University of Cape Town

Deep Learning for Supernovae Detection

Deep Learning for Supernovae Detection

Author:
Gilad Amar

Supervisor:
Dr. Bruce Bassett

A Minor Dissertation submitted in partial fulfilment of the requirements for the degree of a Master of Science

in the

Cosmology Group at AIMS
Department of Mathematics and Applied Mathematics

August 2017

Declaration of Authorship

I, Gilad Amar, declare that this thesis titled, 'Deep Learning for Supernovae Detection' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest."

Isaac Asimov

UNIVERSITY OF CAPE TOWN

Abstract

Faculty of Science
Department of Mathematics and Applied Mathematics

Master of Science

Deep Learning for Supernovae Detection

by Gilad Amar

In future astronomical sky surveys it will be humanly impossible to classify the tens of thousands of candidate transients detected per night. This thesis explores the potential of using state-of-the-art machine learning algorithms to handle this burden more accurately and quickly than trained astronomers. To this end Deep Learning methods are applied to classify transients using real-world data from the Sloan Digital Sky Survey. Using cutting-edge training techniques several Convolutional Neural networks are trained and hyper-parameters tuned to outperform previous approaches and find that human labelling errors are the primary obstacle to further improvement. The tuning and optimisation of the deep models took in excess of 700 hours on a 4-Titan X GPU cluster.

Acknowledgements

This thesis would not have happened if not for my introduction to machine learning by my supervisor Prof. Bruce Bassett. Thank you for welcoming me to the AIMS Cosmology group. Your knowledge, enthusiasm and advice were invaluable when wading into unknown territory.

To Dr Arun Kumar, thank you for helping with all the technical details, your advice and casual discussions helped me retain what sanity I have remaining.

To Kai Staats, thank you for bouncing around ideas with me, the office humour and surfing lessons. It was a pleasure to learn more about machine learning with you.

To Lise du Buisson, this work builds on all the work you did in your Thesis. Thank you for providing me with a grand tour of the data, your research and Rocky Horror experiences.

To the team at the SKA, in particular Laura Richter and Nadeem Ozeer, thank you for including me in the SKA world and the access to computing resources. If not for your help my models would still be training.

To my family, thank you for all your support along the roller coaster that was my MSc and the start of my career. Without you we couldn't all be functionally dysfunctional. Rachel and Carin, thank you for the detailed spelling and grammar comments. It is from your efforts that my english-written thesis is actually in english.

To my girlfriend, Francesca, thank you for checking every single word and concept and pushing me to complete this thesis. Without you this thesis would never have left my laptop. I'm glad I could teach you about machine learning in the process.

Contents

List of Figures

List of Tables

Abbreviations

AGN	**A**ctive **G**alactic **N**uclei
AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
AUC	**A**rea **U**nder **C**urve
CNN	**C**onvolutional **N**eural **N**etwork
DL	**D**eep **L**earning
DNN	**D**eep **N**eural **N**etwork
FN	**F**alse **N**egatives
FP	**F**alse **P**ositive
FPR	**F**alse **P**ositive **R**ate
GPU	**G**raphics **P**rocessing **U**nit
LDSS	**L**ow **D**ispersion **S**urvey **S**pectrograph
LSST	**L**arge **S**ynoptic **S**urvey **T**elescope
ML	**M**achine **L**earning
MLP	**M**ulti-**L**ayer **P**erceptron
MSE	**M**ean-**S**quared **E**rror
RMS	**R**oot **M**ean **S**quared
ROC	**R**eciever **O**perating **C**haracteristic
SDSS	**S**loan **D**igital **S**ky **S**urvey
SGD	**S**tochastic **G**radient **D**escent
SN	**S**uper**n**ova
SNe	**S**uper**n**ova**e**
TN	**T**rue **N**egatives
TP	**T**rue **P**ositives
TPR	**T**rue **P**ositive **R**ate

Dedicated to our future machine overlords.

Chapter 1

Introduction

Future astronomical surveys will generate far too much data for astronomers to tackle as they have traditionally where it was the norm for astronomers to eyeball images of the sky, looking for interesting objects or transients. *Transients* are objects that appear for a short amount of time. These include, but are not limited to, asteroids, gamma ray bursts, supernovae and objects unknown to science altogether. When transients are discovered, spectroscopic follow up with other telescopes are necessary in order to accurately identify the object. Not all transients are created equal; some, like supernovae, are more valuable to science than others. Supernovae are rare gems for cosmologists which allow for learning about the history, curvature and content of the Universe.

Telescope technology has improved greatly in recent years; transients are discovered at an increasing rate. However, astronomers lack the resources to follow up on every one. The amount of data generated will be so large that the mere storage of it becomes a technological challenge, never mind making use of it. In addition, there is increased interest in studies that use data from telescopes sensitive to different wavelengths. Such *multi-wavelength* astronomy exacerbates the problem of handling and computing large amounts of data. To appreciate the extent of the problem, consider the upcoming Large Synoptic Survey Telescope (LSST) which will become scientifically active in 2022 [1]. Over its ten-year lifespan it is estimated to discover over 10 billion galaxies. Each day around 30 TB of raw data will be generated requiring processing and reduction [2]. The final archive will be around 60 PB, with the final object catalogue approximately 10-20 PB [3]. This challenge calls for a new approach to astronomy, one that is capable of dealing with Big Data. *Big Data* is a broad term for data-sets so large that traditional data processing and storing methods are inadequate. Many fields and industries now face this problem. Maintaining the status quo will be impossible; other tools must be explored.

As data challenges have been building, the computer science world has had a renewed interest in Machine Learning. *Machine Learning* (ML) is the study of computer programs able to 'learn' from data. For example, many pictures of a person can be shown to an algorithm such that it learns to identify him/her in another photo without the explicit instruction of a programmer. The reinvigorated interest in ML has largely been the result of an exponential growth in computing power. Newly discovered training methods are impractical to implement on older computers. Recently ML has shown great success in Artificial Intelligence (AI) and Big Data applications, from the popularly used cellphone assistant Siri [4] to driver-less Google cars [5]. Convolutional Neural Networks (CNNs), a ML method loosely inspired by the brain, have achieved near-human level performance in image recognition [6] tasks. This challenge, trivial for people, has evaded the capability of computers and computer programmers for decades. CNNs are an example of *Deep Learning* (DL) algorithms. They are 'deep' in the sense that they use a multi-layered architecture to learn. DL has had incredible success in other domains too, from voice recognition to brain-machine interfaces and medical diagnoses, thus illustrating its promise for many applications.

ML is applied in many arenas but is still fairly new to astronomy. The aim of this study is to establish whether or not ML has a positive role to play in the future of astronomy. To answer this problem there are several research objectives to lead us toward our aim. First, we must come to understand ML fundamentals. In particular, we must learn what DL is and how to successfully apply it. Second, we will need to identify a problem we can use as a test-case of DL's effectiveness in astronomy. For this we will apply DL to transient classification, using data from the Sloan Digital Sky Survey (SDSS), and observe the accuracy and efficiency achieved on a real-world astronomy dataset. As CNNs have proven successful at image recognition, they are the ML method of choice for this application. This will act as a useful playground for exploring the role ML has for the future of astronomy. The final research objective is the evaluation of ML's future potential in astronomy.

The primary interest is to ascertain the extent to which ML is effective. To be a viable solution, ML must be both accurate and efficient. Classification accuracy needs to be competitive with trained astronomers.

As supernovae are quite rare, occurring about once in a hundred years per galaxy, ML must also be very stringent as to what constitutes a supernova. If not, then the pool of objects classified as supernovae will be contaminated with countless false positives and imaging artifacts that will waste resources on unnecessary and expensive follow ups. The ML algorithm must be able to operate in real-time with a large data influx in order to be efficient. Given its superiority in other computer vision tasks, DL promises to

outperform more classical ML approaches [7]. We will investigate the validity of this claim.

This thesis has been structured in the following way: Chapter 2 will use neural networks as a drawing board for understanding the inner workings of ML methods and their successful application. We will need to know the subtleties of what is involved in ML and how to evaluate our results. In addition the inner workings of DL methods will be explained. Chapter 3 will layout the nature of the test case. We will detail what the SDSS sky survey entails; what data are collected and their structure. Next, the current status of ML transient classification will be investigated to see what others have discovered in their research. Here we will tackle the question of how to best achieve the research objectives in a logically sound way and will provide an analysis and discussion of the results. Finally we conclude with closing remarks and future recommendations in Chapter 4.

Chapter 2

Literature Review

2.1 Machine Learning

2.1.1 What is Machine Learning

Computers often cannot solve problems that people find trivial, despite performing significantly better than humans on many tasks. Where we see objects with different depths and shapes, a computer 'sees' only pixels - intensity values of red, green and blue dots on a two dimensional grid. It is not clear how to program a computer to recognize all the different pixel intensity configurations that correspond to a face in the real world. Faces will vary in size and structure as well as hair, eye, and skin colours. In fact the same face could be in different locations in 'pixel space' if obscured, in different orientations or lighting conditions. Machine Learning (ML) is what allows the computer to teach itself how to recognize faces when the required explicit instructions for facial recognition are unknown. This complex task will require a mapping from input data (the facial image pixels) to output classifications, shown in Figure 2.1. An example of ML is when the mapping is not designed by a programmer but is instead 'learned' from example data; this is a ML application. To this end many different algorithms have been designed that develop complex models of input data. These models may be used to find clusters, make predictions and even generate new data. Each of these algorithms try to find the best possible model from the complete hypothesis space of models it can create. This process of searching for model parameters that best encapsulates the data is known as *training*.

ML comes in three main categories; *Supervised*, *Unsupervised* and *Reinforcement Learning*. In Supervised Learning the algorithm is provided with many examples of input and the desired output for training. To be more precise, given a set of data

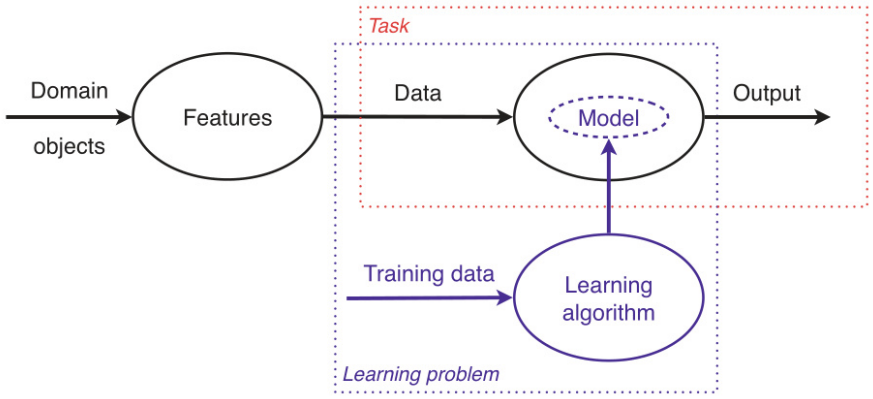


FIGURE 2.1: ML is used to perform complex tasks. A task will require a mapping - a model - of inputs to outputs. The creation of the mapping by using sample data the model will encounter is a ML problem [8].

$D = ((\vec{x}^{\,n}, y^n), n = 1, \ldots, N)$, the algorithm must learn the relationship between the input vector $\vec{x}$ and the output y. $\vec{x}$ is a vector of m values, $\vec{x} = (x_1, x_2, \ldots, x_m)$ [9]. A trained model can then take new input $\vec{x^\star}$ and predict the correct output $y^\star$. We measure the degree of predictive success with the *cost function* (also known as the loss function), $C(y_{\text{pred}}, y_{\text{true}})$, which takes the predictions and correct output as parameters. Training the model is done by minimizing the cost function with respect to the model parameters.

Supervised Learning itself has two sub-types - *Regression* and *Classification*. Regression problems are concerned with giving real-valued output as a function of the input [10], see Figure 2.2. There are many applications requiring numerical predictions on a continuum such as stock prices, temperatures and power generation.

Classification is predicting discrete classes having learned from examples. For example, a bank may wish to classify loan applicants into 'high-risk' and 'low-risk' groups. Unlike regression, the output y can only take on the discrete values of 'high-risk' or 'low-risk'. A model is built from the customers' historical data of incomes, savings, professions, ages and debt histories that can predict the risk class of a new applicant. These items of information about the applicant are known as *features*. If we look into the *feature space*, a space made with each feature as a spacial dimension, we can create a model that separates the two classes, see Figure 2.3. Features relevant to an applicant's financial capacity can provide many weak relations associated with their loan risk [12].

This resulting model can also be analysed to reveal the data's underlying structure. In the bank example, researchers can find common attributes of low-risk customers useful for target advertising for new and safe loans.

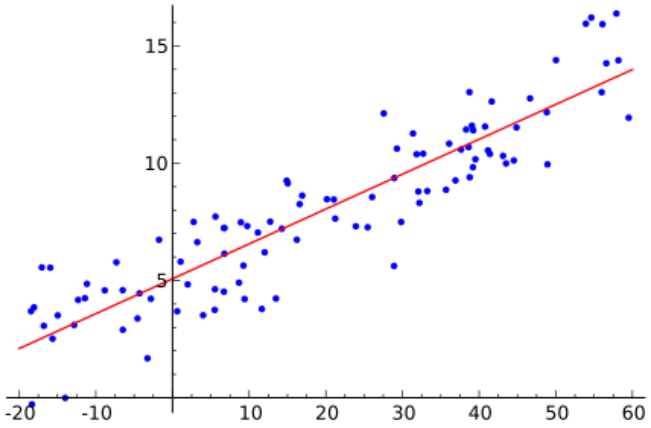


FIGURE 2.2: In Regression a continuous output as a function of the features is learned. Here the examples (blue dots) have an output shown on the $y - axis$ as a function of a single feature, corresponding to the x-axis. While the examples clearly have the unmistakable pattern of an ascending straight line, the regression model must not be confused by the noise. The fitted model (shown in red) can be used to make predictions, y_{pred}, for the output of a new example input, $x^{\star}$, that is close to the true output $y^{\star}$ [11]

.

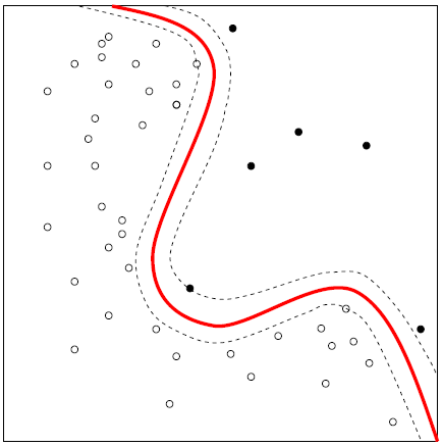


FIGURE 2.3: Two different classes (open and filled circles) fill the two dimensional feature space. Classification is the building of a model that can separate the classes such that for any new feature vector, $x^{\star}$, the correct class can be predicted. The model shown by the red line cuts across the feature space to differentiate the class of objects on one side versus the other. This algorithm used here tries to produce a boundary that is equally distant between the nearest examples, as shown by the broken lines to either side of the border. If a new object which lies to the top-right of the feature plane were discovered the model can be used to predict that it should be a filled circle [13].

A regression model can tackle classification problems by assigning a level of confidence for an object belonging to a class. This is often done by giving a prediction between zero and one. The continuous output is then discretized, often by thresholding, to get a discrete label. For example, output above 0.5 would be classified as a high-risk and below 0.5 as low-risk. However, multi-label problems cannot be answered with a single-valued output. A classic example used within ML texts [14] is that of optical character recognition; recognizing printed or written numeric characters from their scanned images. Such multi-label problems are treated by producing an output vector, $\vec{y}$, of ten values between zero and one. Should the second value in the vector be the largest this would mean the digit is classified as a two. The vector $\vec{y}_i = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$ would symbolise the label being a two as the one is in the second position. Similarly $\vec{y}_i = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$ represents a three. This method of numerically encoding discrete labels is known as *one-hot encoding*.

Ideally machines would learn about the world like babies in that they learn to distinguish between objects and people without being given explicit labels. ML could then be much more useful as there is far less labelled data than not. Learning models without data labels is known as Unsupervised Learning. Unsupervised algorithms attempt to find an accurate but compact description of the data [15]. Once built, the model can be used to find clusters, as in Figure 2.4, compress the data or generate more with the same structure.

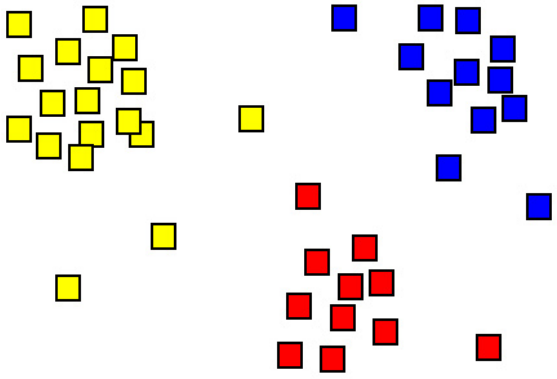


FIGURE 2.4: While data may not have labels, Unsupervised Learning may still be used to uncover some inherent structure. Here indistinguishable objects, shown as squares, are in the two dimensional feature space. A common Unsupervised Learning method is Clustering. Uncovering how many clusters there are within the data and where in the feature space they lie [16].

Semi-supervised learning is the common scenario in which there is a minority of labelled and a majority of unlabelled data. For example, there are many millions of images of trees on the internet, but few of the images will specify the species. Pure supervised learning would be forced to use only the labelled images and cast away the rest, losing

valuable information. Semi-supervised learning takes a more advanced approach and uses the unlabelled data to enhance the learning process. The unlabelled data are used to 'prime' the model, narrowing in on a range of useful parameter initializations before supervised learning begins. The *pre-training* makes the model come to recognize trees, even though it cannot yet distinguish species. It will come to expect trunks, branches, green leaves and bark. Consequently, the subtleties of variations that distinguish species can then be taught using the limited labelled data. The final model will likely be better than one trained on the labelled data alone as shown in Figure 2.5.

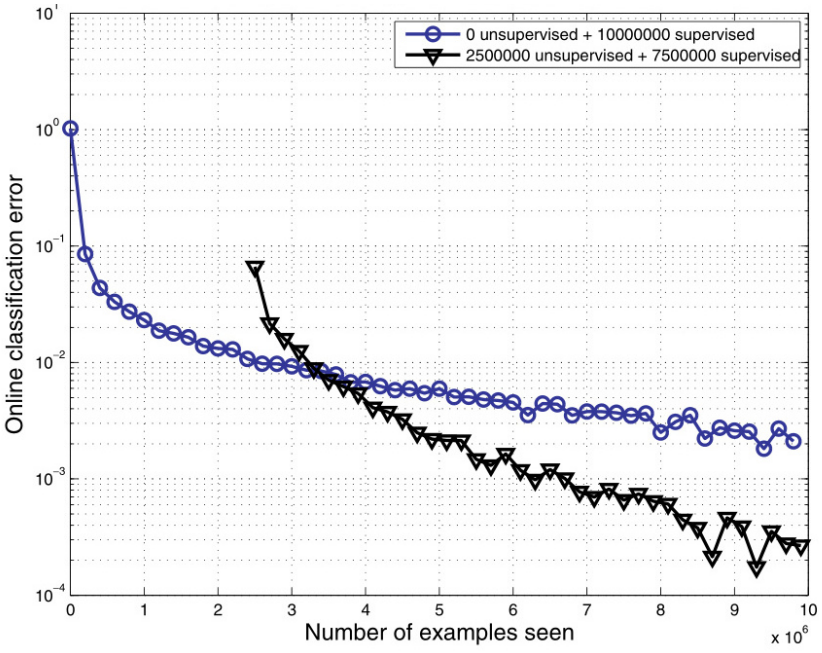


FIGURE 2.5: A semi-supervised model (shown in black) was shown 2.5 million examples without labels. These were used to instantiate the model parameters with better starting values than randomness would have produced. In this case, the semi-supervised model outperforms the supervised model (shown in blue), with parameters settled on sub-optimal values due to the worse initial parameter values. Analogously, it is much easier to teach someone to recognize different bird species (the labels) when they are accustomed to what birds generally look like. The oscillations to the far right of the semi-supervised model result from the error being so close to zero that sampling variations appear large on the log-scale [17].

Instead of predicting output or uncovering structure, Reinforcement Learning handles an agent in an environment, like a character in a game world. The objective is to maximize the agent's behaviour - its actions in response the environment - when a reward is received over time or only at the very end. Figure 2.6 shows an AI system that can learn to play Atari games without explicit instructions at all. Unlike supervised learning the agent is not informed of correct and incorrect behaviours but unlike unsupervised learning there is some feedback on performance. If the game character's reward is its

survival time it must learn life-span enhancing behaviour, such as eating, and avoiding dangerous actions, like drinking poison. [15]

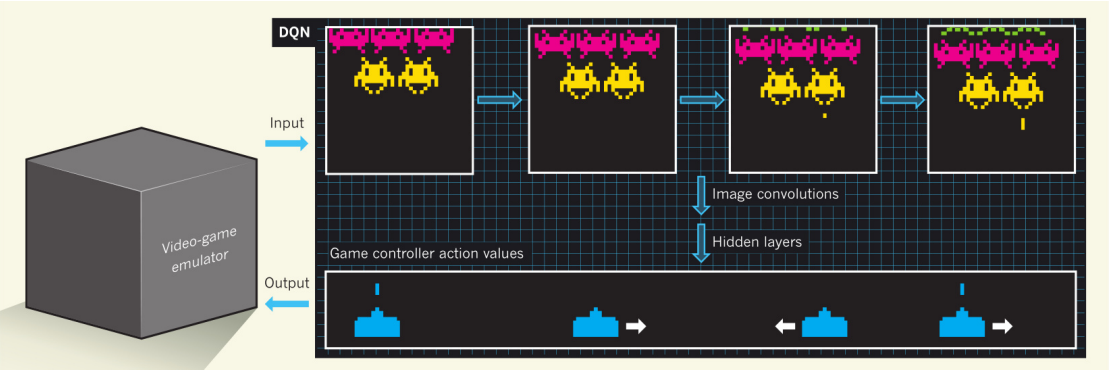


FIGURE 2.6: The 'Deep Q-Network' analyses four sequential video frames and predicts future game score for each possible action. This system is not even given instructions as to what objects to avoid, the characteristic elements of the game or how each control will impact the score. Instead information is extracted from the pixels through image convolutions and hidden layers (described in Section 2.2) and fed to the Reinforcement Learning algorithms which must learn to recognize game elements and their attributes and which actions in different game states most improve the score [18].

Sometimes new information becomes available over time. This means the model would have to be entirely retrained on the expanded data set, a time-consuming task. *Online Learning* is used to combat this problem. Predictions are made on each instance in a series and it will receive a reward or loss after each one [10]. It is not necessarily supervised or unsupervised as incoming data may or may not have labels. The model's objective is to maximise accumulated reward (or minimize the accumulated loss). Expressed formally: For each item in the sequence $s = 1, 2, 3, \ldots$, the algorithm:

1. Takes input $x_s \in X$
2. Makes prediction $y_s \in Y$
3. Receives a response $z_s \in Z$
4. Incurs a cost $c_s = c(y_s, z_t)$

Online learning is similar to reinforcement learning, however, the reward is instantaneous.

2.1.2 Features

In all ML applications features come in three types. *Binary Features* can take on only true or false values (numerically represented as one and zero). For instance, a feature

'being human' can only be true or false. A categorical feature, a person's career for instance, may be thought of as a series of binary features. The first binary variable may detail if the person is a doctor or not; the second for if they are a doctor etc. *Nominal Features* are multi-valued but discrete, such as a person's shoe size. *Real Features* can take on a continuous range of values such as someone's height.

Not all features are equally useful for a ML practitioner. Some features are irrelevant and only serve to add noise to the data set which can decrease performance. A person's favourite colour would likely not help predict their income. Correlated features are ones that are related to each other. For example a person's weight is strongly correlated with their height. As a person's height increases, their weight usually does too. Including their weight to the feature-set would add information but not as much as an uncorrelated (but relevant) feature would. Lastly, a feature may be entirely redundant (a correlation coefficient of one or minus one) adding no new information that is not already captured in the data. For example, a copied column of useful features would be relevant but redundant.

2.1.3 Common Algorithms

2.1.3.1 SVMs

Support Vector Machines(SVMs)[19] are a commonly used machine classification algorithm. This is done by finding the maximum-margin hyperplane that separates two classes of points. The hyperplane is defined so as to have the maximum distance possible between the hyperplane and the nearest point from either of the classes. A hyper-plane can be written as:

$$\vec{w} \cdot \vec{x} - b = 0 \tag{2.1}$$

where $\vec{w}$ is the vector normal to the hyperplane and $\vec{x}$ is the feature vector. Where the two classes are linearly separable, two parallel hyperplanes can be constructed that separate the classes where the gap between planes is maximised. The region between planes is known as the *margin*. A hyperplane made between the two hyperplanes would then maximally separate the classes.

The two boundary planes are described by:

$$\vec{w} \cdot \vec{x} - b = 1 \tag{2.2}$$

and

$$\vec{w} \cdot \vec{x} - b = -1 \tag{2.3}$$

where the distance between them is $\frac{2}{||\vec{w}||}$. To ensure that no points fall in the margin implies the constraints that for all i,

$$\vec{w} \cdot \vec{x} - b \geq 1, \text{ if } y_i = 1 \tag{2.4}$$

and

$$\vec{w} \cdot \vec{x} - b \geq 1, \text{ if } y_i = -1. \tag{2.5}$$

These constraints may be rewritten as

$$y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1. \tag{2.6}$$

As such this becomes an optimization problem for the parameters $\vec{w}$ and b subject to the constraints such that $||\vec{w}||$ is minimized.

However, classes will often not be linearly separable. In this case the *hinge loss* function is used:

$$max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b)). \tag{2.7}$$

Where the the constraint is satisfied, and the point is in the correct side of the hyperplane the cost is zero, otherwise the cost is proportional to the distance from the margin. This changes the problem to optimising the parameters such that

$$\left[\frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i(\vec{w} \cdot \vec{x}_i - b))\right] + \lambda||\vec{w}||^2 \tag{2.8}$$

where λ shifts the balance between preferring increased margin size or ensuring that all points are on the correct side.

SVMs may be further generalised by using the *kernel-trick*. Here the dot product is instead replaced by a non-linear kernel function. This can be thought of as applying a standard SVM to a feature space which has been non-linearly transformed from the original features.

2.1.3.2 Random Forest

In order to understand Random Forests[20] the concept of the decision tree needs to be discussed. A decision tree works by subdividing the training examples using one feature at a time in order to create subsets that are purely of one-class or another. For example, in order to classify if a transaction is fraudulent or not the set of transactions can be split by whether or not the account associated has fraudulent history or not. Each subset can then be further subdivided by whether it is a current or savings account and so on. The

choice of which variables to separate on first, and at which point they are to be split, can be guided by several algorithms. Iterative Dichotomiser 3 (ID3)[21] for instance, chooses the feature and point at which to split the data based on an entropic measure of the purity of the child groups.

Decision trees, if allowed to grow too long can learn very spurious patterns to distinguish classes, patterns which do not hold generally beyond the training dataset. Tree bagging tackles this issue by created many decision trees and averaging their decisions. Each tree is itself trained on a random subset of the data. Random Forest takes this concept further by also randomly selecting a subset of features that each tree may use.

2.2 Neural Networks

Artificial Neural Networks (ANNs) are one of many ML algorithms developed over the last century. In deep learning applications, which we shall cover in Section 2.5, ANNs are almost exclusively used. For us they will make an indispensable start toward deep learning techniques and serve as a platform for more in-depth ML understanding. A simple definition of an ANN was provided by an influential figure in neuro-computing, Dr Robert Hecht-Nielson. He defines a neural network as "*...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs*" [22]. ANNs derive their name from being simplistic models of biological neural networks, see Figure 2.7.

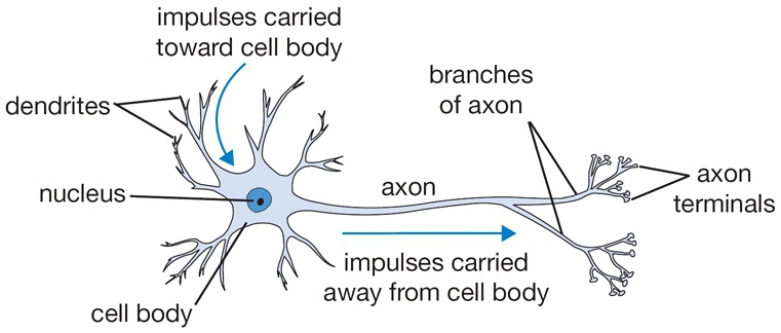


FIGURE 2.7: ANNs are loosely inspired by the structure and operation of biological neurons. The neuron receives information in the form of potential differences at the tips of the dendrites. These electrical impulses are then carried toward the cell body which will itself emit an electrical signal along the axon to other neurons. The emission of an electric spike is dependent on the information received at the dendrites and the 'chemical computation' of those inputs within the cell. Biological neurons either fire or they do not, but do not emit a continuous range of potential differences as an artificial neuron does [23].

The first neural network model was developed with electrical circuits by Warren McCulloch and Walter Pitts in 1943 [24]. However, the model lacked a mechanism for learning. Since then there have been many improvements [25]. The advent of significantly faster computers in the 1950's allowed for larger networks [25] leading IBM to form a research group to study pattern recognition and information theory under the leadership of Nathaniel Rochester [25]. They developed and simulated abstract neural networks on the new IBM 704 computer. By 1959 neural network models called "ADALINE" and "MADELINE" were designed that were similar to the ANNs of today [25]. MADELINE was more advanced than her counterpart and was the first ANN applied to solve a real problem; eliminating echoes on phone lines. Then in 1962 a significant step was made; a learning algorithm, the Windrow-Hoff rule, was developed that could change parameter values as a function of the prediction error [25] making the neuron perform better.

Despite the promising success with neural networks, interest faded in favour of Von Neumann computing architecture in a period known as the *AI winter* [26]. This was partly the result of the early successes which led to over-expectations for what neural networks are capable of. The inevitable disappointment led to research and funding being drastically decreased.

In 1986 three independent research groups tackled a method to extend the Windrow-Hoff rule to multi-layer networks and came up with similar ideas regarding what is now called *back-propagation* [27]. The development of this powerful training algorithm combined with faster computers thawed neural networks out of obscurity.

No understanding of neurons would be complete without first understanding *Perceptrons* [28]. Developed in the 1950's and 1960's by Frank Rosenblatt, a perceptron consists of one or more inputs, a processor and a single output [28] illustrated in Figure 2.8. The flow of information follows a "feed-forward model". In this model information enters as inputs as is processed the network, layer by layer till a result is produced; there is no feedback between layers. For the perceptron this means m inputs, represented by the real-valued $\vec{x} = (x_1, x_2, x_3, \ldots, x_m)$, are fed to the perceptron which computes a single result. Each input value, x_i, is multiplied by a corresponding real-valued *weight* w_i. All weighted inputs are summed, which can be represented as a dot-product

$$\sum_{i=1}^{m} x_i w_i = \vec{x} \cdot \vec{w} \tag{2.9}$$

where m is the number of inputs and $\vec{w} = (w_1, w_2, w_3, \ldots, w_m)$. Should the sum be greater than the threshold 0, the perceptron outputs a 1, otherwise a 0. Expressed as a

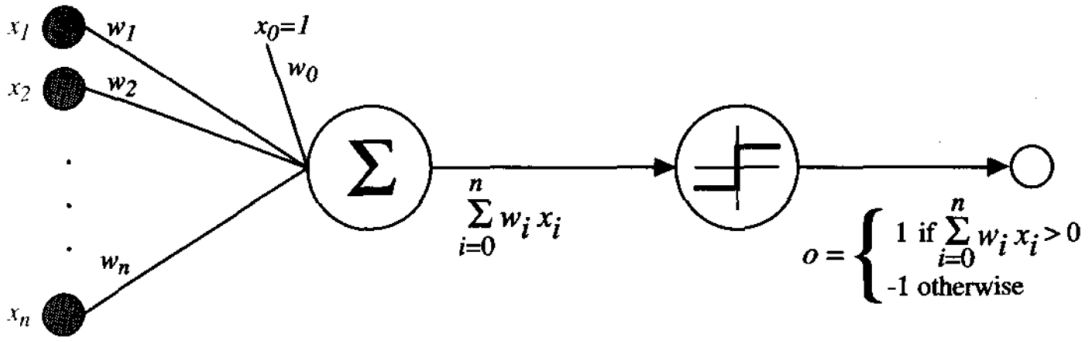


FIGURE 2.8: A Perceptron is a simple model of a neuron. Input is given in the form of a linear weighted sum, $\sum_{i=0}^{n} w_i x_i$, over all n input values. The result is used as input for the (often non-linear) activation function. For the first perceptrons this was the step-wise function which outputs one if the sum is greater than 0, otherwise a -1 [29].

function this reads,

$$f(\vec{x}) = \begin{cases} 1, & \vec{x} \cdot \vec{w} \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

The larger the absolute value of a weighting $|w_i|$ the more the corresponding input x_i contributes to the sum. A positive weighting increases the likelihood of 'exciting' the neuron which leads to an output of 1. Negative weights will reduce the sum and so 'inhibit' the neuron. Weightings can be tuned so the neuron is more responsive to specific inputs, some of which will be excitatory and others inhibitory. Similarly changing the threshold affects the output but in the reverse manner to the weightings. Decreasing the threshold increases the likelihood of the neuron 'firing' whereas increasing the threshold decreases the likelihood. Threshold varying is equivalent to holding the threshold constant and adding a *bias*, a constant b, to the dot-product.

$$f(\vec{x}) = \begin{cases} 1, & \vec{x} \cdot \vec{w} + b \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.11}$$

A positive bias is then excitatory, and negative is inhibitory. This still is not optimal as now weightings and an additional bias term must be tuned. Instead the bias is absorbed into the dot-product by introducing an extra input of x_0 which is always equal to one with its corresponding weighting of w_0. By varying the weighing w_0 this is equivalent to changing the value of the bias term. Now all neuron tuning is done on the weightings.

