Faculty of Science and Engineering

Connor Boyd-Lyon

BSc. (Hons) Mathematics

Image Classification with Convolutional Neural Networks

May 2024

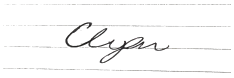
Department of Computing and Mathematics

Abstract

This project explored the role of convolutional neural networks in computer vision. Computer vision is a field of artificial intelligence that focuses on enabling machines to interpret and understand the visual world. Computer vision has a diverse range of applications including document scanning, facial recognition and self-driving cars. Convolutional neural networks (CNNs) are a specialised type of neural network which can learn the features of images. CNNs are the underlying mechanism behind the interpretation of objects in the real-world and are based on the biological brain. First, the mathematical principles of neural networks were explored, including a breakdown of the convolution operation and a visual description of the artificial neuron. Then a model was created on the modified NIST database (MNIST) to recognise handwritten digits, which was a success; the model benchmarked at 98.85% accuracy during testing. The MNIST model was then re-purposed using the concept of transfer learning to create a facial recognition model that classified a face as one of 16 learned individuals. After some amendments to the MNIST architecture the facial recognition model was complete. The model successfully recognised the learned individuals' faces 93.75% of the time. Further refinement of the facial recognition model would allow for it's implementation in industry, either as a method of security or as an identification software for surveillance.

Plagiarism Declaration

With the exception of any statement to the contrary, all the material presented in this report is the result of my own efforts. In addition, no parts of this report are copied from other sources. I understand that any evidence of plagiarism and/or the use of unacknowledged third party materials will be dealt with as a serious matter.

Signed 

Acknowledgements

I'd firstly like to thank my supervisor Dr. Jon Shiach for being a great teacher and role model. This project was undoubtedly an ambitious one; it wouldn't have been possible without his inspiration (and weekly debugging sessions). I'd also like to thank Dr. Stephen Lynch for access to his textbook, "Python for Scientific Computing and Artificial Intelligence". The entire textbook is worth reading for anyone interested in applying Python to their respective fields. I found particular use in Chapter 20 which covers convolutional neural networks.

Finally I'd like to thank my family and friends for giving me the drive to succeed. I'm very lucky to be surrounded by people that see the best in me, and I hope that this project is a result of that.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial intelligence (AI) is most-often associated with creating a machine that can think for itself (W.-M. Lee 2019). The term was coined by John MacCarthy in 1956 during the *Dartmouth Summer Research Project on Artificial Intelligence*. Prior to this, Alan Turing, who Jack Copeland (2004) claims "was the first to carry out substantial research in the field", used the term 'machine intelligence'. Today artificial intelligence is, for the most part, synonymous with machine learning. Machine learning is useful for approaching tasks that would typically require human levels of intelligence, such as understanding speech (Hannun et al. 2014), examining medical scans or x-rays (El Naqa and M. J. Murphy 2015), driving a car (Bojarski et al. 2016) or playing chess (Figure 1.1). Deep Blue (IBM 1997) was a major break through and turning point of artificial intelligence; Garry Kasparov was the greatest chess player in the world at the time, so when Deep Blue beat Kasparov, it showed that machine intelligence had surpassed human intelligence. Now the most advanced chess machine is AlphaZero (*AlphaZero* 2018) with a rating of 4650, whilst the best human chess rating ever achieved is 2882, by Magnus Carlsen. The difference between human and machine intelligence is increasing exponentially, which is, controversially, creating opportunities to apply machine learning in more complex domains.

Figure 1.1: Chess world champion Garry Kasparov vs IBM's Deep Blue, May 11 1997. To Kasparov and the World's surprise, Deep Blue defeats Kasparov $3\frac{1}{2}$-$2\frac{1}{2}$.

Image recognition is a process learned by humans and animals from the very first time they open their eyes. The photoreceptor cells of the eye translate electromagnetic energy into neural signals which the brain can analyse (Schnapf and Baylor 1987). The brain then learns what it sees; for a baby this may be learning what their mother looks like, and as a child it may be what the letters of the alphabet look like. This learning process takes place over several years throughout a human's youth, in which the brain is able to learn the appearance of millions of objects, in various colours, angles and light levels. This process becomes much harder when trying to teach a machine how to see and understand the world in the same way that we do. A computer cannot see depth, as it's input is only a 2D image. Additionally, a computer doesn't have years to learn the features of the world like we do - it's training time is limited by the patience of it's developer. Instead, machines must rely on their strength of arithmetic to learn the features of images. By using matrix multiplication, key features of images can be extracted. If a feature continues to be extracted in multiple instances of the same type of image, it becomes clear that this feature is a telling sign of the type of image that is being handled.

This report explores the relevance of convolutional neural networks (CNNs) in computer vision, and specifically how CNNs are applied to a data set, in aim of labelling the images as one output class or another. Fundamental principles of neural networks were discussed, including machine learning techniques, the convolution operation, the development of the artificial neuron and how these are applied to convolutional neural networks. Two convolutional neural network

models were made on Python, using the Keras-Tensorflow library. The first model was able to successfully recognise handwritten digits (0-9). By using the first model as a foundation, a second model that could successfully recognise faces was made. By investigating the effectiveness of CNNs in these two domains, the possible applications of a high-functioning CNN were made clear. Options for further study were given, including a model which could recognise multiple faces within a single image, or a handwritten document scanner which could recognise joined letters as well as numbers and symbols.

Firstly, Chapter 1 discusses the fundamentals of computer vision, including all of the techniques required in further chapters, as well as additional information on alternative learning methods and alternative types of neural network. Chapter 2 begins the focus on practical implementation, as a CNN is applied to the modified national institute of standards and technology (MNIST) database of handwritten digits (Lecun 1998). The aim of the MNIST CNN is to recognise a digit in it's handwritten form, despite the small discrepancies that are natural for handwritten numbers and letters. In Chapter 3, the MNIST CNN model of Chapter 2 was converted into a new model by making changes to the MNIST model's architecture and by providing new input data - closeup images of 16 different people. The new model aimed to learn the faces of the 16 individuals and correctly recognise which face is present in an input image.

Chapter 2

The Fundamentals of Computer Vision

This chapter will begin by reviewing key statistical techniques and machine learning fundamentals before introducing artificial neural networks (ANNs). Extensive detail will then be given on convolutional neural networks (CNNs), including a worked example to identify the underlying operations that allow an algorithm to learn.

2.1 Statistical Techniques

This section covers the relevant statistical concepts essential in understanding and applying neural networks. Each of the concepts here can be explored via further reading which is recommended throughout.

2.1.1 Problem Types

There are many types of problems that can be solved using machine learning algorithms. The three main types of problem are regression, clustering and classifications. Classification problems are the focal problem type of the later chapters.

Regression

Regression problems are concerned with a continuous output which describes a relationship between variables (W.-M. Lee 2019). Machine learning could be applied to make predictions

or forecasts on how a change in one variable affects the other. Some examples of regression problems include weather forecasting (Dadhich et al. 2021) and stock price predictions (Gilliland et al. 2021). Figure 2.1 shows a diagram of a basic regression model.

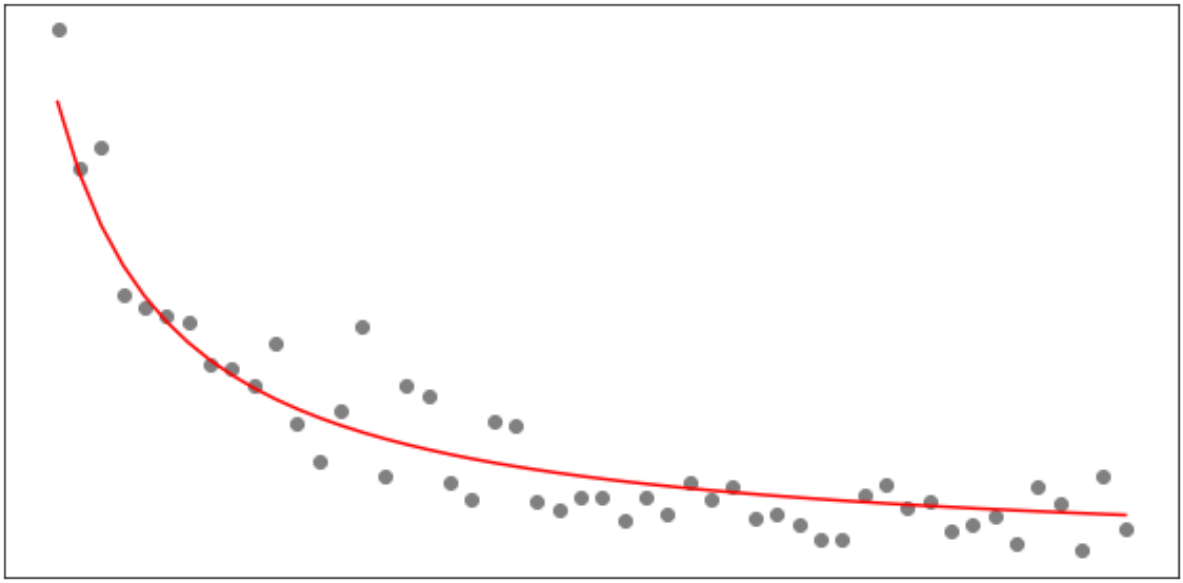


Figure 2.1: A diagram of a well-fitted regression model, approximately $\frac{1}{x}$ (Python 3.9). The concept of fit is covered later (section 2.1.3).

Clustering

Clustering problems are based on grouping data into 'clusters' by their characteristics (W.-M. Lee 2019). The model's objective is to sort the input data into groups containing statistically similar observations. In 1936 a statistician and botanist Ronald Fisher noticed that the iris flower could be grouped into the correct subspecies by measuring their dimensions and collecting like measurements into groups (Fisher 1936), which is the seminal example of clustering in modern statistics. Figure 2.2 shows a hierarchical clustering of Fisher's iris data set. Clustering is also useful to quantify the relation between clusters; the Versicolor and Virginica species in Figure 2.2 are shown to be more closely related to each other than to the Setosa species.

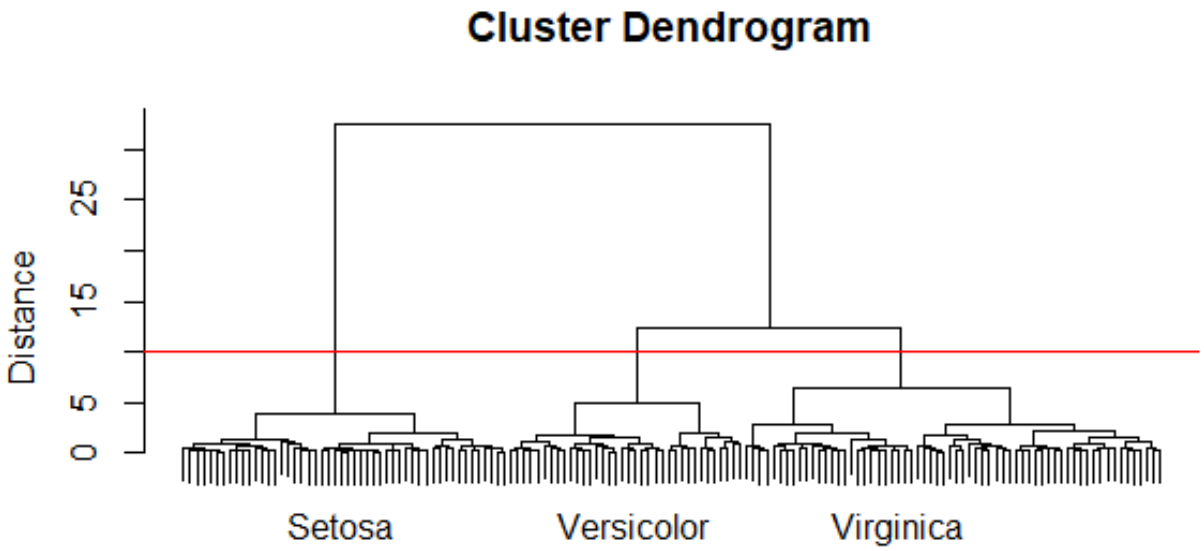


Figure 2.2: A hierarchical clustering of the iris data set (Fisher 1936), displayed as a dendrogram. The data set contains three subspecies of the iris flower; setosa, versicolor and virginica. Each observation is shown at the bottom as a vertical line, where they are paired to the most similar observation by a horizontal line. This pair then finds the most similar pair to it and forms a 'cluster' of pairs. This process repeats until all of the observations are related. We use the distance metric to determine similarity between two clusters. The red line indicates a chosen clustering threshold; we see below it there are three distinct clusters representing the three subspecies of iris. Ward's D2 method was used (R, 2024).

Classification

The main problem type in computer vision is the classification problem. The model's objective is to assign an input into one of k pre-defined categories (Goodfellow, Bengio, et al. 2016). The output is always discrete, as one choice out of k possible choices. In Chapter 3 a handwritten digit is classified as a digit from 0:9, hence $k = 10$. Lee (2019) likens a model's approach to a classification problem as asking the model "is this x or y?".

2.1.2 Frequentist & Bayesian Statistics

When estimating a parameter there are two approaches to consider. Both approaches are used extensively in the domain of Machine Learning (K. P. Murphy 2007).

Definition 2.1. ***Parameter.*** *A parameter, usually denoted as θ, is a quantitative measure of a certain aspect of the data set in question (Casella and Berger 1990). Common examples include the mean or standard deviation. In a data set about people, 'average age' or '% that are married' might be parameters. A parameter estimate is the 'best guess' of θ from the input, and is notated $\hat{\theta}$.*

Frequentist Estimation

The frequentist approach is often referred to as the classical approach to parameter estimation. Here the parameter is unknown and fixed. This parameter can be estimated using statistical methods such as maximum likelihood estimation or method of moments, from which the most likely value of the parameter $\hat{\theta}$ is found given the input data (Casella and Berger 1990). In frequentist statistics, uncertainty in a parameter estimate asks how much the estimate would change given a different input (K. P. Murphy 2007).

Bayesian Inference

When using the Bayesian approach there is a non-fixed parameter estimate with variation given by a prior distribution (Casella and Berger 1990). The aim is not the parameter estimate $\hat{\theta}$ but instead the posterior distribution $p(\theta|x, y)$, which is found using the prior distribution $p(\theta)$. In this context, uncertainty refers to how likely it is that the parameter estimate is correct (K. P. Murphy 2007).

Definition 2.2. ***Priors.*** *Priors are pre-existing knowledge about data before any analyses takes place. In sales data, the prior might be that 70% of customers are male. For data set* $\boldsymbol{x}$ *and output* y*,* $p(\boldsymbol{y})$ *might be our prior (Kotu and Deshpande 2019).*

2.1.3 Model Complexity

When creating a model it's important that the model doesn't underfit or overfit the training data. A model can underfit the data if the model itself is too simple, or if the training data lacks variance. Likewise it's possible to *overfit* a model by over-complicating it (Goodfellow, Bengio, et al. 2016). Overfit models tend to perform very well on training data but struggle with unseen data. The diagrams in figures 2.3/2.4 demonstrate the effect of fit on a regression model's performance. Finding the right fit and complexity for a model is known as *regularisation*.

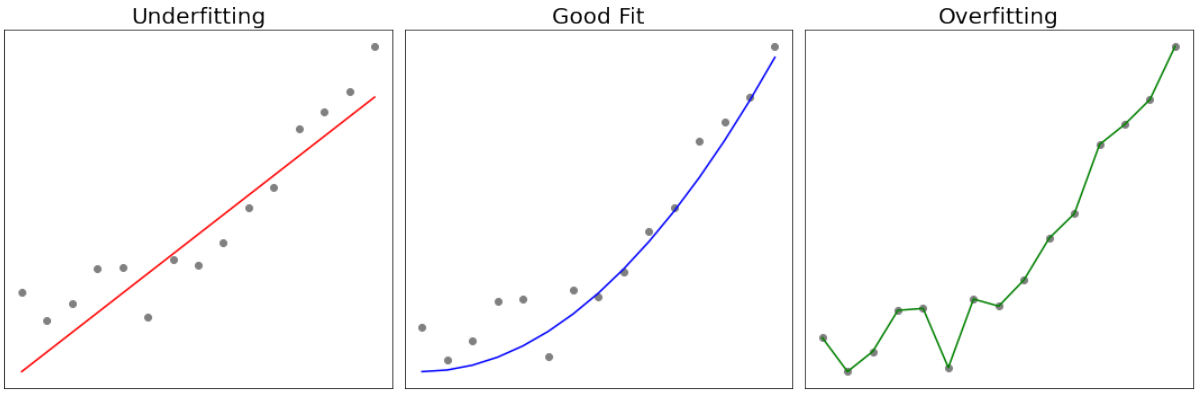


Figure 2.3: Three regression models created to model the training data, which is randomly drawn from the function $y = x^2$ with small standard deviation $0 < \sigma < 0.5$ (Python 3.9). *left:* An underfitted model that doesn't capture the shape of the training data. *centre:* A well-fitted model that captures the shape of the training data quite well. *right:* An overfitted model which captures the shape of the training data exactly.

