecture that will be used to obtain empirical evidence on the effectiveness of CNNs on real-world datasets. The focus here is based on certain elements of a state-of-the-art architecture such as having a sufficient amount of parameters so that the model is easily deployed on a computer containing a single GPU; model interpretation eliminating the black box effect, and finally gaining classification accuracy that equalises or surpasses human recognition and TL model ability.

The architecture should reach similar performance to that of the above-mentioned state-of-the-art architectures. This architecture will be called LiteNet and will be considered as the baseline model. It will contain fewer parameters than that of AlexNet and VGGNet while containing fewer layers than that of GoogLeNet and ResNet. In order to reduce computation time compared to the other state-of-the-art architectures, a reduced amount of input pixels of each image will be used. This will be fixed to 128×128 for most of the datasets with some exception to small or greyscaled datasets which will be fixed at either 28×28, 32×32, or 48×48.

LiteNet will make use of ReLU, ELU, or SeLU as activation function to obtain a clear comparison on how these activation functions change the accuracy and test loss of the chosen model. The reason for the use of SeLU is to aid the research done on this activation function since research

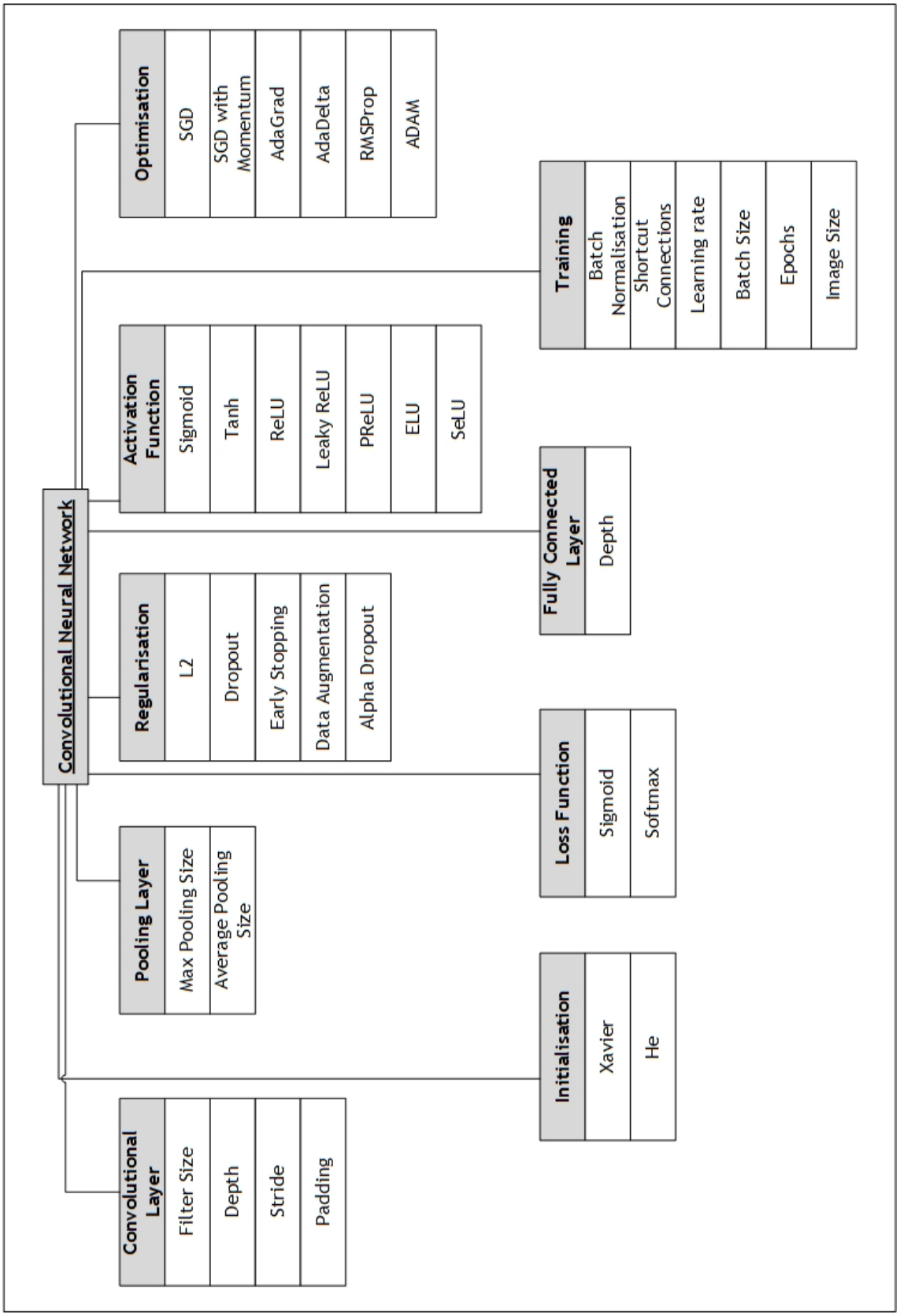


Figure 5.10: Structured taxonomy of CNNs indicating all the different hyperparameters discussed in this thesis.

on ReLU and ELU is far more in comparison to that on SeLU. An investigation on the number of parameters and layers needed to reach the desired accuracy on different datasets will also be done. For this reason, a subset of LiteNet, called ShallowNet, will be designed. The purpose of ShallowNet is to obtain an understanding of whether CNNs need a large number of parameters and layers to reach high accuracy and low test error. This architecture will shed light on the power of CNNs as to whether a deep model is needed or rather a small network to obtain similar performance. Smaller models enjoy the benefit of having faster training meaning that deployment of an accurate small model could prove beneficial in real-world tasks.

Finally, in the design of these two architectures, the stages outlined in Figure 5.9 will be evident in the Jupyter notebooks to ensure that these two architectures can be compared to the state-of-the-art architectures mentioned in Section 5.2. Appropriate specification on the hyperparameters will be clear on each dataset in Chapter 6. The last part of this chapter introduces the structure of the two architectures with some fixed hyperparameters.

5.5.1 LiteNet

This architecture consists of 11-layers as illustrated in Figure 5.11. The architecture is fed an input image, of size 128×128 for colour images or size of 28×28 for greyscaled images. The first block contains two convolutional layers of depth 64 each with a kernel of size 3×3 followed by batch normalisation operations. At the end of the first block, the dimension of the model is reduced through a max pooling procedure of size 2×2. Thereafter, dropout is done to further reduce the number of parameters in the full model.

The second block contains similar operations as the first block, but with a convolutional depth of 128. This is also the case for the third and fourth block where the only change is the depth of the convolutions, 256 and 512, respectively. These blocks are then sent to two fully connected layers with depth 512 followed by batch normalisation and dropout operations to improve the architecture's generalisation ability. The final layer is the classification layer having either Sigmoid or Softmax loss functions depending on the number of categories given as inputs.

5.5.2 ShallowNet

This architecture consists of 6-layers as illustrated in Figure 5.12. The architecture follows the same procedure as that of LiteNet while only having 2 blocks of convolutional operations each. The first block contains two convolutional layers with depth 128. The increased depth is to ensure that the model still has the ability to capture as much information as possible since it has fewer

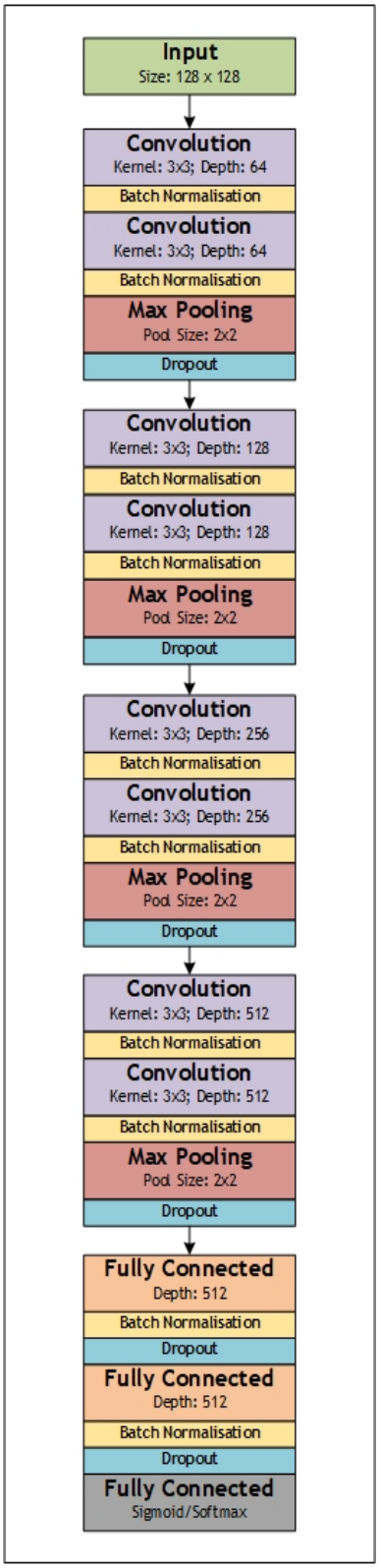


Figure 5.11: The LiteNet architectural design and structure.

layers compared to LiteNet. The second block takes on convolutional operations with depth 256. The pooling size after each block is also increased to 4×4 to ensure that the amount of parameters trained is minimal. These blocks are then sent to only one fully connected layer having a depth of 512.

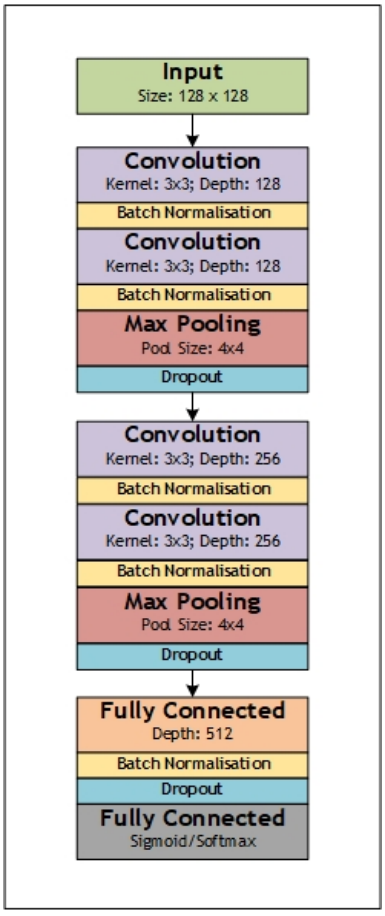


Figure 5.12: The ShallowNet architectural design and structure.

5.6 SUMMARY

In this chapter, famous state-of-the-art CNN architecture was introduced and discussed. The structure and inner workings of these models were outlined. Additionally, the phases of design combined with each architecture were discussed. These phases were directly used to design two custom CNNs for practical deployment. Lastly, TL was discussed which introduces a way to compare large architectures to other architectures on any given dataset.

CHAPTER 6
EXPERIMENTAL EVALUATION

6.1 INTRODUCTION

In the preceding chapter, the state-of-the-art architectures for image classification were introduced followed by two of our own designed architectures. In this chapter, the mentioned state-of-the-art architectures will be compared against the two designed architectures while the main objectives are to understand certain behaviours of these models. To gain a fundamental understanding of these architectures, they will be compared against one another using several datasets in several different settings. The reason for the use of several datasets is to compare different architectural generalisations.