Till now we assign the neuron output only a zero or one according to a threshold. There is no reason why the output cannot be some other function $f(\vec{x} \cdot \vec{w})$, known as the *activation function*. Many different activation functions have been used throughout the development of ANNs. The function favoured today is the Rectified Linear Unit (ReLU)

or $\max(0, X)$. ReLU activations will grow linearly for $X > 0$ but are zero where $X < 0$. Research has shown that ReLUs make training quicker and have better overall results than sigmoidal activations [30].

A perceptron can learn a limited number of relations between inputs [28]. In particular the exclusive-or (XOR) function was determined to be beyond its learning capability in 1969 as discussed by Marvin Minsky and Seymour Papert [31]. However, the output of one neuron may feed as the input to another neuron and so on forming a network. Combining many simple elements into a network can produce a much more complex system [32]. Networks differ in kind but all share the same components: *nodes*, and connections between them [33]. Individual nodes can be thought of as computational units or neurons. Each receives input and processes the information to generate an output. The processing may be very basic (such as a simple summation of the input) or more complex. Connections between the nodes determine the direction of information flow. Interactions between the nodes lead to the complex emergent behaviour of the network.

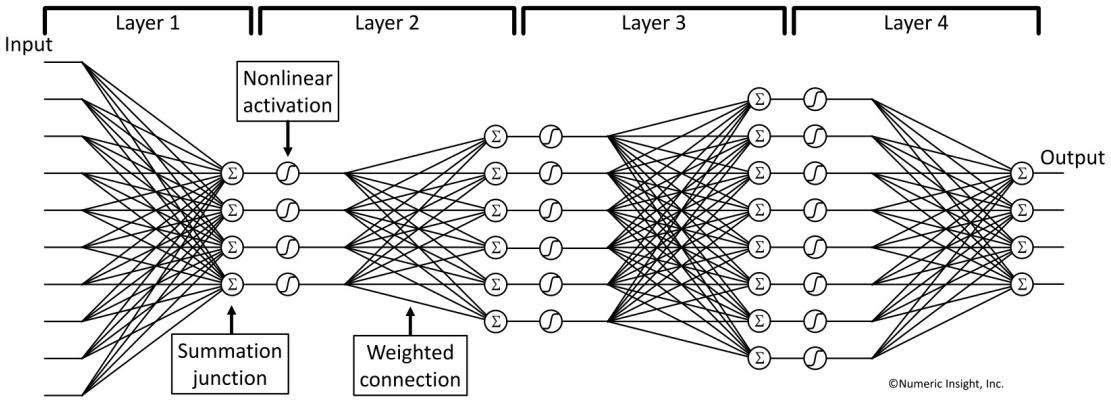


FIGURE 2.9: Pictured above is the architecture of a Multi-Layer Perceptron (MLP). An MLP is an example of a 'feed-forward' network as information flows through the network in one direction only (left to right in the figure). The network is composed of an arbitrary number of layers but will always have an input layer (on the far left) which receives the features and an output layer (on the far right) which returns the model results. Each perceptron within layer 1 takes a weighted sum - connection weightings can differ between neurons - of the input features through a non-linearity The singular output of each input node forms one feature for the next layer to the right, the first hidden layer. Similarly output from each of the layer 2 neurons serves as input features to layer 3, the second hidden layer. Finally layer 4, the output layer, computes the model result. As there are four output nodes, this MLP provides four different answers. For example each of the four neurons may give output representing the model's confidence that the input example image features belong to a cat, dog, man or mouse respectively. The layer 4 neuron with the largest output will then determine to which of the classes the item belongs [34].

Feed-forward ANNs are organized into layers. There are at least two layers to a neural network such as in Figure 2.9; the first layer, or *input layer*, and the last layer, the *output*

layer. Should there be layers between, they are known as *hidden layers*. The output from all input layer neurons are fed as input to second layer neurons. The second layer's output is now based on the results from the input layer. A feed-forward system allows proceeding layers to make decisions at a more abstract level than preceding layers [35]. These networks are much more capable of fitting to arbitrary feature relations. In fact they can compute any continuous function [36]. Changing one neuron to act as an AND logic gate is easily done by hand. However, for ANNs that have hundreds or thousands of neurons an algorithm must be used to train the weightings.

2.2.1 Training

Geoffrey Hinton, around 1985, developed a multi-layer neural network that could learn more functions than the single layer perceptron could [28] [27]. Hinton also developed the training algorithm for multi-layered neural networks called *back-propagation* [37]. Back-propagation is a supervised learning training method where the cost is calculated. The cost is a measure of the difference between output nodes of the network and their target values.

Mean Squared Error (MSE) is a common cost function used in regression models. It is just the average squared difference between the true value of example i, y_i, and the predicted value, $y_i^\star$.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - y_i^\star)^2}{n} \tag{2.12}$$

where n is the number of examples in the test-set, y_i is the true label for example i, and $y_i^\star$ is the predicted value,

Back-propagation is aptly named as the cost is propagated backward though the network, adjusting connection weightings, moving to a different location in 'weight space', to reduce the error at every stage.

Gradient descent is the typical method used to tune the weights once we have the cost. The cost function can be pictured as a surface of height parametrised by the weightings, see Figure 2.10 for a detailed picture. The cost-surface will have many peaks (local maxima) and troughs (local minima). Weights are usually initialized to take on random values; the exact distribution from which they are sampled can vary. All the weight co-ordinates will specify a point in the cost-landscape. The lowest possible cost, and so the best fit of the neural network, lies in the deepest trough, the global minima. Gradient descent looks at the immediate area around the current location, see Figure 2.11, and walk down the path of steepest descent. As gradient descent only walks downhill, and not up, there is no guarantee that it will reach the global minima. Gradient descent

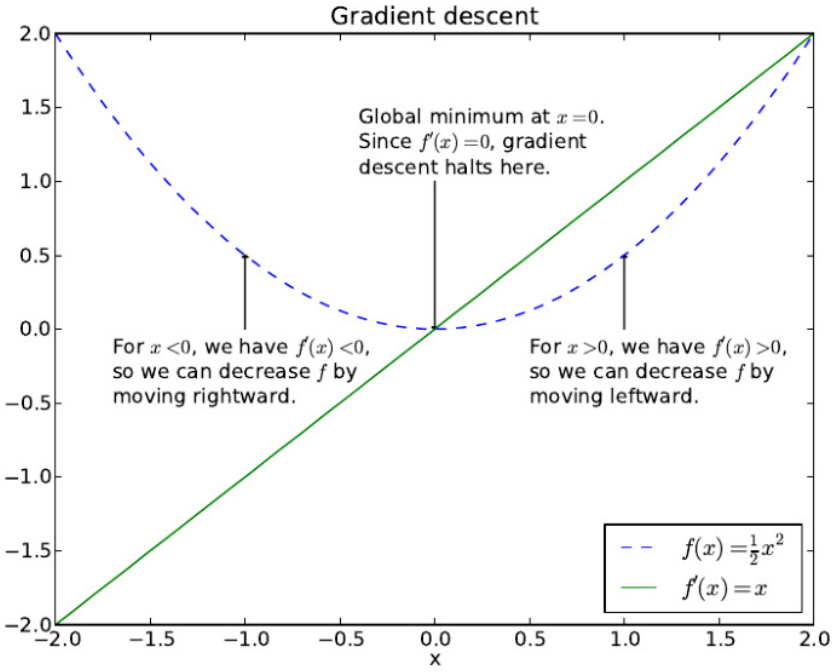


FIGURE 2.10: A neural network like many ML models can be highly complex, non-linear and non-convex. A function, $f(x)$, is convex over an interval $[a, b]$ if the second derivative $f''(x) \geq 0$ for all x in $[a, b]$. It may not be possible to analytically calculate the optimal parameters for the model. However, we can use gradient descent to slide down the surface of the cost function to a minima. Consider the simplified cost equation of $f(x) = \frac{1}{2}x^2$. Left of the minima at $x = 0$ the derivative, $f'(x) = x$, will be negative. So we can decrease the value of $f(x)$ by choosing a new value for x to the right of its previous position. Similarly where x is positive the derivative will be positive and so moving leftward will bring us closer to the minima [38].

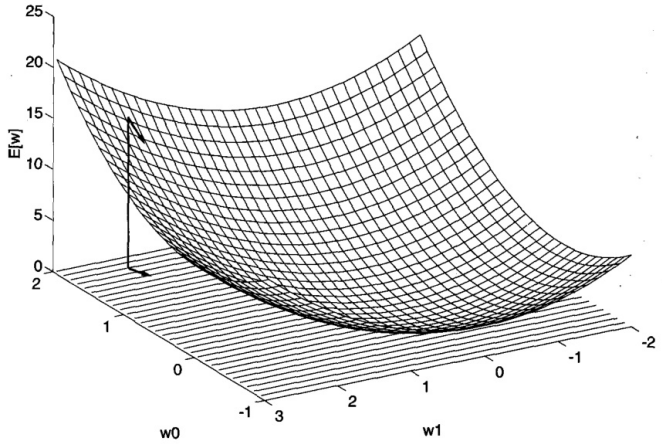


FIGURE 2.11: Where the cost function is a surface in a multi-dimensional space, here w_0 and w_1, the negative of the derivative at any point will point in the direction of steepest decent. Updating the weights to new values close to the last position but in the direction of steepest descent will reduce the cost of the model [29].

cannot climb a peak to get to a deeper trough. Finding the gradient of a function is in principle straight forward but can get messy. First let us define some notation referring to Figure 2.12: z_j is the weighted input sum of node j in layer l. g_j is the activation function applied to z_j at node j in layer l. a_j is the activation, the output of $a_j = g_j(z_j)$ at node j in layer l. w_{ij} is the weight connecting node i in layer $(l-1)$ to node j in layer l. t_k is the target prediction for the k'th neuron in the output layer.

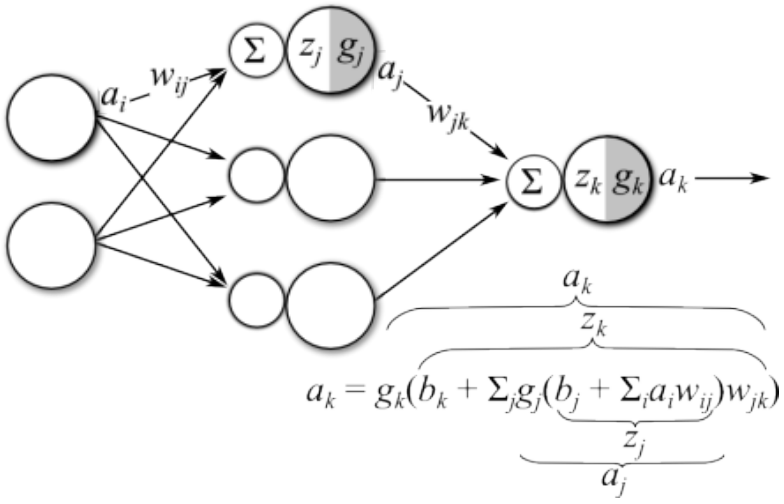


FIGURE 2.12: The effective algebraic function of an MLP takes the form of activation functions nested inside each other. Input activation values, or features, a_i are used in the weighted summation, $z_j = b_j + \sum_i a_i w_{ij}$, as part of the input to the next activation function, g_j, which determines the activation, $a_j = g_j(z_j)$. These values are in turn used in the weighted sum, $z_k = b_k + \sum_j a_j w_j k$ as input to the final activation function, g_k, to produce the activation of a_k. If the model parameters are tuned correctly a_k should be close to the true output, t_k, corresponding to input features a_i. The training method of back-propagation uses the chain-rule of calculus to see what the effect a slight change to each parameter would make to the final result. The model will then update each parameter in the direction that to first-order brings the output closer to the true value [39] [40].

In order to understand the back-propagation algorithm we will consider a *Multi-Layer Perceptron* (MLP) with one hidden layer. The first case we will look at is the weights connecting the final hidden layer to the output layer. Nodes in the hidden layer are indexed by j, where there are J nodes in the final layer. The K output nodes are similarly indexed by k. For the cost function we take the sum of squares of the differences between target values and the network predictions. This is often used in regression problems, here we use it for simplicity. The cost function C is therefore

$$C = \frac{1}{2}\sum_{k \epsilon K}(a_k - t_k)^2 \tag{2.13}$$

A factor of $\frac{1}{2}$ does not affect the results and is used for later convenience. The function returns the cost of a single feed-forward prediction relating to a single example's features.

The total cost is the sum of every individual prediction's cost. This is because we do not wish to make the network efficient at recognizing only one item, but instead accurately predict on the whole data set.

We can use the chain rule to calculate the gradient of the cost with respect to each weight, w_{ij}.

$$\begin{aligned}\frac{\partial C}{\partial w_{jk}} &= \frac{\partial}{\partial w_{jk}}\left(\frac{1}{2}\sum_{k\epsilon K}(a_k - t_k)^2\right) \\ &= (a_k - t_k)\frac{\partial}{\partial w_{jk}}(a_k - t_k)\end{aligned} \tag{2.14}$$

The summation drops out as only one term survives the differentiation. The weight w_{jk} connects the j'th neuron in layer $l-1$ to only the k'th output node therefore every other output node has no dependence on w_{jk} and so all other derivatives are zero. Looking at the partial derivative in Equation 2.14 we get

$$\frac{\partial C}{\partial w_{jk}} = (a_k - t_k)\frac{\partial}{\partial w_{jk}}(a_k) \tag{2.15}$$

since t_k is independent of the weights. Now we use the chain rule again on the activation $a_k = g_k(z_k)$

$$\begin{aligned}&= (a_k - t_k)\frac{\partial}{\partial w_{jk}}[g_k(z_k)] \\ &= (a_k - t_k)g_k'(z_k)\frac{\partial}{\partial w_{jk}}z_k\end{aligned} \tag{2.16}$$

The weighted sum in the output layer is given by $z_k = \sum_{j\epsilon J} g_j(z_j)w_{jk}$. Taking the partial derivative we get $\frac{\partial z_k}{\partial w_{jk}} = g_j(z_j) = a_j$. Again only one term survives from the summation, but this time of index j. Substituting this simple result in place of $\frac{\partial z_k}{\partial w_{jk}}$ in Equation 2.16 we have

$$\frac{\partial C}{\partial w_{jk}} = (a_k - t_k)g_k'(z_k)a_j \tag{2.17}$$

where g_k' is shorthand for $\frac{\partial g_k(z_k)}{\partial w_{ij}}$. We see the derivative is a product of three terms: The difference between prediction and target value of neuron k; second is the derivative of node k's activation function from the last layer; lastly is the activation of node j in the previous layer. Grouping all terms involving k we get

$$\frac{\partial C}{\partial w_{jk}} = \delta_k a_j \tag{2.18}$$

where

$$\delta_k = (a_k - t_k)g_k'(z_k) \tag{2.19}$$

The δ_k can be thought of as a measure of the error at t_k. The error is back-propagated to all the weights $w_{1k}, w_{2k}, w_{3k}, \ldots, w_{Jk}$ in proportion to activation values $a_1, a_2, a_3, \ldots, a_J$.

In doing this we find each weight's contribution toward the error we wish to minimize. So we now update the weights as $w_{jk} \leftarrow w_{jk} - \eta \frac{C}{w_{jk}}$ where η is the magnitude of the step, or the *step size*. For all N examples within the dataset we sum the N steps $\eta \frac{C}{w_{jk}}$ so as to average the best possible step to minimize the total cost. Too small a step size means the algorithm will take too long to converge, and perhaps get stuck in local minima. Too large a step size and gradient descent can overshoot minima altogether, or oscillate over the minima never settling in the local minima, see Figure 2.13 for better understanding. We now proceed with how to update the weights for all layers other

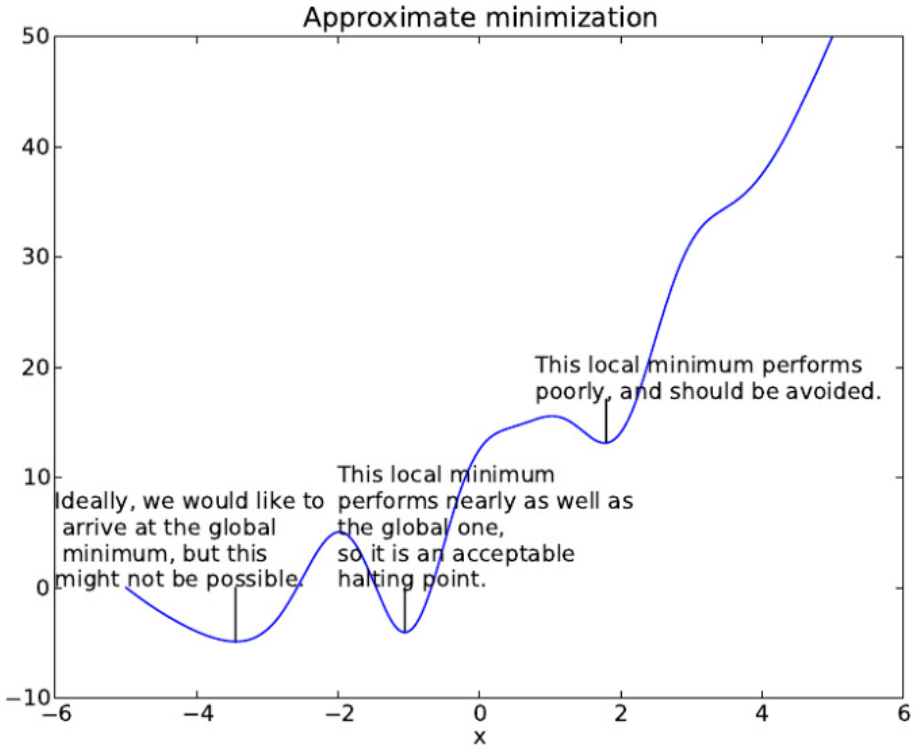


FIGURE 2.13: Navigating the cost function is not as simple as the case in Figure 2.11. The surface will likely not have one local minima but many. Training the network by sliding down the cost surface can trap the model in a sub-optimal local-minima. The ideal is to find the global minima as in the image above, however, this may be too hard to find. Therefore local minima which perform nearly as well as the global minima are acceptable halting points for training [38].

than the last hidden one. Now i indexes nodes in layer $(l-1)$ and j those of layer l. The derivative with respect to inner layer weights is

$$\begin{aligned} \frac{\partial C}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \frac{1}{2} \sum_{k \epsilon K} (a_k - t_k)^2 \\ &= \sum_{k \epsilon K} (a_k - t_k) \frac{\partial}{\partial w_{ij}} (a_k) \end{aligned} \tag{2.20}$$

Unlike with Equation 2.14 the sum is not dropped as every node in the hidden layer affects every node in the output layer. Substituting $a_k = g_k(z_k)$ into Equation 2.20 we obtain

$$\frac{\partial C}{\partial w_{ij}} = \sum_{k \epsilon K} (a_k - t_k) g_k'(z_k) \frac{\partial}{\partial w_{ij}}(z_k) \tag{2.21}$$

Now

$$\begin{aligned} z_k &= \sum_{j \epsilon J} a_j w_{jk} \\ &= \sum_{j \epsilon J} g_j(z_j) w_{jk} \\ &= \sum_{j \epsilon J} g_j \left(\sum_{i \epsilon I} z_i w_{ij} \right) w_{jk} \end{aligned} \tag{2.22}$$

After decomposing z_k in Equation 2.22 we determine $\frac{\partial z_k}{\partial w_{ij}}$ with the chain rule.

$$\begin{aligned} \frac{\partial z_k}{\partial w_{ij}} &= \frac{\partial z_k}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} \\ &= \frac{\partial}{\partial a_j}(a_j w_{jk}) \frac{\partial a_j}{\partial w_{ij}} \\ &= w_{jk} \frac{\partial a_j}{\partial w_{ij}} \\ &= w_{jk} \frac{\partial g_j(z_j)}{\partial w_{ij}} \\ &= w_{jk} g_j'(z_j) \frac{\partial z_j}{\partial w_{ij}} \\ &= w_{jk} g_j'(z_j) \frac{\partial}{\partial w_{ij}} \left(\sum_{i \epsilon I} a_i w_{ij} \right) \\ &= w_{jk} g_j'(z_j) a_i \end{aligned} \tag{2.23}$$

plugging in this solution for $\frac{\partial z_k}{\partial w_{ij}}$ into Equation 2.21 we have

$$\begin{aligned} \frac{\partial C}{\partial w_{ij}} &= \sum_{k \epsilon K} (a_k - t_k) g_k'(z_k) w_{jk} g_j'(z_j) a_i \\ &= a_i g_j'(z_j) \sum_{k \epsilon K} (a_k - t_k) g_k'(z_k) w_{jk} \\ &= a_i g_j'(z_j) \sum_{k \epsilon K} \delta_k w_{jk} \end{aligned} \tag{2.24}$$

Again we see errors back-propagate with the error signal now being

$$\delta_j = g_j'(z_j) \sum_{k \epsilon K} \delta_k w_{jk} \tag{2.25}$$

such that

$$\frac{\partial C}{\partial w_{ij}} = \delta_j a_i \tag{2.26}$$

To update the weights at layer $l-1$ the error signal of layer l is back-propagated and weighted by the activation a_i.

What is particularly convenient about back-propagation is that all the operations are easily vectorizable. There are many programming modules that perform vectorized equations significantly faster than iterating through all weights in a layer. Graphics processing cards have thousands of cores which can do many thousands of algebraic equations at one. We will come to the use of graphics cards later.

We can express this as a pseudo algorithm. For each example n in the training-set of N in total [41]:

1. Feed forward through each layer $l = 2, 3, \ldots, L$ with

$$z^{x,l} = w^l a^{x,l-1} \tag{2.27}$$

where

$$a^{x,l-1} = g(z^{x,l}) \tag{2.28}$$

2. Compute the error at layer L:

$$\delta^{x,L} = \Delta_a C_x \odot g'(z^{x,L}) \tag{2.29}$$

where

$$\Delta_a C_x = \left[\frac{\partial C}{\partial a_1^L}, \frac{\partial C}{\partial a_2^L}, \frac{\partial C}{\partial a_3^L}, \ldots, \frac{\partial C}{\partial a_J^L}\right] \tag{2.30}$$

The element-wise product e.g. $\begin{bmatrix}2\\3\end{bmatrix} \odot \begin{bmatrix}6\\7\end{bmatrix} = \begin{bmatrix}2\times 6\\3\times 7\end{bmatrix}$ is denoted by $\odot$.

3. Back-propagate through layers $l = (L, L-1, L-2, \ldots, 0)$ and compute the error

$$\delta^{x,L} = \left[(w^{l+1})^T \delta^{x,l+1}\right] \odot g'(z^{x,l}) \tag{2.31}$$

4. Update all weights according to gradient descent for each layer according to

$$w^l \leftarrow w^l - \frac{\eta}{N} \sum_{n \epsilon N} \delta^{n,l} (a^{n,l-1})^T \tag{2.32}$$

Each sweep through the entire data set is called an *epoch*.

2.2.2 Advanced Training Methods

Training an MLP with back-propagation as it was originally designed can take a long time. Every training iteration requires several time-consuming stages even with vectorized computation. Each training example must be fed-forward through the network. Then each error must be back-propagated through the network. Finally all the weights must be updated. And that is just one iteration. Back-propagation may not converge for thousands.

Standard gradient descent computes the gradient of the cost function, $C(\theta)$, with respect to the parameters θ for the entire training-set. This is known as *Batch Gradient Descent* and where weights are updated according to the following rule,

$$\theta = \theta - \eta \dot{\Delta}_\theta C(\theta) \tag{2.33}$$

where Δ_θ is the partial derivative with respect to θ. As the gradients need to be computed for the entire training-set this can be a very slow training process and may be limited by memory size. Such training is guaranteed to converge to the global minimum for convex surfaces such as in Figure 2.11, and to a local minimum for non-convex surfaces. *Stochastic Gradient Descent* (SGD) differs from batch gradient descent in that the model parameters are updated for a single randomly selected example $x^{(i)}$ and label $y^{(i)}$

$$\theta = \theta - \eta \dot{\Delta}_\theta C(\theta; x^{(i)}; y^{(i)}) \tag{2.34}$$

Batch training may perform redundant calculations in large data sets where there are similar examples. SGD however, performs an update from just one sample at a time, is much faster and can be used for online training. On average the gradient will point in the correct direction with much less computation.

Mini-batch gradient descent learns from both SGD and batch training and instead performs an update for every batch of n training samples.

$$\theta = \theta - \eta \dot{\Delta}_\theta C(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.35}$$

In this way the high variance of parameter updates from SGD is reduced and training is faster [42]. Common batch sizes vary between 2 to 256 examples. Mini-batch training is what is commonly employed for training neural networks. The acronym SGD is often used to refer to mini-batch training.

The stochastic nature of the batch means the batch gradient will often differ from the complete set. While that seems to be counter-productive the occasional misstep means it is possible to step over local minima and find a deeper one. Fewer samples increases

the random missteps allowing much more exploration of the parameter-space. However, using too few samples has the side-effect of increasing the training time. Every batch loaded contributes to the latency overhead. SGD will spend a lot of time randomly jumping through the parameter space so training will take a long time to converge. However, the likelihood of finding a deeper minima is increased. Too many samples within the batch will dampen the stochastic effect. The more samples in a batch the less time each epoch will take. For best results the order in which the examples are fed to SGD in each epoch should be randomized [42].

Momentum is a method that helps SGD navigate surfaces which curve more steeply in one direction than in another 2.14 [43] [44]. This is done by smoothing out the SGD updates by adding a weighted average of prior gradients

$$\mu_t = \gamma\mu_{t-1} - \eta.\Delta_\theta C(\theta) \tag{2.36}$$

$$\theta = \theta - \mu_t \tag{2.37}$$

where γ, a hyper-parameter, is usually set to 0.9[45]. The momentum term increases the step-size for dimensions where gradients tend to point in the same direction and reduces step-size for dimensions where gradients often change directions. The idea is that it removes some of the noise and oscillations SGD has, especially in the directions of high curvature, see Figure 2.14 for an illustration.

Other extensions like Adagrad by Duchi [46], Nesterov accelerated GD [47], Adadelta by Zeiler [48] and Adam by Kingma and Ba [49] are known to work equally well, if not better than standard momentum in certain cases.

Too much momentum will cause gradient descent to overshoot. If the updates have a high momentum in one direction they will tend to overshoot the local minima repeatedly. *Nesterov Accelerated Gradient* tries to take into account the levelling off near the bottom in order to forcefully reduce the momentum. If we use the momentum term $\gamma\mu_{t-1}$ to do the updates we can get an approximation for the next position by computing $\theta - \gamma_{t-1}$. We can use this information to calculate the gradient not with respect to the *current* parameters but with respect to the approximate future parameters:

$$\mu_t = \gamma\mu_{t-1} - \eta\dot{\Delta}_\theta J(\theta - \gamma\mu_{t-1}) \tag{2.38}$$

$$\theta = \theta - \mu_t \tag{2.39}$$

where γ is again typically taken to be 0.9.

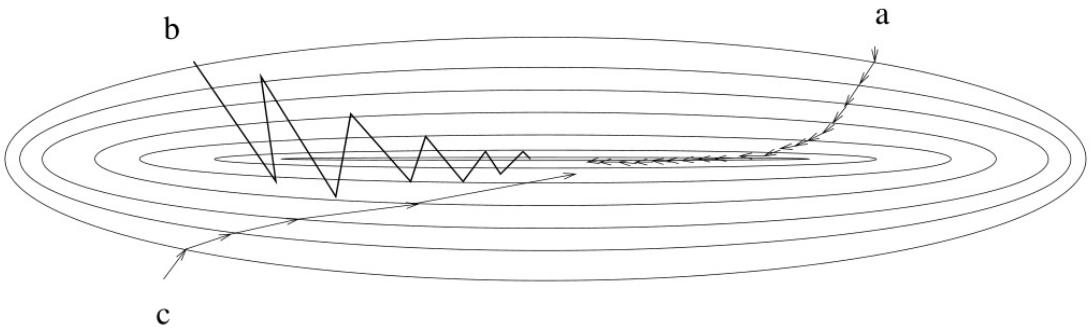


FIGURE 2.14: The size of the step in weight space should be small enough so as to be in the region where the derivative is tangential to the surface, but too small and training may take a long time to converge, as in the case of path *a*. Too large a step-size will encourage the model to leap over the minima to the other side where the gradient has changed direction altogether. This leads to oscillatory behaviour, path *b*, where the model may never converge. Momentum, path *c*, starts with a small step-size, but increases the step size for the next iteration in the direction in which it last moved reaching the minima in much fewer steps [35].

2.3 Optimization

2.3.1 Normalization

In many ML problems some features will consistently have much larger values than others. Consider the case of predicting house market value. Two of the features may be the last sale price and the number of rooms. Pricing may vary between R500, 000 to R20, 000, 000, leaving a huge range in between, whereas the typical number of rooms varies over the small range of one to four. Some ML classifiers are immune to feature scaling, however, gradient descent is negatively affected by features with values over different scales, making the process take longer [50]. For many algorithms, if there is a large discrepancy in the magnitude of features this can translate into the model overestimating the role of the larger magnitude feature. Some algorithms for clustering, or for feature reduction, can produce completely different results based on whether features are scaled or not as they are sensitive to the variance within each feature. Even in cases where the training algorithm is immune to such confusion, training is significantly sped up by the appropriate normalization of features.

To solve this issue we can *normalize* the features to have a similar range. The most basic form of normalization is *rescaling*:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)} \tag{2.40}$$

where x is the original feature and x' is the rescaled value. This effectively forces each feature to vary between zero and one.

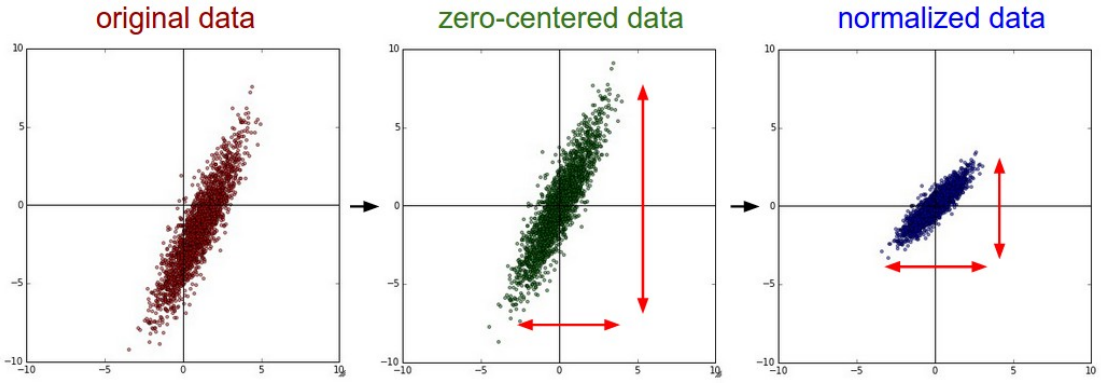


FIGURE 2.15: Initialized weights of a neural network are usually randomly assigned between -1 and 1. So on average a feature that has large values will contribute more to the weighted sum than another smaller scaled feature by virtue of having a greater product with the weighting. The arbitrary scaling of feature values should ideally not alter the results but it has been found to at best slow learning down, and at worst severely reduce model performance. Data will be normalized such that features are zero centred (centre graph), and the range of each feature made equivalent (right graph) [51].

Standardization is another normalization method that not only rescales values over a similar range centered at the origin but to also have unit-variance as in Figure 2.15.

$$x' = \frac{x - \bar{x}}{\sigma} \tag{2.41}$$

where $\bar{x}$ is the mean of feature x and σ is the standard deviation.

In addition, feature variables may often be correlated. This causes a problem for gradient descent, making optimization take longer [50]. Frequently a decorrelation method is used to transform the current set of features to a new set without correlation between variables. Consider Figure 2.16 where two features are strongly correlated with one another. This can be treated simply with a linear transform to the feature vector which effectively rotates the axes to lie along the lines of correlated variables. While most decorrelation algorithms are linear, non-linear decorrelation transforms do exist. Decorrelation of features is often known as *data whitening*.

Batch normalization is a method to counteract covariate shift [52]. *Covariate shift* is, informally, when the distribution of a function domain changes. In practice, what this means is that the n'th layer of neural network is not really learning $P(Y|X)$, that is the output Y based on the input X. What is really happening is that layer n is learning $P(Y|X,\theta)$ where θ are the model parameters of previous layers. After a single training cycle all parameters are updated; the parameters of the $n-1$'th layer are now $\theta^\star$. In the

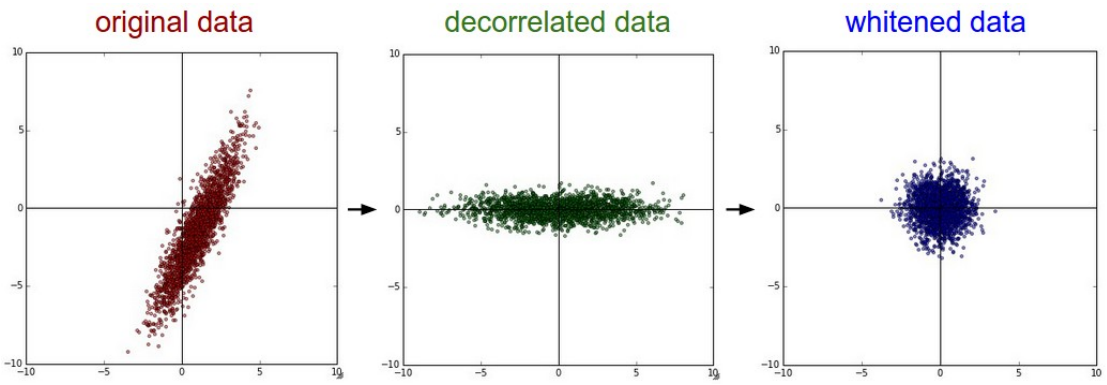


FIGURE 2.16: Rescaling and centering data will still not be optimal for many ML algorithms. After zero centering, as in Figure 2.15, there are still correlations in the data. That is, knowing the value of feature x can inform you about the value of feature y. The slanted elliptical data distribution means the greater in value feature x is, the more likely feature y will be larger. This is not ideal for learning because weightings on different features are now related which slows down the learning process and makes model interpretation more difficult. This issue can be dealt with by decorrelation. For linear correlations a rotation in feature space after zero-centering will work (centre image). The features will still be on different scales. Decorrelation followed by rescaling the data (image on right) is known as *data whitening* [51].

next forward-pass the n'th layer was expecting input with the same statistics as before, that is from the previous layers of parameters θ. Instead, input has changed and layer n is now learning $P(Y|X, \theta^{\star})$. The more layers there are in the network the more the effect compounds through layers. What batch normalization does is to standardize the statistics of each layers output such that the layer above can expect input not to vary significantly since the previous batch. This is done by normalizing the output activations to a mean of 0 and standard deviation of 1.