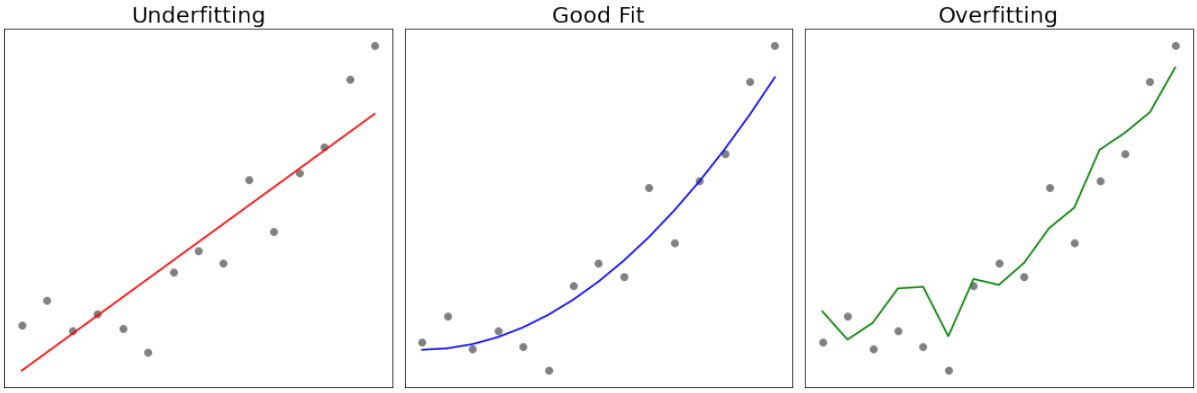


Figure 2.4: The result of using the models to fit unseen, similar data. It's clear that the well-fitted model fits unseen data the best.

Bias and Variance

Bias is a type of statistical error, defined as the difference between the predicted output of a model and the ground truth; Variance is the measure of spread within a data set (Fortman-Roe 2012). A model with high bias and low variance will tend to underfit the training data, whereas a model with low bias and high variance will overfit the training data (Singh 2021). Figure 2.5 shows the relationship of bias and variance and how they contribute to the Mean squared error (MSE) of a model (see below). This balancing act is known as the *bias-variance tradeoff*. It's critical to ensure that MSE stays low by creating an adequately complex model.

Mean squared error is a widely used metric in regression problems within ML. MSE is defined

as,

$$\text{Mean Squared Error} = \text{bias}^2 + \text{variance} \tag{2.1}$$

Figure 2.5 shows that bias^2 and variance are inversely proportional to one another (Fortman-Roe 2012), meaning that a problem only suffers from a high bias *or* a high variance. Because of this, one way of lowering MSE is to add more bias when variance dominates, and introduce more variance or noise when bias dominates.

Definition 2.3. ***Ground Truth.*** *The ground truth is the true value x of a parameter estimate $\bar{x}$. In the context of machine learning the ground truth can refer to a labelled data set used to train a supervised model on (Nwanganga and Chapple 2020).*

Definition 2.4. ***Noise.*** *Noise refers to random variations or disturbances within data or model parameters. Bhattacharyya and Ghosh (2022) define noise as the "unwanted part" in data. Whilst usually this is true, some noise is desirable to prevent underfitting.*

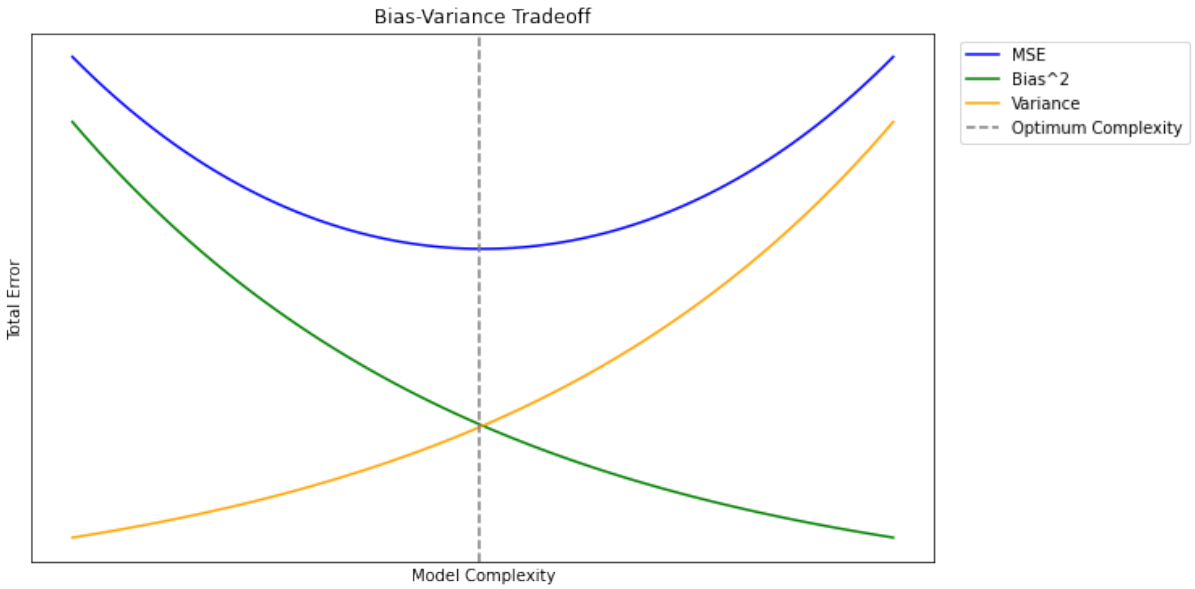


Figure 2.5: The mean squared error (MSE) shown as a sum of bias^2 and variance (equation 2.1). Underfitting or overfitting results in a high MSE; the least error is found when balancing bias and variance. (Python 3.9)

2.1.4 Data Augmentation

When a data set lacks variance data augmentation techniques are often employed to provide additional variance and act as a counterweight against bias. For classification problems this can be done mathematically, by applying minor transformations to the input data such as rotating, cropping and stretching (Goodfellow, Bengio, et al. 2016).

2.1.5 Gradient Descent

To reduce a model's error the developer should aim to minimise the model's cost (error) function. Minimum points of a function are found by locating where the derivative is zero. For simpler problems this can be done explicitly, but this becomes very complex for functions of more than a few variables. The technique of *gradient descent* (Cauchy 1847) is used to iterate towards a critical point by taking small steps in it's direction, given by the opposite sign of the derivative (Goodfellow, Bengio, et al. 2016). Figure 2.6 visualises the first step of the gradient descent process in 2D, and Figure 2.7 shows the algorithm result in 3D. Note that in Figure 2.6 the local minimum is not the global minimum, which demonstrates how gradient descent provides an *approximation* to a functions minimum rather than a solution to it. Whilst gradient descent cannot be appropriately visualised in a higher dimensional setting, it still applies for higher dimensional functions, which is why gradient descent is a fundamental operation in the field of neural networks (Sanderson 2017).

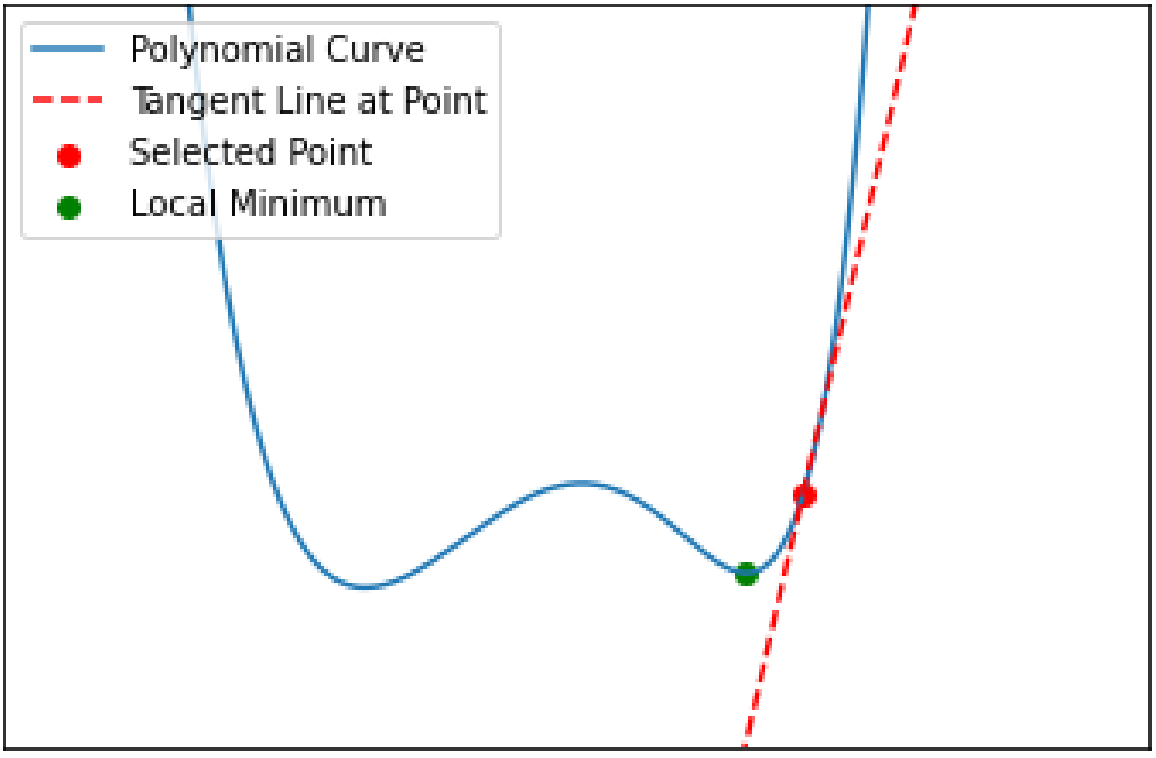


Figure 2.6: A typical polynomial curve with three critical points (Python 3.9). *In red:* a randomly selected point on the curve, with a positive gradient shown by it's tangent line. To move towards the local minimum, a negative step to the left is taken. The process is then repeated. *In green:* the local minimum that the selected point would tend towards, shown by it's tangent line with zero gradient.

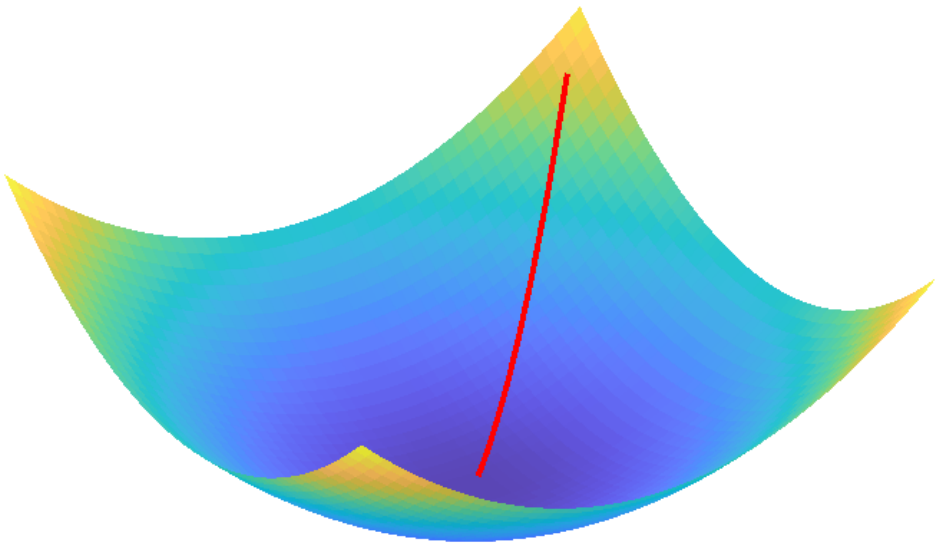


Figure 2.7: A gradient descent algorithm used to find a local minima of an arbitrary, multivariate function. The path of descent can be seen in red (MATLAB R2022b).

Learning rate

The learning rate in gradient descent is a positive scalar which represents the step size per iteration (Goodfellow, Bengio, et al. 2016). The learning rate is a hyperparameter (see section 2.2.2) and is set depending on the function in question. Sometimes a high learning rate can be useful for low gradient functions, but might overshoot the minimum for higher gradient functions. Likewise, a low learning rate is good for steep or unstable functions but might take too long to converge for functions with low gradients.

2.2 Machine Learning techniques

This section of the literature review defines the techniques and concepts more specific to machine learning. The term developer is used throughout and refers to the person(s) that engineer the model.

2.2.1 Learning approaches

The goal of machine learning is for a machine to improve it's output over time (hence 'learning'). There are two main methods of improvement that a model can use, supervised and unsupervised, based on what information the model has to 'learn' from (Goodfellow, Bengio, et al. 2016). Reinforcement learning is an alternative learning method which is covered briefly. The classification problems of the later Chapters are examples of supervised learning; the other learning methods are included for comparison purposes.

Supervised Learning

When a model learns from labelled training data it's learning process is described as supervised (W.-M. Lee 2019). These labels tell the model the desirable outcome, therefore giving the model an objective to aim towards. For example, a model could attempt to recognise images containing a football, and it would do so by learning from images with and without footballs. The images might be labelled *true* or *false*, whether they contain a football or not. Then, given a new image, the model would ask itself "does this image look more like the *true* images, or the *false* images?". Of course, the model doesn't know what a football is, only that a circular object containing black and white polygons appears in the images labelled *true*.

Unsupervised Learning

In contrast, if a model is not given a desired output then it's learning process is 'unsupervised'. Unsupervised learning models are particularly useful for finding relationships in unlabelled data (Malik and Tuckfield 2019). Continuing from the previous example, if the footballs data set was not labelled *true* and *false*, the model would be unsupervised, and it would try to discern the images from one another by finding differences between them. If it's successful the model might tell us that in some of the images there is a circular object, and in some images there is not. It is up to the developer then to draw the conclusion that some of the images contain

footballs.

Reinforcement Learning

Reinforcement learning (RL) is an unorthodox learning method which learns through trial and error (Winder 2020). RL models don't learn by aiming for a known output, instead the model attempts to maximise a reward function, which is provided by the developer. Lanham (2019) compares the training of a RL model to the use of positive reinforcement when training a dog - when the dog performs the trick correctly, it receives a treat. To complete the previous 'images of footballs' example, a reinforcement learning model might have to select the images which contain footballs. The model is given a reward function, suppose it is $y = true - false$ for simplicity. The model's aim is to maximise y, so (by trial and error) it will learn over time that this is done by only selecting items labelled *true*.

Reinforcement learning is a very complex subfield of machine learning and is beyond the scope of this thesis. More mathematical detail and the applications of RL can be found in Phil Winder's comprehensive book 'Reinforcement Learning'.

Transfer Learning

Transfer learning is the use of a pre-existing model for a new purpose (Azunre 2021). By making small changes it's possible to repurpose a machine learning model, which is much simpler than starting from scratch for every machine learning solution.

2.2.2 Hyperparameters

Hyperparameters are parameters that are not learned from the training data but are set prior to the training process. A hyperparameter may be set by the developer if the parameter is too difficult to learn, or if it's unsuitable; Goodfellow (2016) gives an example of an unsuitable parameter as a model's complexity. A model would always choose to overfit the training data to achieve the highest accuracy, so the developer should choose the complexity themselves.

When creating a ML model it's parameters are trained by the training data, whilst the hyperparameters are manually tuned by *validation* data (Goodfellow, Bengio, et al. 2016). The validation data is a second set of test data used by the developer to check the effectiveness of chosen hyperparameters. It's important that the training data and validation data don't share

any observations; they should be unique and likely disjoint subsets of the input vector (see Section 2.2.3 below). Other examples of hyperparameters include number of hidden layers and neurons in a neural network (see section 2.3) or the learning rate in gradient descent (Agrawal 2021).

2.2.3 Cross Validation

Validation is, generally, the process of dividing the training data into an arbitrary number of sections and using all but one of those sections to train the model. Then, to *validate* the model, the last section is used, and the output is scrutinised. Rather than making the decision of which section to use for validation, *cross validation* iterates the validation process, using each section as the validation section once (Starmer 2018a). It's common practice for the data to be divided into ten sections, known as *10-Fold Cross Validation*. Cross-validation is essential for tuning hyperparameters.

2.3 Artificial Neural Networks

Deep machine learning is the term used to describe when a learning process takes place over multiple stages, or layers. Deep learning is essential in extracting meaningful information from complex data, such as images. Mathematically, deep learning can be thought of as function composition. Figure 2.8 shows a traditional network and figure 2.9 shows a deep network for comparison.

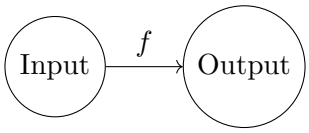


Figure 2.8: A visual example of traditional (non-deep) learning. The model takes an input, applies a function f, and returns an output.

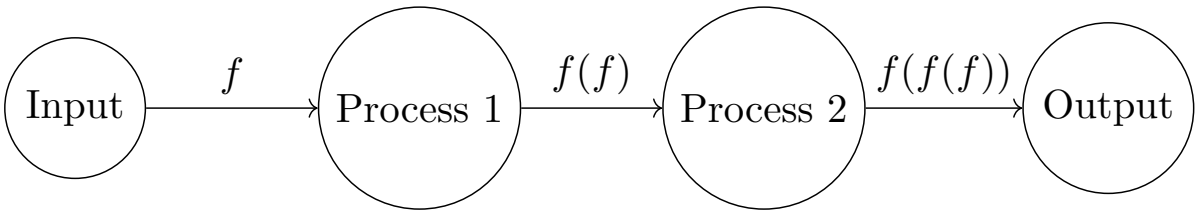


Figure 2.9: A network with multiple layers, hence it is 'deep'. The output of one function f is fed as an input into the next layer. It's clear that the output is a composite of previous functions.

Artificial neural networks, or 'ANNs', are deep learning networks which take inspiration from the biological brain. The nodes in an ANN are referred to as neurons, and each neuron contributes to the output of the network. A neuron acts as a mathematical function, which takes the input(s) from the previous layer's output(s). Feedforward Neural Networks (FNNs) are the fundamental type of ANN. In a FNN the data is fed forward from the input layer into the hidden layer(s) before reaching the output layer. Hidden layers are named as such because generally only the input and output values are visible - the intermediate processes are hidden. FNNs are the simplest and most common type of neural network. A general example of a feedforward neural network is given in Figure 2.10.