Additionally, certain hyperparameters will also be tested against one another in different model settings. Hyperparameters such as the type of activation function, the amount of dropout used and the learning rate should be chosen to successfully train the model to convergence. The empirical work done here will complement the general observations found in literature as reported in the previous chapters. Therefore, the specification of the type of hyperparameters will be done with care and will be stated clearly.

The goal is to analyse if a custom-designed architecture can obtain high enough accuracy and low enough loss to that compared to the state-of-the-art architecture where TL will be used. Another goal is to compare the two custom-designed architectures in the same setting to obtain a clear overview of the relationship between model depth and the number of parameters needed to obtain satisfactory results. A satisfactory result in this setting is that of high accuracy on unseen data and low test loss without excessive overfitting.

In the following sections, the different datasets will be outlined followed by an in-depth evaluation of different architectures with different parameters on these datasets. The overall goal is not to beat the state-of-the-art architectures but rather to show that a smaller designed architecture can obtain satisfactory results on real-world datasets. These results could prove beneficial for companies employing these models in different settings rather than that of ImageNet.

6.2 THE DATASETS

Five datasets will be considered for proper evaluation of the proposed architectures. These are:

1. The MNIST dataset;
2. The Cats and Dogs dataset;

3. The Malaria dataset;
4. The CIFAR-10 dataset;
5. The Facial Expression Recognition 2013 dataset.

6.2.1 The MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is one of the most popular greyscale image datasets to test different ML algorithms (LeCun *et al.*, 1998). The dataset was indirectly introduced in Chapter 1 and Chapter 5. It consists of zip code data describing handwritten digits from zero to nine, i.e. 10 different classes to classify. Each image in this dataset consists of 28×28 pixels resulting in 784 attributes. The dataset consists of 60 000 training images and the test set of 10 000 images. Figure 6.1 illustrates the different classes. The aim is to use the digits in the format provided and correctly identify or classify the digit that was written. The dataset is pre-loaded into the Keras package. The analysis of this dataset could provide aid to the research done on classifying handwritten objects. The benefit is that an accurate model could convert handwritten digits into a computerised form by scanning written documents. This could reduce the time it takes to manually convert written documents into a computer.

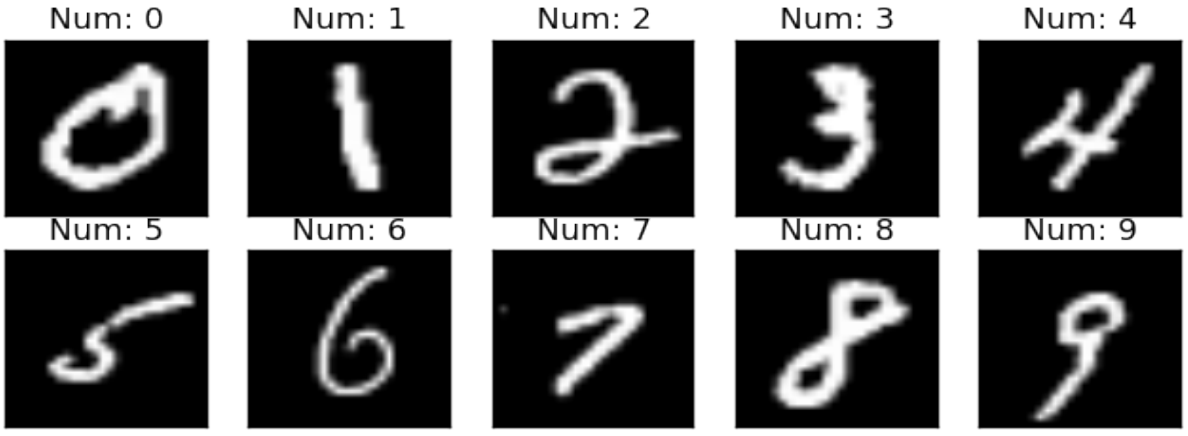


Figure 6.1: Sample images from the MNIST dataset with their associated class labels.

6.2.2 The Cats and Dogs Dataset

This dataset is known as the Dogs and Cats set and can be found online on Kaggle, a website containing thousands of freely available datasets and ML competitions (Kaggle, 2013). It consists of 25 000 colour images of dogs and cats in the format of different pixels and attributes. There are 20 000 training images and 5 000 test or validation images of the two classes.

Due to the different sizes of each image, it was decided to resize these images to 128×128 for appropriate and consistent training. The aim is to use the dogs and cats in the format stated and correctly identify or classify whether a previously unseen image contains a dog or a cat. Figure 6.2 illustrates the two different classes. The analysis of this dataset could potentially benefit social media companies to identify animal trends from online posted photos, to boost or change their marketing strategies.

Figure 6.2: Sample images from the Dogs vs. Cats dataset with a sample image of a cat on the left and a dog on the right.

6.2.3 The Malaria Dataset

This dataset was firstly used by Rajaraman *et al.* (2018), to improve Malaria parasite detection in thin blood smear images. The dataset can be found online at Kaggle (2018). It contains two classes, that of infected cells and uninfected cells with a total of 27 558 coloured images. There are 24 958 images in the training set and 2 600 images in the test or validation set, each equally distributed. The images have different pixels and attribute sizes, and will be resized to 128×128. The analysis of this dataset can potentially save lives. The accurate prediction of a Malaria infected cell, could aid the decisions done by Microscopists in resource-constrained areas and improve diagnostic accuracy. Rajaraman *et al.* (2018) managed to obtain accuracies between 90% and 95%. Therefore, the baseline accuracy score for architecture comparison purposes will be narrowed down to 90% for successful classification. Figure 6.3 illustrates the two different classes.

6.2.4 The CIFAR-10 Dataset

This dataset was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton (Krizhevsky *et al.*, 2009). It is freely available online at Kaggle (2014) and is pre-loaded into the Keras package. The dataset consists of 60 000 32×32 colour images in 10 classes, with 6 000 images per class. There

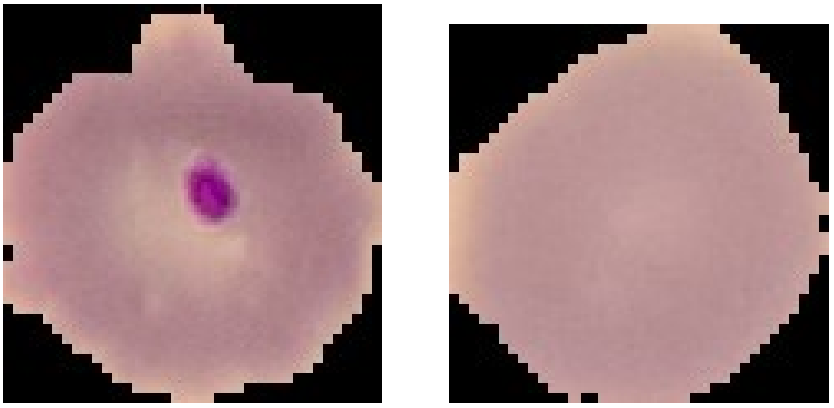


Figure 6.3: Sample images from the Malaria dataset with an infected cell on the left and an uninfected cell on the right.

are 50 000 training images and 10 000 test or validation images. The 10 different class labels are as follows:

0. airplane
1. automobile
2. bird
3. cat
4. deer
5. dog
6. frog
7. horse
8. ship
9. truck

This dataset can be seen as a very small subset of the ImageNet dataset. A proper and accurate model can aid object recognition tasks from small images such as images from mobile phones. The dataset can also provide aid to the performance of different hyperparameters with accelerated training due to the small number of attributes contained in each image. State-of-the-art results were obtained by Wistuba *et al.* (2019), with a test loss value of 1.33 which will be used as the baseline result for architectural comparison. Figure 6.4 illustrates the different classes found in the dataset.

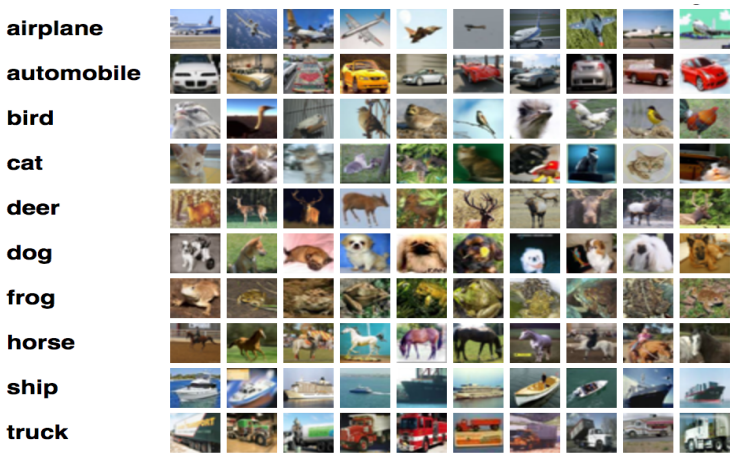


Figure 6.4: Sample images from the CIFAR-10 dataset.
Source: Kaggle (2014).

6.2.5 The Facial Expression Recognition 2013 Dataset

This greyscale dataset consists of 35 685 images of 7 different human facial expressions all of which are of size 48×48 (Pramerdorfer and Kampel, 2016). The 7 different class labels are as follows:

1. angry
2. disgust
3. fear
4. happy
5. sad
6. surprise
7. neutral

An illustration of the dataset is given in Figure 6.5. This is a challenging dataset with the state-of-the-art results only reaching around 70% accuracy (Pramerdorfer and Kampel, 2016). The reason for this is that the dataset itself consists of persons with different age, face pose and facial features. As seen in Figure 6.5, the expression of surprise can easily misguide the architecture to think that the classified emotion should be angry. For this reason, 3 of the emotions or classes in this thesis will be removed from the dataset. That is, disgust (might be misguided to be angry or sad), fear (might be misguided to be angry or sad) and surprise (might be misguided to be happy or neutral). The reduction of the dataset might reduce the model's ability to generalise well on new data but as found in the results section, this did not have a significant impact on model classification accuracy. Given this reduction in the data, the set now consists of 24 565 images of which 21 975 are found in the training set and 2 590 in the validation or test set. The analysis of this dataset can potentially

increase the effectiveness of facial recognition applications. The distribution of the classes is given in Table 6.1.

Figure 6.5: Sample images from the Facial Expression Recognition 2013 dataset. Source: Carrier and Courville (2013).

Table 6.1: The number of classes in the Facial Expression Recognition 2013 dataset.

Expression	Training Samples	Testing Samples
Angry	4965	491
Happy	7215	879
Sad	4965	626
Neutral	4830	594

6.3 EXPERIMENTS

The following section applies the two designed architectures as well as the mentioned TL architectures on the above-mentioned datasets. Different hyperparameters as shown in Figure 5.10 will be specified clearly and the processing pipeline stated in Figure 5.9 will be followed when experiments on the sets are done. The phases of design will be set out sequentially in the Jupyter notebooks where the reader can obtain the code that resulted in the final architectures of each dataset.

The main goal is to assess the performance of the top-performing activation functions in each architecture, that is, ReLU; ELU; and SeLU while maintaining model generalisation on unseen data. Additionally, optimisation techniques such as Adam and SGD with Momentum will also be tested against one another. Each dataset has a baseline performance measure as stated in Section 6.2. There are, however, some hyperparameters that were pre-chosen which will remain constant in all the experiments. These hyperparameters, including the way in which the two architectures were designed with corresponding filters, are given as follows:

- Loss functions:
  - → Binary cross-entropy will be used for 2 classes with Sigmoid as the classification function.
  - → Categorical cross-entropy will be used when there are more than two classes with Softmax as the classification function.
- Dropout (when used):
  - → Convolutional layers with $p = 0.10$ for LiteNet and $p = 0.25$ for ShallowNet.
  - → Fully connected layers with $p = 0.15$ for LiteNet and $p = 0.35$ for ShallowNet.
- Padding will be set as 'same'.
- Learning rate will start at 0.001.
- Weight initialisation will be set as He initialisation.
- L2 weight decay will be set at 0.0005 (as used in previous architectures).