2.3.2 Learning Decay

Throughout training, learning-rates tend to be either too small or too large for the local cost-surface. Early in training a large step-size is desirable to quickly navigate down to a minima. If the learning rate were to be smaller in these regions gradient descent can converge to the local minima. Near the bottom a smaller step-size prevents over-shooting or oscillating over the minima.

Learning decay tackles this by making the learning-rate, η, a function of the epoch number i.

$$\eta(i) = \eta_0 \frac{1}{(1 + \gamma i)} \tag{2.42}$$

where η_0 is the initial learning-rate and γ is the decay constant. Empirically learning decay has been shown to speed up learning and train more accurate models [43].

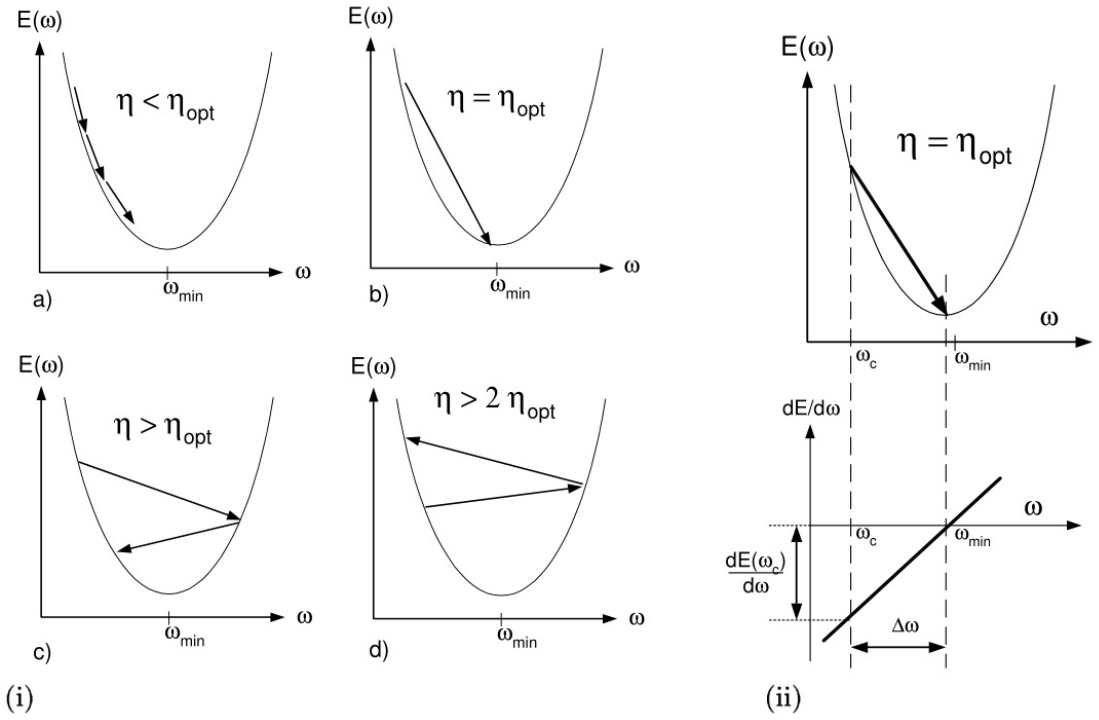


FIGURE 2.17: Where the gradient is negative the minima lies to the right, so adding the negative of the derivative multiplied by some constant η will take the weight toward $\omega_{\min}$. An optimal magnitude for η will update the weight perfectly, as in (ii). This optimum value is not known ahead of time, so (i) explores the results of having a differing value. Where $\eta < \eta_{\text{opt}}$, see a, the weight updates will slowly approach the minima. Where η is slightly larger than η_{opt}, see c, weight updates will oscillate over the minima. Too large a learning rate - $\eta > 2\eta$ in the simplified case of a parabolic cost function - and the model weights will diverge [42].

2.3.3 Activation Functions

Non-linear activation functions are used so as to be able to approximate non-linear relationships in the data [25]. A commonly used activation function is either the sigmoid, shown in Figure 2.18, which is given by the equation

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.43}$$

or the hyperbolic *tanh* function [25].

Weights must be initialized such that activations are not saturated, especially in higher layers. A saturated neuron is one that outputs a value near an activation function asymptote where the gradients are tiny. Should the neuron start or become saturated during training, the back-propagated signal multiplied by the gradient will approach zero, effectively stopping the back-propagating signal [30].

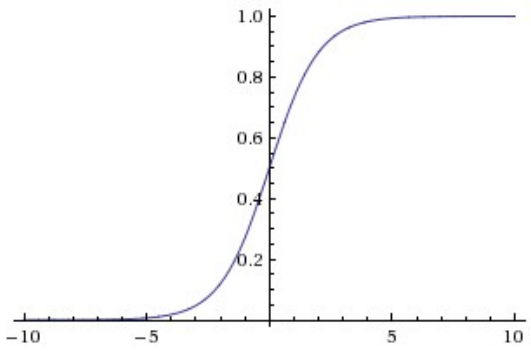


FIGURE 2.18: The sigmoid function.

In theory, no non-linearity has more expressive power than any other, as long as they are continuous, bounded and monotonically increasing [36]. However, the rectified linear unit (ReLU), shown in Figure 2.19, has been found to train faster and produce better models [53].

Being a simple $\max(0, x)$ operation, ReLUs quickly compute and are trained faster than both sigmoidal and *tanh* activations, see Figure 2.20 and Figure 2.21 [30]. In addition they do not require normalization to prevent saturation [30]. If just a few examples produce a positive input to a ReLU, then learning will take place on that neuron. However, ReLUs can die during training. A dead ReLU is one in the $-\infty < x < 0$ region where the gradient is zero which prevents training of the neuron and layers below.

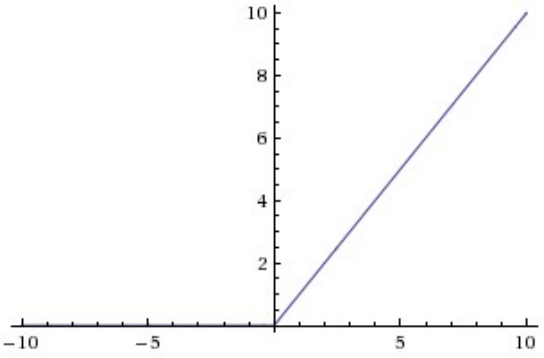


FIGURE 2.19: Of recent invent [53] was the use of the ReLU activation function in MLPs. $g(z)_{\text{ReLU}} = \max(0, z)$ has been found to train networks significantly faster and perform better than equivalent MLPs using sigmoidal activation functions [54]. However, where $x < 0$ the neuron is *dead*, or has a gradient of zero [54].

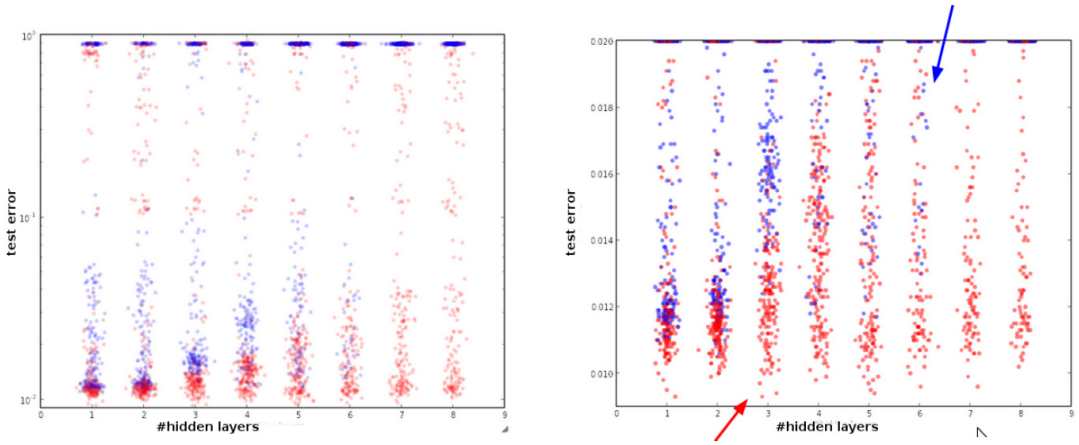


FIGURE 2.20: Models learned after being initialized with different weight parameters may differ in performance. One study [53] explored the effects of using ReLU over the more traditional sigmoidal activation functions. Many models of differing number of hidden layers and initialized weights were trained on the same data to see how activation functions affect performance. The graph on the left showing log (Test Error) versus number of hidden layers shows a clear pattern of ReLU models (in red) outperforming sigmoidal models (in blue) by two orders of magnitude at peak model performance densities. Zooming in on errors of order 10^{-2} on the right, the affect becomes more pronounced in larger models where ReLU models not only perform better on average but have error-rates no sigmoidal MLP ever achieves [55].

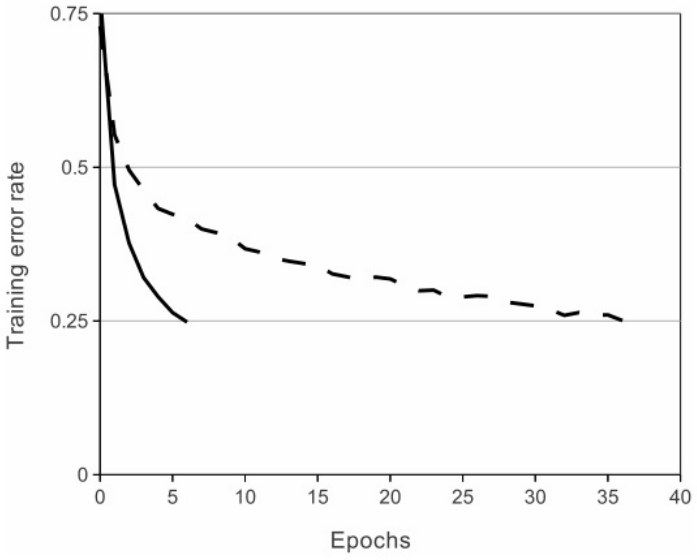


FIGURE 2.21: ReLU activation functions have been demonstrated to not only produce more accurate models but to train significantly faster. Shown above is the training error rate versus the number of training epochs. A four-layer neural-net achieves 25% training error rate on the popular image recognition CIFAR-10 data set [56] six times faster than the equivalent network using *tanh* activation functions. Learning rates were optimized independently for each model so as to make training as fast as possible for that activation function. The magnitude of the effect will vary for different model architectures, however, models trained with ReLUs tend to consistently learn several times faster than their *tanh* or sigmoidal model equivalents [30].

2.3.4 Weight Initialization

Initial weights have significant importance in training speed and accuracy [42]. Weights cannot all be set the same value. If all neurons where equally weighted they would produce the same output leading to the same gradients in back-propagation making training futile. This is not true for biases which are generally set to 0. However, ReLU biases are set to be a small positive value such as 0.01 to ensure that all ReLU activations are in the positive regime.

It has been discovered that output variance grows with the number of neuron inputs. Training is more effective when the variance is normalized to one with scaling by the $\sqrt{\text{fan}_{\text{in}}}$ [57].

$$w = \frac{\Sigma(0,1)}{\sqrt{\text{fan}_{\text{in}}}} \tag{2.44}$$

where fan_{in} is the number of inputs and Σ is a Gaussian of mean zero and standard deviation one.

More recent analysis by Glorot *et al.* [58] suggests weights should be sampled from a uniform distribution between $-r$ and r where $r = \sqrt{\frac{6}{\text{fan}_{\text{in}}+\text{fan}_{\text{out}}}}$ and fan_{out} is the number of output neurons.

A comparative study Larochelle *et al.* found that using the same neuron-count in each layer fares better than a decreasing or increasing neuron-count. The first hidden layer cannot be too small or else it cannot capture enough information from the input, but too many neurons risks over-fitting. However, for most tasks an over-complete first hidden layer works better than an under-complete one. A rule of thumb for the first hidden layer is $2n + 1$ where n is the number of input features.

2.3.5 GPU Speed Up

When dealing with neural networks of many millions of parameters and tens of thousands of training samples the training process can take weeks. Most of the training time is spent on matrix operations. These are matrix-vector products, used in forward and back-propagation, and vector outer-products when calculating weight updates.

Matrix-matrix multiplications can be done substantially faster than an equivalent sequence of matrix-vector products. Firstly by smart caching mechanisms as implemented in the BLAS library, and secondly thanks to parallelism [59]. The speed-up is generally not in proportion to the number of cores used due to high data transfer and the associated latency [60]. For small matrices the multi-core computations may take even longer than a single core because of this. However, parallelism is more efficient with larger

matrices. Each neuron in a layer receives input as a weighted sum or dot-product. As such, neural networks are highly parallelizable by nature. Entire training batches can be processed simultaneously.

In recent years GPUs decreased training times by some two-orders of magnitude compared to CPUs [60]. The typical CUDA-capable [61] GPU has several multi-processors (MP) [62]; each of which contains several streaming multi-processors (SMs) to form a building block. Within each SM there are several stream processors (SP) that share control logic and global low-latency high bandwidth memory bank. The high bandwidth comes at the cost of a small amount of RAM and lower clock speed. GPUs typically have between 2-8GB compared to up to a terabyte for a CPU. Unfortunately information must be sent to the GPU by the CPU via the PCI-E connector which is slow and has large latency. A GPU's architecture allows for thousands of threads, or processes, to run concurrently, shown in Figure 2.22. While CPUs are designed to handle general computing workloads, with units capable of processing high accuracy 64-bit floating point numbers, GPUs are less capable in the kinds of operations they can handle but can execute many threads at once making them excel at linear algebra and matrix multiplications.

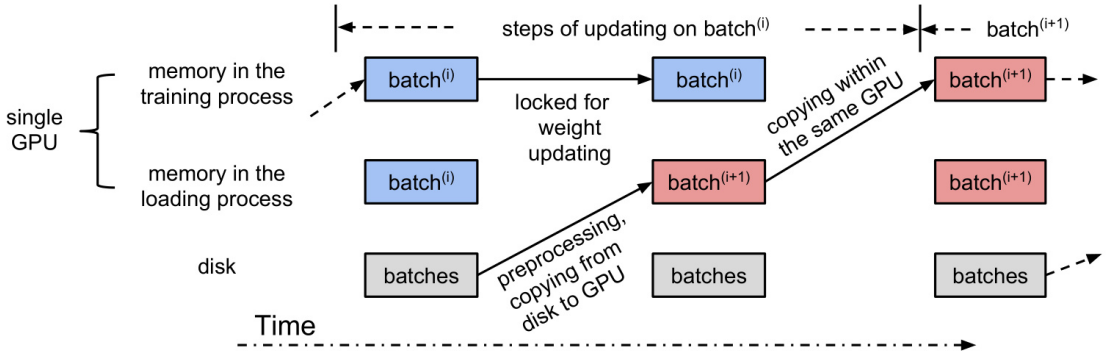


FIGURE 2.22: Graphics Processing Units (GPUs) excel at algebraic operations. GPUs were initially developed to use many cores for image-processing and the gaming industry. Despite each core being slower and with less allocated memory than a CPU, parallization allows for faster algebraic operations. Using a GPU, SGD on a GPU is handled like a production-line. Training batches are loaded onto the GPU (often pre-processed by the CPU). Then batches are simultaneously processed through the network to determine weight updates. Finally, model weights which are stored on the GPU are updated while the next training batch is loaded [63].

Unfortunately, programming GPUs is difficult, hard to optimize and requires specialized compilers. Fortunately many software libraries, as in Figure 2.23, have been developed at a higher level of abstraction to the GPU instructions. Together with libraries such as Theano and TensorFlow, models have been trained with over 100 million parameters.

Property	Caffe	Neon	Theano	Torch
Core	C++	Python	Python	Lua
CPU	✓	✓	✓	✓
Multi-threaded CPU	✓Blas	**x** Only image decoder	✓Blas, conv2D, limited OpenMP	✓**Widely used**
GPU	✓	✓customized Nvidia backend	✓	✓
Nvidia cuDNN	✓	**x**	✓	✓
Quick deploy. on standard models	✓**deepEasiest**	✓	**x** Via secondary libraries	✓
Auto. gradient computation	✓	✓Supports Op-Tree	✓**Most flexible** (also over loops)	✓

FIGURE 2.23: Several deep learning frameworks have been developed to accelerate model construction and training. Popular packages are shown above. The research in this thesis has been implemented using libraries built on top of the Python Theano framework. The open-source Theano, like several other frameworks, includes built in automatic gradient computation, commonly used functions, CPU multi-threading and GPU parallel processing on Nvidia hardware [64].

2.4 Over/Under-fitting

For a model to be useful it must generalize beyond the training data. A model could have 100% accuracy on the training data just by memorizing every example, but then it has not generalized at all. For an analogy; a student can learn to score perfectly on past paper questions if they appear in an exam but s/he has sacrificed general understanding for knowledge of particular questions and will likely perform poorly on unseen questions [8]. S/he has *over-fit* the training-set of past papers but would not perform well on the test-set (the exam), see Figure 2.24.

This illustrates the need for splitting all the data into a *training-set* and *test-set*, illustrated by Figure 2.25. In order for this to work, training and test samples must be representative of the underlying problem. The test-set must not be used in any way for training to remain an unbiased evaluation of the model's performance. Should it be used for training, even indirectly, then the model can learn to fit the test-set. This suggests that the model should not fully optimize performance on the training-set, see Figure 2.26. Over-fitting occurs more often in regression problems where the output is dependent on more than the information contained in the data set, such as environmental noise. Should the model over-fit training data, it is fitting to the noise and not the general trend.

The 'no free lunch' theorem formalized by Wolpert [67] stipulates that no algorithm can beat random guessing over all possible functions to be learned, seemingly supplanting ML altogether. However, real-world functions are not uniformly drawn from the set of

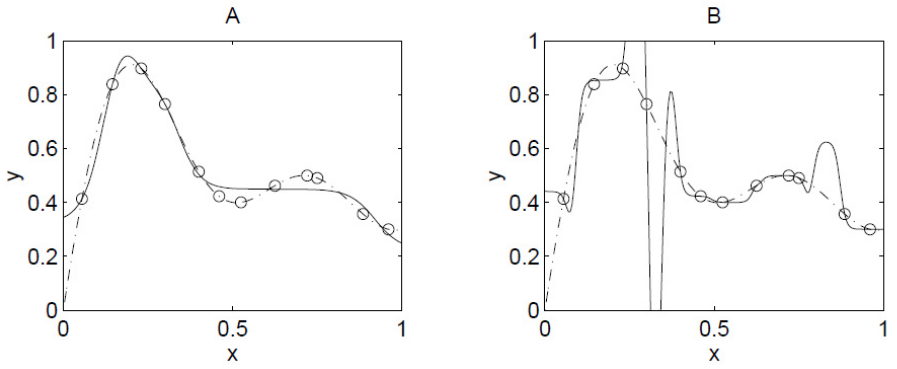


FIGURE 2.24: A neural network is trained to predict y from the x feature based on the 12 examples, shown as circles. The dashed line shows the function from which the points were sampled. *A*) A network of five neurons in the hidden-layer is fitted to the data producing the function approximation shown by the solid line. The model fits well but appears to have some bias in the $0.4 < x < 0.9$ region where the model is insensitive to the variance within the training data. *B*) A network of 20 hidden units is fitted. The significant number of parameters allows the model to fit perfectly, giving the correct prediction for every example feature. Bias is quite low in that the model is quite sensitive to variations within the data. However, variance is high as a small change in x has extreme effects on the prediction. The model thus over-fits the training data and would not generalize to new data sampled from the original function [35].

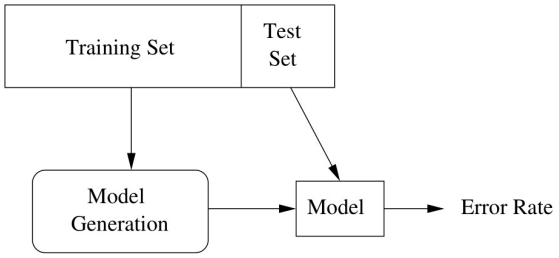


FIGURE 2.25: Model performance cannot be measured with the training data as the result can be misleading. With enough degrees of freedom a model can fit the training data perfectly but generalize badly to unseen data. To avoid over-fitting, the data are split into a training-set, and a test-set used only for an unbiased performance measure of the model [65].

all possible functions. Therefore it is reasonable to make assumptions that functions should be smooth, similar examples should have similar classes and simple functions are favoured over complex ones. Every algorithm must make assumptions beyond the data in order to generalize beyond the training data [9].

Over and under-fitting can be described in term of *bias* and *variance*. With too little complexity, loosely the number of parameters, a fitted model is *biased*; insensitive to variation in the data. For example a straight line fitted to points sampled from a sine function will not capture any of the oscillatory behaviour; the model has *under-fit*. In contrast, a highly complex function, as in the second frame of Figure 2.24, can go

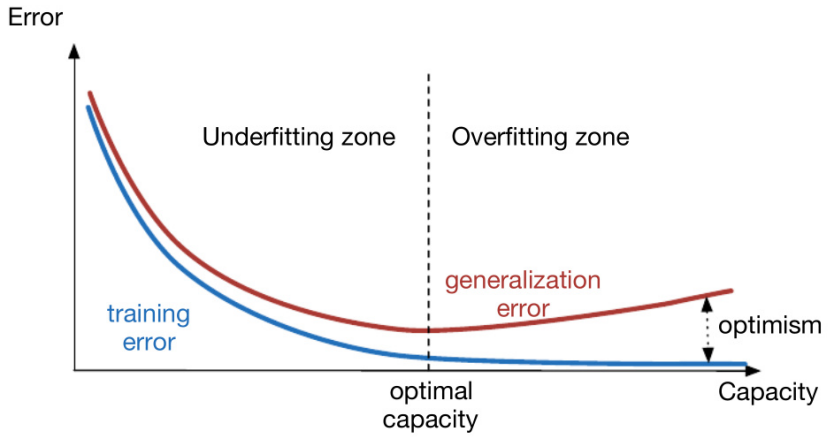


FIGURE 2.26: The more parameters the more capacity (horizontal axis) a model has to fit training-data (bottom curve, blue) thereby reducing test error (top curve, red). Increasing capacity also expands the number of training samples a model can fit. Training performance increases because the model fits the noise inherent to training data more than the general trend. Beyond the optimal capacity is the *over-fitting regime*. Prior to the optimal model capacity, in the *under-fitting regime*; there are not enough parameters to characterise the training data producing poor training and test performance results [66].

through every point but introduces unnatural wiggles. The model has high *variance*; minor variations in the data would result in a very different model. This illustrates the *bias-variance dilemma*; too low complexity introduces systematic bias that no amount of data can resolve but too much complexity increases variance and generalization is lost.

2.4.1 Curse of Dimensionality

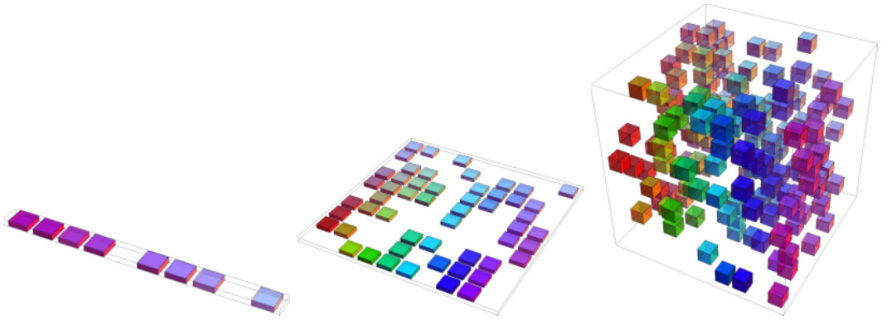


FIGURE 2.27: If eight of ten regions in a 1D feature space (left image) are known; a density of 8/10 is sufficient to estimate values for the two unknown regions. In a 2D plane (centre image) the number of regions grows to $10^2 = 100$. The number of samples needed to maintain the density is $0.8 * 10^2 = 80$. For a 3D feature-space (right image) this escalates to 800 samples. In general the number of samples needed scales as $O(V^d)$ where d is the number of dimensions and V is the number of distinguishable regions per dimension [66].

Naively, intuitions formed in a 3D world suggest that the addition of features could only increase classifier performance. However, with a large number of features potential benefit may be outweighed by the *Curse of Dimensionality* - a phrase coined by Bellman in 1961 [9]. Models generalize poorly in high dimensional data sets for several reasons:

1. When features are added to a fixed number of samples they cover a dwindling fraction of the feature-space. If n samples are required for an R^1 feature space to be considered dense, then n^d samples are required to maintain the same density in d dimensions, see Figure 2.27. The low data density in high dimensional space will require stronger and more accurate constraints [68].

2. Algorithms relying on similarity between samples (often a measure of the distance between points) break down. Consider the K-nearest neighbour classifier. Unknown points are classified by comparing to the closest K samples. This is powerful in a densely packed low-dimensional space such as R^2. However, with the addition of 98 irrelevant features (R^{100}), the irrelevant noise from the many features dominate the distance measure effectively making decisions random.

3. Even if all additional features are relevant, in a high dimensional space all examples can appear as nearest neighbours. Consider the idealized case where examples are laid out on a regular grid. In one dimension, the two nearest neighbours will lie on either side at the same distance from a sample x_t. In two dimensions the four nearest neighbours lie at the corners of the square surrounding x_t. In general for a sample x_t in d dimensions the number of nearest neighbours is given by $2d$. In a high dimensional space so many samples can be considered nearest neighbours, effectively making the K nearest neighbours random.

4. Objects in higher dimensional spaces have a larger amount of surface area for a given volume than objects in low dimensional spaces. For example, most of the volume of a multivariate Gaussian distribution comes not from near the mean (where the values are large) but much further out where the tails of the distribution are small but sweep into a much larger volume than near the core. Put in another way, most of the volume of a higher dimensional orange comes not from the pulp but the thin sliver of skin far from the centre. It follows that in order to enclose even a small fraction of the data a large radius is required [68]. The edge length of a unit-hypercube required to contain a fraction p of the samples is given by

$$e_d(p) = p^{1/d} \tag{2.45}$$

where d is the number of dimensions. In order to enclose just 10% of the total data the edge length is $e_{10}(0.1) = 0.8$. Therefore very large regions are required

to capture a small amount the of the data making it difficult to provide a local estimate for high dimensional data.

5. Most samples will be closer to an edge than to any other sample. For a scenario in which n data points are uniformly distributed in a d dimensional sphere of unit radius, the mean distance between the origin and the closest data point is

$$D(d,n) = \left(1 - \frac{1}{2}^{1/n}\right)^{1/d} \tag{2.46}$$

In a 10 dimensional space with 200 data points this translates to a median distance of $D(10, 200) \approx 0.57$. The nearest point to the origin is over half the way from the origin to the radius, which is closer to the boundary of the data. Most data points are outliers in their own projection. This can be illustrated by the idea of standing at the end of a porcupine quill. Every other quill will appear far away and clumped near the centre. This illustrates the difficulty in prediction of the label at a given point as any point will on average be closer to the edge than the training data point and so requires extrapolation.

There is some respite from the curse of dimensionality. In general, examples do not populate the feature space uniformly but are concentrated in a lower dimensional manifold. For example, k-nearest neighbours performs well for handwritten digit recognition even though the feature space is large. A 28×28 pixel image equates to a feature space of $28^2 = 784$ dimensions, however, the digits only live on a much smaller manifold in the full feature space.

2.4.2 Data Augmentation

Consider training a dogs and cats classifier. If it so happens that many of the cats face toward the left the network may learn to only recognize leftward facing cats and not 'realize' the class is not dependent on orientation. To reduce over-fitting of this kind for images, it is common practice to apply label-preserving transformations to already existing data to generate more. This is known as *Data Augmentation* [69] [70] [30]. Using augmented data typically boosts performance by about 3% [71], even in large data sets such as Galaxy Zoo [72] [70]. Commonly applied image transformations are skewing [28], translation, reflection, brightness adjustment [30] and rotation [69].

The use of data augmentation does introduce a new problem. If every image has many variations made, the resulting data set will be many times larger than the original. Some data sets, such as ImageNet [73], are already massive and it would be impractical

to store all this data [70]. This problem is treated by applying data augmentation on the fly while the original image is in memory, producing a temporary training batch before being discarded. In many implementations image transformations are applied by the CPU while the GPU is training on the previous batch of images being even more economical with computing time.

2.4.3 Regularization

If the model has more parameters than necessary it will tend to over-fit the training data [70]. *Regularization* is a method of ensuring over-fitting does not occur even when the model is over-complex for the task.

2.4.3.1 Weight Decay

The idea behind *weight decay*, also known as *L2 regularization* is to add a *regularization term* to the cost C_0 to obtain the modified cost

$$C = C_0 + \frac{\lambda}{2n} \sum_{\omega} \omega^2 \tag{2.47}$$

where ω is the weights vector, n the number of samples and λ is a scaling constant. All things being equal the network will now prefer to use small weights.

To see why this helps reduce over-fitting consider back-propagation with the modified cost function of Equation 2.47. The partial derivative used in back-propagation will be

$$\frac{\partial C}{\partial \omega} = \frac{\partial C_0}{\partial \omega} + \frac{\lambda}{n}\omega \tag{2.48}$$

and

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b} \tag{2.49}$$

Substituting this into the weight update rules from Equation 2.34 we see

$$b \to b - \eta \frac{C_0}{\partial b} \tag{2.50}$$

and

$$\omega \to \omega - \eta \frac{C_0}{\partial \omega} - \frac{\eta\lambda}{n}\omega = \left(1 - \frac{\eta\lambda}{n}\right)\omega - \eta \frac{C_0}{\partial \omega} \tag{2.51}$$

The update rule for the biases is unchanged but the weights are rescaled by a factor of $1 - \frac{\eta\lambda}{n}$. This rescaling is what is referred to as *weight decay*. The first term favours few and small weights, however, weights can still increase if it causes a decrease in the unregularized cost function [43].

In L1 regularization the added term is the sum of weight absolute values:

$$C_0 + \frac{\lambda}{n}\sum_w |w| \tag{2.52}$$

Naturally this is similar to L2 in that the model prefers smaller and fewer weights. To consider the difference between L1 and L2 regularization let us again consider the update rule with the modified cost function:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} sgn(w) \tag{2.53}$$

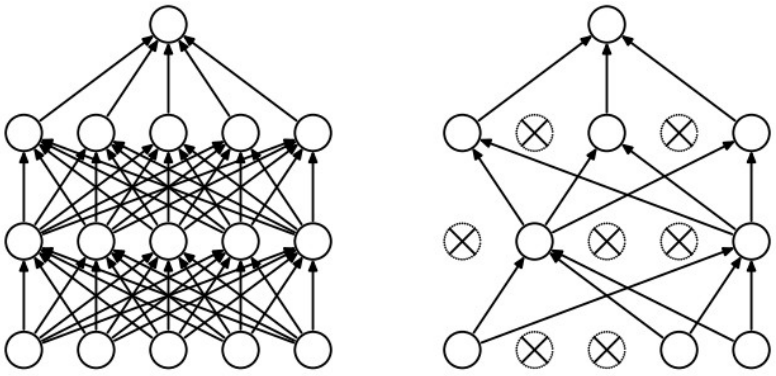
where $sgn(w)$ is the sign of w. A small caveat is that the derivative $\partial C/\partial w$ is not defined when $w = 0$. So in that case the standard update rule will be followed [41], equivalent to defining $sgn(0) = 0$. The update rule is then

$$w \to w - \frac{\eta\lambda}{n} sgn(w) - \eta\frac{\partial C_0}{\partial w} \tag{2.54}$$

Both L1 and L2 regularization shrink weights, however, in L1 regularization weights are shrunk by a constant amount, whereas in L2 it is in proportion to w. For a large weight the L1 regularization term $|w|$ is smaller than that of L2, and so the decay is less than for L2. However, when weights are small L1 will decay more than L2. L1 therefore tends to result in a more *sparse* network; one with a small number of heavily weighted neurons with the rest close to zero. Of course both types of regularization terms can be included in the model. The linear combination of both L1 and L2 is termed an *elastic net*.

2.4.3.2 Drop-out

Drop-out is a powerful regularization method introduced by Hinton [74] in 2012 which has been shown to work well for large neural nets. In drop-out a fraction f of randomly selected neurons within a layer are prevented from propagating their signal to the layer above. Neurons dropped out in this way contribute neither to forward-propagation nor back-propagation, as in Figure 2.28. Instead of the layer computing $a = g(w \cdot x)$ it now computes $a = m \star g(w \cdot x)$ where m is a masking vector and $\star$ is the element-wise operator. Activations of remaining neurons are multiplied by $1/f = 2$ to account for there being less active neurons [70]. Now every time the input is injected, the neural network effectively samples from a different architecture sharing the same weights [30]. As a result, a neuron cannot rely on particular neurons firing from the layer below. Instead, individual neurons learn to detect more robust features, see Figure 2.29, which are helpful regardless of the large variety of internal contexts.