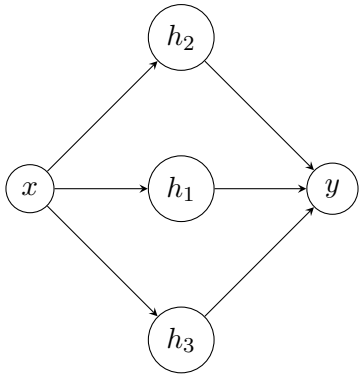


Figure 2.10: A general FNN with one input, x, one hidden layer, $\boldsymbol{h}$, and one output, y. The term 'feedforward' comes from the forward flow of data; there are no cycles (Goodfellow, Bengio, et al. 2016). This neural network could be a regression model containing three parameters. More complex ANNs contain multiple inputs and outputs, and many hidden layers.

2.3.1 The Neuron

The neuron originated from work by neuroscientists on modelling the biological brain. Warren McCulloch and Walter Pitts (McCulloch and Pitts 1943) created the first widely used schematic of the neuron, which was intended for biological purposes. Frank Rosenblatt (Rosenblatt 1958) later created the perceptron, which added weight coefficients to the McCulloch-Pitts neuron. The bias term was soon introduced by Rosenblatt (1962) and the blueprint for the modern neuron was established.

Each neuron in a neural network takes it's input from the previous layer's outputs (except the input neurons) into it's activation function. The result of this activation function is the neurons output. Figure 2.11 shows a schematic of a neuron.

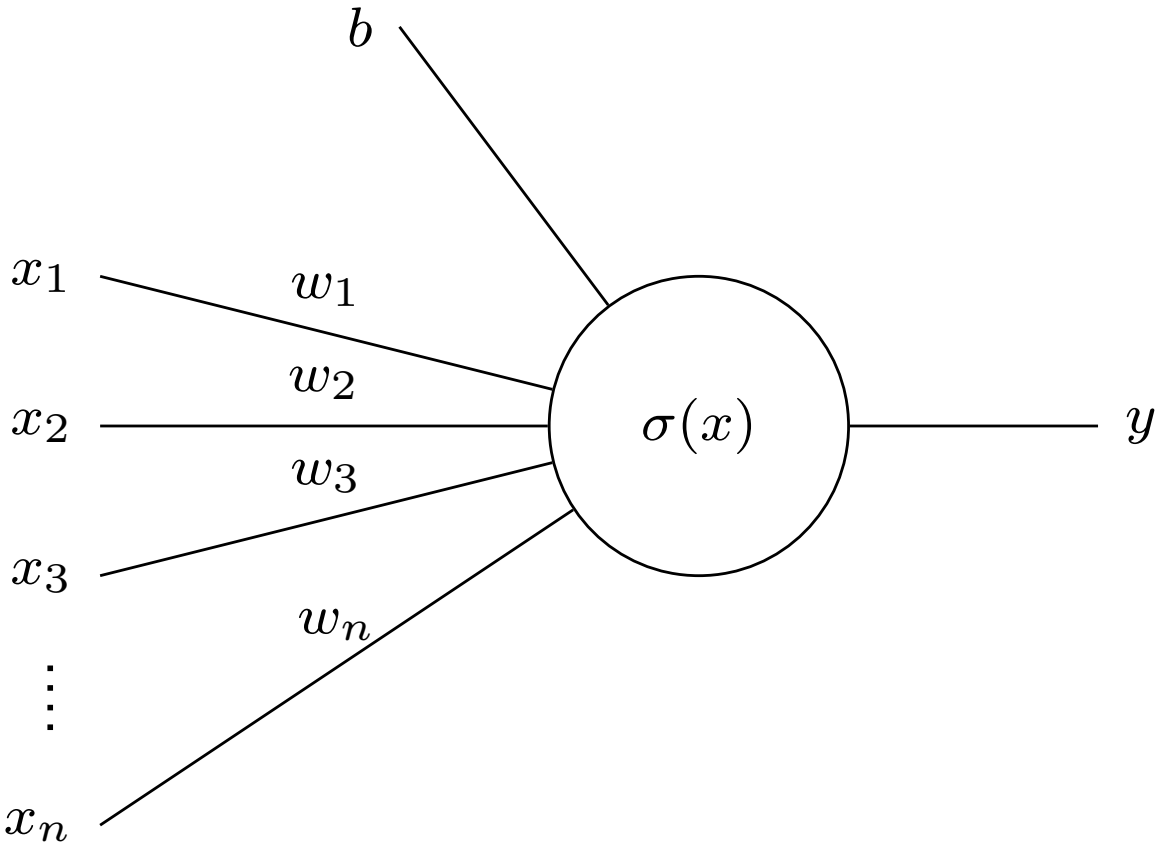


Figure 2.11: The mathematical composition of the modern neuron (Lynch 2023). The inputs x_i have weight coefficients w_i and a bias b. The summation of these terms is entered into the activation function σ, which returns y.

2.3.2 Activation Functions

Each neuron receives inputs which are processed through an activation function. The activation function decides how active the neuron is; the activation value is a real number between zero and one, where zero is inactive and one is completely active. The general activation function for a neural network consists of the sum of all inputs, each multiplied by their respective weight coefficients, plus a bias (Lynch 2023).

The Sigmoid function

The sigmoid curve is a 'S' shaped curve with two asymptotes at the upper and lower bounds. The sigmoid function is given in Equation 2.2. Graphically sigmoid is represented by a logistic curve, shown in Figure 2.12. The adoption of ReLU activation in modern neural networks has led to a decline in the use of the sigmoid function.

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \qquad \{x \, \epsilon \, \mathbb{R}\} \tag{2.2}$$

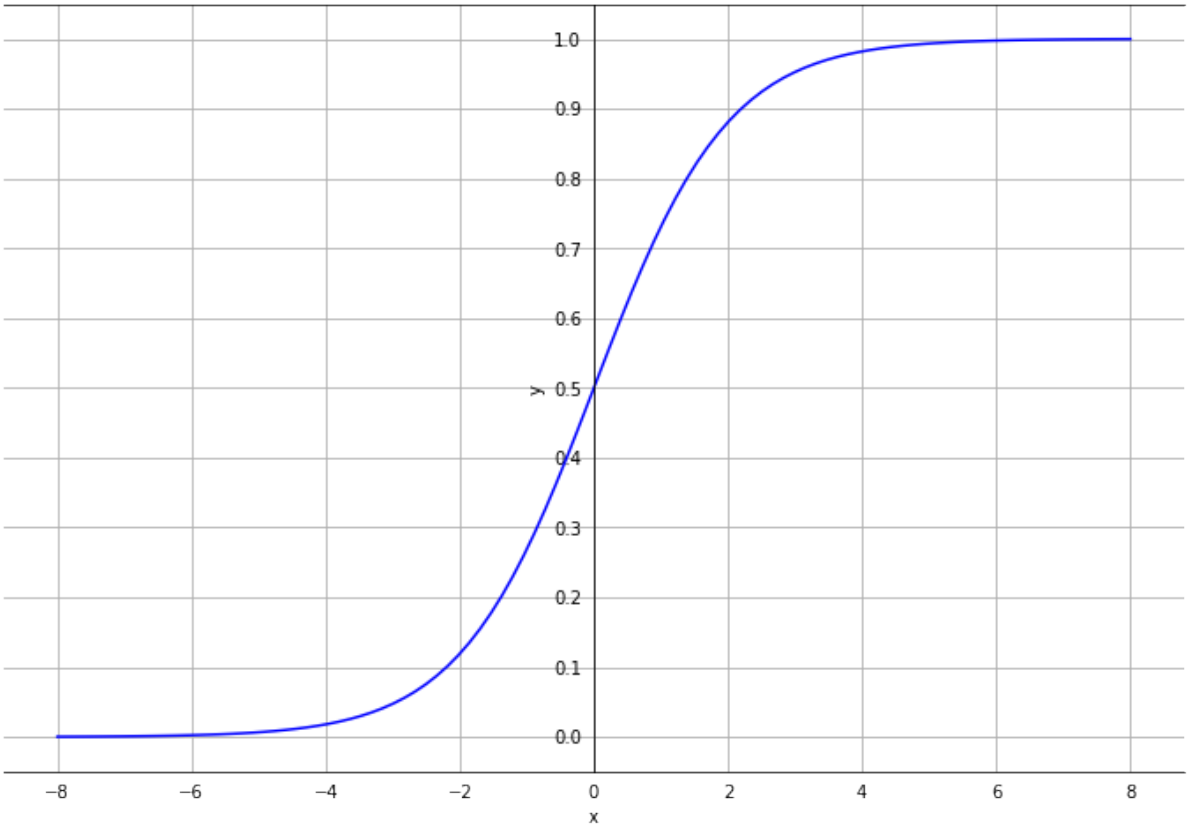


Figure 2.12: The logistic function (Python 3.9). It's clear that the function cannot output a value lower than zero or higher than one. The main disadvantage of the sigmoid function is that it results in positive activation for a negative input, and returns are diminishing for very high inputs.

A unique case of the sigmoid function is softmax activation (Lynch 2023). Due to it's discrete nature it's often used as a normalisation function in the output layer of multi-class classification tasks (see Chapters 3 and 4). Softmax transforms any real number into a probability, so that outputs all sum to 1. The softmax function is shown in Equation 2.3, for the i^{th} class of a k-class neural network output layer.

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}, \qquad \{x \, \epsilon \, \mathbb{R}\} \tag{2.3}$$

ReLU

The 'Rectified Linear Unit' function, or ReLU, is now the prominent activation function in deep learning practises (Sanderson 2017). The ReLU activation function is given in Equation 2.4. The function takes the value zero for all inputs less than zero, and the value of the input for all inputs greater than zero. the ReLU function can be seen in figure 2.13.

$$\sigma(x) = \max\big(0,\, x\big)\ , \qquad \{x\,\epsilon\,\mathbb{R}\} \tag{2.4}$$

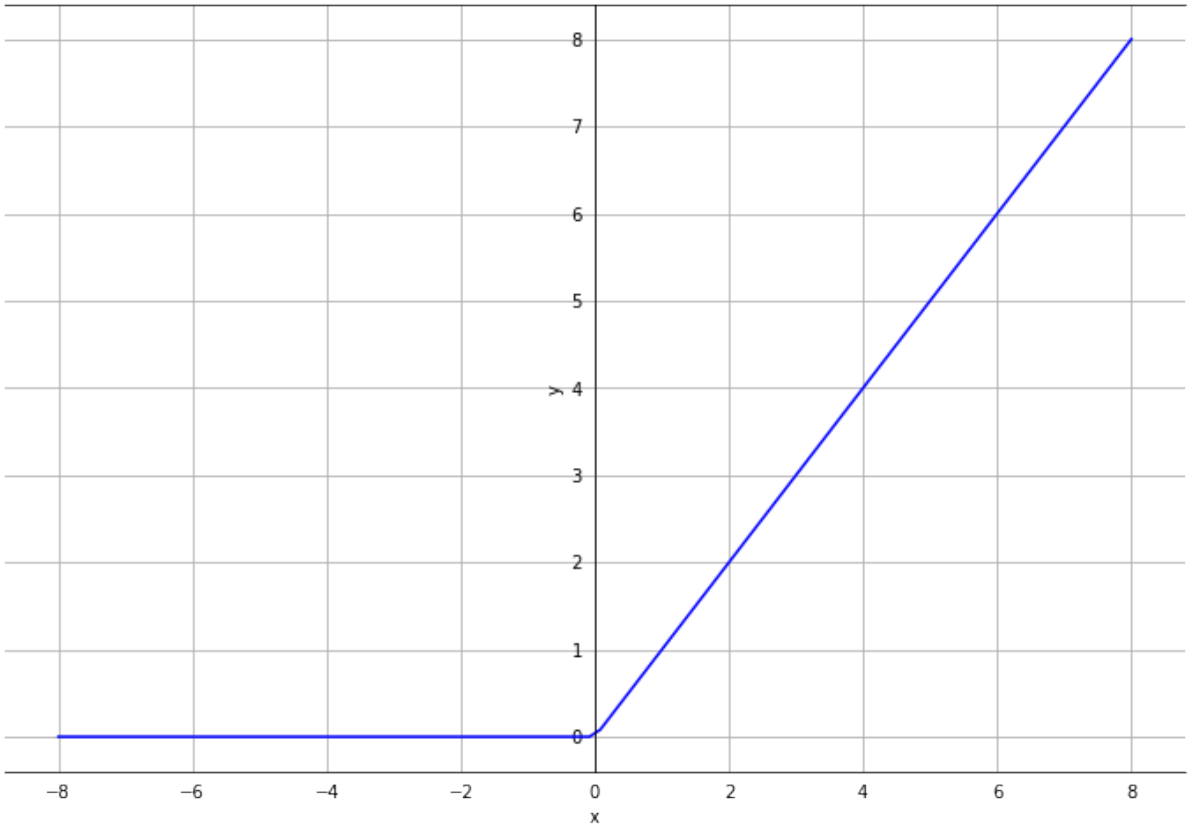


Figure 2.13: The ReLU function (Python 3.9). This improves on the sigmoid function as ReLU results in zero activation for all negative inputs and does not feature diminishing returns for higher inputs.

2.3.3 Regularisation

Section 2.1.3 discussed the issues of overfitting test data. Methods to reduce variance, specifically, to reduce the final model error at the cost of increasing training error are known as *regularisation* techniques (Goodfellow, Bengio, et al. 2016). In the context of neural networks, regularisation is used when one or more neurons become dominant. A neuron can become dom-

inant if it's weight coefficient(s) in the next layer is much higher than the weight coefficient of other neurons, meaning that other neurons in the layer suffer a diminished effect. Figure 2.14 shows neuron with a dominant weight coefficient. There are many ways of regularising a model and the choice of method is situational.

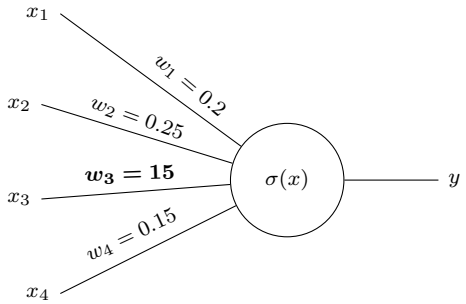


Figure 2.14: A neuron that needs regularising. The weight coefficient w_3 is much higher than the other weights, meaning that the activation value will be dominated by x_3.

L^1 and L^2 regularisation

One way of regularising a model is by adding a penalty term to the model's cost (error) function. This increases the error between the predictions and target values. By doing this the model gains some bias and loses variance (refer back to section 2.1.3), to reduce overfitting. There are two main methods of L (Lebesgue) regularisation.

L^1 regularisation, also known as lasso regression, penalises the model by adding a multiple of the L^1 norm to the objective function. The L^1 norm is the sum of the absolute value of the weights (Starmer 2018b). Regularising with the L^1 norm makes the weight matrix more sparse. L^2 regularisation, or ridge regression, penalises the model by adding a multiple of the L^2 norm to the objective function. The L^2 norm is the sum of the squares of the weights (Starmer 2018c). This results in smaller weightings, but they cannot become zero. The choice between L^1 and L^2 regularisation depends on the relevance of the weightings in the model; penalising with the L^1 norm provides a simpler model by eliminating some weight terms; the L^2 norm creates a complex model with many small but non zero weight terms. Figure 2.15 shows the effect of L regularisation for a regression problem.

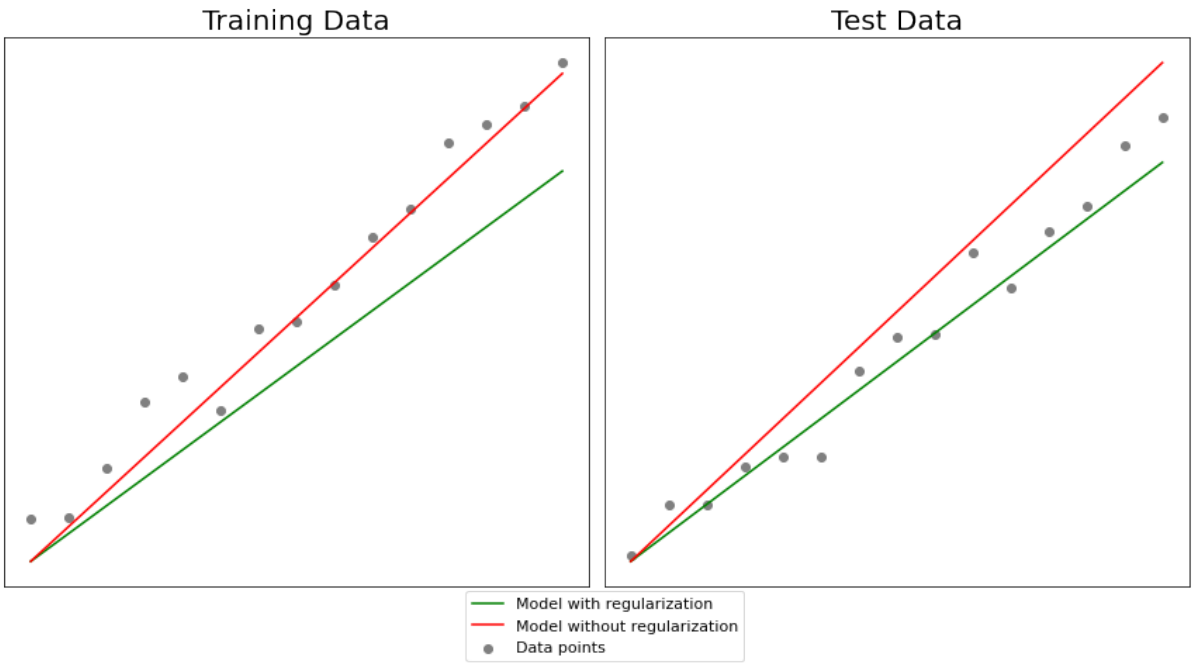


Figure 2.15: A simple regression model with and without L regularisation, for an arbitrary data set; either method could be used to achieve this result (Python 3.9). *left:* In red, the model is trained on the training data, and overfits due to a lack of bias. In green, a new model is created with a Lebesgue penalty term. *right:* Both models are tested on unseen data. It's clear that the regularised model has a better fit.