6.3.1 MNIST

This dataset will be used to compare LiteNet against ShallowNet and LeNet. The first step is the data pre-processing step that was done. This resulted in the data being processed in the following form allowing a direct input tensor to the respective CNNs, that is, for the training set after normalisation and one-hot encoding the shape was given as (60 000, 28, 28, 1) and for the testing set the shape was given as (10 000, 28, 28, 1). The inspection step was done and the architectures were built without using any dropout since the original LeNet did not make use of dropout. The following hyperparameters were chosen additionally:

- Epochs = 10.
- Batch size = 128.

The following results were obtained in the tables below, respectively, using the different activation functions and optimisation techniques.

Interestingly, the architectures with their different hyperparameters all gained results above 90% accuracy meaning that any chosen architecture, in this case, has a high chance of correctly predicting a given unseen digit. Considering LiteNet, Adam as an optimisation method managed to outperform SGD in terms of test loss. This could be due to the fact that Adam converges faster than SGD in a 10 epoch situation. The opposite scenario can be observed when test accuracy is considered. In this case, SGD outperforms Adam especially in the case where ELU is used with SGD. This seems to be the overall trend when all the architectures are observed, that is, ELU with SGD and Momentum provides exceptional results when test accuracy is considered. Also, ReLU

Table 6.2: LiteNet results on the MNIST dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	5 220 554	0.24903	0.97589
ReLU with SGD and Momentum at a rate of 0.90	5 220 554	2.72734	0.98930
ELU with Adam	5 220 554	0.27854	0.95630
ELU with SGD and Momentum at a rate of 0.90	5 220 554	2.72943	0.99019

Table 6.3: ShallowNet results on the MNIST dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	1 173 386	0.13125	0.98229
ReLU with SGD and Momentum at a rate of 0.90	1 173 386	1.21363	0.98809
ELU with Adam	1 173 3864	0.29318	0.92909
ELU with SGD and Momentum at a rate of 0.90	1 173 386	1.20970	0.98879

Table 6.4: LeNet results on the MNIST dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	1 256 080	0.09331	0.98839
ReLU with SGD and Momentum at a rate of 0.90	1 256 080	0.59049	0.98369
ELU with Adam	1 256 080	0.10599	0.98320
ELU with SGD and Momentum at a rate of 0.90	1 256 080	0.58589	0.98530

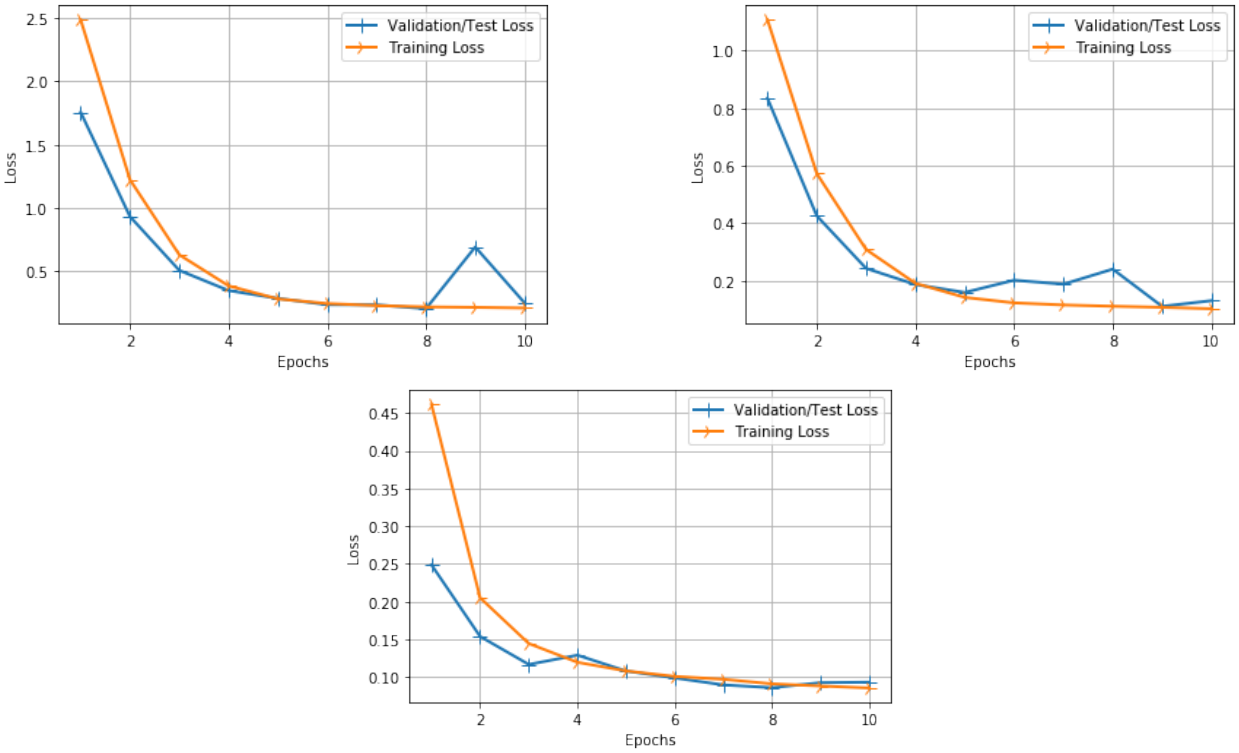


Figure 6.6: Top left (LiteNet), Top right (ShallowNet), and Bottom centre (LeNet) graphs indicating model generalisation ability using ReLU with Adam outputting the loss on the y-axis and the number of epochs on the x-axis.

with Adam managed to obtain the lowest test loss in each architecture.

Another interesting observation is that LiteNet contained 5 times the amount of parameters than that of ShallowNet and LeNet meaning training took longer. This could also give insight to the large test loss when LiteNet is used since 10 epochs might be too small of a number for such a large and deep architecture to reach satisfactory error. Consequently, ShallowNet and LeNet show that a large number of parameters or a deep architecture is not needed to obtain satisfactory results, however, LeNet was only designed for small-sized input images. The statement that ShallowNet performs almost as well as LiteNet in a general setting, making it the ultimate architecture between the two designed architectures, needs to be investigated further.

Lastly, to investigate the generalisation ability in terms of overfitting or underfitting of the different architectures, we considered only the top-performing model in terms of test loss. This was stated as ReLU with Adam in each architecture. The following graphs in Figure 6.6 indicate that each architecture using these specified hyperparameters generalises well on unseen data. Additional outputs of the other architectures in terms of training and validation or test loss graphs are available in the Jupyter notebooks.

6.3.2 Cats and Dogs

This dataset will be used to compare LiteNet against ShallowNet and VGGNet. Following the data pre-processing step, data augmentation was also implemented since the dataset is relatively small. The set contained 20 000 images as training data and 5 000 images as validation or test data. The data augmentation steps were done as follows:

- Rotation of 15 degrees.
- Rescaling by a factor of 1/255.
- Shearing by a factor of 0.1.
- Zooming by a factor of 0.2.
- Horizontal flip.
- Width and height shift by a factor of 0.2.

After this step, an inspection was done and the architectures were built. The following hyperparameters were chosen additionally:

- Epochs = 20.
- Batch size = 32.

The results in the tables below were obtained, respectively, using the different activation functions and optimisation techniques. Additionally, SeLU was also added for comparison.

Table 6.5: LiteNet results on the Cats and Dogs dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	21 732 161	0.34623	0.89483
ReLU with SGD and Momentum at a rate of 0.90	21 732 161	2.66825	0.89563
ELU with Adam	21 732 161	0.40903	0.90605
ELU with SGD and Momentum at a rate of 0.90	21 732 161	2.97800	0.83734
SeLU with Adam	21 732 161	0.55761	0.79727
SeLU with SGD and Momentum at a rate of 0.90	21 732 161	2.42888	0.75901

Notice a large number of parameters here which is due to the input images being of size 128×128 meaning that each image consists of 16 384 attributes. This significant increase in the size of the parameters will increase the time at which the model trains and might slow down convergence also.

Table 6.6: ShallowNet results on the Cats and Dogs dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	9 428 609	0.43417	0.94030
ReLU with SGD and Momentum at a rate of 0.90	9 428 609	1.61393	0.80689
ELU with Adam	9 428 609	0.33032	0.93970
ELU with SGD and Momentum at a rate of 0.90	9 428 609	1.34991	0.81049
SeLU with Adam	9 428 609	0.52273	0.90404
SeLU with SGD and Momentum at a rate of 0.90	9 428 609	2.21778	0.75260

VGGNet will not be trained using SeLU as it was not designed to have self-normalising features. However, LiteNet and ShallowNet will be compared against each other when using SeLU.

Table 6.7: VGGNet (using TL) results on the Cats and Dogs dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	4 195 329	0.25240	0.89623
ReLU with SGD and Momentum at a rate of 0.90	4 195 329	0.22292	0.89342
ELU with Adam	4 195 329	0.16076	0.90685
ELU with SGD and Momentum at a rate of 0.90	4 195 329	0.20863	0.89503

These architectures all display high accuracy on its unseen test data cases. Theoretically, any architecture can be chosen as the final model for binary classification in this setting. Practically, the TL architecture, VGGNet, output superior results in terms of test loss, accuracy and the number of parameters used to train the architecture. This is because VGGNet is pre-trained on ImageNet which does contain images of cats and dogs making training easier for these type of datasets. Another interesting observation is that using SGD and Momentum in VGGNet managed to output significantly lower test loss values than that by the other two architectures. This is because VGGNet was trained using SGD with Momentum.

Comparing LiteNet and ShallowNet, a clear difference in the number of parameters is again observed. The results given by both architectures are very similar, indicating that a large number

of parameters to reach a satisfactory result is not always needed in this case. The depth of the few layers in ShallowNet managed to successfully learn different features on these images in a short period of epochs.

Considering the activation functions used, ELU with Adam managed to obtain the lowest test loss on unseen data cases while having the highest accuracy rate. Especially in the case of ShallowNet. This result could complement research done on using ELU, showing that architectures with fewer layers and higher amount of depth or filters have the ability to successfully train and extract different features on images in a short period of epochs. In all cases in the tables above, ShallowNet indicated its power as a predictive architecture.

Now, to obtain an overview of how the architectures generalised using ELU with Adam, Figure 6.7 is illustrated. Slight characteristics of overfitting are displayed when using VGGNet, however, this might have not been the case if the model stopped training at 7 epochs where the architecture still managed to obtain a very low test loss with high accuracy. ShallowNet and LiteNet using ELU with Adam managed to generalise particularly well. SeLU as activation function did not perform well in most cases and will not be considered in the following evaluations. Additional outputs of the other architectures in terms of training and validation or test loss graphs are available in the Jupyter notebooks.

Following Section 4.5, an additional output regarding filter visualisation on a single cat image was also analysed. All filter visualisation outputs of each architecture can be viewed in the Jupyter notebooks. Considering the case of ShallowNet where ELU was used with Adam, Figure 6.9 and Figure 6.10 indicate the manner in which the CNN learned certain representations from the image given in Figure 6.8. Note how certain parts are more bright than other parts indicating how the particular layer perceived the image at hand.