(a) Standard Neural Net (b) After applying dropout.

FIGURE 2.28: Drop-out is an effective regularization method for MLPs. For every sample fed to the network only a fraction of randomly selected neurons may propagate the signal forward. This effectively creates a different architecture that shares weights with the original network. As a result neurons cannot rely on particular neurons from the layer below. This forces the network to generate a more robust representation of features from one layer to the next [75].

With drop-out the resulting model is the equivalent of training multiple networks and averaging their predictions. Drop-out may add up to a factor of two to training time but generates the equivalent of an ensemble in much less time [30].

2.4.3.3 Validation

It is not enough to split the data into training and test-sets to avoid over-fitting. The practitioner could then tune model hyper-parameters by seeing which values lead to the best performance on the test-set. Doing so would be indirectly training on the test data and so the test-set can no longer function as an independent source with which to evaluate the model [65].

This concern is what motivates the split of all the data into a training, validation and test-set illustrated in Figure 2.30. Typically the validation-set is smaller than the training-set so as to maximise training data. Common practice is to split 70/30 for training and validation-sets respectively.

Hyper-parameters are tuned to maximise performance on the validation-set and the completed model's performance is measured on the test-set [15]. Once the hyper-parameters have been selected and the model evaluated on the test-set, it is possible to retrain the model on all the data. With some well-behaved learning methods this is a reliable method to enhance the model [65]. However, to evaluate this new model a new source of test data must be used.

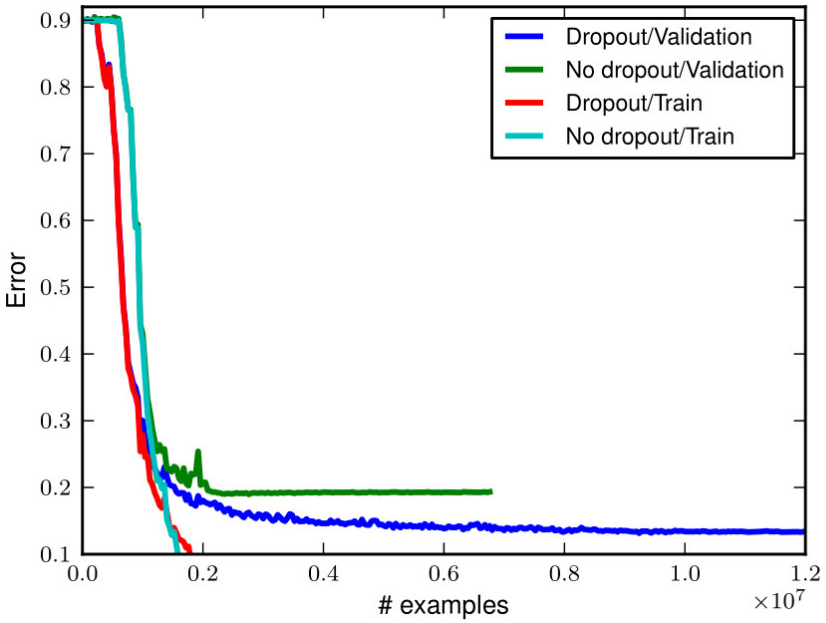


FIGURE 2.29: A study, *Towards drop-out training for convolutional neural networks* [76], was done on the effects of drop-out on CNNs. Plotted above is model error versus the number of training samples. Error on the training-set with drop-out (in red) is lower than without (in teal). The trend changes after $\approx 0.16 \times 10^7$ training examples where the standard model over-fits the training-set. Standard model validation error (in green) gets stuck in a local minima whereas the model with drop-out (in blue) is continuously improving. This can be attributed to the changing network structure which alters the cost function and back-propagation signal. For few samples, less than 1.6×10^7, performance is similar, however, drop-out helps prevent over-fitting [76].

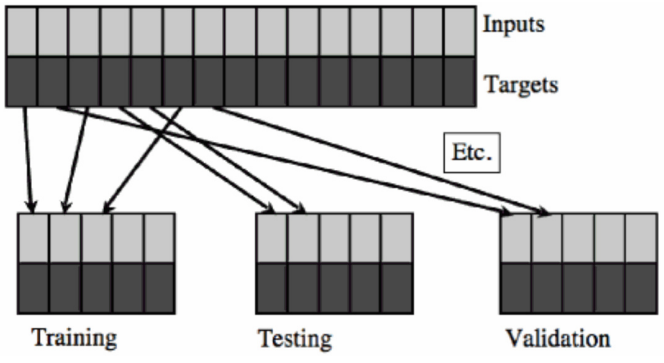


FIGURE 2.30: If hyper-parameters are chosen based on test-set performance then the test-set is being indirectly fit to. This motivates splitting the data into three parts for training, validation and testing. The validation-set is used for tuning hyper-parameters and the test-set is used for measuring performance at the end [77].

2.4.3.4 Cross-validation

The more training data; the better the model. However, too little validation and test data makes for unreliable performance measures. A commonly used method of K-fold cross-validation is used to handle the trade-off and is especially useful when there is little data altogether.

In K-fold cross-validation all the data are split into K equal sized partitions [15]. See Figure 2.31 for an illustration of 5-fold cross-validation. The model is trained on the first $K-1$ partitions, leaving out the K'th partition as the validation-set. This is repeated for all K combinations. The validation performance is measured by taking the average performance of all K models. There is no objective rule what value to use for K. In practice it is common to use 10-fold validation. An entirely unused test-set must be used to measure the models final performance.

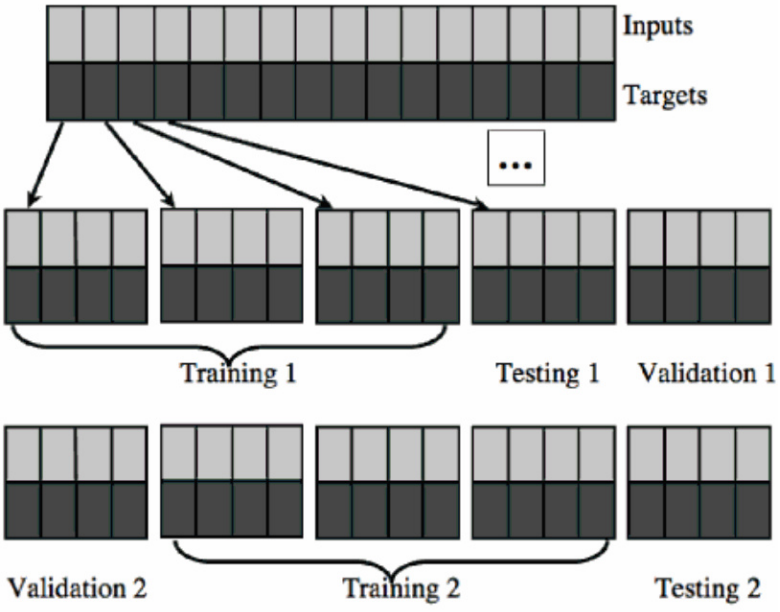


FIGURE 2.31: For small data sets there may not be enough for training and validation-sets. In K-fold cross-validation, data are split into K pieces to train K different models each withholding a different piece for validation. The validation score is the average over all models [77].

2.4.3.5 Early Stopping

If the model is over-complex it may over-fit if trained to convergence. *Early stopping*, see Figure 2.32, is an inexpensive way to avoid over-fitting by stopping training ahead of convergence.

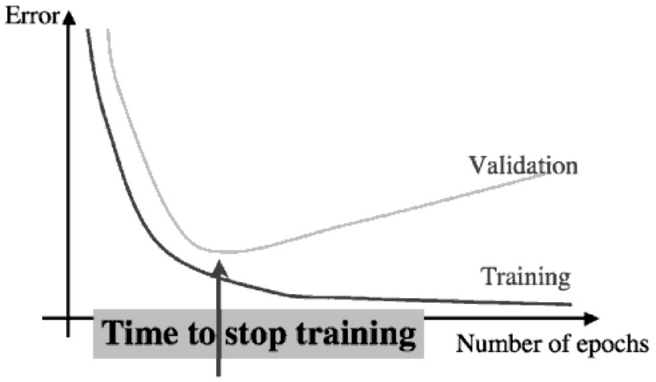


FIGURE 2.32: Early-stopping is a method of stopping training after a set amount of epochs to prevent over-fitting [77].

In an early stopping implementation the model is tested on the validation-set after every N updates. Training is stopped when the model has performed best on the validation-set [43]. Practically, training must go beyond the point of best validation error in order to detect the optimum epoch number. Even if the model overshoots the optimum epoch number, early-stopping will reduce over-fitting damage. Unfortunately, there is no a-priori method to determine when to stop. Instead, a few heuristics exist. The one used in this thesis uses patience, which is the minimum number of training epochs before validation scoring which saves time in early training stages. Once the threshold has been reached the validation score will be recorded after a further N updates. Should the new validation score be better than the previous, training will continue for another N updates otherwise training is halted [78] [79].

2.5 Deep Learning

ML methods typically use shallow-structured architectures. These techniques have only one layer of non-linear feature transformations such as an ANN [80]. Shallow architectures have proven to be successful in solving many simple or well-constrained problems. However, their limited modelling and representational power is often insufficient for various real-world problems. For example, image recognition [80] problems require the model to be insensitive to irrelevant variations, see Figure 2.33, such as lighting conditions, translations, rotations, pose, scale conformation and obstructions [81] while being sensitive to relevant variations such as distinguishing features of dog breeds. A linear classifier would likely classify two different breeds of dogs as the same if they are in the same position and background. Similarly, a single breed of dog could be classified differently when in differing image contexts.

As a result shallow classifiers require smart feature extraction that solves the *selectivity-invariance dilemma*; features must be selective of important variations and invariant to irrelevant ones. Constructed features tend to be case-specific and require deep knowledge of the problem and data [25]. Depending on initial model performance, features would then be redesigned in an iterative and time consuming process, see Figure 2.34.

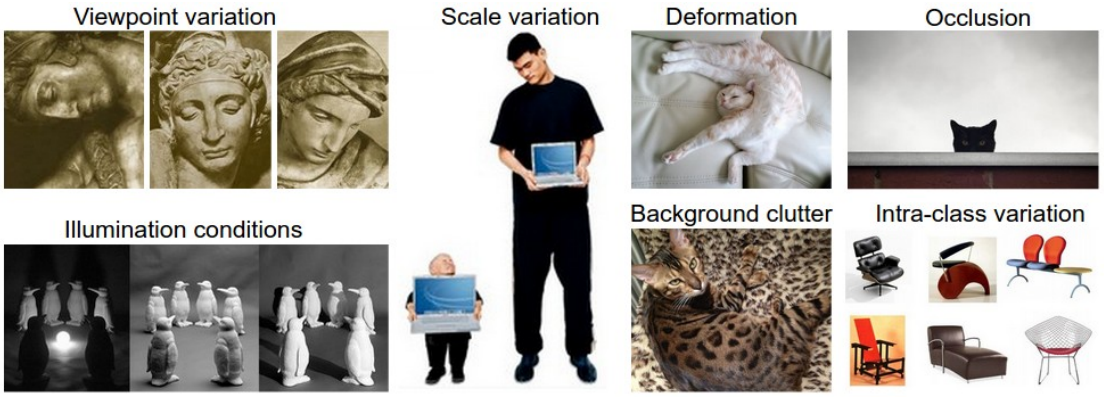


FIGURE 2.33: Image recognition is a difficult task as the very same object can appear with many irrelevant variations. For a model to be successful it must be impervious to these changes. For example, an object may vary in orientation (top left), lighting (bottom left), scale (middle) and distortion (one left of top right). There may be contextual changes such as background clutter (one left of bottom right) and occlusion (top right). In addition, the object class may vary in structure and colouring (bottom right) and yet still belong to the same class [82].

Speech exhibits hierarchical layered structure from the waveform level of sound to the linguistic level of speech [84]. Similarly the visual systems are also hierarchical in nature. Pixels may be assembled into edge-lets, edge-lets to motifs, motifs to parts, parts to objects, and finally objects to scenes [85]. This suggests that ML methods adopt hierarchical structure. A deep learning architecture is a multilayer stack of simple modules which compute non-linear input-output mappings. Each module in the stack performs a transformation on the input to increase selectivity and invariance of the feature-set. See Figure 2.35 for an example of a multi-layered architecture developing better feature representations at every layer for facial recognition. A deep architecture can eliminate feature extraction altogether [86]. The first few layers learn features directly from the input data, illustrated in Figure 2.36.

Deep Learning (DL) algorithms are inspired by the human brain [28] [62]. The brain quickly processes complex data, learns in different domains and solves complicated tasks. DL aims to deal with high-dimensional data efficiently and quickly; performing complicated tasks such as image and sound recognition. As Bengio and Lecun put it, "Deep architectures are compositions of many layers of adaptive non-linear components, in

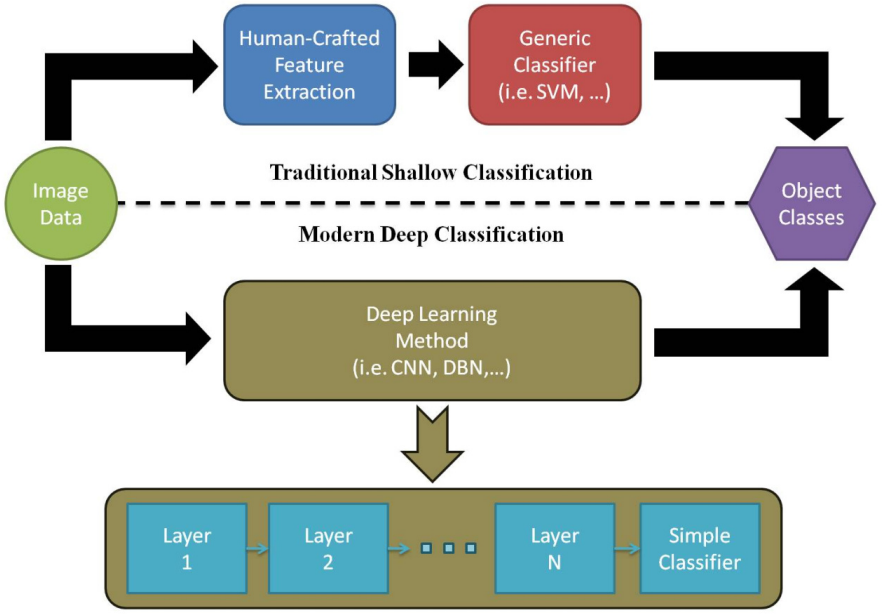


FIGURE 2.34: In shallow architectures features are manually created based on human analysis of the problem. Crafted features are then fed to one of many possible classifiers and tuned for peak performance. In contrast, no human-crafted features are necessary in deep learning. Instead, the multi-layered architecture will learn the appropriate feature transformations to construct the most useful features. The final layer of the network presents these learned features to a simple classifier [83].

other words, they are cascades of parameterized non-linear modules that contain trainable parameters at all levels" [89]. DL refers to supervised or unsupervised algorithms with many layers of information processing that learn hierarchical representations of the data [62] [80]. While there were multiple deep architectures before 2006, almost none were successful is terms of accuracy and efficiency with the exception of Convolutional Neural Networks (CNNs) [86].

DL methods have had many successful applications: visual document analysis [90] [91], facial [92] [93] and speech recognition, natural language processing, human action recognition [94] [28], facial location [81], image semantic discovery [81], image compression [70], collaborative filtering [62], and medical diagnosis [70]. Companies such as Google, IBM, Apple and Facebook have all been aggressively pursuing DL projects. Google has used DL for voice recognition, Street View and image recognition; Microsoft for their real-time language translation in Bing voice search [95], and IBM for their Jeopardy winning Watson [96] [62]. Siri, the iOS assistant, gives weather reports, provides news, answers questions and gives reminders all with the power of DL [62]. The recent enthusiasm for DL is largely a result of increased hardware capabilities, specifically GPUs, cheaper computing equipment and recent advances in research [80].

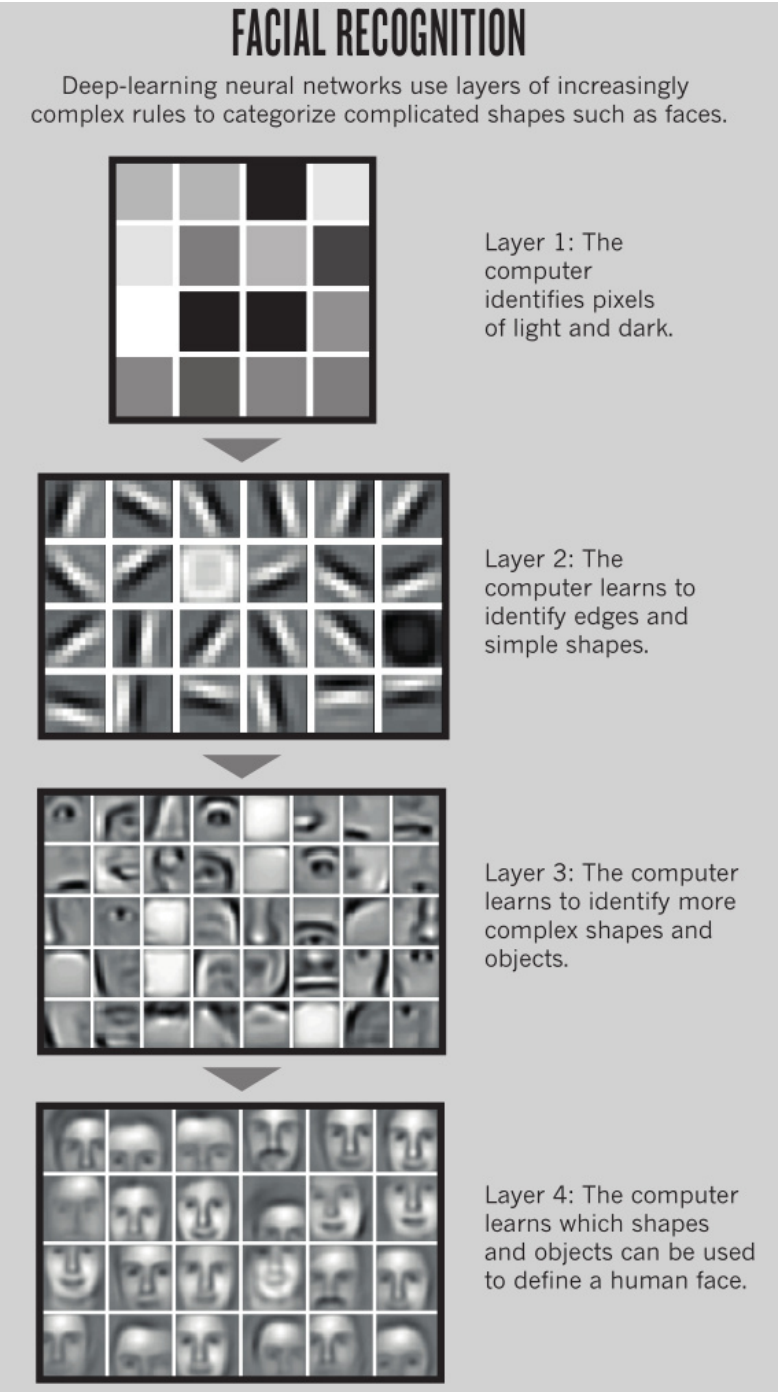


FIGURE 2.35: Deep learning utilizes a multi-layered architecture to construct complex features of the input data. A model for facial recognition takes in pixel values as input. Layer 1 identifies light and dark pixels. Layer 2 learns to identify edges with different orientations and simple shapes from the output features of layer 1. Layer 3 identifies more complex shapes and features from simpler shapes and edges. Layer 4 learns which complex shapes and objects define a human face. A facial image decomposed into its different parts and shapes can then be used to identify the person [18].

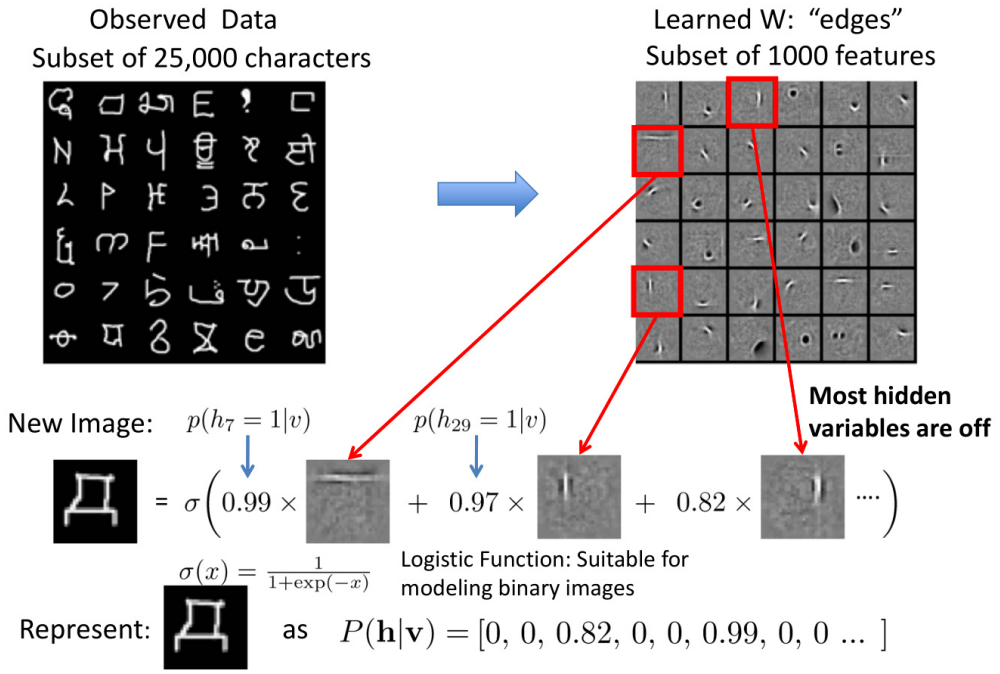


FIGURE 2.36: Sampled characters from a 25,000 character data set are shown above [87]. Designing features that best capture information relating to these characters would take plenty work and insight. Deep learning can learn complex features directly from the data. On the right we can see a sample of what image characteristics provide the most activation to neurons higher up in the network. The network has learned to use edge detectors at different locations and orientations. Now a character can be decomposed into 1000 coefficients of the feature maps rather than manual feature construction [88].

Bengio and Lecun proposed the following requirements for a successful DL implementation [89] [17] [62]. The algorithm must:

1. Function with a large range of architectures.
2. Handle deep architectures, and manipulate intermediate features with many levels of non-linear transformations.
3. Sample from a large space of possible functions with many millions of parameters.
4. Scale efficiently with the number of parameters and samples. This prohibits algorithms that iterate many times over the data.
5. Discover multi-use concepts (multi-task learning) and may use unlabelled data.

2.5.1 Convolutional Neural Networks

2.5.1.1 Introduction

The Universal Approximation theorem by Hornik [36] states that a neural network with a single hidden layer can model any continuous function. However, Bengio [17] showed that a shallow network would require an exponentially large number of neurons when compared to a Deep Neural Network (DNN) with many hidden layers. Recently in 2014, Romero [97] and Ba and Caruana [98] showed that a deeper neural network can be trained to perform much better than a comparatively shallow one. However, DNNs typically require huge computational resources that make their training and real-time application difficult to implement [70].

DNNs face several challenges [28]:

1. DNNs cannot train on unlabelled data, which greatly outnumbers labelled data.
2. Back-propagation correction signals get severely weakened travelling through the neural network. This results in layers near the top being altered, but layers near the bottom are largely unaffected; the *gradient dilution problem* [17].
3. Learning is very slow for networks of many layers due to the many millions of weighted sums, activations and back-propagation signals that need to be computed.
4. The network is likely to end up in a poor local minima rather than the global one. The severity of poor local minima increases significantly as network depth increases [80].
5. The main deficiency of unstructured nets is their lack of built-in invariance with respect to translations and distortions [86]. For example, digits may appear with differing slants, sizes and positions. Words are spoken with varying pitch, speed and intonation. In principle, a sufficiently large network can learn to be invariant to such transformations, however, that will likely result in multiple units with identical weight patterns and the number of training instances required is very large [86].
6. Due to the input data being vectorised and fed to the first layer, the topology of the input is ignored. This is contrary to the fact that spectral representation of speech and images have strong local 2D structures and times series strong 1D local structure. For example, pixels that are spatially or temporally nearby are highly correlated [86].
7. The large number of trainable parameters allows for easy over-fitting.

In 1962 Hubel and Wiesels' studies of cats' primary visual cortex [86] [25] [70] revealed that there is a hierarchy operating within neurons of the visual cortex in living organisms. Simple neurons in the up-most layer connect to a small region of complex neurons below [99]. The first neural nets based on this insight were Fukushima's Neocognition and Lecun's Net-3 [25]. In such an architecture, the lower layer is divided into a smaller number of regions called *Receptive Fields*, each of which is mapped to a single neuron of the layer above [25]. The connection is called a *feature extractor*. Many such feature extractors are applied to the same receptive field generating a corresponding feature vector.

Advantages for this architecture include [66]:

1. Sparse Connectivity - Rather than connect an entire layer to the layer above, each receptive field is connected to a single neuron, see Figure 2.37. This significantly reduces the number of parameters which makes training much easier [25] and faster. Trials have consistently shown that sparsely connected networks outperform their fully connected versions.

2. Shared weights - Each receptive field is connected to the upper layer by an identical set of weights [30] [86], see Figure 2.38. Elementary feature detectors useful in one part of the image are likely to be useful everywhere [86]. This is a strong, yet justified, assumption in the locality of pixel dependencies. This significantly reduces the number of parameters that need to be learned therefore improving training [28] [25] and model generalization while the best possible theoretical performance is only slightly worse than that for a fully connected DNN [30] [86].

3. Sub-sampling - Between layers input data is sub-sampled in what is known as *pooling* which significantly reduces the number of parameters and computations required [25] [62] [60].

4. Deep Architecture - The many layered architecture allows for extracting highly non-linear and robust features.

The resulting *Convolutional Neural Network*(CNN) [80] learns hierarchical representations of the data using local receptive fields, shared weights and sub-sampling and is more resilient to variations within the data [85].

A typical CNN is composed of many layers; some for feature representations, known as *feature maps*, and others as conventional neural networks. The input and output of each stage is a set of matrices called *feature maps*. In the case of a colour image, the input to the first layer can be thought of as an input of three feature maps; each a 2D array

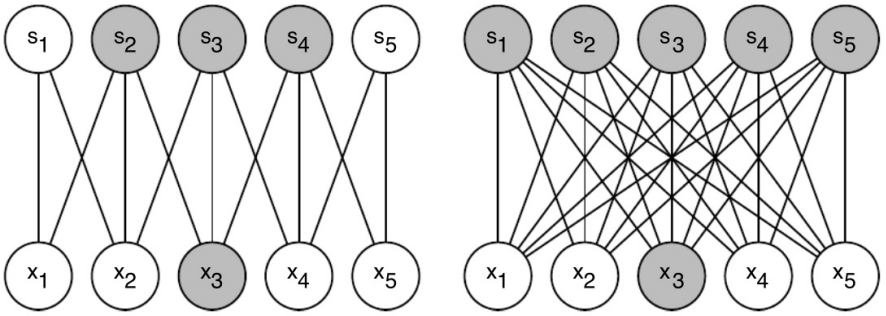


FIGURE 2.37: MLPs with full connectivity between layers L_1 and L_2 have $N_1 \times N_2$ parameters, where N_i is the number of neurons in layer i. For two layers of five neurons each there will be $5 \times 5 = 25$ connections. However, if neurons in L_2 may only connect to a local subset of neurons there is a significant reduction in the number of parameters $3 \times 3 + 2 \times 2 = 13$ connections [38].

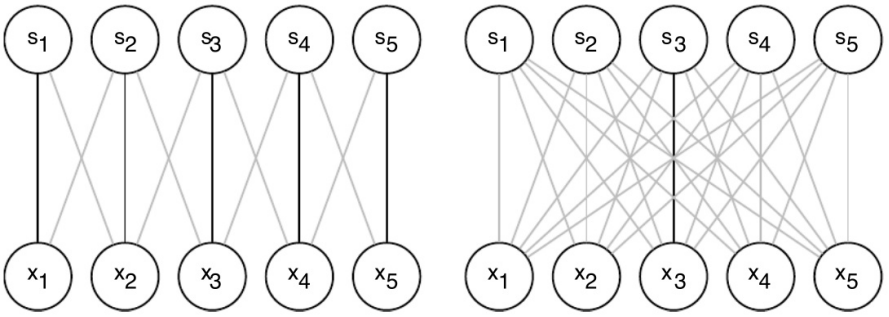


FIGURE 2.38: The network on the left makes use of parameter-sharing; bold lines show all the connections equal to the x_3 to s_3 weight. In total there are three unique weights. In contrast, the fully-connected network on the right has $5^2 = 25$ unique weights [38].

containing a colour channel of the image. For audio, input is a 1D feature map, for video or a volumetric image (such as an MRI) the feature maps are 3D. Each feature map represents a particular feature at all locations over the image, such as vertical lines. A CNN alternates between two types of layers, *convolutional* and *pooling*, which together make up a single CNN stage [62]. There will be several such stages followed by a set of fully connected neurons to form a classification module [85].

CNNs have been successfully used in many commercial projects including Optical Character Recognition (OCR), handwriting recognition (including Arabic and Chinese), video surveillance [25] and speech recognition [30] [80] [100] [101] [102] [103] [104]. The first commercial deployment was for cheque-reading ATM machines in Europe in 1993 [25].

2.5.1.2 Convolution

The value of each pixel in a feature map is the result of connecting a small defined region of the input below to a single neuron in the layer above, refer to Figure 2.39. The

weights used in this connection, called the filter, move over the whole image to generate different pixels in the feature map. Every filters applied will produce its own feature map [28] [85]. Expressed as a vector equation with pixel location indices suppressed,

$$y_j^{(l)} = f\left(\sum_i K_{ij} \otimes x_i^{(l-1)} + b_j\right) \tag{2.55}$$

where $y_j^{(l)}$ is the j'th feature map of the l'th convolution layer, and f is a non-linear activation function [28] [62] [85]. K_{ij} is a trainable filter (or *kernel*) that convolves with feature map $x_i^{(l-1)}$ from the previous layer to produce a new feature map in the current layer, illustrated in Figure 2.40. Symbol $\otimes$ is the discrete convolution operator and b_j the bias term. Each filter K_{ij} can connect to some or all of the feature maps from the previous layer. The weights K_{ij} are trained via back-propagation. As all kernel weights are learned via back-propagation CNNs can be understood to be synthesizing their own feature generators [86].

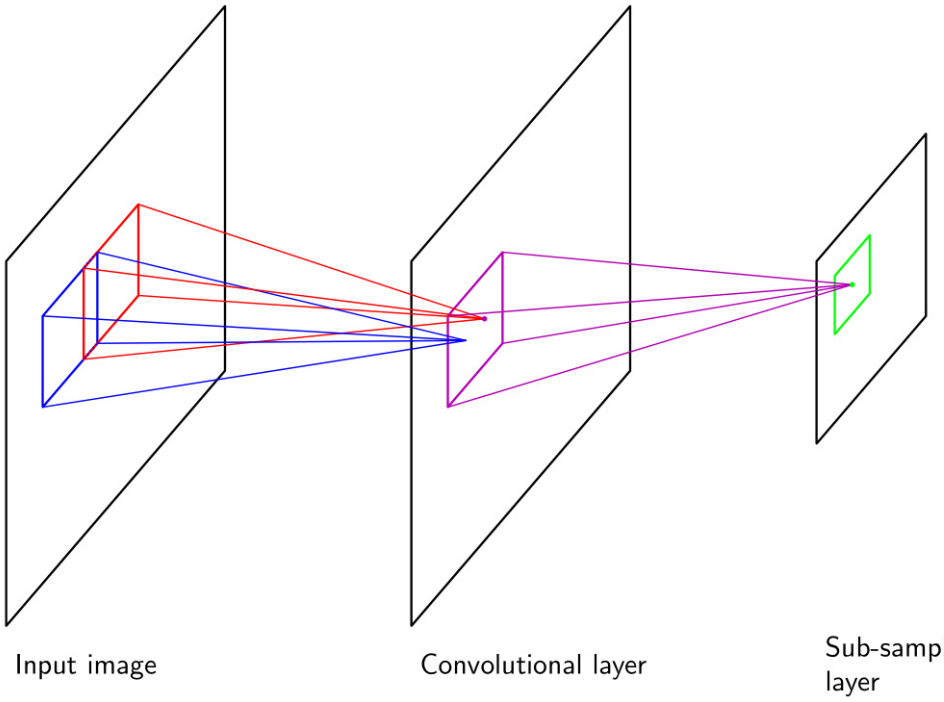


FIGURE 2.39: A CNN is made up of many alternating convolution and pooling layers. Each convolutional layer can make use of many kernels. The output feature maps are then subjected to a sub-sampling layer, known as pooling. A convolution and pooling stack together form one of several successive layers of a CNN [105].