Dropout regularisation

Another form of regularisation is *dropout*. If one neuron's activation value is very high compared to other neurons in it's layer (Figure 2.14), it might have a comparitively overwhelming effect on the following layer (Goodfellow, Bengio, et al. 2016). This can cause overfitting so to prevent this dropout can be introduced. Dropout randomly disables some neurons in the network for each iteration of the training process, allowing all neurons to have a relevant effect on the output. A basic network with dropout is shown in figure 2.16.

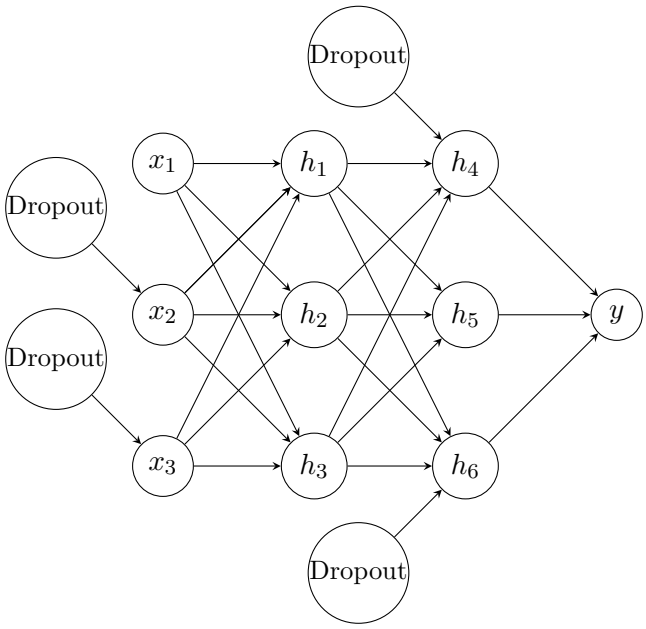


Figure 2.16: An arbitrary neural network featuring dropout regularisation. The dropout neurons have a hyperparameter determining the probability of becoming active. When a dropout node is active it multiplies connected the node by zero, causing the node to inactivate. In this example, x_1 and h_5 might be dominated by the other neurons in their layers, so dropout is applied so that x_1 and h_5 contribute meaningfully to the output. Some neural networks apply a dropout algorithm across the entire network, that chooses which neurons are active and which are not for each iteration (Goodfellow, Bengio, et al. 2016).

2.3.4 Optimisation

The biggest challenge in modern deep learning applications is the balancing act of regularising a model to prevent overfitting whilst optimising the model to prevent underfitting. A fundamental principle of machine learning is optimisation, which mostly involves minimising *cost functions* to reduce bias. It is the counterpart to regularisation.

2.3.5 Backpropagation

Section 2.1.5 covered how the gradient descent algorithm uses the negative gradient to minimise a cost function. Backpropagation is the algorithm used to find the negative gradient when it can't be found explicitly (Goodfellow, Bengio, et al. 2016). This is almost always the case for ANNs, which regularly deal with functions of high dimensions; M6, Alibaba's financial forecasting model, is trained on 100 billion parameters (Lin et al. 2021).

To tell a network that an output is desirable, that output's neuron should be maximised, whilst other output neurons should be minimised (Sanderson 2017). There are multiple ways of increasing the activation value for a neuron. The bias or weightings of that neuron can be

changed, but to do this manually for each neuron, for each training sample is far too labor intensive for the developer. Instead the neuron activations in the previous layer $n-1$ can be changed by maximising the neurons which contribute the most to the desired output (Sanderson 2017). To do this, the process must be repeated, by maximising the relevant neurons in layer $n-2$. This is repeated until the input layer is reached. This process of *propagating backwards* is known as backpropagation. Figure 2.17 shows backpropagation for a small network.

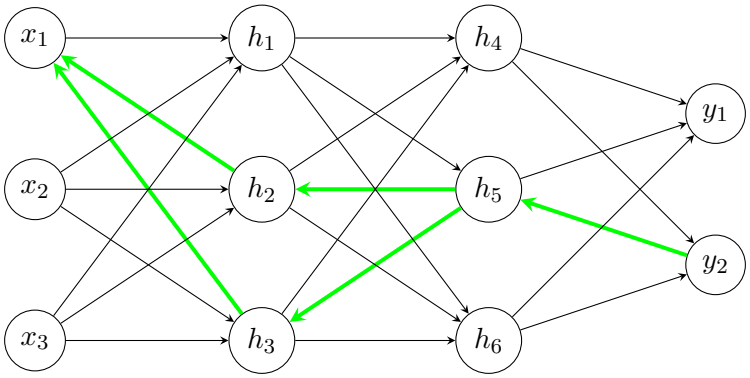


Figure 2.17: A demonstration of backpropagation, or *backprop*, for a neural network. Given that y_2 is the desired output, y_2 should be maximised by working backwards to find which neuron(s) contribute most. If h_5 contributes most, the process is repeated, looking to maximise h_5.

The collection of changes to all of the neurons' activation functions is stored as an n^{th} dimensional vector, which is the n^{th} dimensional gradient that can be used in the gradient descent algorithm.

2.3.6 Stochastic Gradient Descent

Running a gradient descent algorithm with the whole 'batch' of training data is computationally expensive so instead *stochastic* gradient descent or SGD is used. SGD takes 'mini-batches' of the training data and calculates one step of the descent with each mini-batch. Whilst the mini-batches are not as accurate as the whole batch (the gradient of a mini-batch is an approximation of the gradient of a batch (Sanderson 2017)) the computational efficiency gained is worth the small loss in accuracy. Stochastic gradient descent is visualised in Figure 2.18.

Definition 2.5. ***Stochasticity.*** *Stochasticity refers to randomness or probabilistic behavior inherent in a system or process (McNeill and Pinheiro 2014). In neural networks, stochasticity can be intentionally introduced to increase uncertainty or variability in a model (Goodfellow, Bengio, et al. 2016). In Stochastic gradient descent, the mini-batch sampling process is stochas-*

tic.

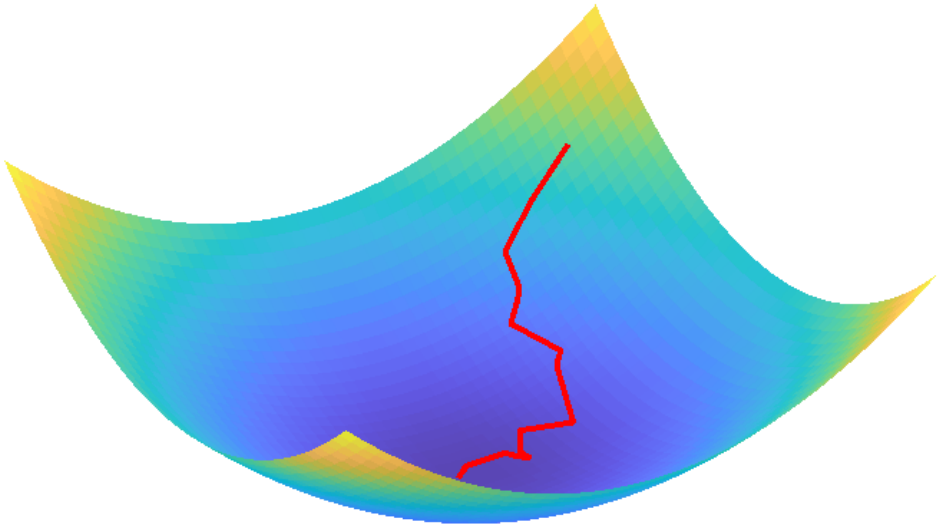


Figure 2.18: A visualisation of the stochastic gradient descent algorithm (MATLAB, R2022b). Comparisons can be made to Figure 2.6. Each step is calculated using one of n mini-batches, which is n times quicker to calculate per step. This means that the GD algorithm is trying to minimise a slightly different function every step, causing stochasticity within the process.

2.3.7 Performance Metrics

In artificial neural networks two important performance metrics are accuracy and coverage (Goodfellow, Bengio, et al. 2016). Accuracy concerns minimising error, whereas coverage concerns the fraction of observations which the model can respond to. It's important to balance these metrics - 100% accuracy can be achieved by processing no samples, and 100% coverage can be achieved by processing all samples - the developer should aim to maximise both. Another important performance metric is loss. Loss quantifies the difference between the predicted output of a network and the target output. Loss is often defined by a loss function, which stochastic gradient descent aims to minimise. When tuning hyperparameters for a neural network a common goal is to find a model that maximises accuracy and minimises loss.

2.4 Types of Neural Network

2.4.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a specialist type of network which are efficient at processing long sequences of values (Goodfellow, Bengio, et al. 2016). Because of this RNNs perform best on sequential problems such as time-series prediction (Ceschini et al. 2022) and natural language processing (Bouarara 2021).

Recurrent neurons take input from the previous layer and also from the neurons' previous state. Because of this recurrent layers are cyclic and have an internal memory (Lynch 2023). The Hopfield network (Hopfield 1982) is an early example of a recurrent neural network 2.19 and introduced the concept of addressable memory. The Hopfield network is still implemented in modern neural networks as a method of retrieving stored information from previous inputs.

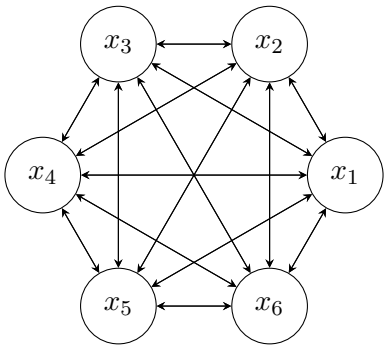


Figure 2.19: Discrete Hopfield Recurrent Neural Network with 6 neurons (Hopfield 1982).

For further reading on Recurrent Neural Networks see the eponymous chapter in 'Python for Scientific Computing and Artificial Intelligence' by Stephen Lynch.

2.4.2 Generative Adversarial Networks

A Generative Adversarial Network (GANs) is a novel network type which consists of a pair of ANNs that are trained simultaneously in a competitive manner (hence 'adversarial'). It's inventor, Ian Goodfellow, gives likeness to a strategy game where each network tries to beat the other (Goodfellow, Pouget-Abadie, et al. 2020). GANs have been used creatively to create artistic landscapes or synthetic images of people (See Figure 2.20). There are many practical uses as well, such as converting satellite data into road maps (Chen et al. 2021) or converting a sketch taken at a crime scene into a realistic face to aid in identification (Wadhwa et al. 2019).

Importantly, GANs can be applied to create synthetic training data for other neural networks

to make use of, which are sometimes referred to as a generative teaching networks or GTNs (Such et al. 2020).

Figure 2.20: 1024 × 1024 images of synthetic 'celebrities' generated with a GAN using the CELEBA-HQ dataset (Karras et al. 2017).)

In a GAN there are two networks, a *generator* and a *discriminator*. There is a cost (error) function which encourages improvement in the networks. The generator function aims to minimise it's cost by having the discriminator incorrectly class it's output as real. Likewise, the discriminator is given either a real image from the training data *or* the output of the generator function. The discriminator minimises it's cost by correctly classifying the input as real or fake.

For more information on GANs see Ian Goodfellow's 2020 article 'Generative Adversarial Networks'.

2.5 Convolutional Neural Networks

A convolutional neural network (CNN) is a specialist type of network designed to solve tasks by making use of convolution, a specialised linear operation which replaces standard matrix multiplication (Goodfellow, Bengio, et al. 2016). CNNs are widely used for image recognition and classification due to the usefulness of the convolution operation in a two dimensional setting; an image can easily be transformed into a grid of numerical values representing the colour or brightness of each pixel. A general convolutional neural network contains five types of layers; the input layer, the convolutional layer, the pooling layer, the fully connected layer and the output layer (Lynch 2023).

Convolution

Convolution is a mathematical operation that takes two functions as an input and gives a third function as an output (Sanderson 2022). Convolution is known for it's 'sliding' mechanism where one of the two functions slides along the other, multiplying the function outputs at each step. Convolution is demonstrated in a statistical context in Figure 2.21. In the context of neural networks convolution is used by sliding a kernel (filter) along an input matrix. The convolution process in a neural network is given in Figure 2.22.

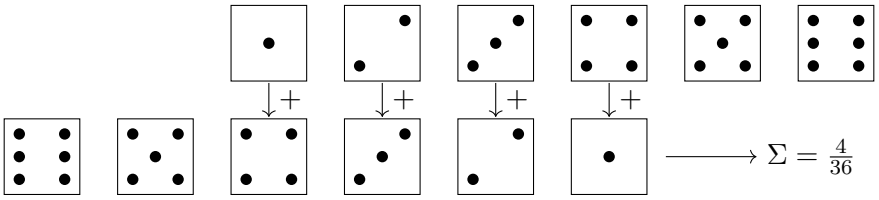
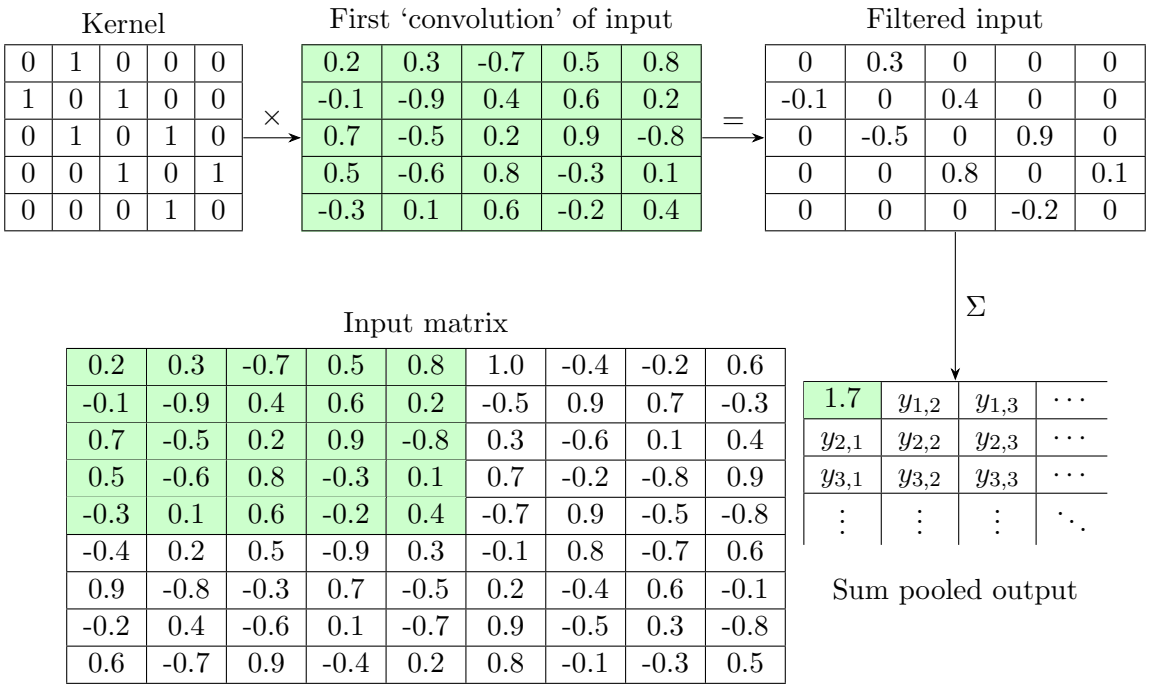


Figure 2.21: The probability of rolling each number on two dice (biased or unbiased) can be calculated with convolution. Given these are fair dice the probability of rolling any number is $\frac{1}{6}$. By multiplying the probabilities of the vertically adjacent die (one and four, two and three, three and two, four and one) and taking the sum of these results, we find the probability of rolling a five. The sliding of the functions then occurs, so that the die on the bottom move one unit to the right. This would find the probability of rolling a six.



0	1	0	0	0
1	0	1	0	0
0	1	0	1	0
0	0	1	0	1
0	0	0	1	0

0.2	0.3	-0.7	0.5	0.8
-0.1	-0.9	0.4	0.6	0.2
0.7	-0.5	0.2	0.9	-0.8
0.5	-0.6	0.8	-0.3	0.1
-0.3	0.1	0.6	-0.2	0.4

0	0.3	0	0	0
-0.1	0	0.4	0	0
0	-0.5	0	0.9	0
0	0	0.8	0	0.1
0	0	0	-0.2	0

0.2	0.3	-0.7	0.5	0.8	1.0	-0.4	-0.2	0.6
-0.1	-0.9	0.4	0.6	0.2	-0.5	0.9	0.7	-0.3
0.7	-0.5	0.2	0.9	-0.8	0.3	-0.6	0.1	0.4
0.5	-0.6	0.8	-0.3	0.1	0.7	-0.2	-0.8	0.9
-0.3	0.1	0.6	-0.2	0.4	-0.7	0.9	-0.5	-0.8
-0.4	0.2	0.5	-0.9	0.3	-0.1	0.8	-0.7	0.6
0.9	-0.8	-0.3	0.7	-0.5	0.2	-0.4	0.6	-0.1
-0.2	0.4	-0.6	0.1	-0.7	0.9	-0.5	0.3	-0.8
0.6	-0.7	0.9	-0.4	0.2	0.8	-0.1	-0.3	0.5

Figure 2.22: One convolution of an arbitrary input matrix. Here the 5×5 kernel is a filter to detect diagonal lines in the input matrix. By element wise multiplication the result is a filtered matrix, which is then pooled to give an output (see section 2.5.1). This output is then given to the neurons activation function. Notice the similarity in concept to Figure 2.21.