Figure 6.9 refers to the activations and filter representations in ShallowNet's first convolutional layer while Figure 6.10 illustrates the activations and filter representations in ShallowNet's last convolutional layer. Additional layer activations can be found in the Jupyter notebooks. Notice the brightness difference between the figures indicating how the last layer in a CNN usually learn more complex, clearer, and abstract features than the first convolutional layer. Lastly, the results obtained from using SeLU could be due to the fact mentioned in Section 3.2.2 that this activation function only performs well in very deep networks, which was not the case here.

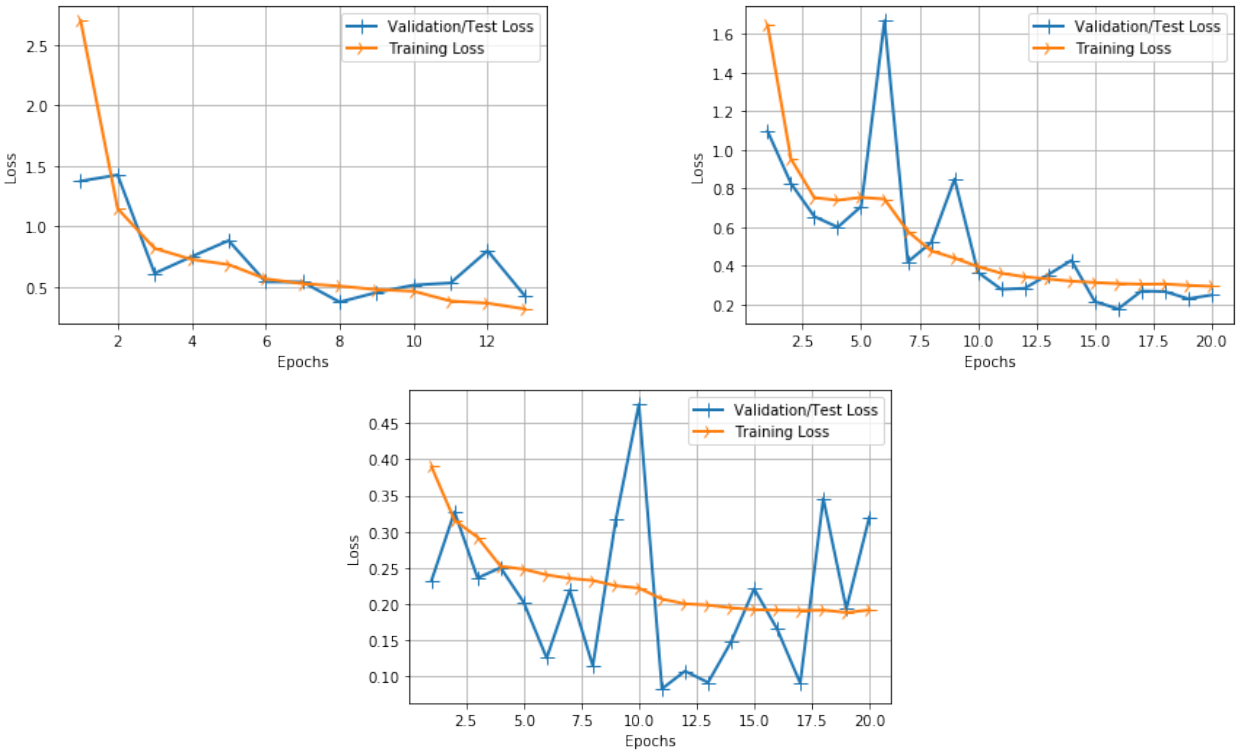


Figure 6.7: Top left (LiteNet), Top right (ShallowNet), and Bottom centre (VGGNet) graphs indicating model generalisation ability using ELU with Adam outputting the loss on the y-axis and the number of epochs on the x-axis.

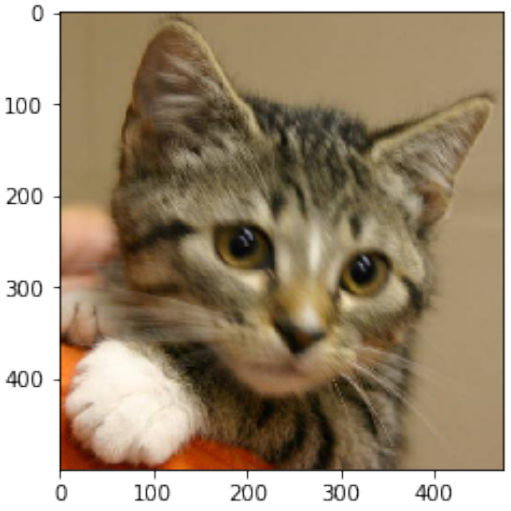


Figure 6.8: Sample image in the Cats and Dogs dataset used to obtain filter visualisation of how certain layers in CNNs perceive the image.

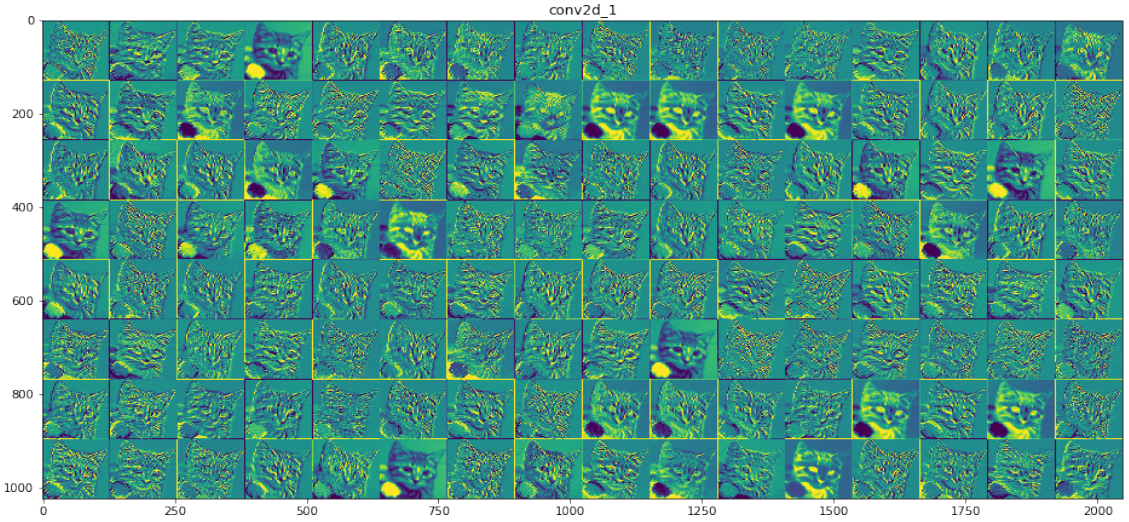


Figure 6.9: Filter representation of the first convolutional layer of the image shown in Figure 6.8.

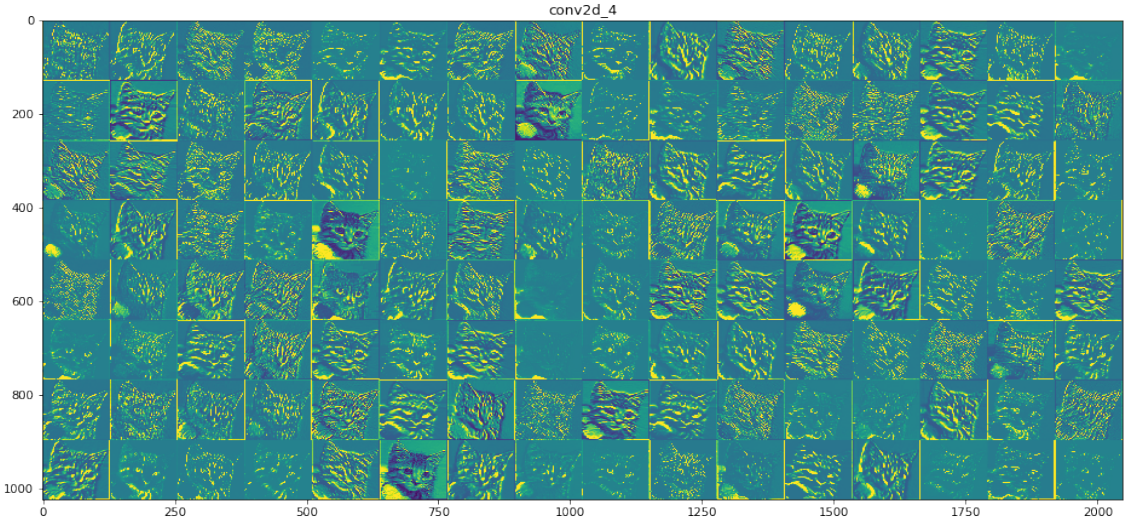


Figure 6.10: Filter representation of the last convolutional layer of the image shown in Figure 6.8.

6.3.3 Malaria

This dataset will be used to compare LiteNet against ShallowNet and VGGNet. Accurate implementation of these models could have the ability to aid the decisions done by microscopists. Following the data pre-processing step, data augmentation was also implemented since the dataset is relatively small. The set contained 24 958 images as training data and 2 600 images as validation or test data. The data augmentation steps were done in the same manner as in the Cats and Dogs dataset. After this step, an inspection was done, and the architectures were built. The following hyperparameters were chosen additionally:

- Epochs = 10.
- Batch size = 32.

The results in the tables below were obtained, respectively, using the different activation functions and optimisation techniques.

Table 6.8: LiteNet results on the Malaria dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	21 732 161	0.53239	0.89384
ReLU with SGD and Momentum at a rate of 0.90	21 732 161	3.12335	0.80923
ELU with Adam	21 732 161	0.19370	0.91192
ELU with SGD and Momentum at a rate of 0.90	21 732 161	2.80266	0.92769

Table 6.9: ShallowNet results on the Malaria dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	9 428 609	0.29492	0.89961
ReLU with SGD and Momentum at a rate of 0.90	9 428 609	1.10699	0.79000
ELU with Adam	9 428 609	0.34636	0.90461
ELU with SGD and Momentum at a rate of 0.90	9 428 609	1.10226	0.87115

In this case, LiteNet reached the lowest test loss combined with the highest accuracy rate. Most

Table 6.10: VGGNet (using TL) results on the Malaria dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	4 195 329	0.23949	0.90923
ReLU with SGD and Momentum at a rate of 0.90	4 195 329	0.20479	0.89730
ELU with Adam	4 195 329	0.47111	0.90499
ELU with SGD and Momentum at a rate of 0.90	4 195 329	0.09482	0.87807

of the architectures performed well here, except for the case of using ShallowNet in conjunction with ELU and Adam. These results indicate how Adam converges, in terms of the Malaria dataset, faster to a lower test loss than that of SGD with Momentum. This is especially evident when the custom-designed architectures are inspected.

An important factor on the classification report was mentioned in Section 2.4, that is, the recall metric found in the classification report is of particular importance when medical data is being analysed. In this case, with LiteNet in conjunction with ELU and Adam (lowest test loss), the report gave an output value of 0.90 for the recall metric. Meaning that a patient has a 90% chance of being positively classified for Malaria if that patient has Malaria while only having a 10% of being discharged. This is a satisfactory result which could aid the diagnoses of Malaria in areas where medical equipment and staff is lacking. LiteNet, however, might seem to be more hardware intensive due to the number of parameters it had to train but with only a single GPU and the risk of classifying false positives, this architecture should be considered above the rest.