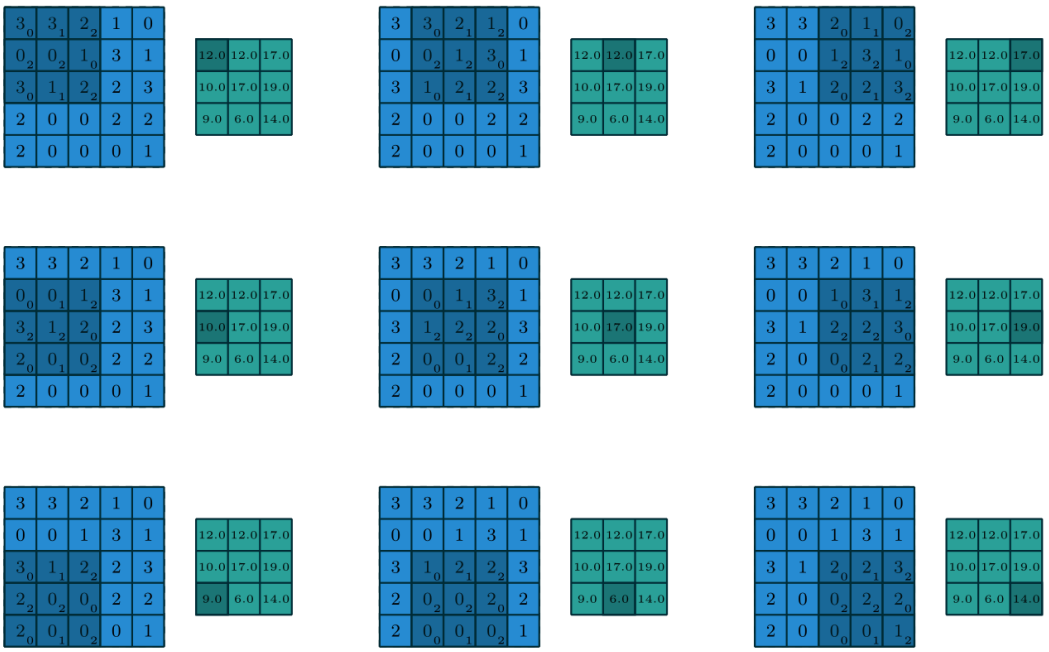


FIGURE 2.40: Activations from layer 1 (the blue grid) are convolved with the kernel (shown by subscript values in the convolution window) to produce a weighted sum for every neuron in the layer above (the green grid). As the kernel size is 3×3, a convolution with a single block stride will produce a 3×3 grid. This can be altered by padding the input with zeros [106].

2.5.1.3 Pooling

Following convolution, pooling is applied. The role of pooling is to merge semantically similar features into one. For example, the relative positions of the features forming a alphabetic letter can vary. A way to reliably detect the feature can be done by coarse-graining (reducing the spacial resolution of) the position of each feature in a manner that preserves relevant information whilst removing sensitivity to noise, translations and distortions [25] [80] [86].

The pooling layer subdivides the convolution layer output into regions of size $z \times z$ known as *pooling windows* [62]. Each window is reduced to a single value with the *pooling function*, see Figure 2.41. Pooling windows may be overlapping but this has empirically been shown to increase over-fitting [25] [30].

The net effect of weight sharing in the convolutional layer followed by a pooling scheme provides the CNN with natural translation invariance properties [80], see Figure 2.42. The pooling function is usually *max-out*. This is the maximum, $\max(f_i)$, where f_i refers to all elements in the pooling window [25] [85].

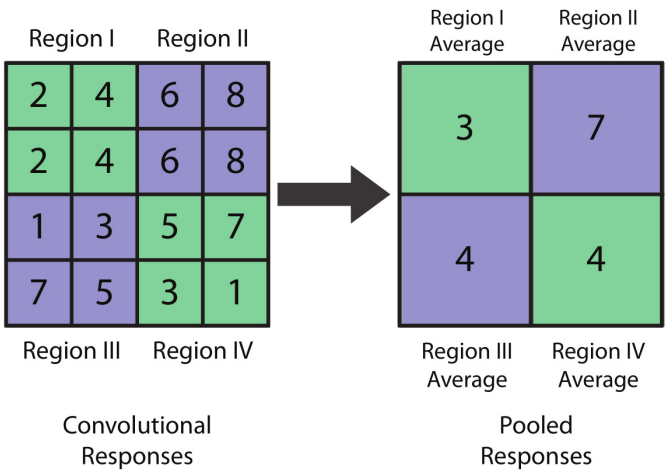


FIGURE 2.41: Convolution responses are pooled with an averaging function using non-overlapping windows [107].

Max-out serves to:

1. Pick out the highest activation in a local region, thereby providing a small degree of translational invariance.
2. Reduce the number of activations for the next layer by a factor of z^2.

Alternatively, the pooling function may be the average, $\sum_i(f_i)/z^2$, as in Figure 2.41. Typically, pooling window size and stride are hand-designed, however, algorithms exist which treat the pooling window size as a hyper-parameter to be optimised [25] [66].

2.6 Measuring Performance

Two primary sources of data exist when producing a model, a training and test-set. The training-set is used to tune the model parameters; the model learns from training data. However, in order to produce an independent measure of the model's performance a test-set must be used. It may be quite intuitive to think a model's success can be completely measured by the accuracy of predictions on the test-set, however, this does not tell the whole story.

2.6.1 Confusion Matrix

If 80% of the test-set are dogs, and 20% are not, then a model can get an accuracy of 80% just by guessing 'dog' every time; clearly not a very useful model. This motivates

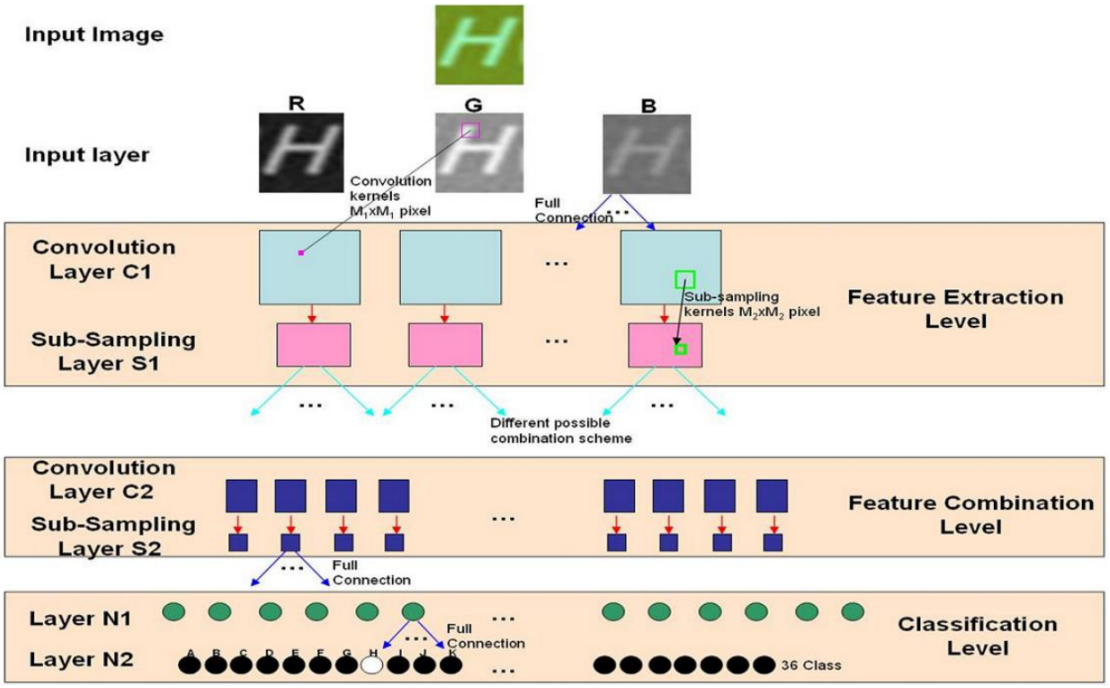


FIGURE 2.42: A standard image will have three layers of intensity values for the red, green and blue pixels respectively. A stacked tensor of RGB values can be convolved with a 3D kernel or, as in the case above, each layer is convolved with a 2D kernel separately. All produced feature-maps are subsequently sub-sampled with pooling. CNN architectures may vary in many ways such as kernel shape, pooling window size, pooling functions and the number of layers. After several stages there are many small feature maps. Commonly they are connected to an MLP used for classification or regression tasks [83].

the use of a *Confusion Matrix*. A confusion matrix is a table counting the occurrences of all the possible model prediction outcomes. For every classification there are four possible outcomes. If the model correctly predicts 'dog', it is called a true positive (TP); if incorrectly classed as 'not a dog' then it is a false negative (FN). On the other hand, if the model correctly predicts the object to *not* be a dog it is a true negative (TN); but if classified as a dog when it is not, then it is a false positive (FP).

We can construct a confusion matrix, also called a *contingency table*, by counting all the outcome types from the test-set. Each row in the table refers to the true classes and each column to classifier predictions [8], see Figure 2.44.

From the confusion matrix we can calculate a range of performance metrics. Accuracy is the simplest

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2.56}$$

that is the sum of the correct predictions over the total.

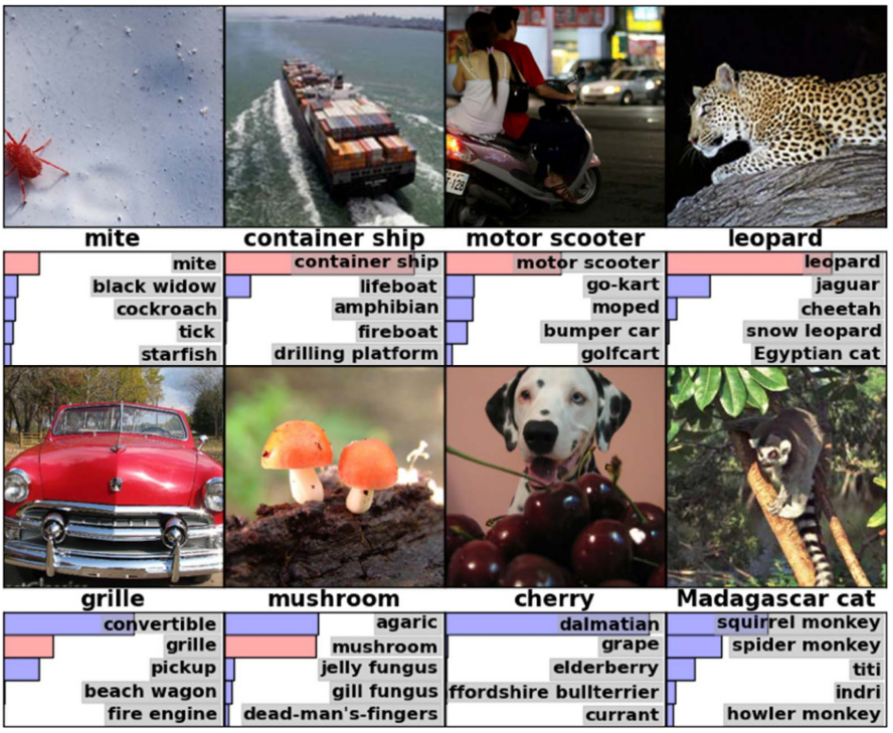


FIGURE 2.43: A CNN was trained as part of the ImageNet Recognition Challenge [73]. The data set was composed of approximately 1000 images for each of 1000 categories. Shown above are the top five predictions for test images. The true label is coloured red and the degree of confidence is indicated by the bar width. The model achieves a top-1 and top-5 error rate of 37.5% and 17.0% respectively. A top-5 error rate is the fraction of test-objects not seen in the top five model predictions. Even where the top-1 is wrong the top-5 predictions seem reasonable. In fact, on the bottom row the model appears to be more accurate than the provided labels [30].

		Assigned Class	
		Positive	Negative
Actual Class	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

FIGURE 2.44: A confusion matrix encapsulates the success of a binary classification model. Ideally all true positive classes will be labelled as positive, *True Positives*; and all negative classes as labelled as negative, *True Negatives*. These will only fill in the table's backward diagonal. The forward diagonal shows cases in which the prediction was incorrect. If predicted positive where truly negative a *False Positive* is committed. Alternatively when predicted negative but is truly positive, a *False Negative* has been committed [10].

2.6.2 Precision and Recall

Two other useful measures of performance are *precision* and *recall* [8] [108]. Recall - also known as *Completeness*, *Sensitivity* or the *True Positive Rate* (TPR) - is the fraction of objects truly belonging to a category that are correctly classified as such,

$$\text{Recall} = \text{Completeness} = \text{Sensitivity} = \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.57}$$

The False Positive Rate (FPR), or 'Type 1 error', is the fraction of positive classifications which are incorrect and is given by

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{2.58}$$

which relates to the *Specificity*, or the fraction of the negative class correctly classified,

$$\text{Specificity} = \frac{\text{TN}}{\text{FP} + \text{TN}} = 1 - \text{FPR} \tag{2.59}$$

Precision (also known as *Efficiency*) is the fraction of objects classified as a class that truly belong to that class:

$$\text{Efficiency} = \text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.60}$$

The ideal would be to have both precision and recall close to one, however, there is often a trade-off between the two. The preferred balance between precision and recall will be problem-dependent and may hinge on not having too much contamination of the positive class, or the economic costs of incorrect classifications.

In practice, precision and recall are particularly important measures in the case of *skewed classes*. Skewed classes occur in the scenario in which one class significantly outnumbers the other class or classes in the data set. For example, when detecting fraudulent behaviour in online banking, the majority of activity will be normal. Should a classifier predict the activity is normal all the time (clearly a bad classifier) it will still be right most of the time and score a high accuracy.

2.6.3 F-measure

A measure that combines precision and recall to give an indication of the algorithm's success is the F_β measure (for real non-negative values of β) [109] [110]:

$$F_\beta = \left(1 + \beta^2\right) \frac{\text{precision} \times \text{recall}}{\beta^2 \times \text{precision} + \text{recall}} \tag{2.61}$$

The F-score favours precision when $\beta > 1$, is evenly balanced when $\beta = 1$ and favours recall where $\beta < 1$. When balanced we have the *F-measure* or *balanced F-score*,

$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{2.62}$$

which ranges from zero to one.

2.6.4 ROC Curve and AUC Score

The Receiver Operating Characteristic (ROC) curve is an encompassing visualization of a binary classifier's performance [109] [110] [111]. They have long been used in signal detection theory as a way of visualizing and evaluating the trade-off between false alarms and correct hits and are commonly used in the design of medical diagnostic tests which are not 100% accurate.

Consider a case of two classes. Formally each example E is mapped to one of the set $\{p, n\}$ which are positive and negative labels respectively. The classifier or model is a mapping from each example to one of the classes. Some models will do this by producing continuous output, often a number between 0 and 1, to which thresholding is applied. If the output surpasses the threshold it is classified as one, for positive, otherwise zero for negative.

The ROC curve is a plot of TPR on the y-axis versus FPR on the x-axis. By changing the threshold in the classification algorithm one can obtain many pairs of (TPR, FPR) values which on a scatter plot produce an *ROC curve*, see Figure 2.45. For a model that has discrete output, even before any threshold, this will only be a single point on the graph corresponding to the only (TPR, FPR) pair. Such discrete scoring classifiers can often have their internal state converted into a score rather than a discrete outcome which will provide a full ROC curve. An example of this is in the decision tree algorithm that produces its discrete output based on the proportion of one instance over another at the final node. By just using the ratio instead of the most common label a continuous score is extracted.

Inst#	Class	Score	Inst#	Class	Score
1	**p**	.9	11	**p**	.4
2	**p**	.8	12	**n**	.39
3	**n**	.7	13	**p**	.38
4	**p**	.6	14	**n**	.37
5	**p**	.55	15	**n**	.36
6	**p**	.54	16	**n**	.35
7	**n**	.53	17	**p**	.34
8	**n**	.52	18	**n**	.33
9	**p**	.51	19	**p**	.30
10	**n**	.505	20	**n**	.1

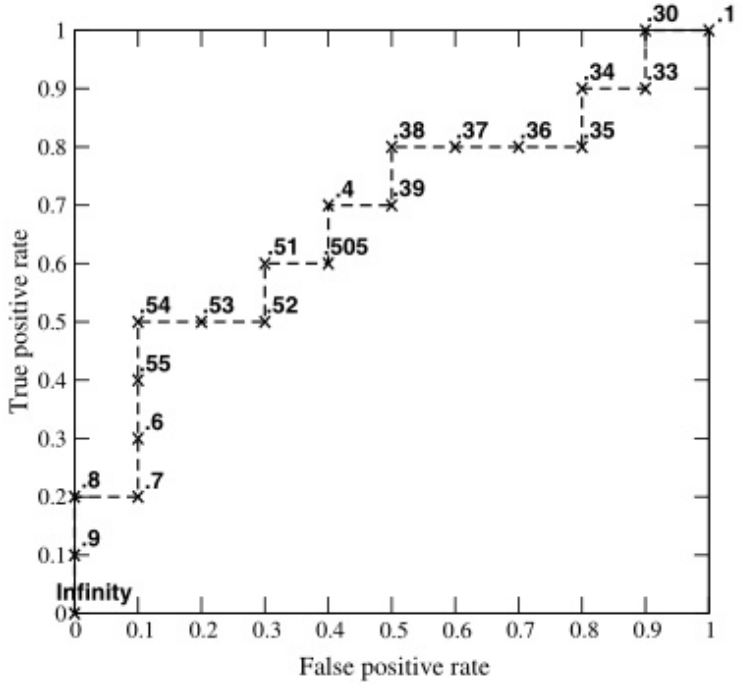


FIGURE 2.45: A classification model that produces continuous output can vary in performance based on the threshold chosen. To provide an unambiguous measure a Receiver Operating Characteristic (ROC) Curve is used. In the example above continuous scores have been assigned to each of the 20 instances where their true class is known. The curve is made by plotting the True Positive Rate (TPR) versus the False Positive Rate (FPR) for all possible threshold values (shown above the crosses). The final threshold selected is problem dependent. In general terms a model with ROC curve that hugs the $(0, 1)$ point is preferred [111].

The ideal model should go through the point $(0, 1)$. That is where there are no false positives and a 100% true positive rate. Classifiers that appear toward the left-hand side of the plot can be thought of as conservative, in that they require strong evidence to make a positive classification. However, their stringent criteria will make few examples classified as positive. Classifiers appearing toward the upper right are quick to label an example as positive, so they are likely not to miss positive classifications. Their lax criteria will likely contaminate the pool of positives with more incorrectly labelled negatives. As there are generally more negative cases than positive, more non-cats then cats, behaviour of the ROC curve toward the upper left tends to be the most informative.

If a model's ROC curve lies along the 45 degree $y = x$ line, it means the classifier is either randomly guessing, or it has not learned a useful data representation. For example, if it guesses positive half of the time it will have a TPR equal to 0.5, however, as now half of the negatives will be incorrectly classified the FPR will equal 0.5 as well. This TPR = FPR trend will be true for any percentage of a random example being positive, yielding a line along $y = x$. In fact, if the proportion of positive to negative classes changes, an ROC curve will not be affected at all. This makes them useful as skew classes are very common in the real world. In some cases the skewness of classes may alter with time, such as the number of infected people over time, but that should not change the fundamental characteristics of what identifies someone as infected by a disease. Should a model appear below the $y = x$ line this means it performs even worse than random guessing. This is not necessarily a bad thing as the model has captured a useful data representation but is making incorrect conclusions. Should label predictions be swapped around, positive to negative and vice versa, the model performance is improved. It will often be the case that the most natural threshold of 0.5, for a model output between zero and one, is not the best. Depending on the model and the problem it may be that selecting the threshold to be 0.6 has a better TPR without increasing the FPR.

A common way of comparing models to each other in a single measure is to use the *Area Under an ROC Curve* (AUC), see Figure 2.46. A perfect classifier would reach a TPR of one and FPR of zero, so the curve would be a straight line along TPR = 1 with an area underneath of 1. A classifier that performs no better than random, a line of $y = x$, would score half of the total area, 0.5. It is still possible for a model with a lower AUC score to perform better than a model of higher AUC score in a particular domain. Therefore, one should not necessarily select the model with the highest AUC score as it depends on the costs of true positives and false negatives. For example, if astronomers make a classifier of abnormal galaxies in order to follow them up with further study, it is more important to have a high TPR, even at the risk of weeding out some abnormal galaxies, than it is to have a low TPR but high FPR. If there are a large number of false positives, the data set is contaminated by a large pool of galaxies that astronomers cannot afford

telescope time to waste on. Here, fewer positives of high quality is better than many positives of low quality. So if a model happens to perform best in the ROC region of interest even though it has lower AUC score it would be the 'economical' choice.

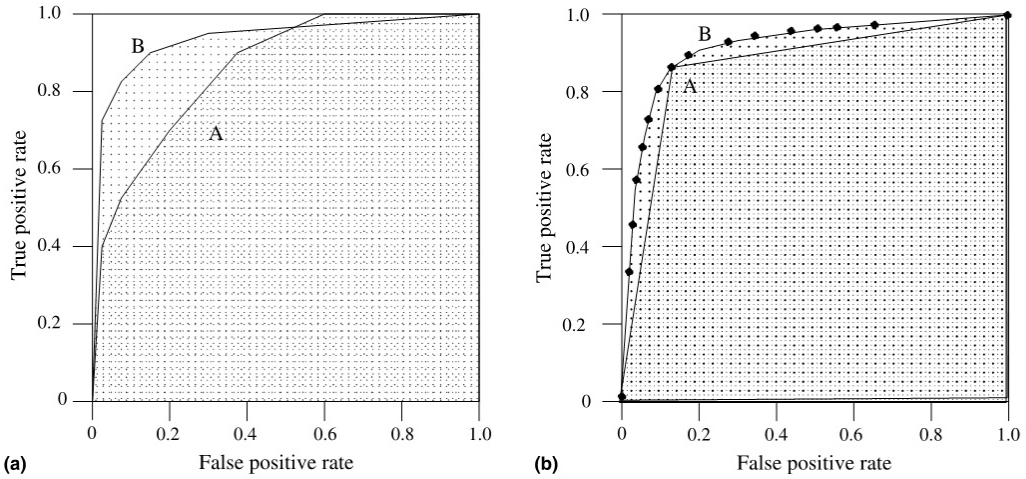


FIGURE 2.46: The Area Under Curve (AUC) score is the area under the ROC curve. A perfect model will have an area of 1 unit^2. This measure is better for evaluating the general performance of one model versus another when the model with higher TPR may vary depending on the FPR domain [111].

Selecting the model with the highest AUC score suffers from the same problem as picking a model of higher accuracy. Without a measure of the variance this score can only be trusted so much. This can be done by testing on multiple test-sets generated via cross-validation and then averaging.

Chapter 3

Research Design and Application

3.1 SDSS Survey

This section supplies the necessary knowledge of supernovae for this thesis. Following this, ML as a tool for handling astronomy data is discussed. Thereafter, we discuss the Sloan Digital Sky Survey (SDSS), the objectives, data and current methodology for data processing. This thesis proposes the use of DL techniques as superior to the more classical ML methods. In this light we discuss results of prior work on the application of ML to supernovae detection.

3.1.1 Supernovae

A supernova (SN), illustrated in Figure 3.1, is a powerful explosive event at the end of a star's lifetime [7]. In fact, so much energy is released that the star briefly outshines the host galaxy [112]. Different supernovatypes may be classified by their spectral features. Type I SNe have no hydrogen lines, see Figure 3.2, unlike Type II SNe which do. Type 1 SNe may be broken up into three subtypes: Type Ia have strong signatures of higher-mass elements such as silicon, sulphur, iron and calcium, but little of hydrogen and helium; Type Ib have prominent helium lines and Type Ic have neither hydrogen nor helium. The identifying features arise from differences in the explosion source. Type Ia are believed to be the result of the thermonuclear explosion of a white dwarf. A white dwarf is the last stage of many stars' lifetimes where the envelope of gases has been expelled to reveal a degenerate core - extremely dense matter that is supported from collapsing in on itself by the Pauli exclusion principle. The other types, Ib, Ic and II, are the result of a star's core collapsing in on itself.

FIGURE 3.1: An image captured by the Palomar Transient Factory of the Pinwheel Galaxy (M101) shows a single exploding star (framed in red), a *supernova*, which can outshine an entire galaxy[10].

Spectral Type	Ia	Ib	Ic	II
Spectrum	No Hydrogen			Hydrogen
	Silicon	No Silicon		
		Helium	No Helium	
Physical Phenomena	Nuclear explosion of low-mass star	Core collapse of evolved massive star (star may have lost its hydrogen or helium envelope during red-giant evolution)		
Compact Remnant	None	Neutron star (typically appearing as a pulsar), black hole or none.		
Light Curve	Standardisable	Large Variations		

FIGURE 3.2: There are several classes of SNe. Type Ia originate from the nuclear explosion of a low mass star and have consistent light curves. Their spectra have no hydrogen but contain silicon. Type Ib has helium, unlike Ic, but both lack hydrogen and silicon. Type II is the only class with hydrogen. Type Ib, Ic and II all originate from core collapse of a massive star and vary greatly in light curve features[113].

The standard theory of Type Ia SNe is that they occur in binary systems where two stars, a white dwarf and a larger companion star, orbit one another. If the companion star enters its red giant phase one, it starts to inflate to the point where some matter is very loosely bound. The free matter leaking off the red giant is then slowly accreted by the white dwarf. The white dwarf's mass will increase, but it cannot do so indefinitely as the nuclear forces preventing collapse can only handle so much. Once the Chandrasekhar limit ($1.4M_{\odot}$, where $M_{\odot}$ denotes solar mass) is reached the core collapses resulting in a thermonuclear explosion which destroys the star within seconds leaving an expanding gaseous remnant rich in metals.

The remaining SNe types occur when a massive star ($M > 9M_{\odot}$) can no longer withstand its own gravitational pressure and collapses. The imploded core can remain either as a neutron star (usually in the form of a pulsar) or collapse further into a black hole, or be entirely destroyed. SNe differ not only in spectra but light curves too. A light curve is a plot of a supernova's intensity over time. The light curves of types Ib, Ic and II SNe can vary substantially depending on the progenitor's metallicity and mass, as shown in Figure 3.3. As a Type 1a SNe occurs when a white dwarf's mass reaches the critical limit; they have consistent light curves. It is for this reason that Type Ia

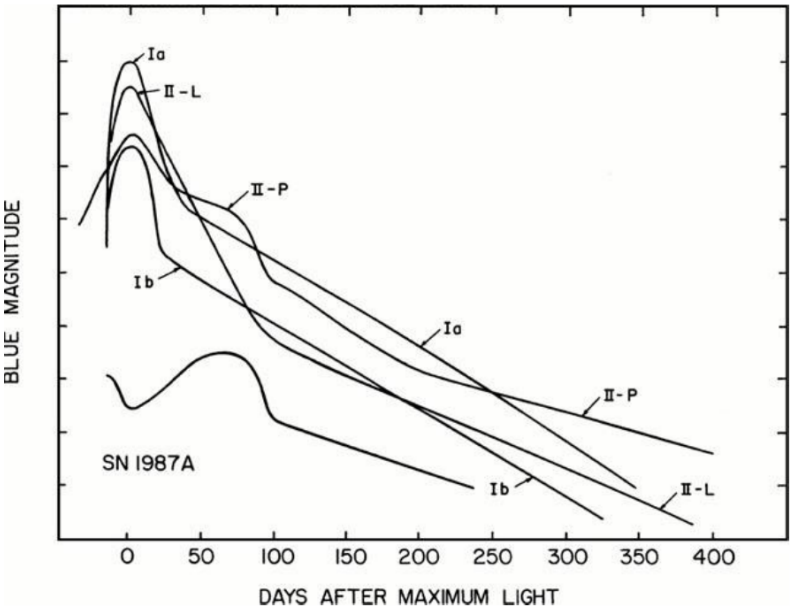


FIGURE 3.3: Star mass, metallicity and other factors cause SNe to have different light curves. Even within different SNe types there is variation with the exception of Type Ia[113].

SNe are particularly useful for cosmologists to measure distance and velocity. If one knows how bright a candle burns and how dim it appears, one can tell how far away the candle is. The further away the SN the dimmer it will appear. In addition, the obtained

spectra will be appropriately red-shifted in accordance with their recession from us due to the expansion of the universe. Type Ia light curves and spectra allow for exactly this on a cosmological scale, making them *standard candles*, see Figure 3.4. In this manner the accelerating expansion of the universe was discovered[114][115] which has since motivated even larger SNe surveys.

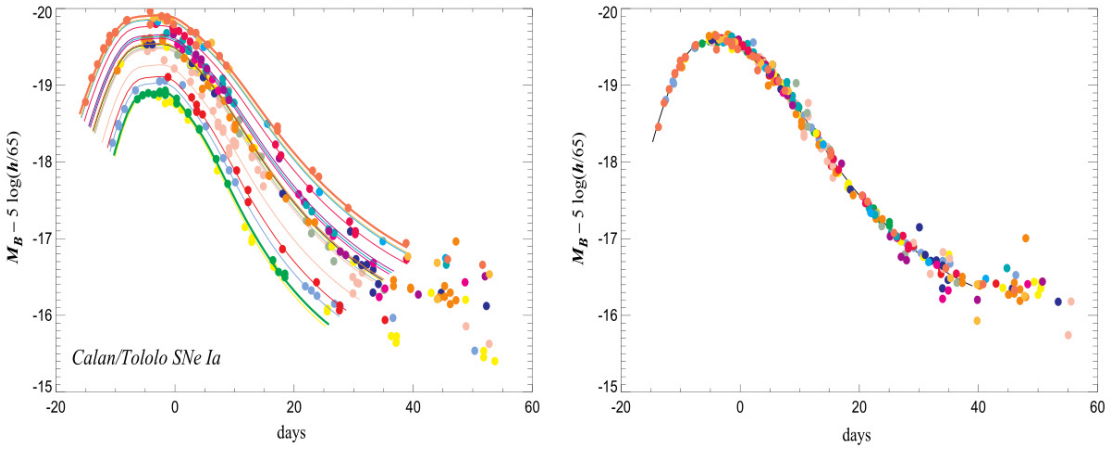


FIGURE 3.4: A SN event can be seen for weeks. A time-series of the light intensity is known as a *light curve*. On the left several Type 1a SNe light curves are shown. Type Ia trigger conditions are so similar as to produce nearly the same light curves viewed at the same distance. As they are equivalently as bright as one another the change in flux, and Doppler shifted light-curves can be used to re-scale the light curves (on the right) and infer their distance and velocity; making them useful *standard candles*[116] in cosmology[117].

3.1.2 The SDSS Survey

The Sloan Digital Sky Survey (SDSS-II) was a spectroscopic red-shift survey which ran from July 2005 to July 2008 as part of an extension to SDSS-I. The SDSS Supernova Survey(SDSS-II SN) was one of three components (along with the Legacy and SEGUE surveys) which ran over three three-month campaigns in the Autumn of 2005, 2006, and 2007. The SDSS-II SNe Survey was commissioned to address the following goals[118]:

1. Refine the SNe Type Ia Hubble diagram. At the time of the survey the Hubble diagram was constructed from low ($z \approx 0.1$) and high red-shift ($0.3 < z < 1$) Type Ia samples from multiple telescopes of differing passbands and selection criteria, which introduce systematic errors. The SDSS-II SNe Survey would gather distance estimates for Type Ia SNe in the sparsely populated red-shift range $0.05 < z < 0.35$ to better constrain the Hubble parameter.

2. Minimise SNe systematics. At the time most SNe surveys had systematic errors comparable to their statistical errors. Systematic errors in the SDSS-II SNe Survey

would be significantly reduced. Photometric calibration errors of 1% in Stripe 82 were achieved from the many years of the large-scale calibration of the data during SDSS-I. In addition, the spectroscopic filters had well-measured transmission curves and were all situated on the same stable camera.

3. Fix rest-frame ultraviolet light curves. SNe at red-shift $z > 1$ have their light curves matched to templates of the rest-frame light curve to reduce systematic errors. For example the 3600 Å (the u-band) in the SN rest-frame corresponds to $\approx$ 8000 Å at a red-shift of 1.2. At the red-shift of $z = 0.3$, the SDSS would be capable of observing this region at 4700 Å, the g-band. The measurements can then be used to improve template data in the rest-frame ultraviolet region.

4. Explore photometric methods to measure SNe characteristics. Spectrum measurements of candidates are a luxury available only when other telescopes have free time and favourable conditions. The SDSS large repository of photometric data would allow a search for a more practical photometric method of SNe classification and red-shift determination.

5. Study SNe types, rates and host galaxies. SNe rates of occurrence per galaxy could be estimated for the different SNe types. In addition, the SNe host galaxies identified could be studied for information regarding progenitor properties.

6. Probe for rare and interesting objects as yet unknown to astronomy.

The SDSS 2.5 m telescope and imaging camera, illustrated in Figure 3.5, are located at the Apache Point Observatory in New Mexico. This produces photometric measurements in each of the u, g, r, i and z spectrum filters spanning wavelengths of 350 to 1000 nm, see Figure 3.6. SNe are difficult to detect in the u and z filters except at low red-shifts ($z \approx 0.1$ for Type Ia) due to the relatively poor filter throughput. Beyond a red-shift of 0.4, Type Ia can still be detected, but after $z \approx 0.2$ the ability to obtain high-quality photometry deteriorates quickly. During the survey, the camera is operated under a *rolling search*. This is where a portion of the sky is repeatedly scanned to discover new SNe and measure their respective light curves.

The SDSS telescope camera is comprised of six sensor columns each of which contains five CCD chips corresponding to the r, i, u, z, and g filters. The telescope is oriented such that a patch of sky drifts through the camera's field of view; known as *drift-scanning*. The focused image will drift across the columns providing 55 seconds worth of exposure to each CCD. Sensor readings for a single point in the sky cannot be taken simultaneously as not all CCDs are exposed to that point concurrently. Instead CCDs are read out in sync with the drifting i.e. 55 seconds apart. Such drift-scanning allows the telescope to image long continuous strips of the sky.