Padding and Strides

Padding is a common matrix operation where a border of elements, usually zero, is applied to an input matrix (Goodfellow, Bengio, et al. 2016). Without padding the corner elements would only appear in one convolution and central pixels would be overfitted by the model. Figure 2.23 demonstrates the padding operation.

The number of 'strides' describes the spacing between each convolution. Skipping a space between each convolution means that the convolutional layer has a stride of two, rather than one. Increasing the stride is used when there are concerns for computational efficiency.

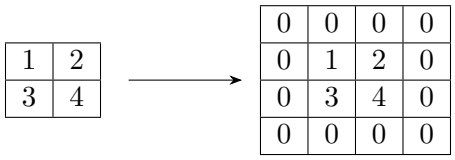


Figure 2.23: A 2×2 matrix is transformed into a 4×4 zero-padded matrix.

2.5.1 Pooling Layers

Pooling is an operation that reduces dimensionality of an input (Lynch 2023). There are multiple pooling methods; the choice of pooling method is a hyperparameter. *Average* pooling works by taking the average of the elements in a vector, and *Max* pooling simply takes on the highest value present in the vector. Figure 2.22 uses *Sum* pooling, where the elements are added together. When pooling a high dimensional matrix, convolution can be used by pooling submatrices (alike to Figure 2.22). Increasing the stride can be done when pooling a very large matrix, which is known as downsampling (Goodfellow, Bengio, et al. 2016).

In summary, pooling reduces model complexity whilst retaining spacial structure. This improves computational efficiency at a small cost in accuracy.

2.5.2 Feature Extraction for Classification

The processes and layers discussed so far are concerned with *feature extraction*. Convolution between a kernel and an input takes place, in aim of extracting a specific feature (usually an edge). The dimensionality of the convolution output is then reduced using pooling. After many iterations of this process the output is a filtered transformation of the input, flattened in the form of a column vector. Following the feature extraction process, the fully connected or 'dense' layers are then tasked with *classification*, by learning the extracted features in aim of

classifying the input as the correct output class. The dense layers of a CNN are identical to the fully connected layers in an ANN. Figure 2.24 shows the architecture of the dense layers of an arbitrary CNN. An applied example with activation values is given in Figure 2.25.

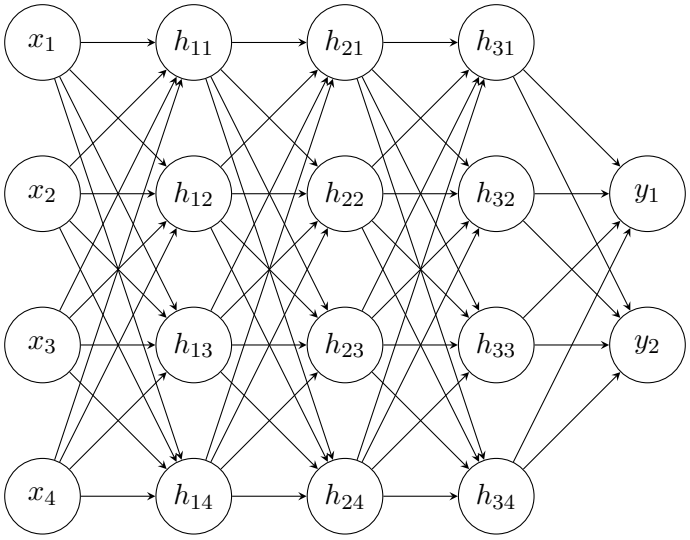


Figure 2.24: The fully connected layers of a two-class convolutional neural network. The input $\boldsymbol{x}$ is the column vector resulting from the flattening of the convolution and pooling processes. The output of the fully connected layers is whichever output neuron has the highest activation.

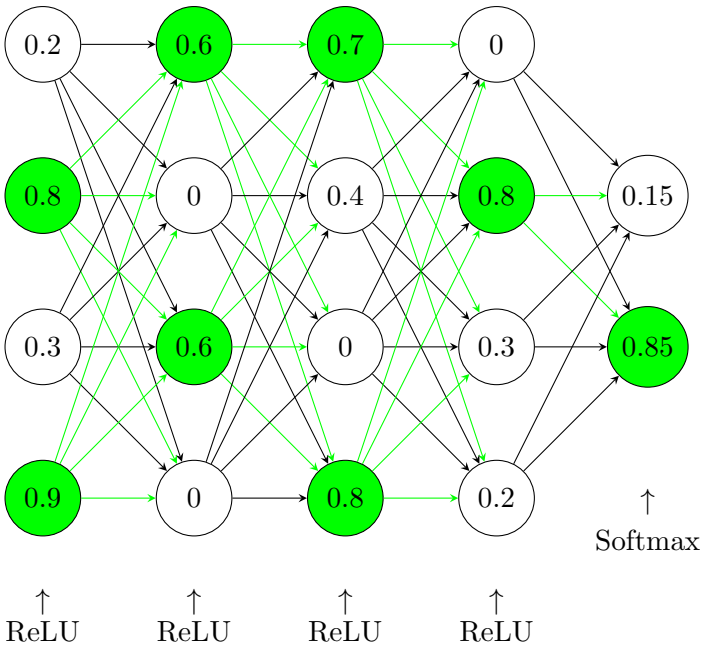


Figure 2.25: The convolutional neural network from Figure 2.24, featuring activation values. Highly activated neurons are in green. It's clear that $y_2 = 0.85$ is the output class. Typically ReLU is used as the activation function, except in the output layer which requires Softmax so that the classes sum to 1.

To further explore how each neuron's output value is calculated, the composition of neuron

$h_{11} = 0.6$ from Figure 2.25 is shown in Figure 2.26. Equations 2.5:2.7 show the calculations for the activation of neuron h_{11}.

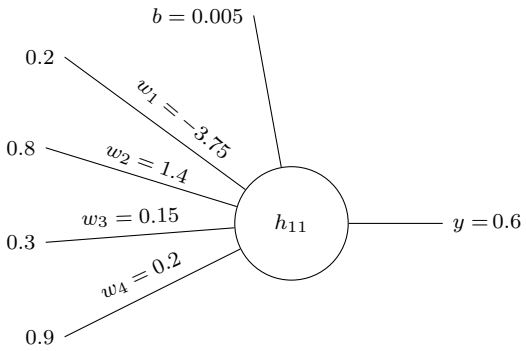


Figure 2.26: The breakdown of neuron h_{11} from Figure 2.25. The inputs x_i are from the previous layer, which are multiplied by the synaptic weights w_i. The bias is added and the result is the input for the ReLU activation function.

To find neuron h_{11}'s output we can manually calculate it using the previous layer's output values,

$$h_{11} = 0.2 * w_1 + 0.8 * w_2 + 0.3 * w_3 + 0.9 * w_4 + b \tag{2.5}$$

The synaptic weight coefficients and bias from Figure 2.26 can be substituted,

$$h_{11} = 0.2 * (-3.75) + 0.8 * (1.4) + 0.3 * (0.15) + 0.9 * (0.2) + (0.005) = \mathbf{0.6}\,. \tag{2.6}$$

Finally ReLU activation is applied to discard negative values,

$$\sigma(h_{11}) = \max(0, h_{11}) = \max(0, 0.6) = 0.6\,. \tag{2.7}$$

Which is the expected output of neuron h_{11} in Figure 2.25.

Chapter 3

MNIST: A Convolutional Neural Network for Handwritten Digits

3.1 Introduction

Handwritten digit recognition is a fundamental part of machine learning and computer vision. Originating from the United States National Institute of Standards and Technology (NIST), the modified NIST dataset (MNIST) has served as a cornerstone for benchmarking various machine learning algorithms, particularly Convolutional Neural Networks (CNNs). The dataset comprises 60,000 training images and 10,000 testing images, each depicting a grayscale image of a handwritten digit (0-9), normalized and centered within a fixed-size image (Lecun 1998). MNIST is often used to test new network architectures before implementing them to solve more complex problems (Goodfellow, Bengio, et al. 2016), and is widely regarded as the *"Hello World!"* of computer vision. Figure 3.1 shows six images from the MNIST database.

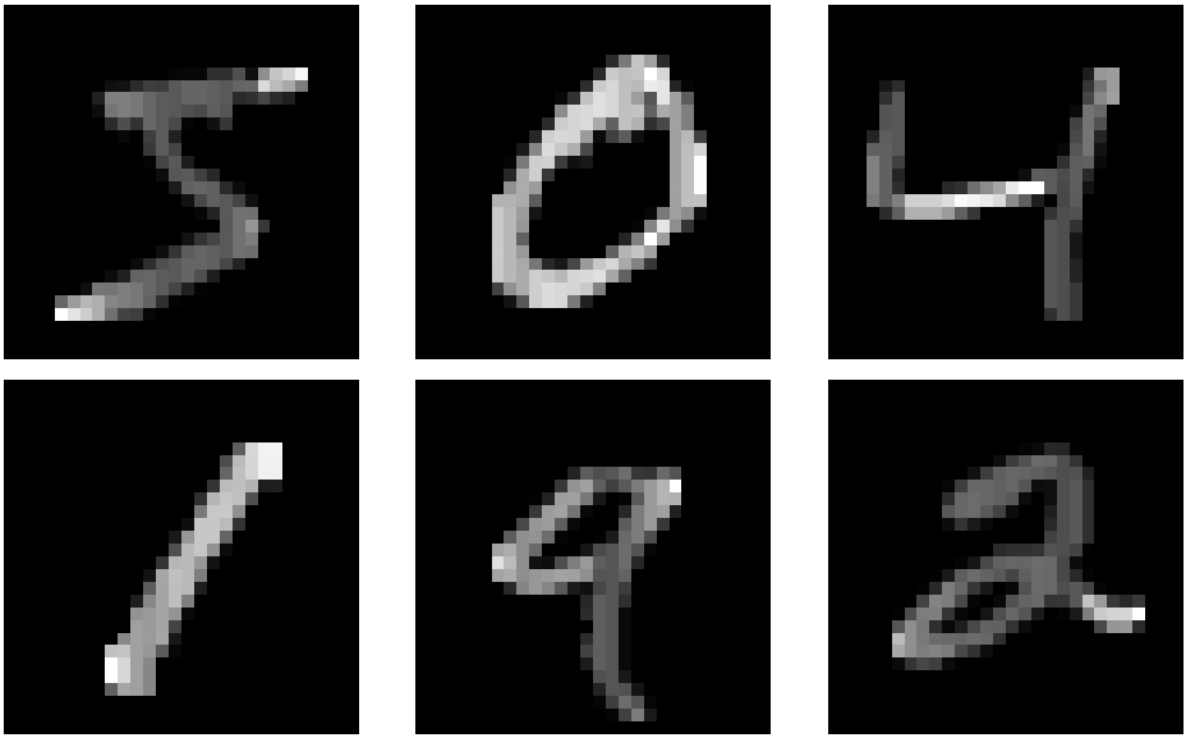


Figure 3.1: The head of the MNIST database. Each image is a handwritten digit, scaled to 28×28 pixels, processed into grey scale and given a black background.

In this chapter a CNN template is modified and applied to the MNIST database. First the methodology for the MNIST neural network is described. The network results are then given, followed by a discussion.

3.2 Methodology

This section will cover the methods implemented to create a neural network to classify images in the MNIST database. The network template in this Chapter was sourced from Chapter 20 of *Python for Scientific Computing and Artificial Intelligence* (Lynch 2023).

3.2.1 Model architecture

The template consisted of a model with two convolutional layers, a flattening layer and two fully connected layers. The basic architecture of the model is given in figure 3.2. Pooling layers were not included; pooling improved the run time of the model however a large fall in accuracy was also recorded.

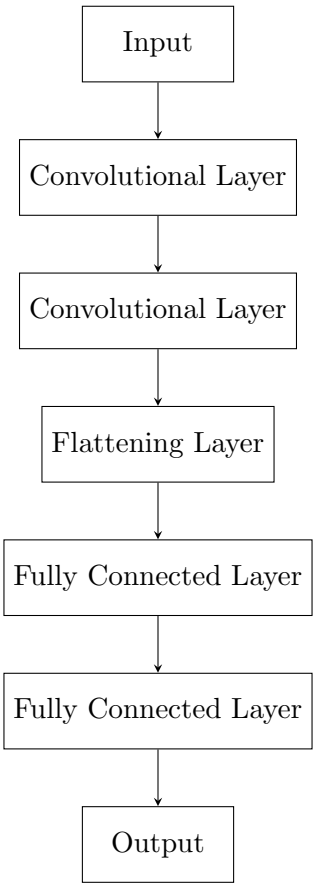


Figure 3.2: Flow chart describing the basic architecture of the MNIST neural network template.

3.2.2 The input layer

The MNIST dataset is included in Keras for engineers to test their architectures on. Figure 3.3 shows MNIST being loaded and the network being initialised.

```
#Loading MNIST (from Keras)
mnist = tf.keras.datasets.mnist # Digits 0-9, 28x28 pixels
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalising the data.
x_train = tf.keras.utils.normalize(x_train, axis = 1)
x_test = tf.keras.utils.normalize(x_test, axis = 1)

# Building the CNN.
input_shape=(28,28,1)
inputs = tf.keras.layers.Input(shape=input_shape)
```

Figure 3.3: The Python 3.10.12 code for setting the MNIST database as the input for the neural network. Normalisation is essential at this step so that patterns learned from the training data can be applied to the testing data.

3.2.3 The convolutional layers

The (28×28) input is processed through two convolutional layers which extract key features through a (5×5) edge detecting kernel. The Python code for the convolutional layers is given in Figure 3.4.

```
#####Convolutional layers#####
layer = tf.keras.layers.Conv2D(filters=64, kernel_size=(5,5), \
strides=(2,2), activation=tf.nn.relu)(inputs)

layer = tf.keras.layers.Conv2D(filters=64, kernel_size=(5,5), \
strides=(2,2), activation=tf.nn.relu)(layer)
#####--------------------#####
```

Figure 3.4: The Python 3.10.12 code for creating two convolutional layers. A 5×5 kernel is used with a 2×2 stride, and ReLU activation is used on the resulting 64 output 'filters', which sets negative values to zero.

Parameter choices

Two experiments were performed on the choice of activation function and stride length in the convolutional layers.

For the convolutional layers three models were compared using ReLU, sigmoid and softmax activation. Both model accuracy and run time were taken into account.

By reducing stride from (2×2) to (1×1) there were four times more convolutions, so features could be extracted more accurately. However computational demand also increased four-fold. Figure 3.5 shows convolution of an image with a larger stride and Figure 3.6 shows convolution of the same image with a smaller stride. Note that different features are detected.

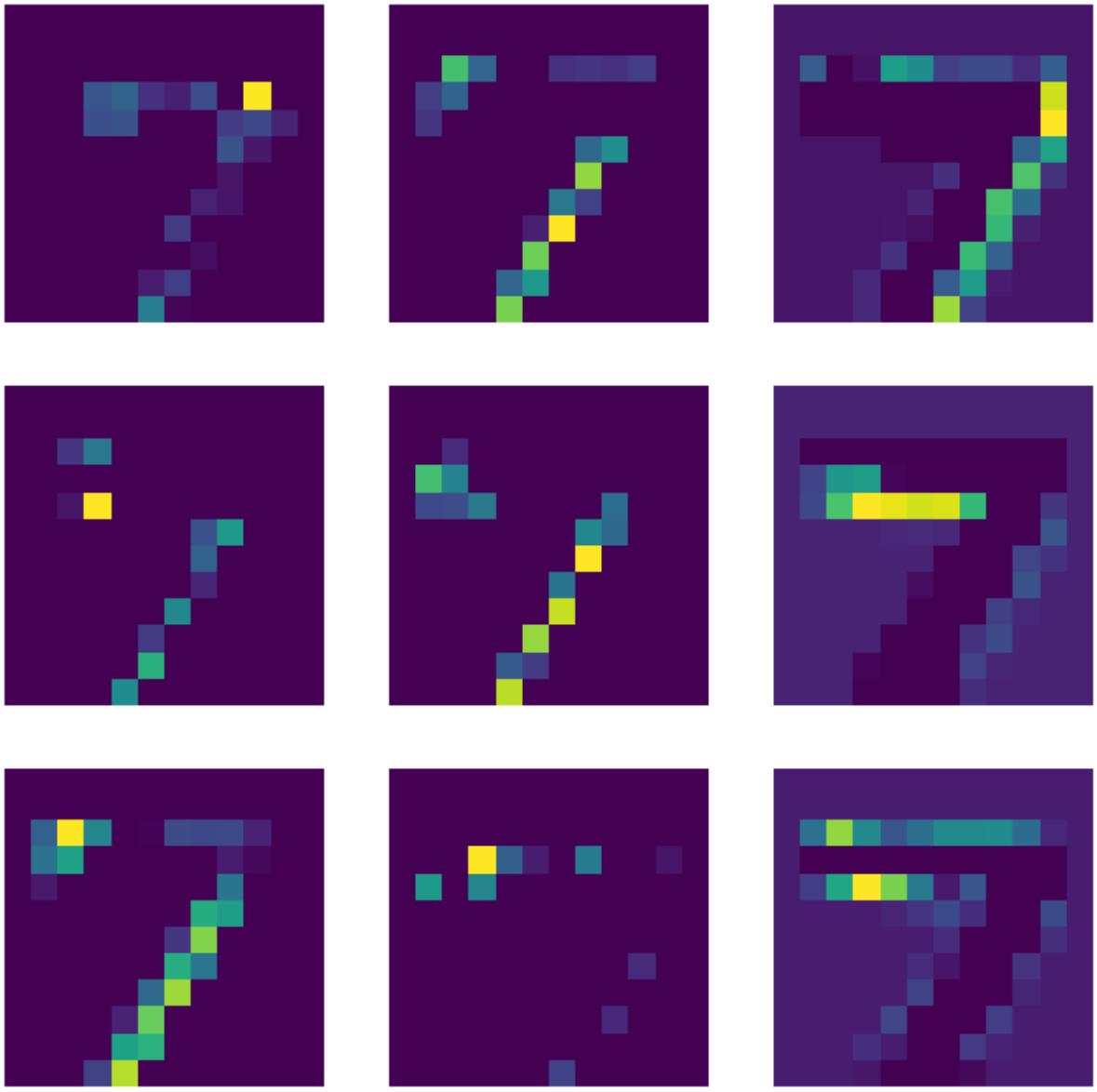


Figure 3.5: The first 9 convolutions of a '7' from MNIST using a 2×2 stride. The output is a 12×12 image of highlighted edges.