In the case of using LiteNet in conjunction with ELU and SGD with Momentum, the model showed lacking generalisation results as shown in Figure 6.11. Even though it scored the highest accuracy, the architecture using these hyperparameters was slightly starting to overfit meaning that this architecture, having these settings, should not be used for practical purposes. This can be seen by the increasing nature of the test or validation line starting after epoch 8. Additional outputs of the other architectures in terms of training and validation or test loss graphs are available in Jupyter notebooks.

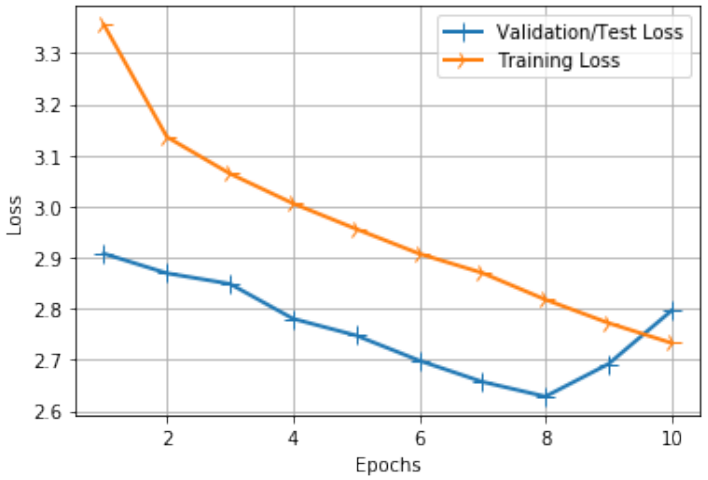


Figure 6.11: Model generalisation ability of the LiteNet architecture when ELU is used as an activation function and SGD with Momentum is used as an optimisation technique.

6.3.4 CIFAR-10

This dataset will be used to compare LiteNet against ShallowNet, VGGNet, and ResNet. The first step is the data pre-processing step that was done. The inspection step was done and the architectures were built in the usual manner. The following hyperparameters were chosen additionally:

- Epochs = 10.
- Batch size = 64.

The following results were obtained, respectively, using the different activation functions and optimisation techniques:

Table 6.11: LiteNet results on the CIFAR-10 dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	6 008 138	1.44429	0.77130
ReLU with SGD and Momentum at a rate of 0.90	6 008 138	3.23276	0.77520
ELU with Adam	6 008 138	1.52482	0.72530
ELU with SGD and Momentum at a rate of 0.90	6 008 138	3.21893	0.78899

Here, ShallowNet managed to outperform all architectures in terms of a combination of parameters, test loss, and accuracy. The accuracy scores under 80% indicate that this is not an easy dataset

Table 6.12: ShallowNet results on the CIFAR-10 dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	1 568 906	1.30962	0.77850
ReLU with SGD and Momentum at a rate of 0.90	1 568 906	1.87475	0.73600
ELU with Adam	1 568 906	1.36240	0.72710
ELU with SGD and Momentum at a rate of 0.90	1 568 906	1.89177	0.72780

Table 6.13: VGGNet results on the CIFAR-10 dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	1 267 786	1.10150	0.62500
ReLU with SGD and Momentum at a rate of 0.90	1 267 786	1.26536	0.55849
ELU with Adam	1 267 786	1.10204	0.61760
ELU with SGD and Momentum at a rate of 0.90	1 267 786	1.27340	0.55470

Table 6.14: ResNet results on the CIFAR-10 dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	24 588 810	0.56192	0.81099
ReLU with SGD and Momentum at a rate of 0.90	24 588 810	0.62893	0.79939
ELU with Adam	24 588 810	0.59366	0.80239
ELU with SGD and Momentum at a rate of 0.90	24 588 810	0.65807	0.78409

to correctly classify. ReLU with Adam again showed its supremacy as it maintained the lowest test loss and highest accuracy in each case of architectures used. In terms of model selection, ShallowNet again triumphs the choice. ResNet contain far too many parameters, even though it reached the lowest test loss and highest accuracy, due to the high number of layers it takes to process different input images. This might not be a problem when the user has hardware intensive processing available, though in general ShallowNet should be preferred as it reaches test loss and accuracy very close to ResNet by using only a fraction of the parameters that need to be trained in ResNet. Additional outputs of the architectures in terms of training and validation or test loss graphs are available in Jupyter notebooks. All architectures in each setting managed to generalise well.

In terms of prediction and classification of unseen data in CIFAR-10, Figure 6.12 indicates the prediction of ShallowNet using ReLU with Adam on an image that is observed as label 9, a truck. ShallowNet managed to correctly predict that it is indeed a truck even though the image contains a small number of pixels making it difficult to notice using the human eye. An output obtained by using ShallowNet in conjunction with ReLU and Adam can be found in Appendix A as an example of how the Jupyter notebooks handle code, as well as for the reader to obtain a basic idea on this type of format.

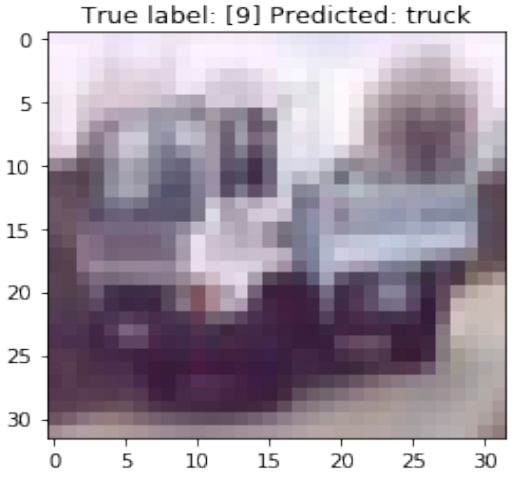


Figure 6.12: Prediction of the label 9, a truck, using ShallowNet.

6.3.5 Facial Expression Recognition

The final dataset in this section will be used to compare LiteNet against ShallowNet using only the activation ReLU and ELU with Adam as the optimisation technique. This final dataset is the

most complex one and thus the main focus in this thesis. The goal here is not to reach state-of-the-art results, but rather to assess the ability of each architecture to classify the different emotions correctly. Having such architectures that could classify human emotions from images using a low amount of training cost could complement current research done in facial recognition models. Live deployment of such architectures could contribute to social media platform tools, in the sense that these models may detect different emotions whereby social media users can share their feelings by simply posting a photo of their face, for example.

After data pre-procession, inspection and model building, the following hyperparameters were chosen additionally:

- Epochs = 50.
- Batch size = 64.

Data augmentation were also done as follows:

- Rotation of 15 degrees.
- Rescaling by a factor of 1/255.
- Shearing by a factor of 0.1.
- Zooming by a factor of 0.2.
- Horizontal flip.
- Width and height shift by a factor of 0.2.

The following results were obtained, respectively, using the different activation function:

Table 6.15: LiteNet results on the Facial Expression Recognition 2013 dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	7 314 628	1.01419	0.64000
ELU with Adam	7 314 628	0.91991	0.62000

Table 6.16: ShallowNet results on the Facial Expression Recognition 2013 dataset.

Hyperparameters	Parameters	Test Loss	Test Accuracy
ReLU with Adam	2 218 884	1.00620	0.62000
ELU with Adam	2 218 884	0.93185	0.62000

ShallowNet again showcased its power in terms of parameter and test loss minimisation. Both

architectures did not manage to reach the 70% baseline state-of-the-art result although both architectures were not as deep as the state-of-the-art models designed in Pramerdorfer and Kampel (2016). The lower score could be from the reduction in class labels done in Section 6.2.5, but overall the results should be considered as satisfactory since a custom-designed architecture with not many layers almost reaches 65% accuracy on this dataset. This does not only show the power of this architecture, but also the power of CNNs in general. Even though more advanced techniques exist to properly perform facial recognition, having small compact and lightweight models can improve the way in which large facial recognition models or software go about their predictions.

The problem of lower accuracy might not be representative from the designed architectures but rather from the dataset itself, as seen in Table 6.16 and Figure 6.13 which is the confusion matrix of LiteNet in conjunction with ReLU and Adam indicating the difficulty of classifying neutral facial expression which could have been misjudged as sad or angry.

Table 6.17: Accuracy metrics of LiteNet using ReLU with Adam on the Facial Emotion Recognition 2013 dataset.

Emotions	Precision	Recall	F1 - Score
Angry	0.59	0.70	0.64
Happy	0.80	0.91	0.85
Neutral	0.40	0.26	0.31
Sad	0.61	0.61	0.61

Not only LiteNet struggled with classifying whether a person is neutral or sad, but ShallowNet also had some problems with these two labels. The architecture, however, managed to be more than 80% certain if a person is happy. So, if the task at hand is to predict whether someone is happy from an input image of size 48×48 then this architecture will perform particularly well. As a test, an unseen image of myself where I take on the emotion of happy has been fed to the architecture while using OpenCV to obtain an object recognition frame around the part where the architecture recognises a label. The outcome is given in Figure 6.14 and was indeed correctly predicted, using LiteNet.

Finally, for comparison purposes, the architecture did manage to achieve approximately the same accuracy scores than that of Agrawal and Mittal (2020), where the authors also made use of a CNN while experimenting with different kernels and depth sizes to surpass human-like recognition. Perhaps if more epochs have been used a higher result could have been obtained but this was not

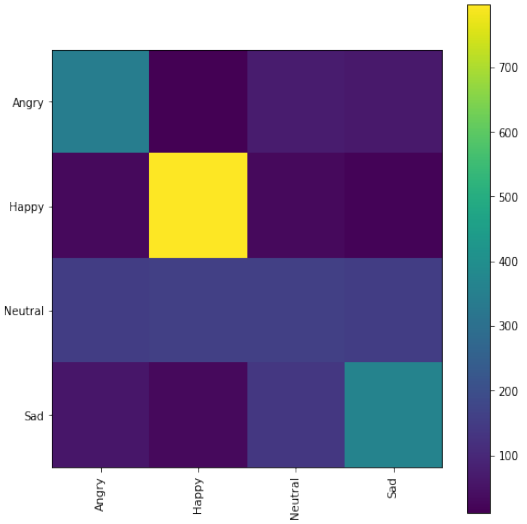


Figure 6.13: Colourised Confusion matrix output of LiteNet using ReLU with Adam on the Facial Emotion Recognition 2013 dataset. Yellow indicates a well-classified score.

the main objective for this dataset. The objective was to assess the strength of a more compact architecture, that is, ShallowNet which could contribute to the classification of small images as those found on mobile phones for example, easily and quickly.

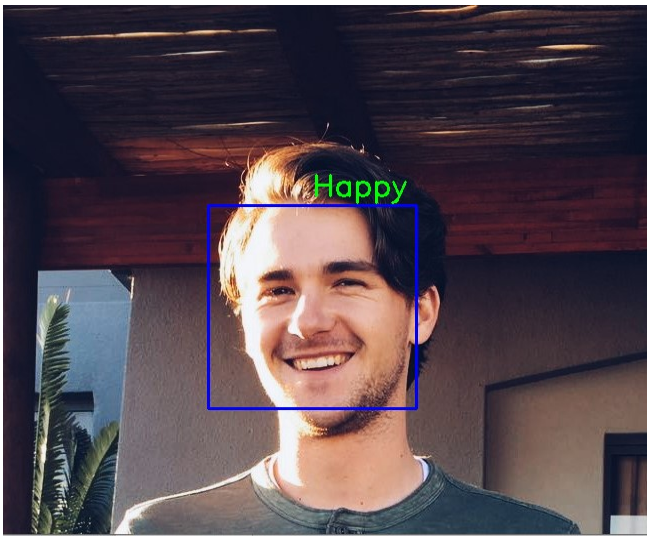


Figure 6.14: Facial emotion image prediction by LiteNet using OpenCV which provided the object recognition green box and label.