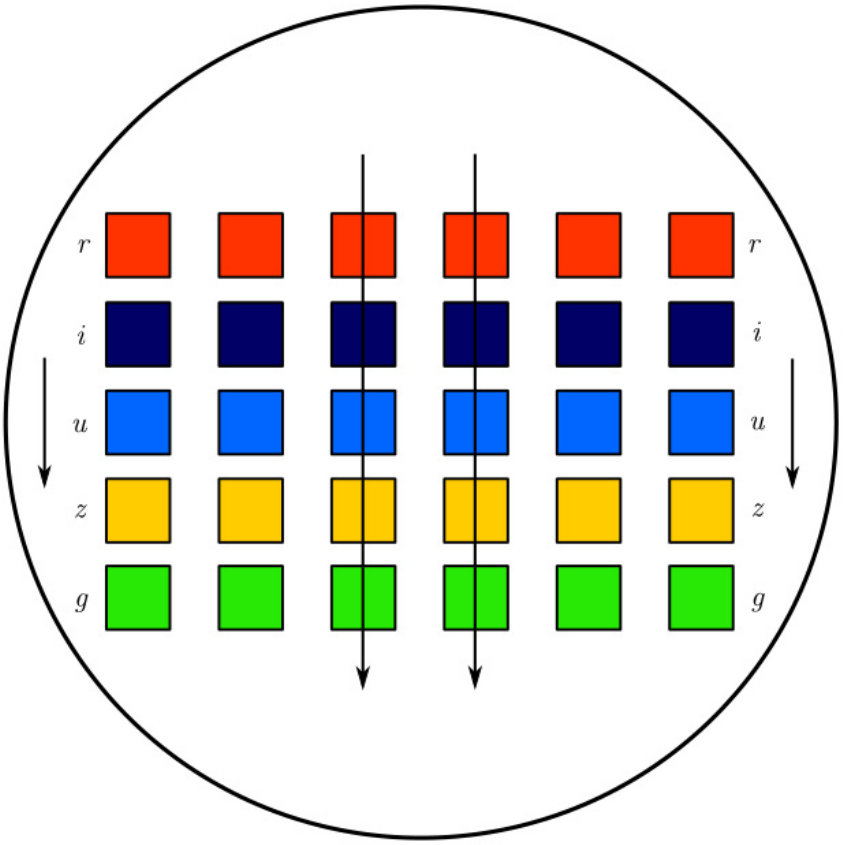


FIGURE 3.5: The SDSS photometric camera has five CCD chips for each of the r, i, u, g and z bands in each of six sensor columns. The camera is oriented such that by the natural rotation of the sky the point of exposure drifts down over the column, taking 55s to cross over each of the rows[113].

Exposure time per unit area limits the survey from operating over the whole sky. Instead only Stripe 82, shown in Figure 3.7, was observed. This is a 300 deg^2 area along the celestial equator, 2.5° wide in declination and between a right ascension of 20 h and 04 h. It takes about two nights to cover all of Stripe 82 in this way. However, the average *cadence* (frequency of observation) was approximately four nights as a result of bad weather conditions and interference from moonlight. Stripe 82 was used to take advantage of the already extensive object databases, reference images and photometric calibration created by SDSS-I. As both surveys used the same telescope for photometrics this minimised systematic errors that may occur from differing photometric standards on different telescopes.

SDSS imaging software[119] was used to process the raw images and SNe were identified via a frame subtraction technique[120]. Each recent image, the *search image*, would be subtracted from the first image taken of the same location, the *reference image*, resulting in a *difference image*, see Figure 3.8. This was only done in real-time using the g, r and

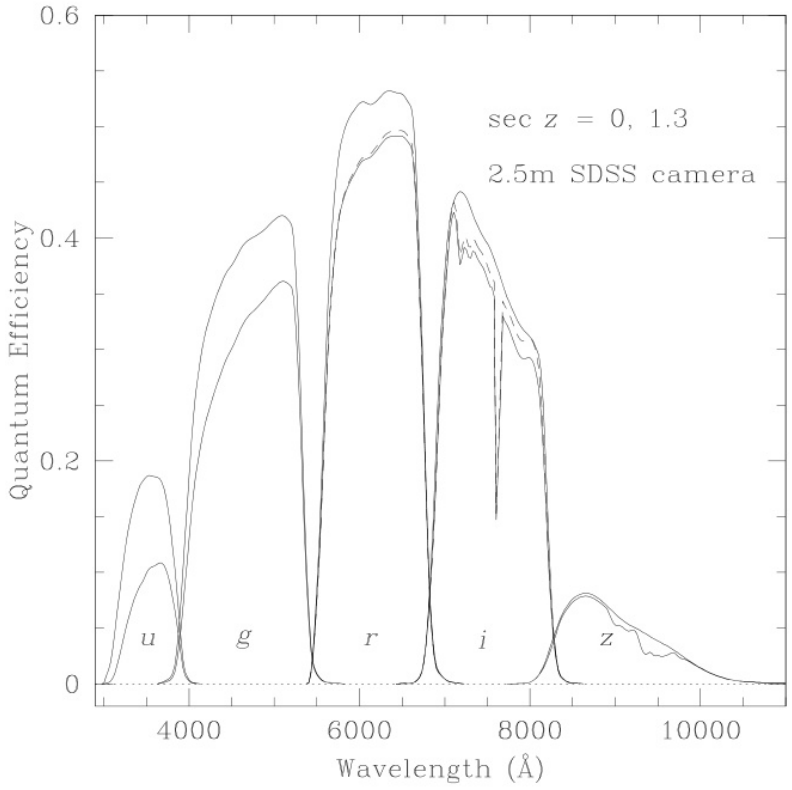


FIGURE 3.6: The upper curves are the quantum efficiencies for light absorption for each of the u, g, r, i and z filters ignoring atmospheric effects. The lower curve includes the atmospheric effects assuming an air mass - the optical path length through Earth's atmosphere - of 1.3. Further scattering within the chips affects only the r and i bands with the corresponding response curve given by the dashed lines[119].

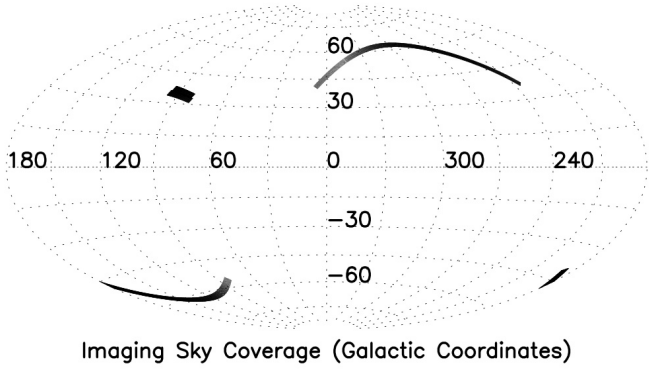


FIGURE 3.7: The SDSS survey is limited in the area of the sky that it can scan by both being ground-based and requiring long exposure times. Shown above is Stripe 82 which was scanned roughly every four nights.[119]

i bands as they are most useful for SNe detection. Using just three bands allows for a more easily interpretable false-colour image to be made. In principal, unless a transient occurred only noise will be left in the difference image. The SNe candidates would then be filtered by an automated object detection algorithm requiring that difference images have a noticeable difference detected in two or more filters, not coincide with existing catalogued stars or variable objects, and were not seen to be moving during the r and g band exposures. In 2006 and 2007, further software cuts were made to reduce the number of images for hand-scanning. The change significantly reduced the number of moving objects, diffraction spikes, and long-term variable objects by cross-matching with a veto catalogue. Now, single-epoch detections would be sent onwards to manual scanners if they were not moving, bright enough ($M_r < 21$ or $M_g < 21$) and detected in at least two epochs.

In the past, removal of false detections was done manually by astronomers, see Figure 3.9 for the decision-tree. During hand-scanning, obvious artifacts such as in Figure 3.10 were removed. Artifacts can be produced via diffraction spikes, CCD saturation, light bleeding and registration errors among others. However, this hand-scanning process was imperfect and led to a large number of artifacts that constituted 70% or more of the candidates. Even if the transient is genuine it may not be a SN but an asteroid, variable star, cosmic ray or artificial satellite. Figure 3.11 shows a summary of the many different genuine transients. Should other telescopes be available, they can be used to validate the remaining SNe candidates[122]. A team of roughly twenty hand-scanners used the g, r and i-band search and difference images (each having a size of 51×51 pixels) and object history to classify each of the candidate objects into one of ten possible classes: dipoles, artifacts, saturated stars, transients, variables, moving objects, SN Gold, SN Silver, SN Bronze and SN Other[123], examples of these are shown in Figure 3.12. For the SDSS-II supernova survey this meant hundreds or thousands of images were being manually scanned each night. Unfortunately, it has been found that classification is done inconsistently between people or even the same person over time depending on mood and exhaustion. As hand-scanning accuracy is paramount, fake SNe were injected into the pipeline to provide quality control and calculate the average detection efficiency per human[121][124].

Regardless of human accuracy, eye-scanning of future sky surveys such as the Large Synoptic Survey Telescope (LSST) will not be possible due to the millions of detections per night. The LSST is expected to find 1,000 new SNe each night for 10 years; effectively generating one SDSS per day[1][126]. Such big data demands new statistical inference and machine ML for processing and analysis. This concern is what motivates this dissertation on applying DL techniques to the detection of SNe.

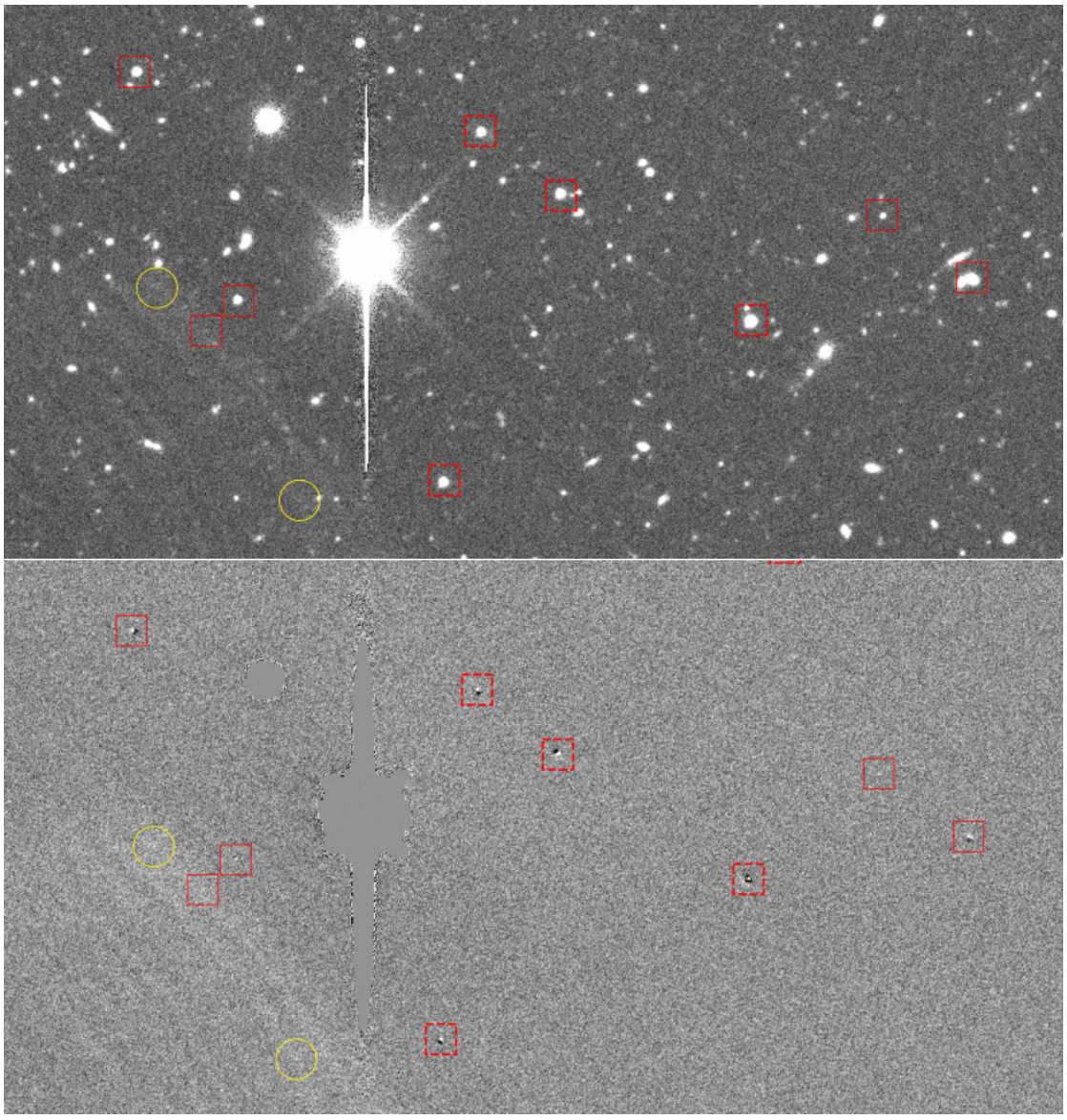


FIGURE 3.8: Shown above is the difference image for the *i*-band in deep field C3 on 13 October 2013. Noted transients are outlined in red. Objects which failed initial detection cuts are outlined in dashed red squares. Objects in solid red squares passed the cuts but were then ruled out by an *autoScan* algorithm. The objects outlined in yellow circles path both tests and are potential SNe candidates[121].

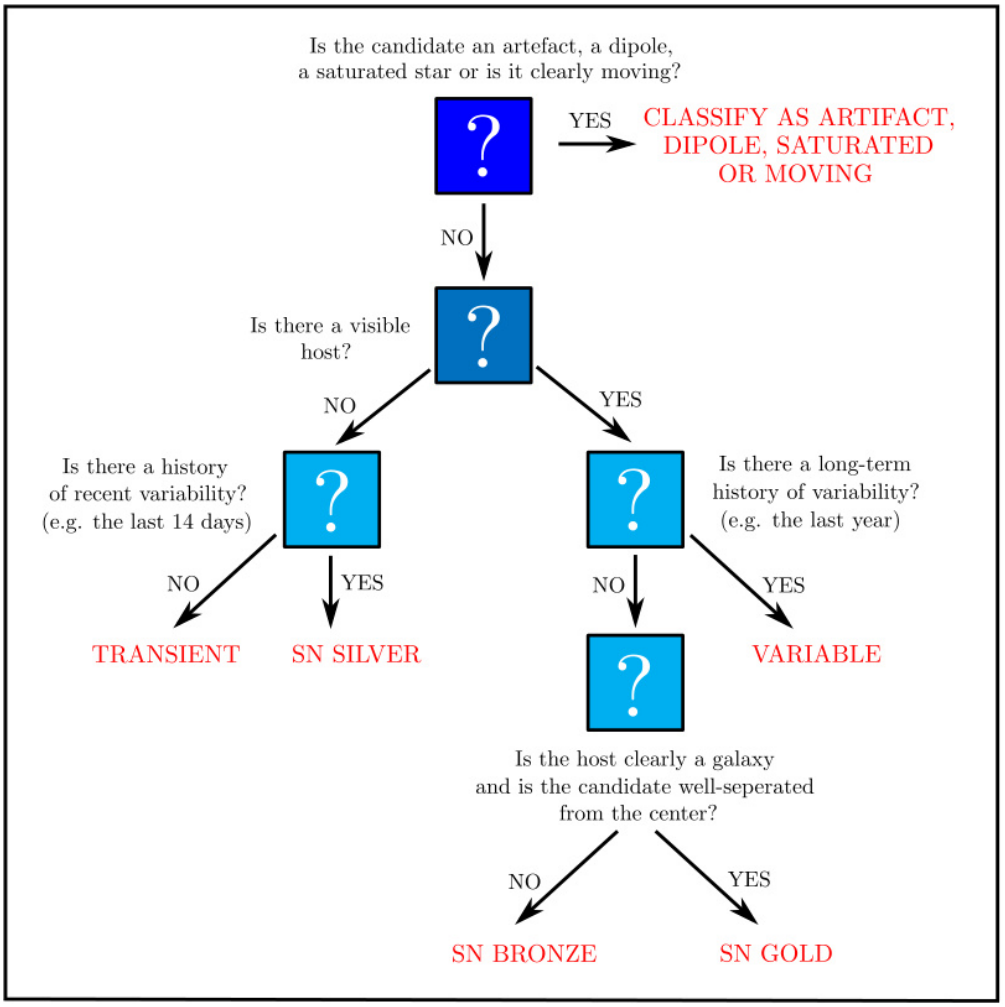
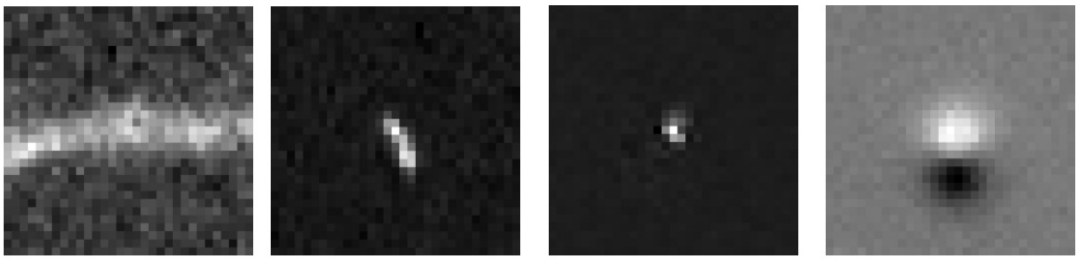


FIGURE 3.9: Hand-scanners labelled the transients in the data set according to the decision tree above[113].



(a) satellite's crossing

(b) cosmic ray

(c) subtraction failure, too faint

(d) high proper motion stellar object

FIGURE 3.10: Many apparent transients are imaging artifacts or are not SNe. Each of the difference images above show artifacts and their source of occurrence[113].

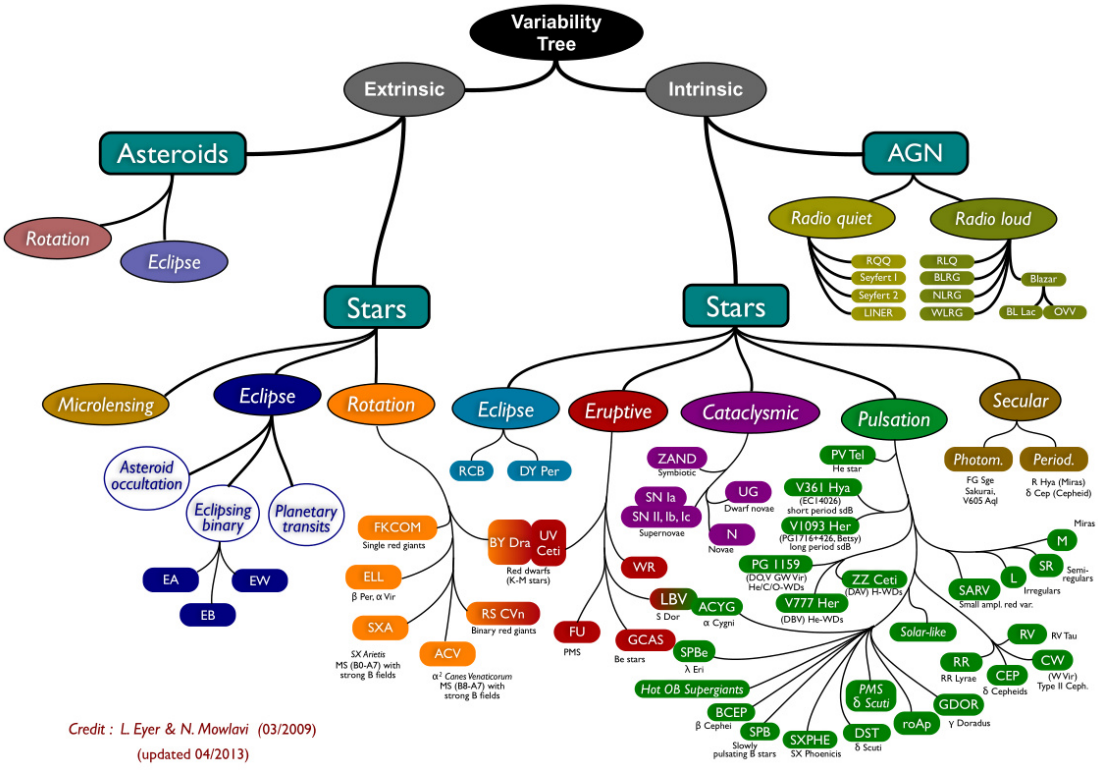
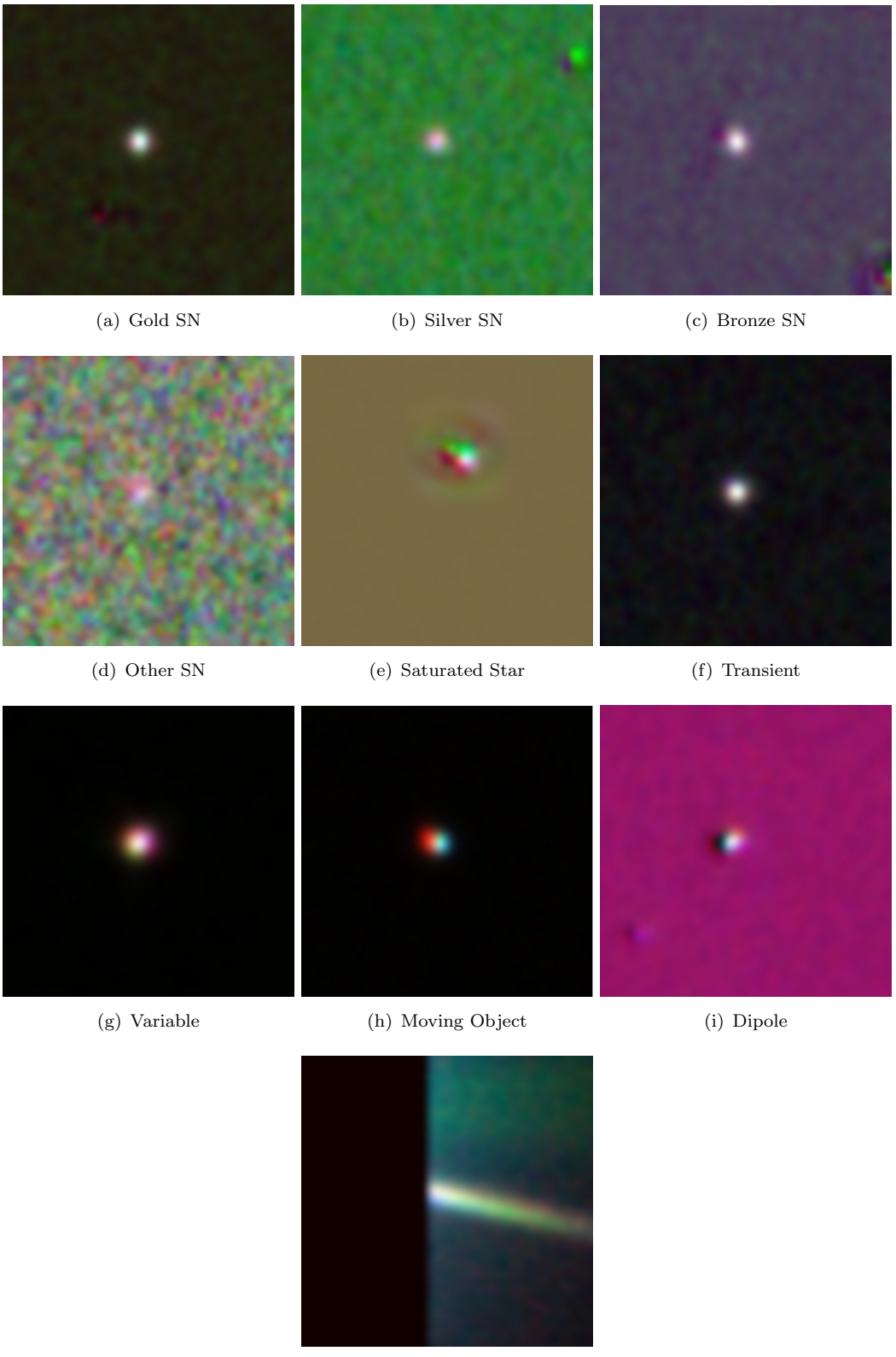


FIGURE 3.11: The variability tree above summarizes the various potential sources of detected transients[125].

Throughout the SDSS-II SNe Survey 10,258 new variable objects were discovered and 500 Type Ia were spectroscopically identified including 81 core-collapse SNe. The final data release is of all transient sources brighter than $\approx$ 22.5 M, that have no known variability prior to 2004, and exclude known active galaxies. The obtained full data set contains 27,480 objects of which 11,959 are non-real artifacts and 15,521 are real. Data from 2005 were omitted because of the different threshold cuts employed at the time so as not to introduce unnecessary variation into the data set. For classification purposes, the original classes were regrouped into three new visual classes, *artifacts*, *real objects* and *dipoles/saturated*. Residuals of real objects are point-like (convolved with the telescope beam and atmospheric seeing), artifact residuals resemble diffraction spikes, and dipoles/saturated objects are often quite point-like, typically with negative flux in some part of the image stemming from saturated CCD effects or registration errors.

3.1.3 Previous Results

Prior work on automatic classification occurred as part of the SNe factory where difference image attributes such as position, shape, Full-Width Half-Maximum (FWHM) and



(a) Gold SN (b) Silver SN (c) Bronze SN

(d) Other SN (e) Saturated Star (f) Transient

(g) Variable (h) Moving Object (i) Dipole

(j) Artifact

FIGURE 3.12: Transient Classes use for the SDSS survey.

distance to the nearest object in the reference image are used to discriminate between fakes and transients[127][128].

Recently several classical ML algorithms have been applied to the classification of candidate SNe[7], see Table 3.1. These include Naive Bayes (NB), Support Vector Machines (SVM), K-Nearest Neighbours (KNN), a three layer MLP, Minimum-Error Classification (MEC) and Random Forests (RF). The features used in all these algorithms were derived with several data-reductions methods. Support Vector Machines and Random Forests are discussed in Sections 2.1.3.1 and 2.1.3.2 respectively. K-Nearest Neighbours[129] determines the class of a new point in the feature space by comparing it to the K nearest neighbours and aggregating the class. Naive Bayes[130] makes use of Bayes' theorem in order to classify. Bayes' theorem states,

$$P(y|x_1,\ldots,x_n) = \frac{P(y)P(x_1,\ldots,x_n|y)}{P(x_1,\ldots,x_n)}. \tag{3.1}$$

Assuming that all variables are independent means that,

$$P(x_i|y,x_1,x_{i-1},x_{i+1},\ldots,x_n) = \prod_{i=1}^{n} P(x_i|y). \tag{3.2}$$

Therefore,

$$P(y|x_1,\ldots,x_n) \propto P(y)\prod_{i=1}^{n} P(x_i|y) \tag{3.3}$$

since the denominator $P(x_1,\ldots,x_n)$ is constant. Several classifiers are possible by making different assumptions regarding the distribution of $P(x_i|y)$ but $P(y)$ is just the relative frequency of each class.

In order to explain what MEC a data reduction technique of PCA needs to be explained. Principal Components Analysis (PCA)[131] performs an orthogonal rotation in the complete pixel feature space such that the first axis of the new set of co-ordinates, or the first principle component, is oriented in the direction of greatest variance. The second PCA component direction will have less variance and so on. This also removes any linear correlation between PCA components. Each sample in the data set can be entirely described by using the n PCA co-ordinates instead of the original n features. However, the reducing variance of the PCA components means that earlier PCA components most likely carry more information. If only the first m components are used and the remaining $n-m$ components discarded then the number of features can be severely reduced without losing much information. Five different feature sets were made with 0, 5, 10, 25, 50, 100 and 200 PCA components respectively.

MEC, was a simple method used as a baseline in the paper by Lise et al. Here, PCA is applied to the data belonging to one class at a time. All images are then reduced to 15 components using the PCA transformation derived from each class. These principal components are used to re-produce the original image resulting in as many reconstructions as there are classes. The reconstruction error is defined as the Euclidean distance between the original vector and the reconstruction. MEC determines the class to be the one corresponding to the smallest reconstruction error.

In addition to PCA, another feature reduction method called Linear Discriminant Analysis (LDA) was used[132]. LDA projects the data in a direction that maximises the variance between classes and minimises the variance within a class. LDA can only produce less than n_{classes}. As there are only two classes (real and non-real) each object had only one LDA component as a feature.

By creating feature sets with the varied number of PCA components from the original 51×51 image or the central 31×31 image, either including or excluding the LDA component and using either normalized or non-normalized versions, a total of 56 data sets were created. During training the choice of data set was treated as a hyper-parameter to be optimized with a validation-set. The results are shown in Figure 3.13. The perfor-

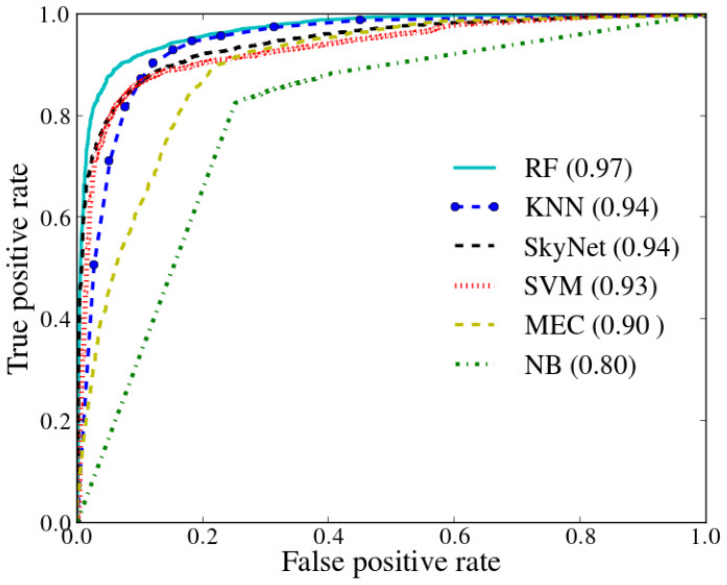


FIGURE 3.13: The ROC Curve and corresponding AUC scores for classic ML methods on the SDSS data set[113].

mance results for each of the classifiers on the fake subset of SNe is listed in Table 3.1. Performance on the spectroscopically confirmed SNe sample are shown in Table 3.2.

Machine Learning Algorithm	Recall
RF	**0.97**
KNN	0.96
SkyNet	0.96
SVM	0.94
MEC	0.96
NB	0.90

TABLE 3.1: Previous performance results on the fake subset of SNe[113].

	RF	KNN	SkyNet	SVM	MEC	NB
SN Ia	0.90	**0.94**	0.92	0.90	0.91	0.83
SNe	**1.00**	**1.00**	**1.00**	0.89	**1.00**	0.63
All	0.91	**0.95**	0.93	0.90	0.93	0.80

TABLE 3.2: Previous results on the spectroscopically confirmed SNe[113].

Set	Count	Percentage
Training	14 425	52.50%
Validation	6 176	22.48%
Testing	6 874	25.02%

TABLE 3.3: Data split for training, validation and testing.

Whilst these results are impressive, DL offers to bypass the laborious creation of feature sets and CNNs have shown impressive results on image recognition tasks. The rest of this dissertation will focus on the application of CNNs to the data set in an effort to improve on the state-of-the-art.

To make the clearest comparison between previous results and this research the exact same data-split was used. 25% is reserved for the test-set and 75% of the remaining data are used for training, leaving the rest for cross-validation, see Table 3.3.

3.1.4 SDSS Data

Looking only at the class distribution, shown in Table 3.4, we see it is distributed heavily in favour of artifacts and bronze SNe with 28.18 and 23.91% respectively. Saturated stars and variables together make up approximately 3% of the data. In order to maintain the comparison between this work and previous work this thesis looks at the binary classification task of real and non-real transients. Re-classifying objects into their real and non-real categories, the binary distribution is more even as seen in Table 3.5

In order to better understand the high-dimensional data distribution t-distributed stochastic neighbour embedding (t-SNE) is used. t-SNE is a non-linear data reduction algorithm

Class	Type	Count	Percentage
Artifacts	Non-real	4 065	28.18%
Moving Objects	Real	909	6.30%
Saturated Stars	Non-real	253	1.75%
Dipoles	Non-real	1 961	13.59%
Variables	Real	186	1.29%
Transients	Real	1 095	7.59%
Other SN	Real	502	3.48%
Bronze SN	Real	3 449	23.91%
Silver SN	Real	616	4.27%
Gold SN	Real	1 389	9.63%

TABLE 3.4: Distribution of classes in the SDSS transient training-Set

Type	Count	Percentage
Real	8 146	56.47%
Non-real	6 279	43.52%

TABLE 3.5: Real and Non-real transient distribution in the SDSS training-set

developed by Geoffrey Hinton and Laurens van der Maaten[133]. Simply put, it is a technique for embedding high dimensional data into a low dimensional space, usually 2D or 3D. This is done in such a way that nearby objects in the high dimensional space remain nearby in the low dimensional space and the reverse for distant objects This occurs in two stages. In the first stage a probability distribution is created over pairs of objects so that nearby objects are likely to be chosen, and dissimilar points less likely. In the second stage t-SNE defines a similar probability distribution on the low-dimensional map. Then the Kullback-Leibler divergence between the two distributions is minimized.

t-SNE is implemented in the Scikit-Learn Python package[134]. As images in this data set have $51^2 = 2601$ dimensions t-SNE is a great method for providing intuition on the data set. t-SNE is used to reduce the entire training-set of 14 425 images to a 2D plane, shown in Figure 3.14. Only the inner 30×30 window of each image was used to reduce the dimensionality before t-SNE, and to not be misled by noise surrounding the potential transients. Hyper-parameters of t-SNE can make a large difference in the success of the algorithm[135]. Here several perplexity values, (30, 50, 75, 100), are used and the results compared. In this instance no noticeable difference was seen between the perplexities. While some natural separation is seen between real (red) and non-real (blue) classes there appears to be a significant overlap. Several layers of non-linear transformations, such as a CNN, may be able to introduce greater separation between the classes allowing for better classification.