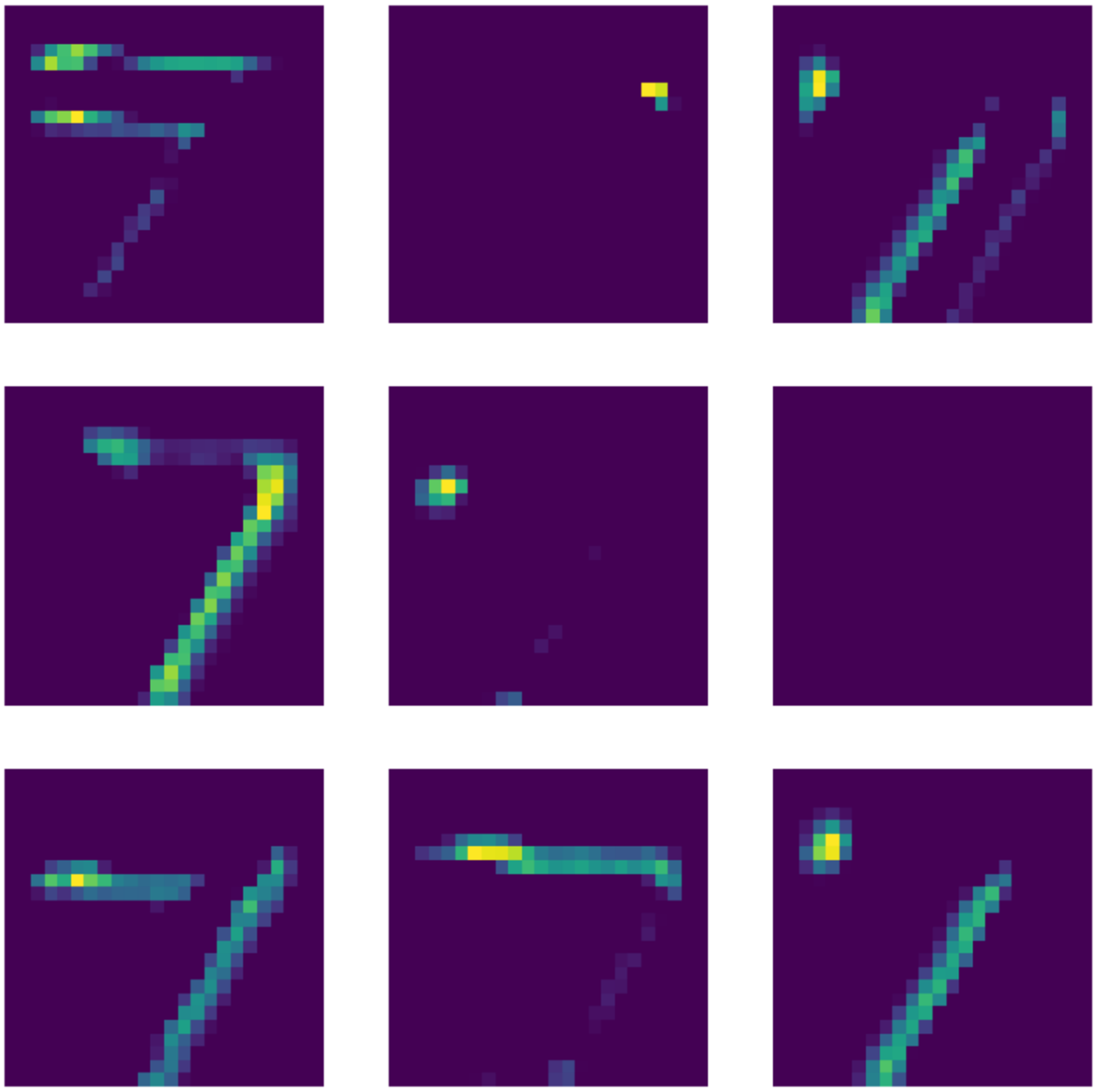


Figure 3.6: The first 9 convolutions of a '7' from MNIST using a 1×1 stride. The extracted features are clearer and the output is larger, at 24×24 pixels.

3.2.4 The flattening layer

The flattening layer takes the output of the convolutional layers, 64 4×4 images, and orders them as a column vector. The process of flattening the convolution output is shown in Figure 3.8.

```
#####Flattening layer#####
layer = tf.keras.layers.Flatten()(layer)
#####----------------#####
```

Figure 3.7: The Python 3.10.12 code for creating the flattening layer. The input is transformed into a column vector.

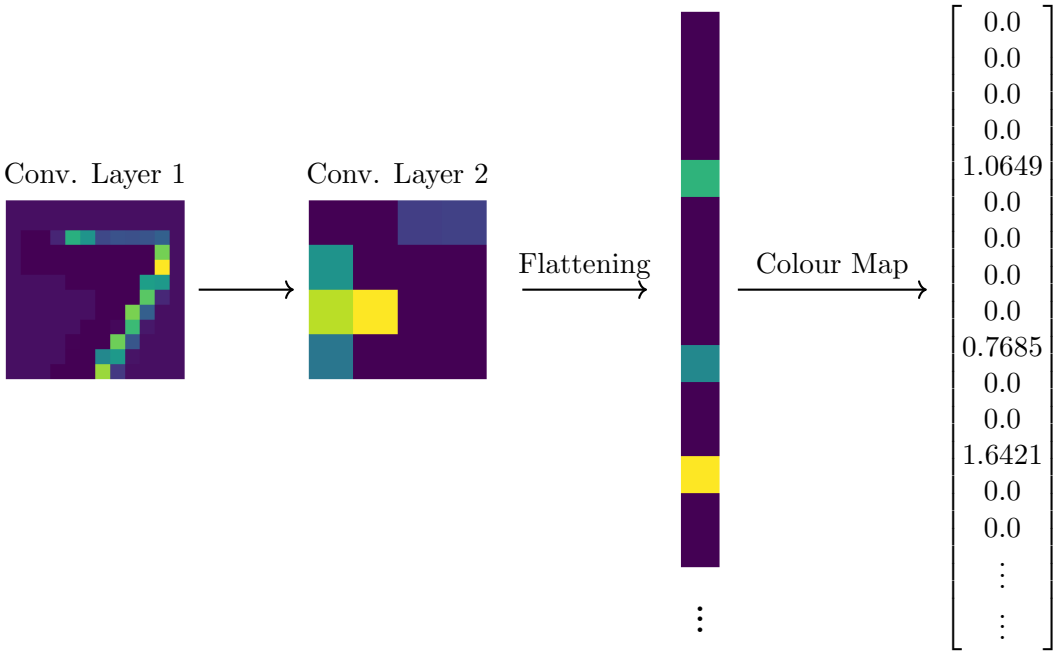


Figure 3.8: The process of the convolutional and flattening layers in the MNIST neural network. The flattened pixel vector is converted into numerical values using the viridis colour map.

3.2.5 The fully connected and output layers

The fully connected or 'Dense' layers' were implemented to learn the features of the handwritten digits. Figure 3.9 shows the code for creating the fully connected layers of the MNIST neural network. The fully connected layers are modelled in Figure 3.10.

```
######Fully connected layers#####
layer = tf.keras.layers.Dense(128, activation = tf.nn.relu)(layer)

layer = tf.keras.layers.Dense(128, activation = tf.nn.relu)(layer)
#####-----------------------#####


#####Output layer#####
outputs=tf.keras.layers.Dense(10,activation=tf.nn.softmax)(layer)
#####------------#####
```

Figure 3.9: Python 3.10.12 code for creating the two fully connected layers and the output layer. Note that the output layer is functionally identical to the previous layers except for the change in number of outputs (10) and the change in activation function. Softmax activation is used in the output layer to convert the previous layers' output into a probability (see Section 2.3.2).

```
# Compile the CNN.
model = tf.keras.Model(inputs, outputs)
model.summary()
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(x_train, y_train, epochs = 3)
```

Figure 3.11: The Python 3.10.12 code for compiling the layers of the MNIST model. Accuracy is decided as the main metric. The Adams optimiser is used and categorical cross-entropy is the default choice of loss function for classification problems; the choice of optimiser and loss function are beyond the scope of this report.

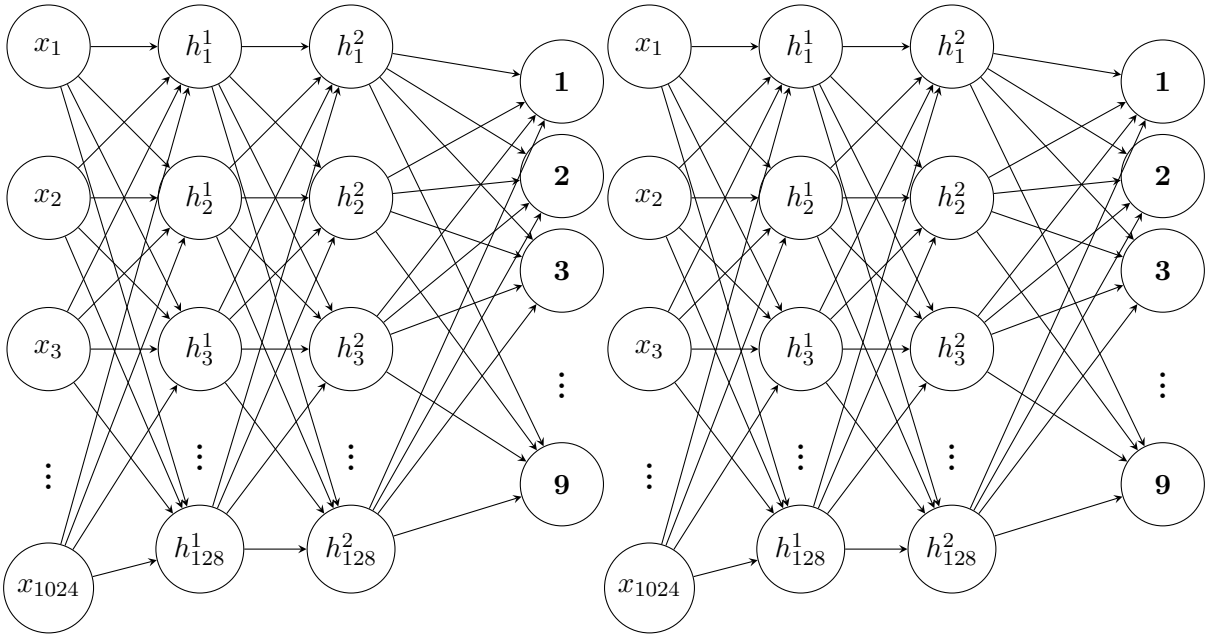


Figure 3.10: The fully connected or 'dense' layers of the MNIST neural network. This is where the learning of features took place. The input $\boldsymbol{x}$ is the 1024×1 column vector resulting from the flattening layer (see Figure 3.8). There are two hidden layers, $\boldsymbol{h^{1,2}}$, each with 128 neurons. The output is a 10×1 column vector containing the probabilities that the input image belongs to each class. The output neuron with the highest probability is the prediction of the model.

3.2.6 Compiling the model

Parameter initialisation is random for each run of a model, which causes slight fluctuations in accuracy and run time. The model was compiled using the code in Figure 3.11.

3.3 Results

In this section the results of the MNIST model are given, including the results of experimentation on hyperparameter choice and model complexity.

For the convolutional and fully connected layers, ReLU activation was chosen due to it's edge

in accuracy over other activation functions (Table 3.1).

Table 3.1: Comparison of activation functions in the convolutional layer. The accuracy of ReLU is higher because it sets negative values to zero, whereas other methods set negative values close to zero, so that they still have some effect.

Method	Performance	
	Time (s)	Accuracy (%)
ReLu	143.5	98.71
Sigmoid	143.4	96.02
Softmax	144.0	97.59

For the choice of stride length in the convolutional layer, a stride length of 1×1 was chosen. Table 3.2 shows the results of the experiment on stride length. Decreasing the stride brought an accuracy increase of 0.5% from the training data. The model with the 1×1 stride had 34 more correct classifications than the model with a 2×2 stride. The difference in run time was significant, as the 2×2 stride model took $\sim$2.5 minutes to run, whilst the more accurate model took ~ 27.5 minutes to run. In a practical setting the 2×2 stride length would be preferred due to significantly better efficiency. However, there was no limit on computational resources, nor any requirement for a model that is trained quickly, hence the more accurate model was chosen.

Table 3.2: Incorrect classifications with varying stride in the convolutional layers. It's clear that the smaller stride length makes less incorrect predictions, at the cost of a far longer run time. Accuracy was calculated manually from the testing data.

Predicted Digit	Incorrect No. Classifications	
	2×2 Stride	1×1 Stride
0	6	4
1	7	8
2	9	13
3	12	10
4	20	7
5	11	14
6	26	18
7	32	14
8	16	13
9	10	14
Total	**149**	**115**
Accuracy (%)	98.51	98.85
Time (s)	144	1643.9

The Final Model

The final model architecture is given in Table 3.3. The final model features a training accuracy of 99.13% and loss of 2.83%.

Table 3.3: The final model architecture for the MNIST neural network. There is one input image of 28×28 pixels, which after two convolutional layers becomes 64 images of 20×20 pixels with extracted features dominating. All pixels are flattened into a $25,600 \times 1$ column vector, and put through the fully connected layers (see Figure 3.10). The output is a 1×10 of probabilities that the input belongs to each class.

Layer (type)	**Output Shape**	**No. Parameters**
input (InputLayer)	[(28, 28, 1)]	0
Conv2d_1 (Conv2D)	(24, 24, 64)	1,664
Conv2d_2 (Conv2D)	(20, 20, 64)	102,464
Flatten (Flatten)	(25,600)	0
Dense_1 (Dense)	(128)	3,276,928
Dense_2 (Dense)	(128)	16,512
Output (Dense)	(10)	1,290
Total Parameters		**3,398,858**

3.4 Discussion

The MNIST model effectively classifies handwritten digits at an acceptable accuracy for further application. The model accuracy is acceptable, at 99.13% from training and 98.85% on the testing data. However, this accuracy was made by decreasing stride length, which increased the model run time almost 12-fold and caused the number of parameters to be $3,398,858$ (Table 3.3). When comparing to leading MNIST benchmark models, SOPCNN (Assiri 2020) achieved 99.83% accuracy with only $1,400,000$ parameters. Decreasing stride length is therefore a brutish approach towards increasing accuracy, which is successful but largely inefficient.

There were some avenues that were not explored, such as the choice of optimiser. Optimisation methods are outside of the scope of this report so the default 'adams' method was used, which is a commonly used alternative to the stochastic gradient descent algorithm.

The model could have been expanded upon by adding 26 more classes for the letters of the alphabet; if a model for handwritten letters and digits was found to be as accurate as the MNIST model, it could be implemented into applications such as document scanning, so that physical information can be scanned and stored non-locally as backups or duplicates. A model for handwritten letters would also have uses in translation; multi-network models such as Google

Translate (Brin 2006) allow the scanning of a document, such as a restaurant menu, from one language to another.

Chapter 4

A Convolutional Neural Network for Facial Recognition

4.1 Introduction

Facial recognition has many real-world applications including identification and security. Facial recognition has been used in PDAs since OMRON released the 'OKAO Vision Face Recognition Sensor' in 2005 (Phys.org 2005). Google (2012) released their first facial recognition software 7 years later on the Android 4.0 update, but it wasn't until the impressive benchmark of the iPhone X that facial recognition became the choice method of security for mobile phones (Apple 2017). There are also identification uses of facial recognition; Clearview AI is used by law enforcement to identify missing persons and wanted criminals using surveilance footage (Ton-That and Schwartz 2017).

Chapter 3 introduced Tensorflow-Keras and created a convolutional neural network to classify handwritten digits successfully. In this Chapter the MNIST model is used as a foundation to create a new model which identifies an input as one of 16 different faces. First, the database of faces is introduced. Then the methodology is described, including the conversion from the MNIST model to the new 'Faces' model. The results of the facial recognition model are then given, followed by a discussion on the model's strengths and limitations.