6.4 SUMMARY

In this chapter, a thorough investigation was done on the comparison of two custom-designed architectures to that of state-of-the-art architectures. Five datasets were used and in each dataset

different architectures with different activation functions and optimisation techniques were compared in terms of test loss and accuracy on unseen images. In each dataset, an appropriate final architecture was chosen which outperformed all the other architectures on that given dataset. Lastly, object detection was done based on a facial emotion dataset of which a successful result was obtained.

CHAPTER 7
CONCLUSION

This thesis studied the effects of using different CNN architectures with different hyperparameters on different datasets. A review was done on the optimal methods to approach these tasks while an experimental study was done to obtain a better understanding of CNNs. The objectives of the thesis were reached by sequentially presenting the results in the empirical chapter, Chapter 6.

Here, it was found that an in-depth understanding of CNNs is needed to fully deploy these models for practical purposes. Additionally, different hyperparameters were analysed and it was found that, in terms of activation functions, using ReLU provides satisfactory results when test loss and accuracy is analysed, while using Adam as optimisation procedure achieved superior results on most of the considered datasets.

Additionally, it was found that the shallower architecture of the two custom-designed architectures produced suitable results in terms of prediction power, parameter depth, and hardware power. The TL architectures used produced fair results on datasets that consisted of features that the respective architecture has already been trained on, for example, ResNet produced exceptional results on the CIFAR-10 dataset compared to the custom-designed architectures since ResNet is well known for being pre-trained on ImageNet which contains all the images found in CIFAR-10.

In conclusion, it was observed that CNNs have the ability to improve patient care in terms of medical disease detection from images as well as to detect emotions from facial features. Having such models ready for deployment in these settings could prove beneficial in other research or practical areas. This thesis hopes to have shown the predictive power that is possible when using CNNs for image classification and object detection.

7.1 LIMITATIONS

There were some limitations in this study, that is:

- Access to ImageNet or larger datasets: CNNs are well-known for the amount of data they need to successfully distinguish between the different objects in images. This study could have shown improved results if there were more training data available in each dataset. Even though data augmentation was done, each dataset still had less than 100 000 training images, whereas all the TL architectures had pre-trained weights from ImageNet.
- Only having a single GPU: Even though cloud-computing exists, making models easier to deploy, the memory usage from CNNs remains large.
- Time constraints: Although in this thesis, time constraint was not a problem this is necessarily

not the case in practice. These models took a while to run on a single GPU meaning that the cost of training a CNN might indicate that it is too time-consuming when working under certain deadlines.

7.2 FUTURE RESEARCH

To complement or improve the research done in this thesis, future directions based on the following ideas can be done:

- Obtaining access to ImageNet, larger datasets, or more realistic datasets than that of CIFAR-10.
- Considering architectures with less layers and more depth in each layer.
- Using generative models such as Generative Adversarial Networks, GANs, for generating richer, more improved, and new training samples (Goodfellow *et al.*, 2016).

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. *et al.* (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Agrawal, A. and Mittal, N. (2020). Using cnn for facial expression recognition: a study of the effects of kernel size and number of filters on accuracy. *The Visual Computer*, vol. 36, no. 2, pp. 405–412.

Algobeans (2016). Convolutional neural networks (cnn) introduction. [Online].
Available at: `https://algobeans.com/2016/01/26/introduction-to-convolutional-neural-network`

Atlas, L.E., Homma, T. and Marks II, R.J. (1988). An artificial neural network for spatio-temporal bipolar patterns: Application to phoneme classification. In: *Neural Information Processing Systems*, pp. 31–40.

Bishop, C.M. (2006). *Pattern recognition and machine learning*. Springer.

Carrier, P.-L. and Courville, A. (2013). The facial expression recognition 2013 (fer-2013) dataset. [Online].
Available at: `https://datarepository.wolframcloud.com/resources/FER-2013`

Chatterjee, C.C. (2019). Basics of the classic cnn. [Online].
Available at: `https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add`

Ciresan, D.C., Meier, U., Masci, J., Gambardella, L.M. and Schmidhuber, J. (2011). Flexible, high performance convolutional neural networks for image classification. In: *Twenty-second international joint conference on artificial intelligence*.

Clevert, D.-A., Unterthiner, T. and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314.

Dahl, G.E., Sainath, T.N. and Hinton, G.E. (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. In: *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 8609–8613. IEEE.

Dauphin, Y.N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S. and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: *Advances in neural information processing systems*, pp. 2933–2941.

Domínguez, A. (2015). A history of the convolution operation [retrospectroscope]. *IEEE pulse*, vol. 6, no. 1, pp. 38–49.

Duchi, J., Hazan, E. and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, vol. 12, no. 7.

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Durán, J. (2019). Everything you need to know about gradient descent applied to neural networks. [Online].
Available at: `https://medium.com/yottabytes/everything-you-need-to-know-about-gradient-descent-applied-to-neural-networks-d70f85e0cc14`

Elgendy, M. (2020). *Deep Learning for Vision Systems*. Manning Publications.

Erhan, D., Bengio, Y., Courville, A. and Vincent, P. (2009). Visualizing higher-layer features of a deep network. *University of Montreal*, vol. 1341, no. 3, p. 1.

Fergus, R. (2015). Neural networks: Mlss 2015 summer school, course notes.
Available at: `http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf`

Forsyth, D. and Ponce, J. (2012). Computer vision: A modern approach. always learning.

Fukushima, K., Miyake, S. and Ito, T. (). Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE transactions on systems, man, and cybernetics*, vol. 1.

Geitgey, A. (2016). Machine learning is fun! part 3: Deep learning and convolutional neural networks. [Online].
Available at: `https://medium.com/@ageitgey/machine-learning-is-fun-part-3-deep-learning-and-convolutional-neural-networks-f40359318721`

Girshick, R., Donahue, J., Darrell, T. and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*. MIT Press. Available at: `http://www.deeplearningbook.org`.

Gupta, T. (2017). Deep learning: Feedforward neural network. [Online]. Available at: `https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7`

Hammel, B. (2019). What learning rate should i use? [Online]. Available at: `http://www.bdhammel.com/learning-rates/`

Harley, A.W. (2015). An interactive node-link visualization of convolutional neural networks. In: *International Symposium on Visual Computing*, pp. 867–877. Springer.

Hastie, T., James, G., Witten, D. and Tibshirani, R. (2013). *An introduction to statistical learning*, vol. 112. Springer.

Hastie, T., Tibshirani, R. and Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction*. Springer Science & Business Media.

He, K., Zhang, X., Ren, S. and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034.

He, K., Zhang, X., Ren, S. and Sun, J. (2016). Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Hesamian, M.H., Jia, W., He, X. and Kennedy, P. (2019). Deep learning techniques for medical image segmentation: Achievements and challenges. *Journal of digital imaging*, vol. 32, no. 4, pp. 582–596.

Hewage, R. (2018). Extract features, visualize filters and feature maps in vgg16 and vgg19 cnn models. [Online]. Available at: `https://towardsdatascience.com/extract-features-visualize-filters-and-feature-maps-in-vgg16-and-vgg19-cnn-models-d2da6333edd0`

Hinton, G.E., Osindero, S. and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, vol. 18, no. 7, pp. 1527–1554.

Huang, T.S. (1996). Computer vision: Evolution and promise. *1996 CERN School of Computing*, pp. 21–25.
Available at: 10.5170/CERN-1996-008.21

Huang, Z., Ng, T., Liu, L., Mason, H., Zhuang, X. and Liu, D. (2020). Sndcnn: Self-normalizing deep cnns with scaled exponential linear units for speech recognition. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6854–6858. IEEE.

Hubel, D.H. and Wiesel, T.N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, vol. 160, no. 1, p. 106.

Inference, G.-B.D.L. and Learning, B.D. (2015). A performance and power analysis. *NVidia Whitepaper, Nov.*

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Jacobson, L. (2013). Introduction to artificial neural networks - part 1. [Online].
Available at: http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7

Jain, P. (2019). Complete guide of activation functions. [Online].
Available at: https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

James, G., Witten, D., Hastie, T. and Tibshirani, R. (2013). *An introduction to statistical learning*, vol. 112. Springer.

Kaggle (2013). Dogs vs. cats. [Online].
Available at: https://www.kaggle.com/c/dogs-vs-cats/overview

Kaggle (2014). Cifar-10 - object recognition in images. [Online].
Available at: https://www.kaggle.com/c/cifar-10

Kaggle (2018). Malaria cell images dataset. [Online].
Available at: https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria

Kalita, J. (2014). Simple neural nets for pattern classification: Mcculloch-pitts. [Online]. Available at: `http://www.cs.uccs.edu/~jkalita/work/cs587/2014/02ThresholdLogic.pdf`

Karpathy, A. (2017). Convolutional neural networks (cnns/convnets). cs231n: Convolutional neural networks for visual recognition. 2017; course notes.

Keras (2020). Keras documentation: Keras applications. [Online]. Available at: `https://keras.io/api/applications/`

Kingma, D.P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Klambauer, G., Unterthiner, T., Mayr, A. and Hochreiter, S. (2017). Self-normalizing neural networks. In: *Advances in neural information processing systems*, pp. 971–980.

Krizhevsky, A., Hinton, G. and Nair, V. (2009). Learning multiple layers of features from tiny images.

Krizhevsky, A., Sutskever, I. and Hinton, G.E. (2012). Imagenet classification with deep convolutional neural networks. In: *Advances in neural information processing systems*, pp. 1097–1105.

Kuhn, M., Johnson, K. *et al.* (2013). *Applied predictive modeling*, vol. 26. Springer.

Kyurkchiev, N. and Markov, S. (2015). Sigmoid functions: some approximation and modelling aspects. *LAP LAMBERT Academic Publishing, Saarbrucken*.

Kyurkchiev, N. and Markov, S. (2016). On the hausdorff distance between the heaviside step function and verhulst logistic function. *Journal of Mathematical Chemistry*, vol. 54, no. 1, pp. 109–119.

LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W. and Jackel, L.D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, vol. 1, no. 4, pp. 541–551.

LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324.

LeCun, Y., Kavukcuoglu, K. and Farabet, C. (2010). Convolutional networks and applications in vision. In: *Proceedings of 2010 IEEE international symposium on circuits and systems*, pp. 253–256. IEEE.

Lee, H., Grosse, R., Ranganath, R. and Ng, A.Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: *Proceedings of the 26th annual international conference on machine learning*, pp. 609–616.

Li, F.-F., Karpathy, A. and Johnson, J. (2015). Convolutional neural networks for visual recognition. [Online].
Available at: `https://cs231n.github.io/`

Li, M., Zhang, T., Chen, Y. and Smola, A.J. (2014). Efficient mini-batch training for stochastic optimization. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 661–670.

Lin, M., Chen, Q. and Yan, S. (2013). Network in network. corr abs/1312.4400 (2013). *arXiv preprint arXiv:1312.4400*.

Maas, A.L., Hannun, A.Y. and Ng, A.Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In: *Proc. icml*, vol. 30, p. 3.