While there appear to be two centers around which real and non-real objects cluster, non-real objects exhibit more variance. This is unsurprising in that there are many more

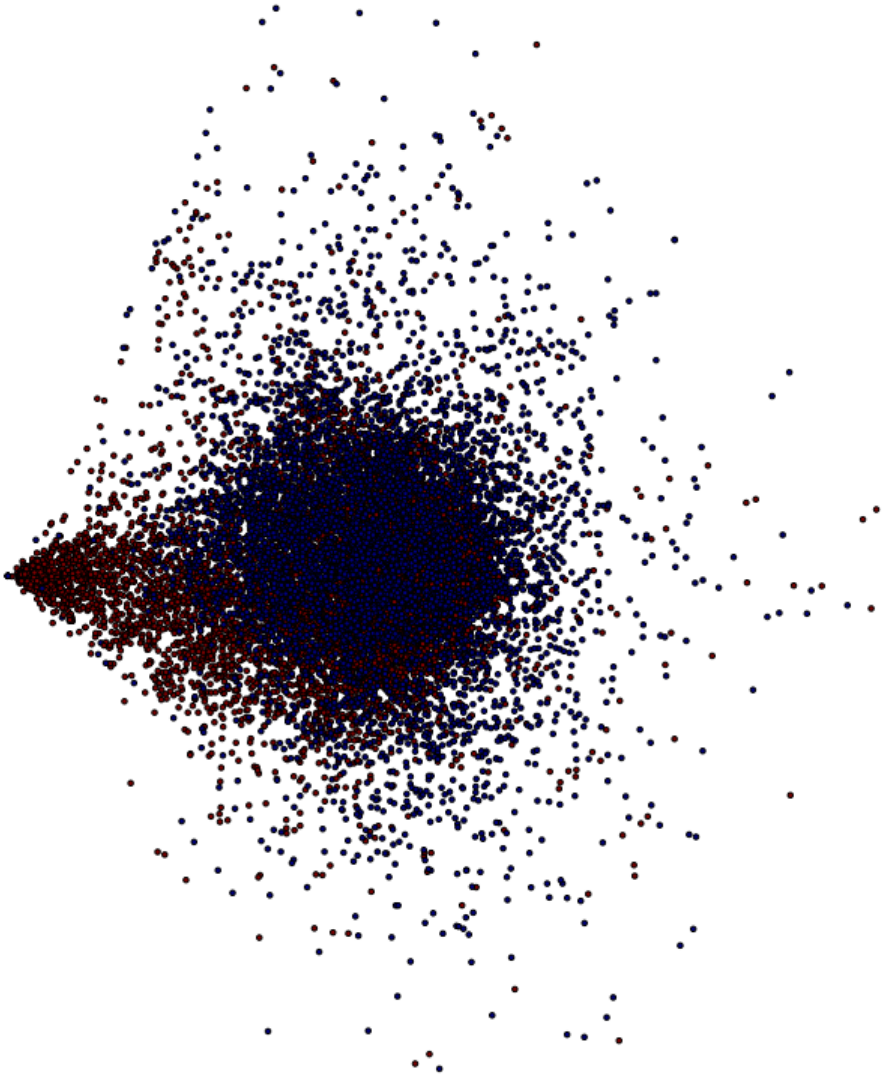
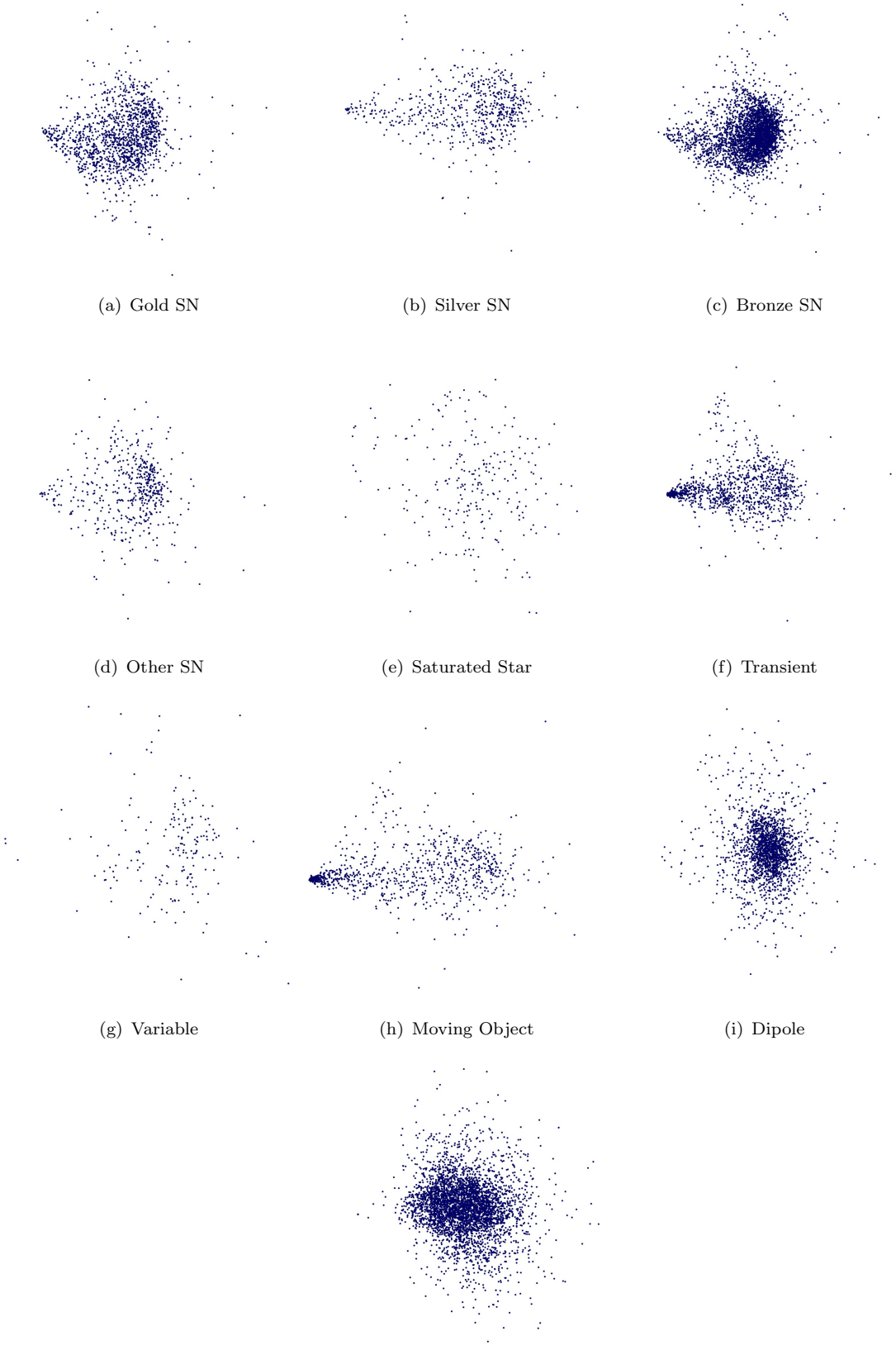


FIGURE 3.14: Shown above is a 2D projection of the central 30×30 window of 14 425 training data samples. There is some separation between real (in red) and non-real (in blue) samples. In addition, there appears to be many outliers as part of the non-real cluster.

ways that a transient is not a SNe than vice versa, a non-real object may appear within a greater parameter volume than SNe. Artifacts in particular may vary drastically from all the possible sources. By the same reasoning we expect SNe to be more clustered.

In keeping with this reasoning there are disproportionally more artifacts at the outskirts, Figure 3.15, of the non-real cluster than toward the centre and all SNe appear most concentrated in a node on the far left. As t-SNE is stochastic, the output is non-deterministic. Due to this all distributions of points on the plane in Figure 3.15 were drawn from the same t-SNE operation but plotted separately.



(a) Gold SN (b) Silver SN (c) Bronze SN

(d) Other SN (e) Saturated Star (f) Transient

(g) Variable (h) Moving Object (i) Dipole

(j) Artifact

FIGURE 3.15: t-SNE plot showing isolated classes.

3.2 Application Design

In order to determine the possibility of using DL for transient detection several architectures and hyper-parameters can be used. Implementation of CNNs for this purpose was done using the Python language including the Theano[64] and Keras[136] packages. Python, as an interpreted language, is not as fast as a compiled language such as C. However, several popular open-source tools for ML are written in Python or have Python wrappers to the faster C++ code below.

Training CNNs is not feasible without the use of a GPU to significantly soeed up the training process. Theano is a Python package which significantly reduces the programming complexity required to use NVidia GPUs without personally digging into Cuda. Theano is a symbolic manipulation library which makes device-agnostic computational graphs. The computational graph may be implemented on a chosen device such as a GPU. The graph allows for symbolic differentiation, required for back-propagation, and significantly reduces programming complexity. Model design and training was all done using Keras. Keras sits at a higher level of abstraction than Theano with pre-built graphs nodes such as a convolutional layer. The abstraction allows for simpler model design by specifying what layers follow which and their respective parameters. Training for all models was performed at the Square Kilometre Array (SKA) offices in Cape Town with a server containing four Titan X GPUs.

Data were obtained in the form of FITS files[137]. In order to convert from this data representation to one Keras would recognize the Python package Pyfits[138] was used. Data directories were recursively read through to create two Numpy[139] tensors for training and testing in the shape of $(n_i, n_c, 51, 51)$ where n_i and n_c are the number of images and colour channels respectively, and 51 is the pixel width and height of all the images. Two different sets of training and testing tensors were generated in this way where the second set stacked the search image behind the difference image as a further three colour channels. Using cPickle, both Numpy tensors were saved in a binary format to reduce size, reduce data processing required for every run and for quick loading into Keras. On loading data to the model, the training-set was further split into a smaller training-set and validation-set using Scikit-Learn[134], a library of ML tools.

In order to evaluate several models and possible configurations, a modular approach was needed for model design. A Keras script was developed which on request returns one of six models with user-specified parameters. The six models are described in the next section, however, all of them would have the following customisable parameters:

- Network Activations - The activation functions used after all convolutional and dense layers.
- Drop-out - Following each convolutional or dense layer an optional drop-out layer could be included. All such layers share the same user-defined drop-out fraction.
- Loss Function - The loss function used in back-propagation.
- Use Search Images - Whether to use only difference images or search images in addition.
- BatchNorm - Similar to Drop-out, each layer had an optional batch normalization layer.
- Optimizer - The weight update method used and internal parameters such as the momentum contribution to SGD.
- Regularization - All layers could have L1 and L2 regularization added with a specified strength.

After the model architecture and parameters have been chosen several global variables were set for Theano:

- device = gpu0 - The device to be used. The device may be 'cpu' or 'gpu' followed by the number of the GPU. By default Theano will use the CPU.
- cnmem = 0.3 - A fraction of the GPU or CPU memory can be reserved for use by only this session. This speeds up training but if the process runs out of memory training will crash.
- fastmath = True - Theano can optimize training by using less accurate calculations, in particular for non-linear functions.
- floatX = float32 - Data types must be 32-bit floats. GPUs only support 32-bit floats and not 64-bit.
- mode = FAST_RUN - Theano needs to compile computation graphs at the start of the session. Compilation may be quick at the cost of unoptimized code, or compilation can take longer but model training and prediction time are optimised.

Following model design and loading, several parameters specific to training were set. In no particular order these are:

- Run Name - A brief overview of what model and parameters were used. This is used as a prefix when saving results, model weights and log files.
- Batch Size - The number of samples within each batch.
- Prediction Batch Size - The number of samples to use in a batch when using the model for prediction.
- Random Seed - Used for reproducibility. A random seed is used in validation data-split, the choice of items in mini-batches, drop-out and weight initialization.
- Number of Epochs - The maximum number of training epochs allowed.
- Early Stopping - Whether or not to include early stopping.
- Patience - The minimum number of training epochs. Early stopping may interrupt training only afterwards.
- Improvement Factor - The factor by which the loss must drop after several epochs in order to continue training.
- Validation Frequency - How often to check validation performance for improvement with early stopping.
- Verbosity - Show accuracy and loss during training.
- Load Weights - The path of model weights used to initialize the model. This is used for renewing training after an interruption. If empty, the model is loaded with random weights.

With Keras, a model may also be loaded with Callbacks. These are functions which are executed after each batch or epoch. They may be used to return preliminary results or alter training parameters on the fly. *Early Stopping* is one such callback. In addition, *ModelCheckpoint* is used to record model weights after each epoch to minimize losses from a system crash or sudden downtime. Developers may also construct their own callback. One created was *LossHistory* which simply records training and validation losses and accuracies after each epoch.

In this implementation data augmentation was optional and individual augmentations could be included or excluded at will. These are:

- Buffer Size - The maximum number of items to pre-generate.
- Predictive runs - The number of predictions the model will use with data augmentation to provide an aggregate prediction. This improves performance by making an ensemble of networks.

- Standardization - Whether or not to rescale features to be centred at the origin and have unit-variance.
- Rotation - Whether or not to include 90° rotations.
- Shift - The maximum fraction of the image that may be shifted horizontally or vertically.
- Flip - Whether or not to include horizontal flips.

Without data augmentation all data are rescaled using the maximum and minimum pixel values such that values lie between zero and one.

In order to maintain quality control, recover from crashes and improve reproducibility, the executed script would be saved to a separate text file under the run name used. Logs of training, validation and testing results would be saved as they occurred. Standard output to the terminal was saved to file in addition to being displayed in the terminal helping track down where errors occurred or where training was cut off abruptly. At the end, both the final model and the best performing model (on the validation-set) would be saved. Performance measures of the models, including accuracy, precision, recall and F1 scores would be saved to file. Using continuous-valued predictions, points for the ROC curves were saved together with the resulting AUC score. Then an ROC curve was plotted for quick and simple model evaluation when tuning hyper-parameters. Finally, the script would send an email providing an alert of which run executed and the total run-time.

3.2.1 CNN Model Designs

As there are currently no theory-based rules for determining the best model architecture, six different CNN architectures were created with varying filter window sizes and number of layers. Three base models (models 0, 1 and 2) were designed with increasing convolution window sizes of 2×2, 3×3 and 5×5 respectively, see Table 3.6. Convolutional window size is maintained through all layers, with the exception of Model 1 where window size is limited so as to produce feature maps of even dimensions for pooling that would not require zero-padding. In this manner, three full convolutional layers (including pooling) were stacked in each of the three base models. Activations from the final convolutional layer are then connected to a single fully-connected layer of 2048 neurons followed by the output layer. Models 3 to 5 are copies of the three base models but have an additional fully-connected layer of 1028 neurons.

Model 0	**Model 1**	**Model 2**	**Model 3**	**Model 4**	**Model 5**
Conv 2×2 64 filters	Conv 3×3 64 filters	Conv 5×5 64 filters	Conv 2×2 64 filters	Conv 3×3 64 filters	Conv 5×5 64 filters
MaxPool	MaxPool	MaxPool	MaxPool	MaxPool	MaxPool
BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm
Drop-out	Drop-out	Drop-out	Drop-out	Drop-out	Drop-out
Conv 2×2 128 filters	Conv 3×3 128 filters	Conv 5×5 128 filters	Conv 2×2 128 filters	Conv 3×3 128 filters	Conv 5×5 128 filters
MaxPool	MaxPool	MaxPool	MaxPool	MaxPool	MaxPool
BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm
Drop-out	Drop-out	Drop-out	Drop-out	Drop-out	Drop-out
Conv 2×2 256 filters	Conv 2×2 256 filters	Conv 5×5 256 filters	Conv 2×2 256 filters	Conv 2×2 256 filters	Conv 5×5 256 filters
MaxPool	MaxPool	MaxPool	MaxPool	MaxPool	MaxPool
BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm
Drop-out	Drop-out	Drop-out	Drop-out	Drop-out	Drop-out
Dense 2048	Dense 2048	Dense 2048	Dense 2048	Dense 2048	Dense 2048
BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm	BatchNorm
Drop-out	Drop-out	Drop-out	Drop-out	Drop-out	Drop-out
Output	Output	Output	Dense 1028	Dense 1028	Dense 1028
			BatchNorm	BatchNorm	BatchNorm
			Drop-out	Drop-out	Drop-out
			Output	Output	Output

TABLE 3.6: Summary of the six model architectures used.

All convolutional layers had optional batchnorm and drop-out layers above. Training of models could take between 6-24 hours, so in order to avoid months of optimization, batchnorm layers were treated as all-or-nothing. Either all layers had batchnorm, or none did. Similarly drop-out fractions for all layers were the same. The number of filters, the choice of model, use of batchnorm and the drop-out fraction were considered hyper-parameters to be determined by performance on the validation-set.

3.3 Data Augmentation

Having both the search and difference images provided a question. While transients are detected with difference images, the reference image can provide additional information. For instance, it could show that a galaxy was already present, or that there was a moving object before (the colours would appear in slightly different positions along a line.) There is definitely information contained that a human could make use of, however, the curse of dimensionality from twice the number of features could negatively impact the model. To determine which data set should be utilized performance on the validation-set was

used. As seen in Figure 3.16, the change in performance between an equivalent model

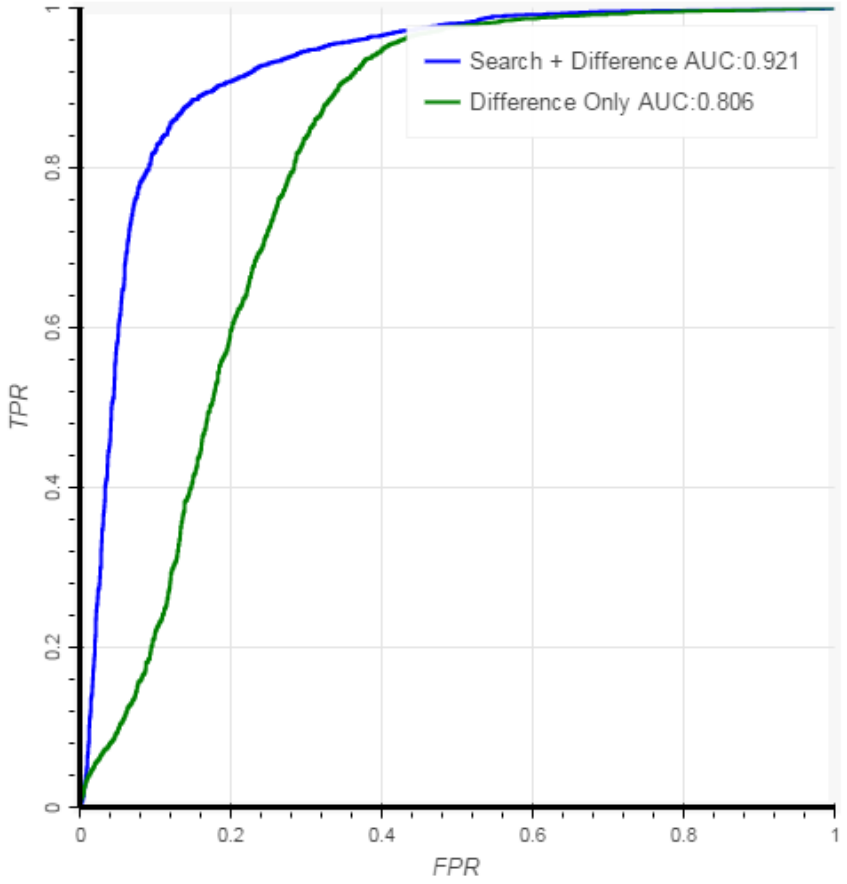


FIGURE 3.16: The difference in performance using only difference images or in combination with search images. Using all the image information results in a significant performance improvement.

with or without search images is significant. Performance results are, unless stated otherwise, all on the validation-set. Using the search images improved performance even with different loss functions. As a result search-images were used for all further hyper-parameter optimization.

3.3.1 Batch Normalization

Batch normalization, described in Section 2.3.1, is a powerful method of improving network performance. To determine whether or not batch normalization should be used, performance on the validation-set was tested. As seen from Table 3.7 batch normalization does improve model performance.

Model Type	AUC Score
Batch Normalization	**0.799**
No Normalization	0.748

TABLE 3.7: Model comparison with and without batch normalization.

Model Type	AUC Score
Normalization and Rotation	0.966
Only Normalization	**0.968**

TABLE 3.8: With the simple addition of rotation and flips we increase the effective amount of data eight-fold. Despite this, there is no statistically significant increase in model performance.

3.3.2 Rotations and Flips

Facial recognition models need not learn to recognize upside down faces as this is an uncommon case and would likely lead to more false positives of upside down faces than true positives. A transient however, has no consistent orientation with respect to us, therefore it makes sense to augment the data set with rotated versions of the original data. Similarly the same would be true of mirrored images. Half of the time the image would be flipped upside down due to the mirror symmetry. Images were flipped only along one axis as a vertical flip combined with 90° rotations can achieve all possible orientations; an additional random horizontal flip would just add unnecessary computation. Common practice in several image recognition problems has been to introduce a random rotation to the image within a range of degrees. However, as these images are small to begin with, allowing any random rotation would introduce the problem of what to do at the corners of the image where there is no data to rotate into. This could be filled with zeros, or perhaps a certain amount of Gaussian noise. In the interests of not having such artificial pixels in the images, it was decided to only make rotations in multiples of 90°, thus providing four times as many images without introducing non-real pixels.

As seen in Table 3.8, there is no significant improvement in performance using both rotation and flips compared to the equivalent model architecture without data augmentation. In fact, there is a small decrease in performance. For other models there was improvement, but it too was negligible and likely not significant. Even though these results suggested rotation and flips were an unnecessary inclusion they are included as the data augmentation step would happen regardless on the CPU which ran parallel with the GPU and so no additional training time is needed.

Model Type	AUC Score
Shift = 0.0	0.967
Shift = 0.1	0.969
Shift = 0.2	**0.97**

TABLE 3.9: Performance comparison with varied amounts of maximum image shifts in data augmentation.

Model Type	AUC Score
No Augmentation	0.921
Full Augmentation	**0.97**

TABLE 3.10: Data augmentation including rotation, shifts and flips significantly increases the effective data set. All these augmented images were computed on-the-fly on the CPU while the previous batch was being computed on the GPU. The unmistakable improvement in performance can be seen in the table above.

3.3.3 Shifts

As a form of data augmentation, images may be shifted horizontally or vertically by a fraction of their dimensions. By providing the model with many instances of the same object appearing at different locations in the image, the model learns to be more invariant with respect to image translations. The maximum degree to which images were shifted was determined with the validation-set. Here shifts of 0, 10 and 20% are tested.

As seen in Table 3.9, improvement is small but steady with increasing maximum shift. Shifts of greater than 0.3 were not used because it was noticed in several objects the transient was quite close to the edge of the image. Shifting too much would remove the transient from the difference and search images altogether. In addition, as there are no pixels to spare surrounding the 51×51 image, any shift would mean the inclusion of artificial pixels. Here such artificial pixels were made equal to the nearest known values. This makes for streaks on a shifted image, which may increase the likelihood of being classified as an imaging artifact.

The benefits of augmentation practices can clearly be seen for Model 0 in Table 3.10 which shows ROC Curves for non-augmented and fully-augmented versions.

3.4 Hyper-parameter Optimization

3.4.1 Batch Size

As discussed in the literature review, there are competing interests with regard to the batch size used in SGD. Smaller batches explore the parameter space better, however, they take more epochs to converge than larger batches. In order to determine optimal batch size, training was done for batch sizes in power of twos to determine the average epoch duration over five epochs. Batch sizes larger than 1024 could not be trained as the GPU has insufficient memory. The smallest batch size that could be used with a reasonable epoch time would be chosen, bearing in mind the number of required epochs may be near 1000 or more but is still unknown at this stage. The results are presented in Figure 3.17.

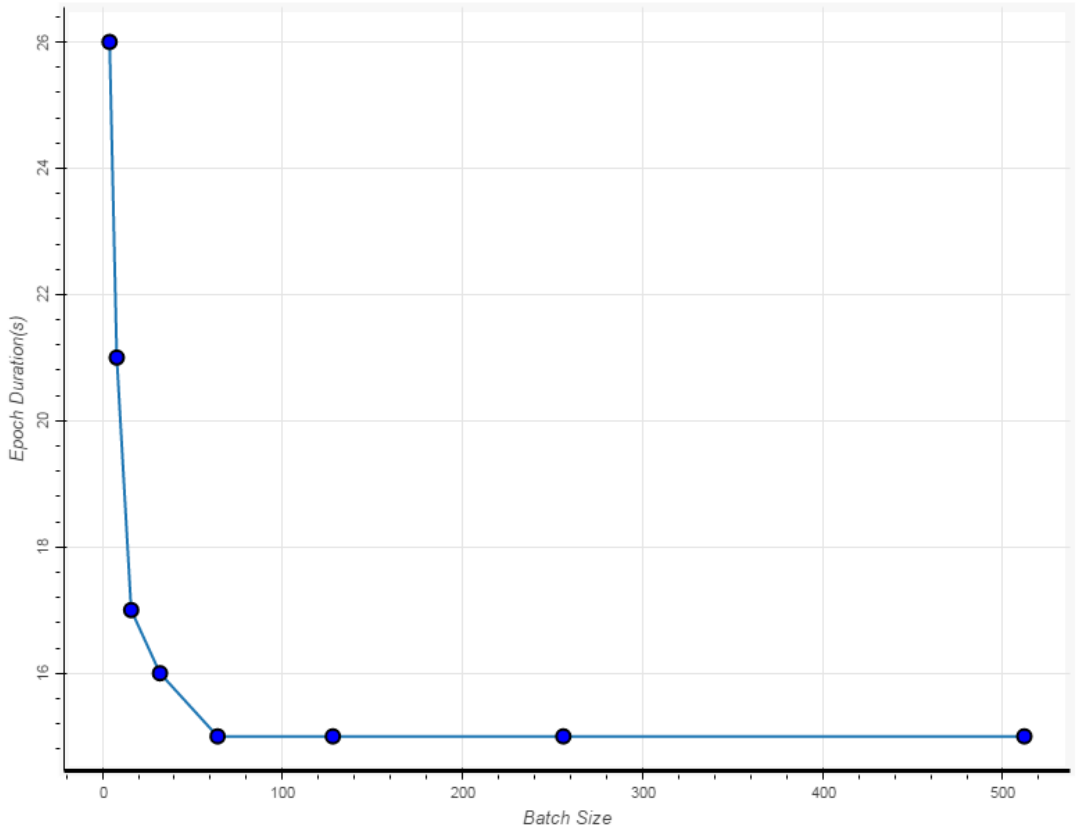


FIGURE 3.17: The average duration of five epochs is measured as a function of the mini-batch size. Batch sizes used were in powers of two. Training was done with Model 0, using only difference images and without batch normalization and data-augmentation. Training time appears asymptotic to 15s for batches larger than 64. Batches greater than 512 failed to run for lack of memory on the GPU.

Training time asymptotically approaches 15s for batches larger than 64. In contrast, for batches smaller than 32 the epoch duration rises steeply. This is consistent with

Model Type	AUC Score
Drop-out = 0.0	**0.972**
Drop-out = 0.1	0.969
Drop-out = 0.2	0.967
Drop-out = 0.3	0.962

TABLE 3.11: AUC score for models of various drop-out values. Without exception, as drop-out increases model performance decreases.

what we would expect. Using large batches reduces the number of data transfers. This suggests that the few data transfers, for batches greater than 64, account for very little of the total training time. Instead, total training time is mostly accounted for by the GPU processing forward and backward propagation. On the other side, batches of half the size require twice the number of memory transfers. With batches less than 64 the accumulated latency of data transfers from the CPU to GPU begin to dominate training time

A batch size of 32 not only appears commonly in the literature of previously applied CNNs but increases training time by approximately 7%. However, as much as small batches are more likely to escape local minima, their enhanced stochastic nature will likely require more training epochs than larger batches. Without complete training for several models of different batch size this is difficult to estimate ahead of time.

3.4.2 Regularization

There are several available methods for regularization of CNNs. For several image recognition tasks, in particular ImageNet, Drop-out has proven to enhance performance. However, as shown in Table 3.11, even small drop-out fractions negatively impact performance.

This could have been due to the network not being complex enough to model the data, and so any regularization would make it even less capable. However, this is likely not the case as increases in the model's complexity through increasing the number of filters leads to no significant improvement, see Subsection 3.4.4. This suggests that the model was in fact complex enough and drop-out removes too much useful information for classification.

In addition to increased generality, drop-out should increase training time. This is shown in Figure 3.18. Drop-out does indeed slow down training. In any epoch models with less drop-out have less loss. However, one would expect models free of drop-out to plateau and models with drop-out to eventually surpass them in performance. In Figure 3.18, training is cut short for several runs as early stopping was triggered.

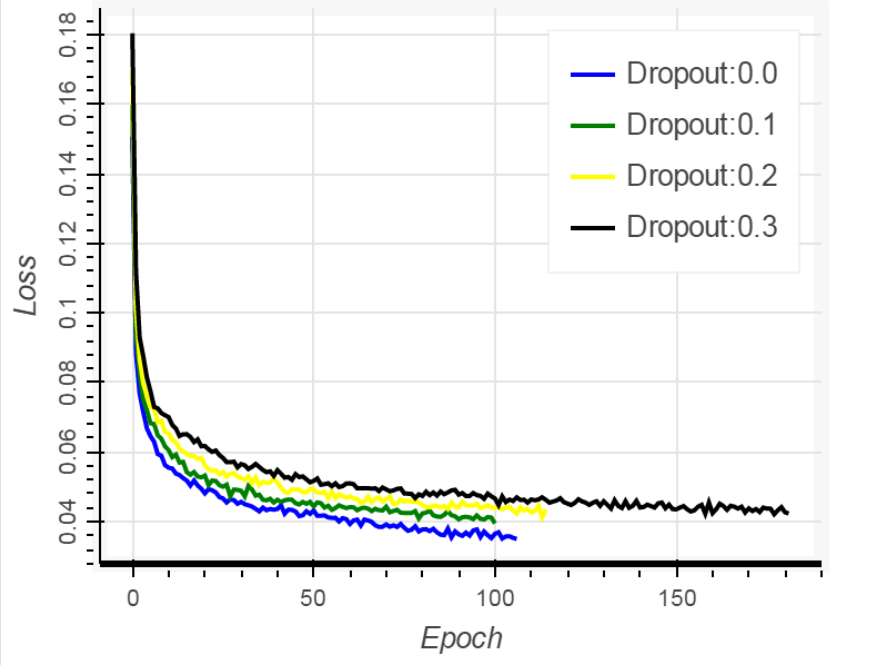


FIGURE 3.18: Model training takes significantly longer with drop-out than without. Training curves stop at various epochs due to early stopping.

Aside from drop-out, L1 and L2 and elastic-net (both L1 and L2) regularization methods may be used. The degree to which these are applied may vary, however, due to limited training time, λ values of only 0.01 were considered. In all instances, see Figure 3.19, regularization degrades performance. In particular, LASSO appears to contribute more to the model degradation considering L1 regularization (in yellow) and Lasso (in green) have AUC scores of 0.954 and 0.726 respectively. Elastic net is unsurprisingly worse than either LASSO or weight-decay. In response to these results no regularization was included.

3.4.3 Network Activation Function

The activation functions used for all convolutional and dense layers were all sigmoids or all ReLUs. Model 0, without batchnorm, data augmentation and using only difference images performs best when using ReLUs rather than sigmoids, see Table 3.12. Meaningful comparisons were not made between the training time required for both network activations as several models would be trained concurrently. For this reason system conditions were not the same when both model versions were trained. However, the number of epochs may still be spoken of fairly. Early stopping kicked in at epoch 98 for the sigmoidal-only version. For the ReLU version the model achieved greater performance in near half the number of epochs at 50 before early-stopping interrupted.

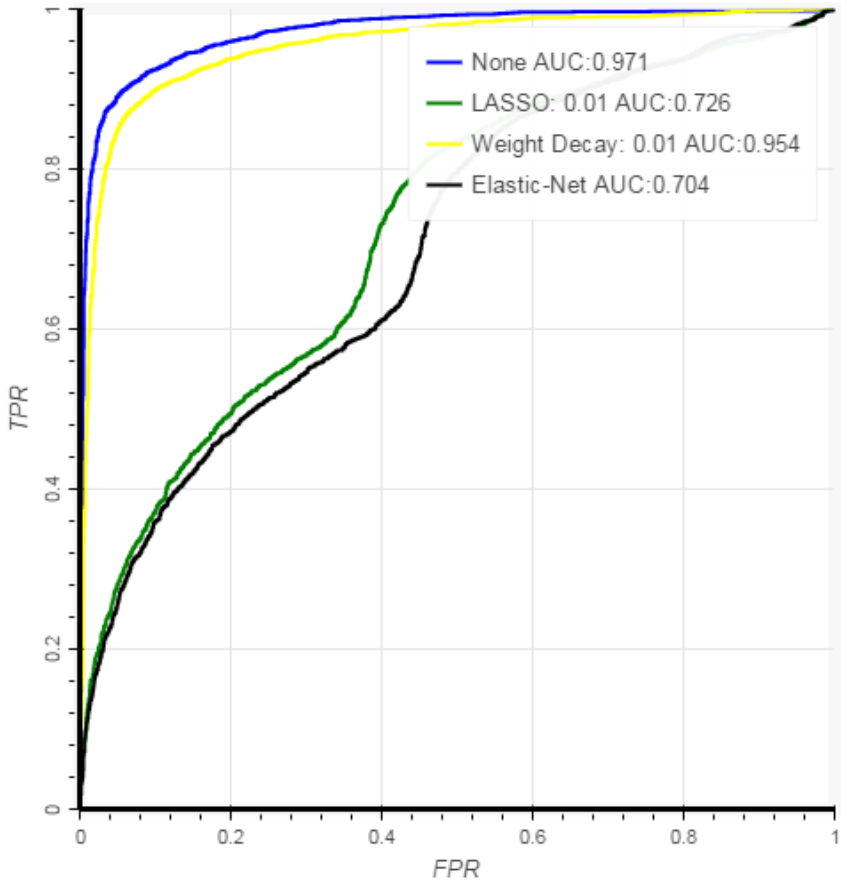


FIGURE 3.19: The addition of weight decay only slightly decreases performance. However, the inclusion of LASSO severely hampers the model. Elastic Net, a double handicap using both regularization methods, predictably reduces performance even more.