4.1.1 Sourcing the database

A large database wasn't desirable due to storage limitations and resource requirements. For this reason a smaller database was found with 227 training images and 64 testing images (Hashmi 2021). Each image is square but varies in size, from 300-350 pixels. The main challenge of using a small database is to secure a good fit of the data. Underfitting was possible if there weren't enough samples to learn features from, and overfitting could have occurred if there wasn't enough variation within the samples. Whilst permission for every image was not given by the individuals photographed, it's taken in good faith that the participants have been made aware of the open-source nature of the images before they agreed to participate in the original study, which was also on facial recognition. Some entries of the 'Faces' database is shown in Figure 4.1.

Figure 4.1: Some images from the 'Faces' database, which contains images of 16 human faces. Each image attempts to capture the face in different conditions, such as angle, lighting, resolution and emotion, so that the model can recognise the faces in different conditions.

The application of transfer learning

The MNIST neural network in Chapter 3 was used as a template for the facial recognition model in this chapter to demonstrate the validity of transfer learning in creating artificial neural

networks.

4.2 Methodology

In this methodology the methods used to create the 'Faces' model are given. By implementing the same architecture and hyperparameters from the previous chapter an initial model was established. The initial model was changed to better fit the new problem - hyperparameter values were changed, a dense layer was removed and two pooling layers were added. The following sections describe the model chronologically, from the input to the output layers. The basic architecture of the facial recognition model is given in Figure 4.2.

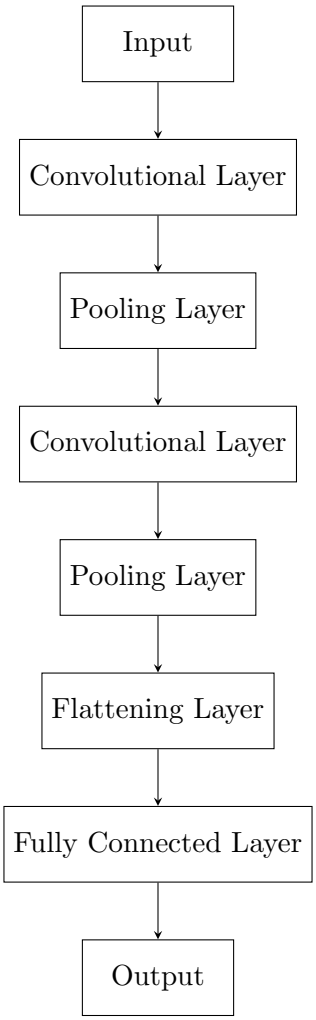


Figure 4.2: Flow chart describing the basic architecture of the facial recognition neural network.

4.2.1 Data pre-processing for the input layer

Data pre-processing was an essential step due to the small size of the 'Faces' database. The training data was augmented by randomly applying shear, zoom and flip to increase variation in the training data. Figure 4.3 shows the code used to pre-process the training and testing data.

```
# Defining pre-processing transformations on raw images of training data
train_datagen = ImageDataGenerator(
                shear_range=0.1,
                zoom_range=0.1,
                horizontal_flip=True)
# Defining pre-processing transformations on raw images of validation data
test_datagen = ImageDataGenerator()
```

Figure 4.3: The Python 3.10.12 code for the *ImageDataGenerator* (IDG) pre-processing tool from Keras (Chollet 2015). IDG added variation by shearing, zooming and flipping the images at random. This process helped prevent overfitting by creating more complex features within the data.

Normalisation

The normalisation methods that the MNIST model employed were removed for the 'Faces' model after experimentation showed that model accuracy improved without it. Normalisation can have adverse effects if the data becomes 'warped' and the model learns warping as a feature of the training data.

4.2.2 The convolutional and pooling layers

The convolutional layers from the MNIST model contained too many parameters for the 'Faces' database. Heavy feature extraction was leading to overfitting, so the number of filters in the first convolutional layer was halved. As well as this, a pooling layer was added after each convolutional layer (two in total) which improved run time without a significant decrease in accuracy. The model performance improved significantly from these changes. The stride length remained at $(1, 1)$ from the change in Chapter 3, so that the features of the input were adequately captured. Figure 4.4 shows the Python code for these layers. The process of convolution and pooling on an input is shown in Figure 4.5

```
##### Convolutional layers #####
layer = tf.keras.layers.Conv2D(filters=32, kernel_size=(5, 5), strides=(1, 1), activation=tf.nn.relu)(inputs)
layer = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(layer)

layer = tf.keras.layers.Conv2D(filters=64, kernel_size=(5, 5), strides=(1, 1), activation=tf.nn.relu)(layer)
layer = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(layer)
##### --------------------- #####
```

Figure 4.4: Python 3.10.12 code for creating the convolutional and max pooling layers of the 'Faces' model. The filters were reduced to 32 in the first layer, and a pooling layer was added after both convolutional layers.

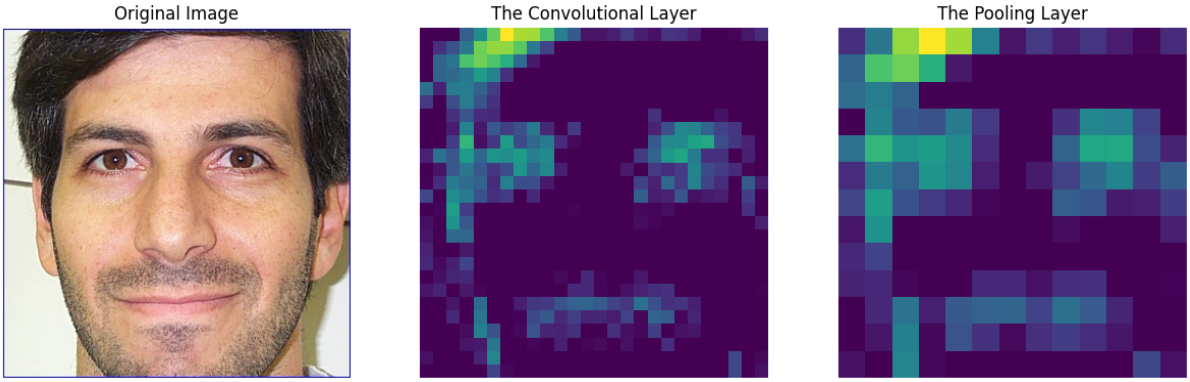


Figure 4.5: One convolution and pooling output from the convolutional layers. There are 32 of these outputs in the first layer and 64 in the second, each created with a different edge-detecting kernel.

4.2.3 The Flattening and Dense Layer

The pooled output in Figure 4.5 was flattened by the flattening layer, which was identical to the MNIST model's flattening layer (Section 3.2.4). The output of the flattening layer was a $10,816 \times 1$ column vector representing the extracted features of each input image. This was entered into a single fully connected layer containing 64 neurons. Here the model learned the features of the 16 faces. The code for these layers is shown in Figure 4.6 and the 'Faces' neural network diagram is shown in Figure 4.7.

```
##### Flattening layer #####
layer = tf.keras.layers.Flatten()(layer)
##### ---------------- #####

##### Fully connected layer #####
layer = tf.keras.layers.Dense(units=64, activation=tf.nn.relu)(layer)
##### ----------------------- #####

##### Output layer #####
outputs = tf.keras.layers.Dense(units=OutputNeurons, activation=tf.nn.softmax)(layer)
##### ------------ #####
```

Figure 4.6: The Python 3.10.12 code for creating the Flattening and Dense layers of the 'Faces' neural network. The flattening layer contains 10,816 elements that are inputs for the fully connected layer. The fully connected layer contains 64 neurons and 692,288 parameters. This layer is the main point where learning takes place in the network. There are 64 inputs to the output layer, which has 16 neurons or 'classes' and 1040 parameters.

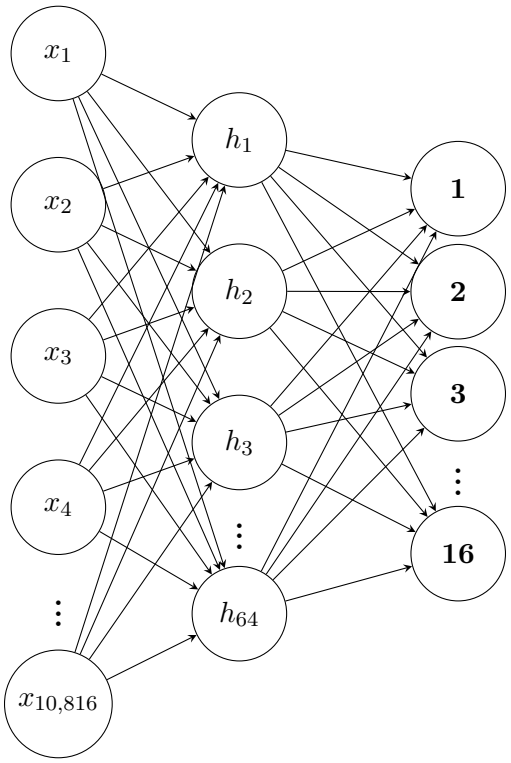


Figure 4.7: The fully connected or 'dense' layers of the 'Faces' neural network. The input $\boldsymbol{x}$ is the 1024×1 column vector resulting from the flattening layer. There is one hidden layer, $\boldsymbol{h}$, with 64 neurons. The output is a 16×1 column vector containing the probabilities that the input belongs to each class. The output neuron with the highest probability is the prediction of the model.

4.2.4 Compiling the Model

The model was compiled using the layers described and the same compiling methods given in Chapter 3. The number of epochs and steps per epoch were changed to fit the new database size. The code for the model compilation is given in Figure 4.8.

```
# Compile the CNN
model = tf.keras.Model(inputs, outputs)  # Initialize the CNN
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()

StartTime = time.time()

model.fit(
    training_set,
    steps_per_epoch=7,  # Adjusted based on the number of batches per epoch
    epochs=10,
    validation_data=test_set,
    validation_steps=len(test_set)
)
```

Figure 4.8: The Python 3.10.12 code to compile and fit the model to the training data. The same optimiser and loss function were used as in Chapter 3.

4.3 Results

A facial recognition model was made using a convolutional neural network. The architecture for the 'Faces' neural network is given in Table 4.1, where the network's layers are given chronologically. The final model contained 8 layers and 747,024 parameters, which required 2.85MB of memory.

Table 4.1: Summary of the 'Faces' neural network architecture. 'Output shape' describes the output dimensions for each layer. the input layer outputs three 64×64 images from the three colour channels, red, blue and green (RGB).

Layer	**Output shape**	**No. parameters**
Input Layer	(64, 64, 3)	0
Convolutional Layer 1	(60, 60, 32)	2,432
Max Pooling Layer 1	(30, 30, 32)	0
Convolutional Layer 2	(26, 26, 64)	51,264
Max Pooling Layer 2	(13, 13, 64)	0
Flattening Layer	(10,816)	0
Fully Connected/Dense Layer	(64)	692,288
Output Layer	(16)	1,040
Total no. parameters		**747,024 (2.85 MB)**

The model performed well on testing data, consistently recording testing accuracy's of over 90%. Figure 4.9 shows six predictions made using the 'Faces' model, five of which were successful. The model successfully classified each of the 16 faces that were learned by the model at least twice; Table 4.2 shows the model results on the unseen images of the 16 faces, where a particularly successful run of the final model yielded 99.59% accuracy on the training data.

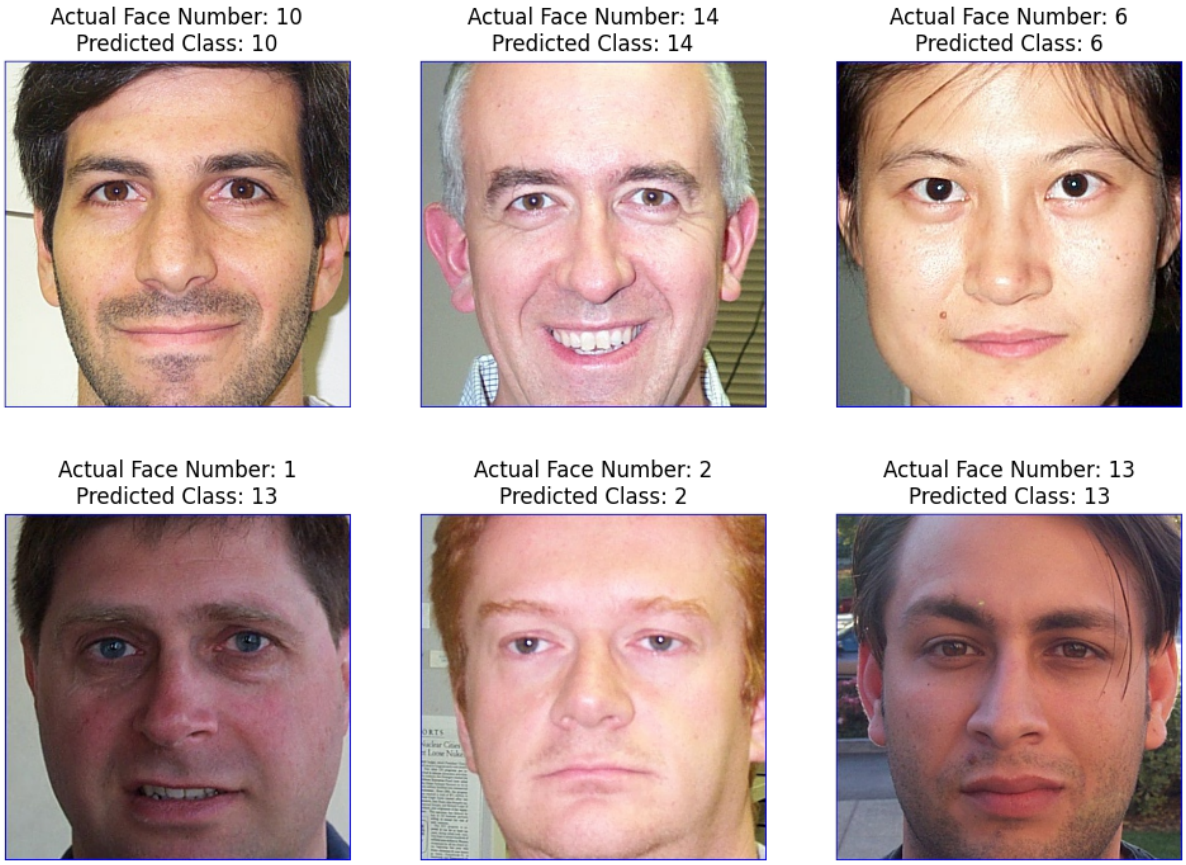


Figure 4.9: The model's predictions for 6 of the 64 testing images. All of the faces are correctly classified except 'Face 1' which was incorrectly classified as 'Face 13'.

Table 4.2: The final model's predictions on the testing data. Ones and zeros represent correct and incorrect classifications, respectively. These predictions yielded a testing accuracy of 93.75%.

Actual Class	Image 1	Image 2	Image 3	Image 4
Face 1	1	1	0	0
Face 2	1	1	1	1
Face 3	1	1	1	1
Face 4	1	1	1	1
Face 5	1	1	1	1
Face 6	1	1	1	1
Face 7	1	1	0	1
Face 8	1	1	0	1
Face 9	1	1	1	1
Face 10	1	1	1	1
Face 11	1	1	1	1
Face 12	1	1	1	1
Face 13	1	1	1	1
Face 14	1	1	1	1
Face 15	1	1	1	1
Face 16	1	1	1	1

Variation in accuracy

The accuracy of the model varied somewhat for each run due to the random initialisation of parameters during the fitting process. Due to this some runs of the model were less successful than anticipated. The accuracy of each iteration is given in Table 4.3. After removing anomalies (runs 6 & 20) the average model accuracy was improved to 94.89% and variance improved significantly to $\sigma^2 = 20.45$. This gives the model a benchmark accuracy of $(94.89 \pm 4.52)\%$, given by the mean and standard deviation of the observed results in Table 4.3.

Table 4.3: The model accuracy from fitting the training data over 20 runs. Anomalies are shown in bold, where accuracy is less than 80%.

Run No.	**Accuracy (%)**
1	93.03
2	98.36
3	89.57
4	97.54
5	96.72
6	**39.13**
7	95.90
8	98.77
9	91.80
10	95.08
11	98.77
12	97.54
13	81.56
14	95.08
15	97.13
16	88.93
17	99.59
18	95.49
19	97.13
20	**54.28**
Average	90.07
Variance	244.28

4.4 Discussion

In this Chapter, a convolutional neural network was made to recognise faces given a small number of images to learn from. Transfer learning was also successfully demonstrated by converting the MNIST convolutional neural network of Chapter 3 into a facial recognition (FR) model. Pooling layers were added to reduce dimensionality and a Dense layer was removed to reduce model complexity. Data augmentation techniques were then implemented to add variation to

the otherwise biased data set. As the model retrieves the input images from a directory, it's very adaptable - other faces can be learned and classified, if enough images are available for training, validation and testing. The memory requirement for the model's parameters, 2.85MB, is sufficiently small; ResNet, a 50 layer neural network, requires 168MB to store it's 26 million weight parameters (*GraphCore* 2017).

One challenge was deciding the model's complexity. The model had to be adequately complex to fit the small training database (see Section 2.1.3). The model was sensitive to any changes in complexity, likely due to the small size of the 'Faces' database. Surprisingly, the final FR model contains less parameters than the MNIST model, despite the more complex input of the FR model. The reduction in parameters is largely because of the reduction in dimensionality caused by the pooling layers.