Marsden, P. (2017). Artificial intelligence timeline infographic – from eliza to tay and beyond. [Online; accessed April 01, 2020].
Available at: `https://digitalwellbeing.org/artificial-intelligence-timeline-infographic-from-eliza-to-tay-and-beyond`

McCarthy, J., Minsky, M., Rochester, N. and Shannon, C. (2006). A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI Magazine*, vol. 27, pp. 12–14.

McCulloch, W.S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133.

McPhee, M., Baker, K. and Corky, S. (2015). Deep blue, ibm's supercomputer, defeats chess champion garry kasparov in 1997. *NY Daily News*.
Available at: `www.nydailynews.com/news/world/kasparov-deep-blues-losingchess-champ-rooke-article-1.76226`

Mehta, A. (2019). A comprehensive guide to types of neural networks. [Online].
Available at: `https://www.digitalvidya.com/blog/types-of-neural-networks/`

Minsky, M. and Papert, S.A. (2017). *Perceptrons: An introduction to computational geometry*. MIT press.

Moawad, A. (2018). Neural networks and back-propagation explained in a simple way. [Online]. Available at: https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e

Moolayil, J.J. (2019). A layman's guide to deep neural networks. [Online]. Available at: https://towardsdatascience.com/a-laymans-guide-to-deep-neural-networks-ddcea24847fb

Nair, V. and Hinton, G.E. (2010). Rectified linear units improve restricted boltzmann machines. In: *ICML*.

NASA's Imagine The Universe (2013). The electromagnetic spectrum. [Online]. Available at: https://imagine.gsfc.nasa.gov/science/toolbox/emspectrum1.html

Nesterov, Y.E. (1983). A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In: *Dokl. akad. nauk Sssr*, vol. 269, pp. 543–547.

Nuric (2019). Imperial college machine learning - neural networks. [Online]. Available at: https://www.doc.ic.ac.uk/~nuric/teaching/imperial-college-machine-learning-neural-networks.html

Nwankpa, C., Ijomah, W., Gachagan, A. and Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*.

Oquab, M., Bottou, L., Laptev, I. and Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1717–1724.

Paliwal, A. (2018). Understanding your convolution network with visualizations. [Online]. Available at: https://towardsdatascience.com/understanding-your-convolution-network-with-visualizations-a4883441533b

Peemen, M. (2019). Convolutional neural network (cnn). [Online]. Available at: https://developer.nvidia.com/discover/convolutional-neural-network

Phillips, D.K. (2015). Speed of the human brain. [Online]. Available at: https://askabiologist.asu.edu/plosable/speed-human-brain

Polyak, B.T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17.

Poole, D., Mackworth, A. and Goebel, R. (1998). *Computational intelligence: a logical approach*. Oxford University Press.

Pramerdorfer, C. and Kampel, M. (2016). Facial expression recognition using convolutional neural networks: state of the art. *arXiv preprint arXiv:1612.02903*.

Rajaraman, S., Antani, S.K., Poostchi, M., Silamut, K., Hossain, M.A., Maude, R.J., Jaeger, S. and Thoma, G.R. (2018). Pre-trained convolutional neural networks as feature extractors toward improved malaria parasite detection in thin blood smear images. *PeerJ*, vol. 6, p. e4568.

Ranzato, M., Boureau, Y.-L. and Cun, Y.L. (2008). Sparse feature learning for deep belief networks. In: *Advances in neural information processing systems*, pp. 1185–1192.

Raschka, S. (2015). *Python Machine Learning*. Packt Publishing, Birmingham, UK. ISBN 1783555130.

Ren, S., He, K., Girshick, R. and Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In: *Advances in neural information processing systems*, pp. 91–99.

Ripley, B.D. (2007). *Pattern recognition and neural networks*. Cambridge university press.

Rojas, R. (2013). *Neural networks: a systematic introduction*. Springer Science & Business Media.

Rosenblatt, F. and Papert, S. (1957). The perceptron. *A perceiving and recognizing automation, Cornell Aeronautical Laboratory Report*, pp. 85–460.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1988). Neurocomputing: Foundations of research. *ch. Learning Representations by Back-propagating Errors*, pp. 696–699.

Russell, S. and Norvig, P. (2003). *Artificial intelligence: A modern approach*. Pearson Education.

Saha, S. (2018). A comprehensive guide to convolutional neural networks — the eli5 way. [Online]. Available at: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

Saxena, A. (2016). Convolutional neural networks (cnns): An illustrated explanation. [Online]. Available at: `https://blog.xrds.acm.org/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/`

Sethi, B. and Goel, R. (2015). Exploring adaptive learning methods for convex optimization.

Simonyan, K., Vedaldi, A. and Zisserman, A. (2013). Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Skansi, S. (2018). Convolutional neural networks. In: *Introduction to deep learning*, pp. 121–133. Springer.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958.

Statinfer (2017). 203.5.10 local vs. global minimum. [Online].
Available at: `https://statinfer.com/203-5-10-local-vs-global-minimum/`

Stewart, M. (2019). Advanced topics in deep convolutional neural networks. [Online].
Available at: `https://towardsdatascience.com/advanced-topics-in-deep-convolutional-neural-networks-71ef1190522d`

Sutskever, I. (2013). *Training recurrent neural networks*. University of Toronto Toronto, Canada.

Sutton, R. (1986). Two problems with back propagation and other steepest descent learning procedures for networks. In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pp. 823–832.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A. (2015). Going deeper with convolutions. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826.

Szeliski, R. (2010). *Computer vision: algorithms and applications*. Springer Science & Business Media.

Thrun, S. and Pratt, L. (1998). *Learning to learn*. Springer.

Tsang, S.-H. (2018). Review: Alexnet, caffenet — winner of ilsvrc 2012 (image classification). [Online].
Available at: https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160

Turing, A. (1950). Computing machinery and intelligence. *Mind*, vol. 49, pp. 433–460.
Available at: https://www.csee.umbc.edu/courses/471/papers/turing.pdf

Wang, Z.J., Turko, R., Shaikh, O., Park, H., Das, N., Hohman, F., Kahng, M. and Chau, D.H. (2020). Cnn explainer: Learning convolutional neural networks with interactive visualization. *arXiv preprint arXiv:2004.15004*.

Wilson, A.C., Roelofs, R., Stern, M., Srebro, N. and Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning. In: *Advances in neural information processing systems*, pp. 4148–4158.

Wistuba, M., Rawat, A. and Pedapati, T. (2019). A survey on neural architecture search. *arXiv preprint arXiv:1905.01392*.

Xu, B., Wang, N., Chen, T. and Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*.

Yang, Y. and Bang, C.S. (2019 04). Application of artificial intelligence in gastroenterology. *World Journal of Gastroenterology*, vol. 25, pp. 1666–1683.

Yosinski, J., Clune, J., Bengio, Y. and Lipson, H. (2014). How transferable are features in deep neural networks? In: *Advances in neural information processing systems*, pp. 3320–3328.

Zeiler, M.D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zeiler, M.D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In: *European conference on computer vision*, pp. 818–833. Springer.

Zhou, T. (2018). Deep learning models for route planning in road networks. [Online].
Available at: https://pdfs.semanticscholar.org/03a7/103af7557c6f58c391a35f069817ae8cf102.pdf?_g
2.46472800.679416010.1598525255-2067317317.1590501326

APPENDIX A
JUPYTER NOTEBOOK OUTPUT

CIFAR ShallowNet (RELU) (ADAM)

August 29, 2020

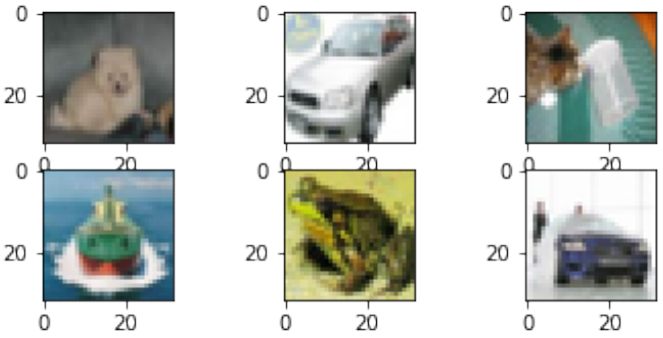
[1]:
```
#Importing Keras packages
from __future__ import print_function
import os
import keras
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, ZeroPadding2D
from keras.preprocessing.image import ImageDataGenerator
from keras.utils import np_utils
from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
from keras.models import load_model
from keras import optimizers
from tensorflow.keras.preprocessing import image
from keras.layers.advanced_activations import ELU, LeakyReLU, PReLU, Softmax,␣
↪ReLU
from keras.regularizers import l2
from tensorflow.keras import initializers
import pandas as pd
import numpy as np
```

```
Using TensorFlow backend.
```

[2]:
```
#Loading the CIFAR-10 dataset
from tensorflow.keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

print(y_train.shape)
print(y_test.shape)
print(x_train.shape)
print(x_test.shape)

#Data Pre-processing and Inspection
img_rows = x_train[0].shape[0]
img_cols = x_train[1].shape[0]

#Inspection
```

```
import matplotlib.pyplot as plt

#Plotting 6 images with color map set to grey since the images are greyscale
plt.subplot(331)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap = plt.get_cmap('gray'))

plt.subplot(332)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap = plt.get_cmap('gray'))

plt.subplot(333)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap = plt.get_cmap('gray'))

plt.subplot(334)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap = plt.get_cmap('gray'))

plt.subplot(335)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap = plt.get_cmap('gray'))

plt.subplot(336)
random_num = np.random.randint(0,len(x_train))
plt.imshow(x_train[random_num], cmap = plt.get_cmap('gray'))
plt.show()

#Normalise the data by changing the range from (0 to 255) to (0 to 1)
x_train =  x_train/255
x_test = x_test/255
print(x_train.shape)
print(x_test.shape)

#Change the image type to float32 data type
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

```
(50000, 1)
(10000, 1)
(50000, 32, 32, 3)
(10000, 32, 32, 3)
```



```
(50000, 32, 32, 3)
(10000, 32, 32, 3)
x_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
```

[3]:
```
from tensorflow.keras.utils import to_categorical

#One-hot encode the outputs
y_cat_train = to_categorical(y_train,10)
y_cat_test = to_categorical(y_test,10)

print("Number of Classes: " + str(y_test.shape[1]))

num_classes = y_test.shape[1]
num_pixels = x_train.shape[1] * x_train.shape[2]
print("Number of Pixels: " + str(num_pixels)) #32*32=1024

y_train.shape
y_train[0]
```

```
Number of Classes: 1
Number of Pixels: 1024
```

[3]:
```
array([6], dtype=uint8)
```

[4]:
```
#Setting up some of the hyperparameters
batch_size = 64
num_classes = 10
epochs = 10
input_shape = x_train.shape[1:]
```

[5]:
```
#Ensuring the model is built on the existing GPU
import tensorflow as tf
print("Num GPUs Available: ", len(tf.config.experimental.
 ↪list_physical_devices('GPU')))
physical_devices = tf.config.experimental.list_physical_devices('GPU')
print("physical_devices-------------", len(physical_devices))
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

```
Num GPUs Available:  1
physical_devices------------- 1
```

[8]:
```
#Initiating ShallowNet for training
model = Sequential()

#1st Layer
model.add(Conv2D(128, (3, 3), padding = 'same', input_shape = input_shape,␣
 ↪activation = 'relu', kernel_regularizer = l2(0.0005),
                 kernel_initializer = "he_normal"))
model.add(BatchNormalization())

#2nd Layer
model.add(Conv2D(128, (3, 3), padding = 'same', input_shape = input_shape,␣
 ↪activation = 'relu', kernel_regularizer = l2(0.0005),
                 kernel_initializer = "he_normal"))
model.add(BatchNormalization())

#Pooling with Dropout
model.add(MaxPooling2D(pool_size = (4, 4)))
model.add(Dropout(0.25))

#3rd Layer
model.add(Conv2D(256, (3, 3), padding = 'same', input_shape = input_shape,␣
 ↪activation = 'relu', kernel_regularizer = l2(0.0005),
                 kernel_initializer = "he_normal"))
model.add(BatchNormalization())

#4th Layer
model.add(Conv2D(256, (3, 3), padding = 'same', input_shape = input_shape,␣
 ↪activation = 'relu', kernel_regularizer = l2(0.0005),
                 kernel_initializer = "he_normal"))
model.add(BatchNormalization())

#Pooling with Dropout
model.add(MaxPooling2D(pool_size = (4, 4)))
model.add(Dropout(0.25))

#5th Layer = 1st Fully Connected or Dense Layer
```

```
model.add(Flatten())
model.add(Dense(512, activation = 'relu', kernel_regularizer = l2(0.0005),␣
 ↪kernel_initializer = "he_normal"))
model.add(BatchNormalization())
model.add(Dropout(0.35))

#6th Layer = Final Layer
model.add(Dense(10, kernel_regularizer = l2(0.0005), kernel_initializer =␣
 ↪"he_normal"))
model.add(Activation('softmax'))

print(model.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 32, 32, 128)       3584
_________________________________________________________________
batch_normalization_6 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 32, 32, 128)       147584
_________________________________________________________________
batch_normalization_7 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 8, 8, 128)         0
_________________________________________________________________
dropout_4 (Dropout)          (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
batch_normalization_8 (Batch (None, 8, 8, 256)         1024
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 8, 8, 256)         590080
_________________________________________________________________
batch_normalization_9 (Batch (None, 8, 8, 256)         1024
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 2, 2, 256)         0
_________________________________________________________________
dropout_5 (Dropout)          (None, 2, 2, 256)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               524800
_________________________________________________________________
batch_normalization_10 (Batc (None, 512)               2048
_________________________________________________________________
```

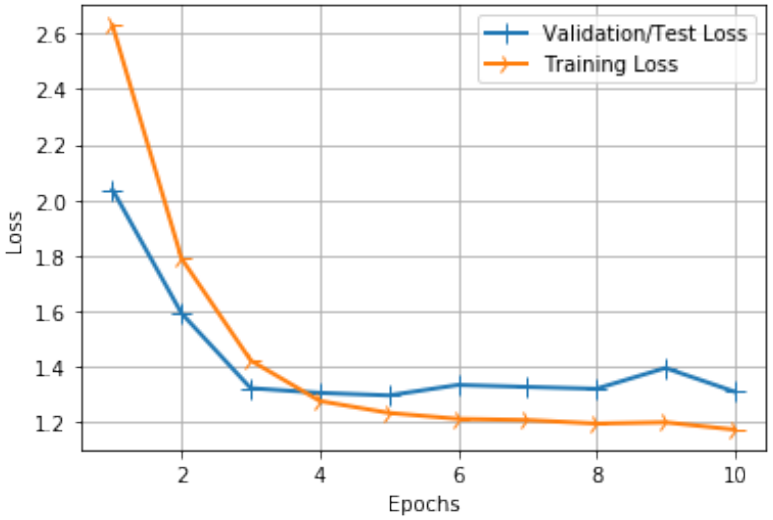
```
dropout_6 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 1,571,466
Trainable params: 1,568,906
Non-trainable params: 2,560
_________________________________________________________________
None
```

[9]:
```
#Model testing and reporting
opt = keras.optimizers.Adam(lr = 0.001)

model.compile(loss = 'categorical_crossentropy',
              optimizer = opt,
              metrics = ['accuracy'])

history = model.fit(x_train, y_cat_train, batch_size = batch_size,
                    epochs = epochs, validation_data = (x_test, y_cat_test),␣
 ↪shuffle = True)

#Saving model weights
model.save("C:\\Users\\Admin\\Desktop\\Research\\Results\\ShallowNet␣
 ↪Results\\Weights\\CIFAR_ShallowNet_RELU_Adam_Weights.h5")

#Evaluate the performance of the trained model
scores = model.evaluate(x_test, y_cat_test, verbose = 1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 24s 473us/step - loss: 2.6298 -
accuracy: 0.5066 - val_loss: 2.0379 - val_accuracy: 0.6447
Epoch 2/10
50000/50000 [==============================] - 21s 428us/step - loss: 1.7891 -
accuracy: 0.6861 - val_loss: 1.5918 - val_accuracy: 0.7044
Epoch 3/10
50000/50000 [==============================] - 21s 426us/step - loss: 1.4222 -
accuracy: 0.7447 - val_loss: 1.3222 - val_accuracy: 0.7606
Epoch 4/10
50000/50000 [==============================] - 21s 428us/step - loss: 1.2760 -
accuracy: 0.7686 - val_loss: 1.3054 - val_accuracy: 0.7551
Epoch 5/10
50000/50000 [==============================] - 22s 431us/step - loss: 1.2332 -
accuracy: 0.7829 - val_loss: 1.2963 - val_accuracy: 0.7636
```

```
Epoch 6/10
50000/50000 [==============================] - 21s 429us/step - loss: 1.2116 -
accuracy: 0.7985 - val_loss: 1.3339 - val_accuracy: 0.7586
Epoch 7/10
50000/50000 [==============================] - 22s 430us/step - loss: 1.2075 -
accuracy: 0.8026 - val_loss: 1.3268 - val_accuracy: 0.7637
Epoch 8/10
50000/50000 [==============================] - 21s 426us/step - loss: 1.1947 -
accuracy: 0.8101 - val_loss: 1.3201 - val_accuracy: 0.7689
Epoch 9/10
50000/50000 [==============================] - 21s 426us/step - loss: 1.1993 -
accuracy: 0.8106 - val_loss: 1.3951 - val_accuracy: 0.7467
Epoch 10/10
50000/50000 [==============================] - 21s 428us/step - loss: 1.1730 -
accuracy: 0.8222 - val_loss: 1.3096 - val_accuracy: 0.7785
10000/10000 [==============================] - 2s 156us/step
Test loss: 1.3096290985107422
Test accuracy: 0.7785000205039978
```

[10]:
```
#Evaluating training and test error
history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']
epochs = range(1, len(loss_values) + 1)
line1 = plt.plot(epochs, val_loss_values, label = 'Validation/Test Loss')
line2 = plt.plot(epochs, loss_values, label = 'Training Loss')
plt.setp(line1, linewidth = 2.0, marker = '+', markersize = 10.0)
plt.setp(line2, linewidth = 2.0, marker = '4', markersize = 10.0)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.show()
```



[11]:
```
#Classification report
model.metrics_names
print(model.metrics_names)
print(model.evaluate(x_test,y_cat_test,verbose = 0))

from sklearn.metrics import classification_report,confusion_matrix

predictions = model.predict_classes(x_test)
```

```
['loss', 'accuracy']
[1.3096290985107422, 0.7785000205039978]
```

[12]:
```
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.84      0.71      0.77      1000
           1       0.92      0.87      0.90      1000
           2       0.65      0.75      0.69      1000
           3       0.67      0.60      0.63      1000
           4       0.66      0.85      0.74      1000
           5       0.69      0.75      0.72      1000
           6       0.79      0.86      0.83      1000
           7       0.93      0.68      0.79      1000
           8       0.94      0.80      0.86      1000
```

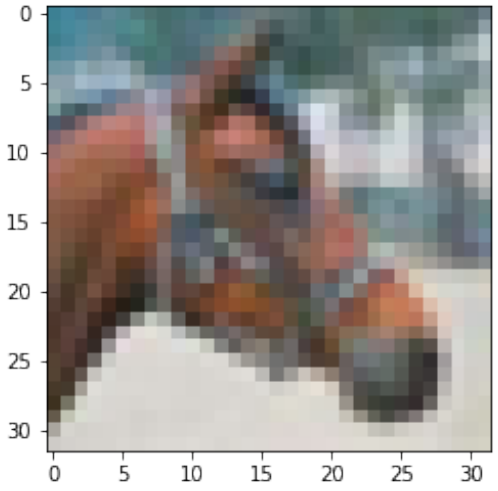
```
           9       0.82      0.92      0.87      1000

    accuracy                           0.78     10000
   macro avg       0.79      0.78      0.78     10000
weighted avg       0.79      0.78      0.78     10000
```

[ ]:
```
#Testing model predictions
import cv2
import numpy as np
from keras.models import load_model

img_row, img_height, img_depth = 32,32,3
classifier = load_model('C:
 ↪\\Users\\Admin\\Desktop\\Research\\Results\\ShallowNet␣
 ↪Results\\Weights\\CIFAR_ShallowNet_RELU_Adam_Weights.h5.h5')
color = True
scale = 8

def draw_test(name, res, input_im, scale, img_row, img_height):
    BLACK = [0,0,0]
    res = int(res)
    if res == 0:
        pred = "airplane"
    if res == 1:
        pred = "automobile"
    if res == 2:
        pred = "bird"
    if res == 3:
        pred = "cat"
    if res == 4:
        pred = "deer"
    if res == 5:
        pred = "dog"
    if res == 6:
        pred = "frog"
    if res == 7:
        pred = "horse"
    if res == 8:
        pred = "ship"
    if res == 9:
        pred = "truck"

    expanded_image = cv2.copyMakeBorder(input_im, 0, 0, 0, imageL.shape[0]*2␣
 ↪,cv2.BORDER_CONSTANT,value = BLACK)
    if color == False:
        expanded_image = cv2.cvtColor(expanded_image, cv2.COLOR_GRAY2BGR)
```

```
    cv2.putText(expanded_image, str(pred), (300, 80) , cv2.
↪FONT_HERSHEY_COMPLEX_SMALL,4, (0,255,0), 2)
    cv2.imshow(name, expanded_image)


for i in range(0,10):
    rand = np.random.randint(0,len(x_test))
    input_im = x_test[rand]
    imageL = cv2.resize(input_im, None, fx = scale, fy = scale, interpolation =␣
↪cv2.INTER_CUBIC)
    input_im = input_im.reshape(1,img_row, img_height, img_depth)

    #Get Prediction
    res = str(classifier.predict_classes(input_im, 1, verbose = 0)[0])

    draw_test("Prediction", res, imageL, scale, img_row, img_height)
    cv2.waitKey(0)

cv2.destroyAllWindows()
```

[13]:
```
import matplotlib.pyplot as plt
my_image = x_test[17]
plt.imshow(my_image) #5 = Dog, #7 = Horse
```

[13]: <matplotlib.image.AxesImage at 0x11cc2f1f988>



[14]:
```
model.predict_classes(my_image.reshape(1,32,32,3))
```

[14]:
```
array([9], dtype=int64)
```

[15]:
```
#Label mapping
labels = '''airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck'''.split()
```

[16]:
```
p_test = model.predict(x_test).argmax(axis=1)
y_test.flatten()
misclassified_idx = np.where(p_test != y_test)[0]
i = np.random.choice(misclassified_idx)
plt.imshow(x_test[i], cmap='gray')
plt.title("True label: %s Predicted: %s" % (y_test[i], labels[p_test[i]]));
```