Another challenge in creating the network was the variation in accuracy across runs. The final model had an observed standard deviation of $\sigma = 15.63$, which is significant. Despite setting a seed, many processes within the Keras framework are stochastic (Chollet 2015), so poor initial parameter values leads to accuracy divergence during the fitting process. This lead to the conclusion that the model has a poor parameter stability region.

A particular success of the model is it's ability to perform adequately with a very small database. The traditional approach towards facial recognition is to provide as much training data as possible; Clearview contains 20 billion images of faces and is among the most accurate FR models available (Ton-That and Schwartz 2017). The FR model was able to consistently reach $> 90\%$ accuracy by utilising data augmentation techniques rather than a larger database, which greatly reduced resource requirement whilst sacrificing minimal accuracy. It's possible that by increasing the amount of training data and retaining data augmentation techniques the FR model could consistently reach industry applicable accuracy's.

One limitation of facial recognition with CNNs is the inability to distinguish between authentic and imitative images. Previous to Apple's iPhone X (2017), it was possible to breach FR security using a printed image of the owner's face. The iPhone X introduced FaceID, which is a highly accurate CNN and the new 'TrueDepth' technology, which is a second layer of security that identifies an individual with depth-capturing infrared. FaceID was the first three dimension facial recognition software and has since set the standard for facial recognition security software's. Another limitation of the model is it's inability to handle multiple faces at once. If

two of the individuals were photographed together, the model would attempt to see one single face, which would likely provide a nonsense output. By allowing for multiple outputs, a group photograph could be used as an input where every learned individual in the image is recognised.

Chapter 5

Conclusion

Chapter 2 provided an insight into the Mathematical operations that lie the foundations for modern artificial intelligence (AI) systems. Common types of problems that can be solved with machine learning (ML) were described, and the importance of model complexity was highlighted. Essential algorithms such as gradient descent were explained and visualised. Machine learning approaches were discussed, giving particular focus to supervised learning, which is the focus of the later chapters. Hyperparameters were then described, including how validation is used to tune them.

Chapter 2 then gave detail on artificial neural networks (ANNs). Deep learning was defined, and a basic feedforward neural network was visualised. This was followed by a representation of the McColloch-Pitts neuron (McCulloch and Pitts 1943) and Rosenblatt's final model (Rosenblatt, 1962). The activation functions that govern each neuron were discussed, including sigmoid, ReLU and the discrete softmax function. L^1 and L^2 norm regularisation techniques were explained as well as dropout neuron regularisation. Following a brief note on the importance of optimisation, backpropagation was visualised in detail; the process of backward flow in aim of minimising a cost function was shown. The optimised version of gradient descent, the stochastic gradient descent algorithm, was explained before finally discussing important performance metrics including accuracy, loss and coverage.

In the final sections of Chapter 2, the important neural network types were described including recurrent neural networks and generative adversarial networks, followed by an in-depth demonstration of each layer of a convolutional neural network (CNN). The convolution operation was

demonstrated and each layer was described including visuals, showing the flow of information through the network. To conclude Chapter 2, a particular example was given in the dense layers of a neural network. Network activations were shown and one of the activations was calculated manually to understand the origin of each value in the network.

Chapter 3 contained the first of two experiments that aimed to use CNNs for practical applications. The MNIST database which contains 70,000 handwritten digits (Lecun 1998) was used in attempt to create a model on Python 3.9 that recognises handwritten digits. The methodology described the layers of the model, detailing hyperparameter values and specifics of the MNIST models' architecture. The results of the model were impressive, featuring a tested accuracy of 98.85%. The model's high accuracy cost a comparably inefficient training time, however given that the aims for this chapter were to create a highly accurate model and not necessarily an efficient one, the results were considered successful.

To further improve the MNIST model of Chapter 3, high-performance computing could have been used to run a more complex model whilst maintaining an acceptable efficiency. Then, by incorporating the MNIST CNN into a multi-network system of convolutional and recurrent neural networks, a document scanner of handwritten digits could be created that successfully identifies and orders digits as they are ordered on the page. The number of output classes could easily be increased to add detection for letters of various alphabets. The discussion of Chapter 3 mentions how Google Translate (Brin 2006) uses a similar system so that restaurant menus and public information can be translated using a smart phone camera. By creating a system of networks that identify each letter in the image and order them successfully, a real-time translator could be created.

Chapter 4 features the second experiment using CNNs for real-world application. A facial recognition model was created on Python 3.9 using a CNN and a data base of 16 faces, each captured between 10 and 20 times. One particular aim of this experiment was to explore the validity of transfer learning in neural networks. By using the MNIST model from Chapter 3 as a template for the facial recognition model, it was made clear that adapting an old model was much simpler than to start from scratch. Next, the new model architecture was changed to extract the more complex features of the new data set. The final model accuracy had some variance likely due to the smaller size of the training data; parameters which were initialised randomly had less training data to converge to. A particularly successful run shown in Table

4.2 featured a testing accuracy of 93.75%. The model was considered partially successful, as training accuracies were consistently above 90%, however it is a considerably lower accuracy than those recorded by the MNIST model. This demonstrates the problem complexity of facial recognition compared to handwritten digit recognition.

If another facial recognition model was created, another data base would have been chosen with more images of each class. The BioID data base (BioID 1998) is a facial recognition benchmark, similarly to MNIST, which contains 1,521 images that are optimised for training a facial recognition model on. Additionally, having no benchmark to compare Chapter 4's model on is a major flaw of using a niche data base. As well as utilising an alternative data base, by utilising local software and high performance computing results may have been improved. Google Colab (Google 2017) is a cloud-based Python provider and was sufficient for the smaller data set. However, for more complex image recognition tasks, Colab may hinder computational efficiency where a local Python installation on a high-performance system would not.

As well as these changes, with more time a multi-step verification process could have been incorporated which would check a result by augmenting the input and confirming that the output is the same. If the results were different, further runs would be done until successive runs agree. This would mitigate many of the errors in a facial recognition model; by 'double-checking' a result of a 95% accurate model, the error would theoretically halve, giving an immediate 2.5% increase to model accuracy. The negative aspect of this process is that computation time doubles for each verification layer.

Despite the lengthy computation time of the MNIST neural network and the variation in accuracy of the facial recognition neural network, the aim of creating neural networks to classify images was done successfully. As well as this, transfer learning was successfully demonstrated by converting a model which classifies handwritten digits to one that recognises one face from another.

References

Agrawal, Tanay (2021). *Hyperparameter optimization in machine learning : make your machine learning and deep learning models more efficient*. Apress. URL: https://learning.oreilly.com/library/view/hyperparameter-optimization-in/9781484265796/.

AlphaZero (2018). URL: https://deepmind.google/technologies/alphazero-and-muzero/.

Apple (2017). URL: https://www.apple.com/uk/newsroom/2017/09/the-future-is-here-iphone-x/.

Assiri, Yahia (2020). "Stochastic Optimization of Plain Convolutional Neural Networks with Simple methods". In: URL: https://paperswithcode.com/paper/stochastic-optimization-of-plain.

Azunre, Paul (2021). *Transfer learning for natural language processing*. O'REILLY MEDIA. URL: https://learning.oreilly.com/library/view/transfer-learning-for/9781617297267/?sso_link=yes&sso_link_from=manchester-metropolitan-university.

Bhattacharyya, Souvik and Ghosh, Koushik (2022). *Noise filtering for Big Data Analytics*. De Gruyter.

BioID (1998). *BioID*. URL: https://www.bioid.com/face-database/.

Bojarski, Mariusz, Testa, Davide Del, Dworakowski, Daniel, Firner, Bernhard, Flepp, Beat, Goyal, Prasoon, Jackel, Lawrence D., Monfort, Mathew, Muller, Urs, Zhang, Jiakai, Zhang, Xin, Zhao, Jake, and Zieba, Karol (2016). "End to End Learning for Self-Driving Cars". In: *CoRR* abs/1604.07316. arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

Bouarara, Hadj Ahmed (2021). "Recurrent Neural Network (RNN) to Analyse Mental Behaviour in Social Media". In: *International Journal of Software Science and Computational Intelligence* 13.3. DOI: 10.4018/IJSSCI.2021070101.

Brin, Sergey (2006). *Google Translate*. URL: http://translate.google.com..

Casella, George and Berger, Roger L. (1990). *Statistical Inference*. Duxbury Press.

Cauchy, Augustine M. (1847). *Methode générale pour la résolution des systémes d'equations simultanées*. URL: https://cs.uwaterloo.ca/~y328yu/classics/cauchy-en.pdf.

Ceschini, Andrea, Rosato, Antonello, and Panella, Massimo (2022). "Hybrid Quantum-Classical Recurrent Neural Networks for Time Series Prediction". In: *International Joint Conference on Neural Networks*.

Chen, Xu, Chen, Songqiang, Xu, Tian, Yin, Bangguo, Peng, Jian, Mei, Xiaoming, and Li, Haifeng (2021). "SMAPGAN: Generative Adversarial Network-based semisupervised styled Map Tile Generation Method". In: *IEEE Transactions on Geoscience and Remote Sensing* 59.5, pp. 4388–4406. DOI: 10.1109/tgrs.2020.3021819.

Chollet, Francois (2015). *Keras*. URL: https://github.com/fchollet/keras.

Dadhich, Shruti, Pathak, Vibhakar, Mittal, Rohit, and Doshi, Ruchi (2021). "Chapter 10: Machine Learning for Weather Forecasting". In: *Machine Learning for Sustainable Development*. de Gruyter.

El Naqa, Issam and Murphy, Martin J. (2015). "What Is Machine Learning?" In: *Machine Learning in Radiation Oncology: Theory and Applications*. Ed. by Issam El Naqa, Ruijiang Li, and Martin J. Murphy. Cham: Springer International Publishing, pp. 3–11. ISBN: 978-3-319-18305-3. DOI: 10.1007/978-3-319-18305-3_1. URL: https://doi.org/10.1007/978-3-319-18305-3_1.

Fisher, R. A. (1936). "THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS". In: *Annals of Eugenics* 7.2, pp. 179–188. DOI: https://doi-org.mmu.idm.oclc.org/10.1111/j.1469-1809.1936.tb02137.x. eprint: https://onlinelibrary-wiley-com.mmu.idm.oclc.org/doi/pdf/10.1111/j.1469-1809.1936.tb02137.x. URL:

https://onlinelibrary-wiley-com.mmu.idm.oclc.org/doi/abs/10.1111/j.1469-1809.1936.tb02137.x.

Fortman-Roe, Scott (2012). [Accessed 20/02/2024]. URL: https://scott.fortmann-roe.com/docs/BiasVariance.html.

Gilliland, Michael, Tashman, Len, and Sglavo, Udo (2021). *Business forecasting: The emerging role of Artificial Intelligence and machine learning*. Wiley. URL: https://ebookcentral.proquest.com/lib/mmu/reader.action?docID=6579254.

Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron (2016). *Deep learning*. The MIT Press.

Goodfellow, Ian, Pouget-Abadie, Jean, Mirza, Mehdi, and Xu, Bing (2020). "Generative Adversarial Networks". In: *Communications of the ACM*, pp. 139–144.

Google (2012). URL: https://www.slashgear.com/motorola-atrix-2-receives-android-4-0-ics-update-09251027.

– (2017). *Colab*. URL: https://colab.google/.

GraphCore (2017). URL: https://www.graphcore.ai/performance-results.

Hannun, Awni Y., Case, Carl, Casper, Jared, Catanzaro, Bryan, Diamos, Greg, Elsen, Erich, Prenger, Ryan, Satheesh, Sanjeev, Sengupta, Shubho, Coates, Adam, and Ng, Andrew Y. (2014). "Deep Speech: Scaling up end-to-end speech recognition". In: *CoRR* abs/1412.5567. arXiv: 1412.5567. URL: http://arxiv.org/abs/1412.5567.

Hashmi, Farukh (2021). *Face Recognition using Deep Learning CNN in Python*. URL: https://thinkingneuron.com/face-recognition-using-deep-learning-cnn-in-python/.

Hopfield, J. J (1982). "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the National Academy of Sciences of the United States of America* 79.8.

IBM (1997). *Deep Blue*.

Karras, Tero, Aila, Timo, Laine, Samuli, and Lehtinen, Jaakko (2017). "Progressive Growing of GANs for Improved Quality, Stability, and Variation". In: *CoRR* abs/1710.10196. arXiv: 1710.10196. URL: http://arxiv.org/abs/1710.10196.

Kotu, Vijay and Deshpande, Balachandre (2019). *Data Science: Concepts and Practice*. Morgan Kaufmann Publishers.

Lanham, Micheal (2019). *Hands-on Deep Learning for games leverage the power of neural networks and reinforcement learning to build intelligent games*. Packt. URL: https://learning.oreilly.com/library/view/hands-on-deep-learning/9781788994071.

Lecun, Yann (1998). *MNIST dataset*. URL: https://keras.io/api/datasets/mnist/.

Lee, Jootaek (2020). "Artificial Intelligence and Human Rights: Four Realms of Discussion: Summary of Remarks". In: *American Society of International Law* Proceedings of the Annual Meeting.v114, pp. 242–245.

Lee, Wei-Meng (2019). *Python machine learning*. Wiley.

Lin, Junyang, Men, Rui, Yang, An, Zhou, Chang, Ding, Ming, Zhang, Yichang, Wang, Peng, Wang, Ang, Jiang, Le, Jia, Xianyan, Zhang, Jie, Zhang, Jianwei, Zou, Xu, Li, Zhikang, Deng, Xiaodong, Liu, Jie, Xue, Jinbao, Zhou, Huiling, Ma, Jianxin, Yu, Jin, Li, Yong, Lin, Wei, Zhou, Jingren, Tang, Jie†, and Yang, Hongxia (2021). *M6: A Chinese Multimodal Pretrainer*. URL: https://arxiv.org/pdf/2103.00823.pdf.

Lynch, Stephen (2023). *Python for Scientific Computing and Artificial Intelligence*. CIC Press. ISBN: 978-1-003-28581-6.

Malik, Alok and Tuckfield, Bradford (2019). *Applied unsupervised learning with R*. Packt Publishing Ltd.

MathWorks (n.d.). *MATLAB*. Version R2022b. URL: https://uk.mathworks.com/products/matlab.html.

McCulloch, Warren and Pitts, Walter (Feb. 1943). "A Logical Calculus of the Ideas Immanent in Nervous Activity (1943)". In: *Ideas That Created the Future: Classic Papers of Computer Science*. The MIT Press. ISBN: 9780262363174. DOI: 10.7551/mitpress/12274.003.

0011. eprint: https://direct.mit.edu/book/chapter-pdf/2248320/9780262363174_c000800.pdf. URL: https://doi.org/10.7551/mitpress/12274.003.0011.

McNeill, Fiona and Pinheiro Andre Reis, Carlos (2014). *Heuristics in analytics: A practical perspective of what influences our Analytical World*. Wiley.

Murphy, Kevin P. (2007). *Frequentist Parameter Estimation*.

Nwanganga, Fred and Chapple, Mike (2020). *Practical machine learning in R*. John Wiley & Sons.

Phys.org (2005). URL: https://phys.org/news/2005-03-world-recognition-biometric-mobile.html.

R Core Team (n.d.). *R: A Language and Environment for Statistical Computing*. Version 4.3.1. R Foundation for Statistical Computing. URL: https://www.R-project.org/.

Rosenblatt, F (1958). "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review*. URL: https://doi.org/10.1037/h0042519.

Rossum, Guido van (n.d.). *Python*. Version 3.9. URL: http://www.python.org.

Sanderson, Grant (2017). *Neural Networks*. URL: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

– (2022). *But what is a convolution?* URL: https://www.youtube.com/watch?v=KuXjwB4LzSA.

Schnapf, Julie L. and Baylor, Denis A. (1987). "How Photoreceptor Cells Respond to Light". In: *Scientific American* 256.4, pp. 40–47. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/24979361 (visited on 05/10/2024).

Singh, Pramod (2021). *Deploy machine learning models to production: With flask, streamlit, Docker, and kubernetes on google cloud platform*. Appress L. P.

Starmer, Josh (2018a). *Machine Learning Fundamentals: Cross Validation*. URL: https://www.youtube.com/watch?v=fSytzGwwBVw.

Starmer, Josh (2018b). *Regularization Part 1: Lasso (L1) Regression*. URL: https://www.youtube.com/watch?v=NGf0voTMlcs.

– (2018c). *Regularization Part 1: Ridge (L2) Regression*. URL: https://www.youtube.com/watch?v=Q81RR3yKn30&t=255s.

Such, Felipe Petroski, Rawal, Aditya, Lehman, Joel, Stanley, Kenneth, and Clune, Jeffrey (2020). "Generative Teaching Networks: Accelerating Neural Architecture Search by Learning to Generate Synthetic Training Data". In: Proceedings of Machine Learning Research 119. Ed. by Hal Daumé III and Aarti Singh, pp. 9206–9216. URL: https://proceedings.mlr.press/v119/such20a.html.

Ton-That, Hoan and Schwartz, Richard (2017). *Clearview AI*.

Turing, Alan (2004). *The essential Turing: seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life, plus the secrets of Enigma*. Ed. by Jack Copeland. Oxford University Press.

Wadhwa, Deepanshu, Maharana, Utkarsh, Shah, Devina, Yadav, Vaibhav, and Pandey, Prashant (2019). "Human sketch recognition using generative adversarial networks and one-shot learning". In: *2019 Twelfth International Conference on Contemporary Computing (IC3)*. IEEE. DOI: 10.1109/ic3.2019.8844885.

Winder, Phil (2020). *Reinforcement Learning*. URL: https://learning.oreilly.com/library/view/reinforcement-learning/9781492072386.