Model Type	AUC Score
ReLU	**0.921**
Sigmoid	0.806

TABLE 3.12: ReLUs not only train faster on a GPU but a significant increase in performance is seen when all layer activations are ReLUs rather than sigmoids.

3.4.4 Filter Size

Another parameter to be tuned is the number of filters to be used at each convolutional layer. Only two versions were explored. Layers 1, 2 and 3 would have either 32, 64 and 128 filters or 64, 128 and 256 filters respectively. While there is a performance improvement using more filters, the improvement is slight, see Table 3.13. Even though the improvement is small it was decided to use the greater number of filters. An under-complex model can only model so much complexity, on the other hand an overly-complex model, should it be seen to over-fit, could simply include more regularization. This increase in over-fitting was not seen.

Model Type	AUC Score
Filters: 64-128-256	**0.971**
Filters: 32-64-128	0.968

TABLE 3.13: Doubling the number of filters at every convolution stage would increase model complexity significantly. The increased complexity barely improved the model.

Model Type	AUC Score
Model 0: 1024 Neurons	0.971
Model 0: 2048 Neurons	**0.972**
Model 1: 1024 Neurons	**0.97**
Model 1: 2048 Neurons	0.968
Model 2: 1024 Neurons	**0.972**
Model 2: 2048 Neurons	0.967

TABLE 3.14: Changing the number of neurons in the final fully-connected layer appears to make very little difference in performance. This suggests the model is already complex enough and gains little, if at all, from increasing the complexity.

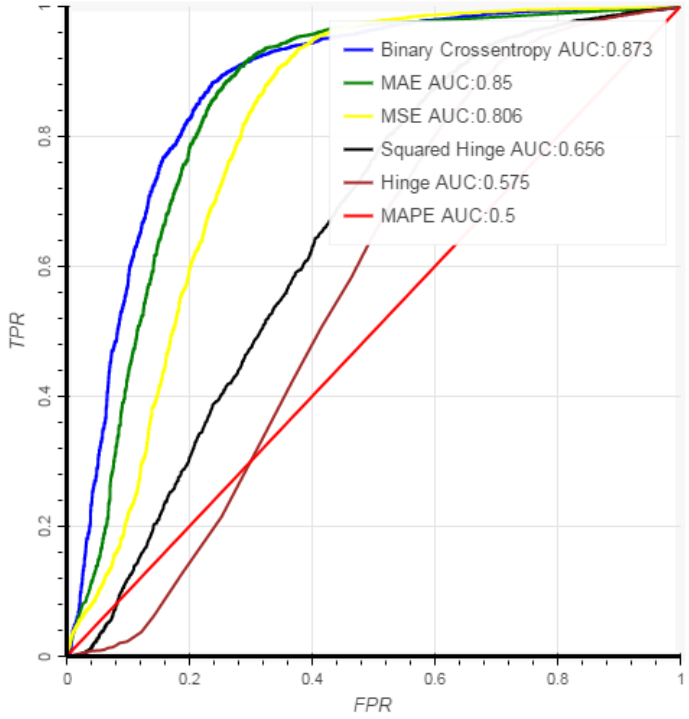
3.4.5 Loss Function

In keeping with the expectation, binary cross-entropy achieves a greater improvement to the AUC score than the generic MSE for Model 0 using only difference images, see top of Figure 3.20. However, when training was performed on Model 5 using search images in addition to difference images, see bottom graph, the MSE loss function achieved a greater AUC score. As MSE exhibited more stable behaviour between implementations, without the time for further loss function tuning, MSE was selected.

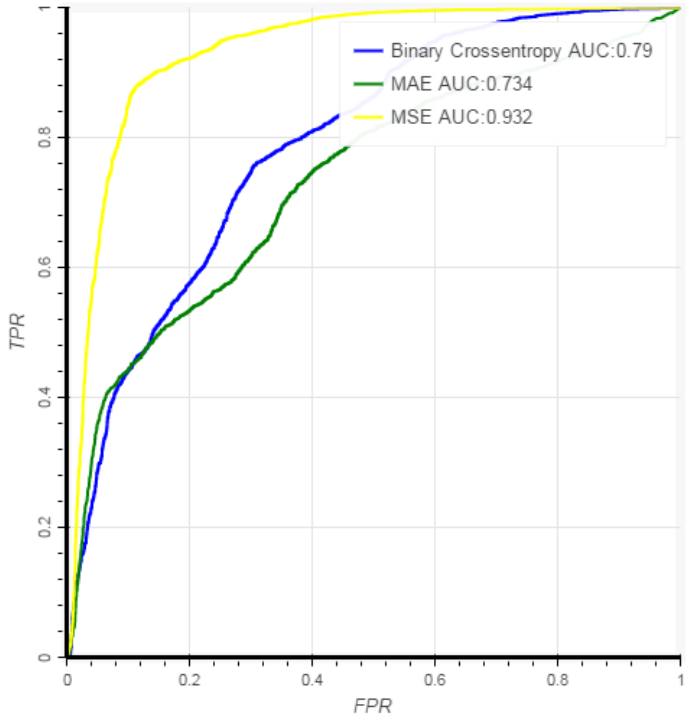
3.4.6 Fully-connected Layers

As with the number of filters per convolutional layer the complexity of the fully connected layer/s need to be determined. As shown in Table 3.14 differences between Models 0, 1 and 2 with a single hidden layer of either 1024 or 2048 neurons show very little difference. With increased layer size, for two of the three models tested, performance actually decreased. However, with AUC scores only differing in the third decimal place noise may account for some if not all the differences observed.

With similar reasoning applied to the number of filters, an overly-complex model may more easily be tuned with regularization methods than an under-complex model. As such, 2048 neurons were used in remaining optimizations.



(a) Model 0: Without search images



(b) Model 5: With search images

FIGURE 3.20: Several loss functions are available. For a binary classification problem literature suggests binary cross-entropy. This appears to be true with Model 0 using only difference images. However, in training Model 5 *with* search images as well, MSE significantly outperforms both binary cross-entropy and Mean Absolute Error (MAE).

Model No.	Drop-out - 0.0	Drop-out - 0.1	Drop-out - 0.2	Drop-out - 0.3
0	**0.9720**	0.9694	0.9673	0.9624
1	0.9677	-	0.9618	0.9661
2	0.9673	0.9664	0.9607	0.9645
3	-	0.9666	0.9713	**0.9722**
4	-	**0.9732**	**0.9718**	0.9673
5	-	0.9689	0.9701	0.9650

TABLE 3.15: Model AUC Scores for Shift = 0.2 and variable drop-out.

Model No.	Shift - 0.1	Shift - 0.2
0	0.9701	0.9678
1	0.9728	0.9714
2	0.9689	0.9702
3	0.9719	**0.9717**
4	0.9705	0.9714
5	**0.9729**	0.9706

TABLE 3.16: Model AUC Scores for shifts of 0.1 and 0.2. A shift of 0.1 means that in the data augmentation pipeline the input is a crop of the original image translated vertically or horizontally by up to 10% of the total image height and width respectively.

3.5 Model Analysis

3.5.1 Slow Training

Several comparisons were made between all six models. Once when determining the best drop-out, results shown in Table 3.15; again when determining optimal shift, summarised in Table 3.16; and lastly, to determine the number of neurons in the final layer, results shown in Table 3.17. Despite all these metrics with which to compare the models, there appears to be no apparent winner, certainly not in the majority of cases. It was not even necessarily true that a deeper model in the same conditions would outperform the single hidden-layer version.

Rather than potentially over-fitting to the validation-set by excessive hyper-parameter tuning and model selection, it made more sense to instead take two extremes with equivalent numbers of layers. Model 0 and 2 fit these criteria, and are largely distinguished by the convolution window size.

Using Models 0 and 2, training was done again with a different random number seed, half the batch size and half the learning rate. As seen in Figure 3.21, while the smaller learning rate and batch size do slow down learning, early stopping does not interrupt the process for over 240 epochs compared to the initial 108. The slower convergence rate on Model 0 made little change, see Table 3.18. However, for Model 2 (with larger convolution kernels) the slower learning improved performance by close to 1%. While

Model No.	AUC Score
0	0.9708
1	0.9699
2	**0.9721**
3	0.9713
4	0.9702
5	0.9656

TABLE 3.17: Model AUC Scores without drop-out and with 1024 neurons in uppermost layer.

still a gain, the 'costs' of increased learning show that fine-tuning at this stage has ever-diminishing returns.

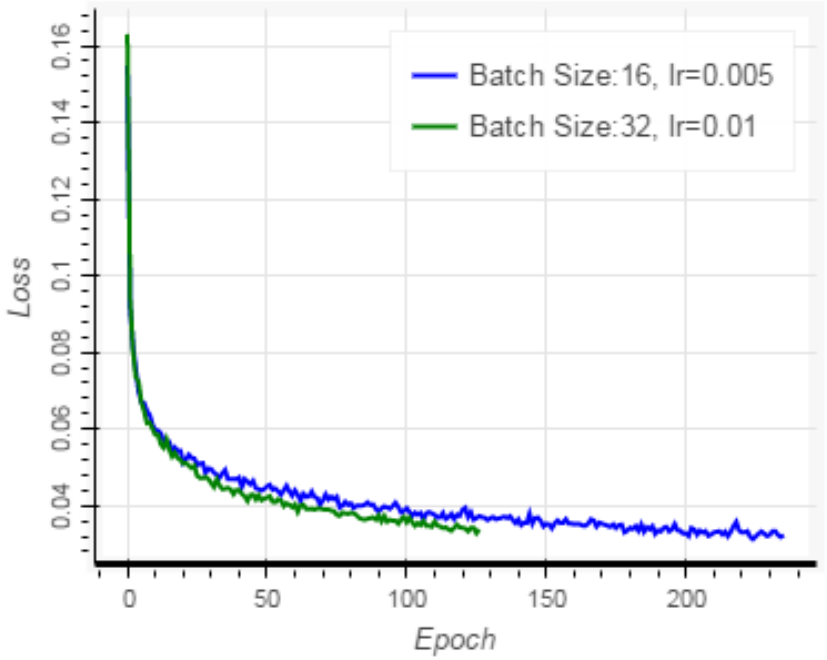


FIGURE 3.21: Using half the batch-size and learning rate reliably increases the number of epochs required. However, the model kept exhibiting sufficient improvement such that early stopping did not interrupt training.

3.5.2 Comparative Performance

Taking the slow-trained Model 2 to be the final version we may now compare to prior work. State of the art performance was achieved with Random Forests[140] and achieved an AUC score of 0.97, see Figure 3.23. This is surpassed by the CNN approach which achieved an AUC score of 0.974. To get a better understanding of what the model is doing well a confusion matrix is used. The confusion matrix, see Figure 3.22, suggests that there will be approximately 8.3% false negatives. In addition, approximately 9.4% of non-real transients will be classified as real (false positives).

FIGURE 3.22: The normalized confusion matrix shows greater than 90% accuracy for both classes.

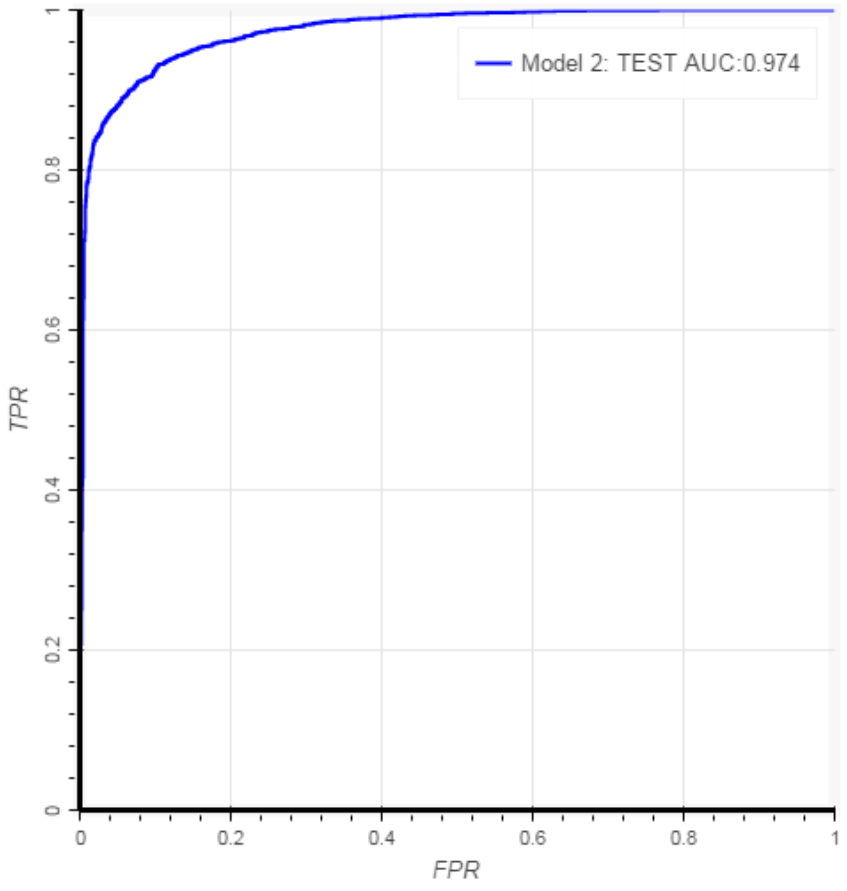


FIGURE 3.23: ROC Curve of the fully-tuned CNN.

Model Type	AUC Score
Model 2: Fast	0.967
Model 2: Slow	**0.974**
Model 0: Fast	**0.972**
Model 0: Slow	0.97

TABLE 3.18: Slow learning was done with both models 0 and 2. Interestingly the less complex model 0 decreases in performance while model 2, with 5×5 convolutional filters, benefits from the slower and more stochastic learning.

There were 625 fake SNe in the test-set. Of these, the CNN has a recall of 0.98, an improvement of just 1% to the Random Forest model used which achieved 0.97, see Figure 3.13. In comparison, fake SNe recall averaged over the human hand-scanners, was 0.956 ± 0.010 [121].

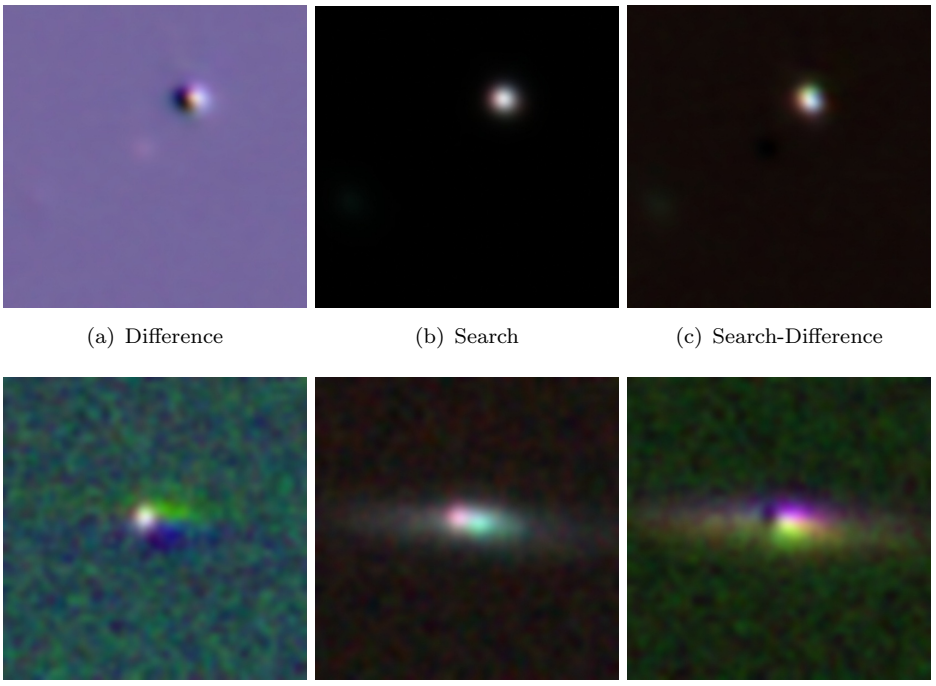
Regarding transients which were confirmed to be SNe, there were 135 in the test-set. Model 2 achieved a recall of 0.96, again a small improvement to the simple K-Nearest Neighbours used in prior research which has a recall of 0.95.

3.5.3 Misclassified examples

Despite there being some improvements to previous research, performance gains were small. To understand why this is the case we take a look at some of the misclassified samples.

Figure 3.24 shows verified SNe samples which were incorrectly classified by the model as non-real transients. Two of these cases are shown. In each case the difference image is shown on the left, with the corresponding search image in the center. On the right, the difference image is subtracted from the search image, before scaling to RGB colours, to demonstrate how noticeable the difference is. This is necessary as both difference and search images shown here have colours scaled to between 0 and 256 which may exaggerate the difference in absolute terms.

The incorrectly classified SN in Figure 3.24 image *a* appears to be a dipole. Confusion in the model may has arisen from the colour differences (green on the top right and red on the bottom left in image *c*). The model likely interpreted these misaligned colours as evidence of a misaligned scan, a dipole, rather than as evidence of a SNe. Image *d* is not interpreted so simply. The clue appears in image *f*. The noise present in the new scan seemed dominant enough to change the colour characteristics of the space surrounding the galaxy. Slight noise would not do this, see the image above in *c*. As such, the model may have interpreted this as an artifact.



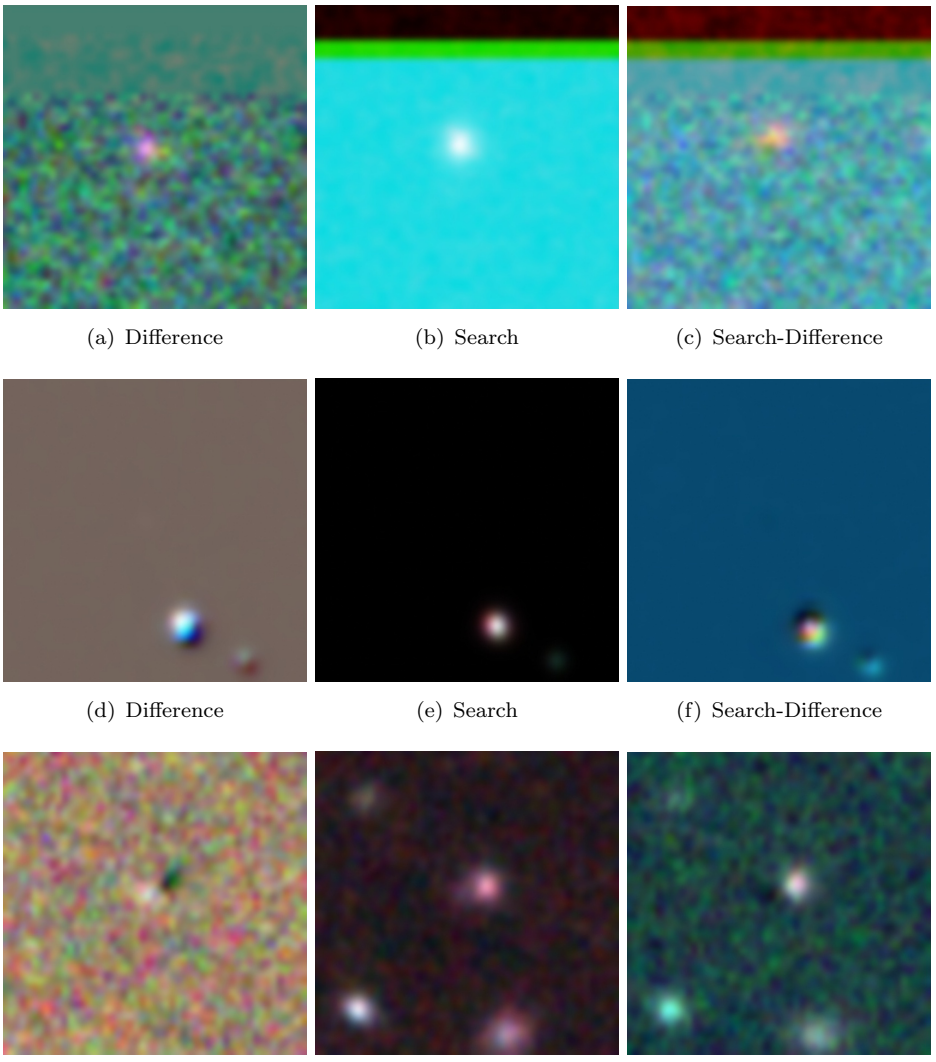
(a) Difference (b) Search (c) Search-Difference

(d) Difference (e) Search (f) Search-Difference

FIGURE 3.24: Verified SNe misclassified by the model as non-real transients. Each row shows the difference image, the search image and the result of the difference image being subtracted from the search image. The subtraction takes place before rescaling colour channels to suit RGB images.

Figure 3.25 shows several fake SNe which are misclassified as non-real. Image *a* is not uncommon, several samples feature stripes of radically different colours. This is very similar to many artifacts making this an understandable error. The off-center fake SN in image *d* resembles a dipole and has a nearby potential transient in the frame. For several samples, such as image *g* the signal to noise ratio is very low. No crafted features were included in the model which contained information on the signal to noise ratio; perhaps the inclusion of such a feature would help the model decide when to ignore all but the most intense pixels.

Figure 3.26 shows real transients incorrectly classified as non-real. The transient in image *a* was likely mistaken for a saturated star that may exhibit some colour separation as in Figure 3.12. A moving object, a real transient, would exhibit the same colour separation from the sequential exposure times for each of the colour sensors. However, one would expect the centers of each colour to be equally spaced in a straight line. The green and blue, or filters *i* and *r* respectively, largely overlap which may have "nudged" the model to consider the item a saturated star instead. Image *d* shows a fake SNe happening on the far right of the picture. Data augmentation alone could shift the image to the right losing all information on the transient and retain only background noise. Image *g*, like *d*, shows multiple transient candidates. This too is not a rare instance within the data set. As



(a) Difference (b) Search (c) Search-Difference

(d) Difference (e) Search (f) Search-Difference

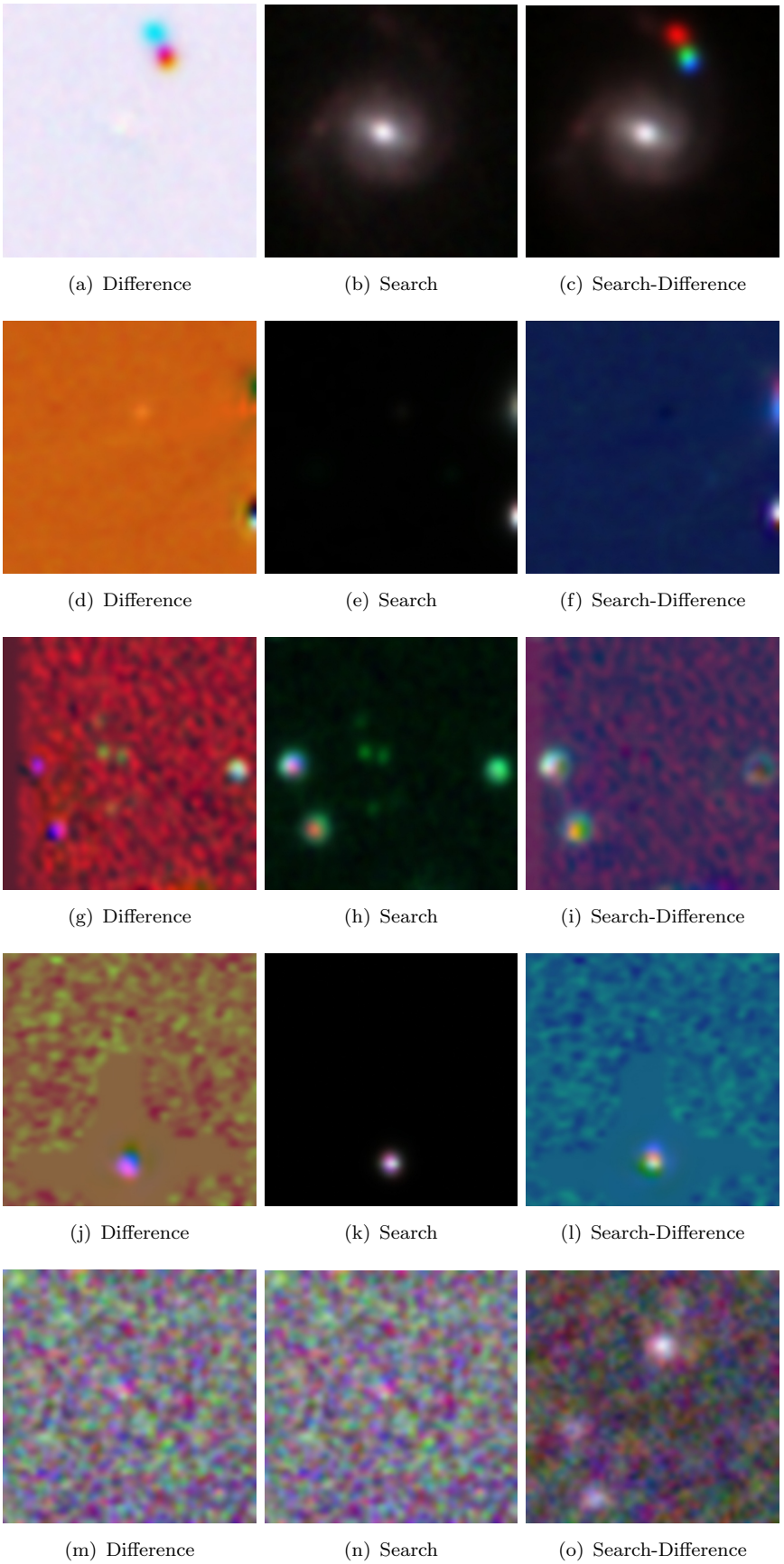
(g) Difference (h) Search (i) Search-Difference

FIGURE 3.25: Fake SNe misclassified as non-real.

the model provides a binary classification from an input image, all objects are classified together. No algorithm is in-built to automatically separate transient candidates and classify each accordingly. Instead, the model at best produces an average score over all the candidates and at worst produces only nonsense. Since SNe are rare, about 1 in 100 years per galaxy, it is highly unlikely for there to be more than one SNe in the image which means two of the three are likely not-real. An average score, for an accurate model, would therefore almost always predict an aggregate of several transient candidates to be non-real. In the case of image *j*, a cross-hair shape surrounds the transient making classification as an artifact more likely. The bottom row of Figure 3.26 shows that the difference and search image appear almost identical. Only when the difference of the two is produced in image *o* do changes appear. While information is not missing with only two of the three images, rarely is the search-difference image necessary to accentuate

differences. As such, the model likely does not take full advantage of having both search and differences available.

Figure 3.27 shows samples of false positives. As with the false negatives, from the fake and verified SNe, there are cases of multiple objects within the images (*a*, *j* and *m*), off-enter transients (*a*, *g* and *j*) and in some cases, such as image *m*, it is not apparent (for an untrained eye) why that is not a real transient. As this is not a fake nor verified transient we have no objective source to say to what class the item belongs.



(a) Difference (b) Search (c) Search-Difference

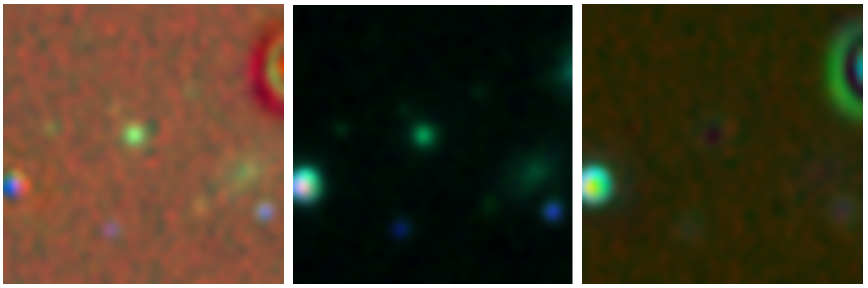
(d) Difference (e) Search (f) Search-Difference

(g) Difference (h) Search (i) Search-Difference

(j) Difference (k) Search (l) Search-Difference

(m) Difference (n) Search (o) Search-Difference

FIGURE 3.26: Real SNe misclassified as non-real.



(a) Difference (b) Search (c) Search-Difference

(d) Difference (e) Search (f) Search-Difference

(g) Difference (h) Search (i) Search-Difference

(j) Difference (k) Search (l) Search-Difference

(m) Difference (n) Search (o) Search-Difference

FIGURE 3.27: Non-real transients misclassified as real.

Chapter 4

Discussion and Conclusions

In conclusion, while machine learning application in astronomy is young, results from this study suggest that it does in fact have a positive role to play in future astronomical surveys. Deep learning methods prove to be on par, if not better, than human scanners; but unlike astronomers they can classify thousands of transients a second. Unlike conventional machine learning algorithms convolutional neural networks require no complex and case-specific features to be crafted. With only data-augmentation during training, convolutional neural networks discover their own abstract features for classification. Deep models can provide continuous-valued scores for classification certainty that may be tuned, unlike human scanners, for best recall and precision. In addition, they are capable of handling such large data throughputs as may be generated by the Large Synoptic Survey Telescope through heavy utilization of GPUs. Deep models, in particular Convolutional Neural Networks, are indispensable for future astronomical sky surveys such as the LSST.

Regarding future work using the Sloan Digital Sky Survey transient data-set, the data itself is the limiting factor behind transient classification success. Many labels appear inconsistent and often have very low signal-to-noise ratios due to hand-scanners minimizing false negatives. For fast and reliable classification the preprocessing algorithms for highlighting and centering candidate transients need improvement so as to consistently provide a single centered transient as input for the classification model. Artifact and moving object detection may be enhanced by cross-referencing sky locations and times with databases of known artificial satellites and asteroids. However, similar applications of deep learning (with better data-sets) may be improved by including a time-series of images of the region in question, including time durations between scans images as features. With this information, not only may variable stars be more easily identified (from their repetitively oscillating light curves), but when supernovae are discovered their age

and type may be quickly identified as well. Supernovae identified in this way with high confidence may immediately be recommended for dedicated telescope follow-ups. Finally, models developed in this thesis were used and assessed in isolation. Ideally several models would be trained with different architectures, hyper-parameters and initializations. Their combined outputs may then be fed as input to an ensemble classifier which may utilise the strengths of different models and produce a superior classifier.

Appendix A

ML Code

A.1 Training Script

```
#*************************************************************************
#                                   Meta Data                              *
#*************************************************************************

__authors__ = 'Gilad Amar'
__copyright__ = 'Copyright 2014-2017, UCT'
__credits__ = ['Gilad Amar']
__license__ = '3-clause BSD'
__maintainer__ = 'Gilad Amar'
__email__ = 'giladamar@gmail.com'

'''
Information about Class
'''
# cnmem = 0.3
# Execution String:
  # THEANO_FLAGS='device=gpu0' python Keras_transient_classifier_V2.0.py &> 'logs/Aug(featurewiz_rotFlip_shift0.2)_50-1000_r2_r1_batch16_lr0.005_ntwkAct(relu-sigmoid)_btchNorm_T_Search_Multi11_Mdl3_(64-128-256)(2048)_mse_drp0._SGD_noReg()_log.txt'

run_descriptor = 'Aug(featurewiz_rotFlip_shift0.2)_50-1000_r2_r1_batch16_lr0.005_ntwkAct(relu-sigmoid)_btchNorm_T_Search_Multi11_Mdl3_(64-128-256)(2048)_mse_drp0._SGD_noReg()'

#*************************************************************************
#                                   ModelIMPORTS                           *
#*************************************************************************
```

```
24
25 from keras.layers.advanced_activations import LeakyReLU, PReLU
26 from keras.optimizers import SGD, Adadelta, Adagrad, RMSprop, Adam
27 import keras.regularizers as Regularizers
28 import model_selection
29
30 #*********************************************************************
31 #                              HYPERPARAMETERS                       *
32 #*********************************************************************
33
34 random_seed = 2
35 random_data_split_seed = 1
36
37 #FEEDBACK
38 verbose = 1 #0: Nothing, 1: Progress Bar, 2: Line per Epoch
39 show_accuracy = True
40 buffer_size = 2
41
42 #TRAINING HYPERPARAMETERS
43 shuffle = True
44 batch_size = 16
45 predict_batch_size = 128
46 use_validation = True
47 validation_split = 0.
48 nb_classes = 11
49 nb_epoch = 1000
50 patience = 100
51 factor_improvement = 0.995
52 validation_frequency = 5 #Only in data augmentation mode
53
54 #LOADING DATA
55 generate_data = False
56 save_as_binary = False
57 data_path = 'Train_Valid_data_andSearch_allclasses.bin'
58 # 'Train_Valid_data_diff_allclasses.bin' #
59 test_data_path = 'Test_data_andSearch_all_classes.bin' if data_path == '
       Train_Valid_data_andSearch_allclasses.bin' else '
       Test_data_diff_all_classes.bin'
60 category_change_list = (0,      1,      2,      4,  5,  6,  7,  8,      9,
       10)
61 #Artifacts, Moving Objects, Saturated Stars, Dipoles, Variables, Transients
       , Other SN, Bronze SN, Silver SN, Gold SN
62
63 #DATA AUGMENTATION
64 data_augmentation = True
65 featurewise_center = True
66 samplewise_center = False
67 featurewise_std_normalization = True
```

```
68  samplewise_std_normalization = False
69  zca_whitening = False
70  rotation_range = 0.
71  width_shift_range = 0.2
72  height_shift_range = 0.2
73  horizontal_flip = True
74  vertical_flip = False
75  save_to_dir = None #Disabled in keras.preprocessing.image
76  save_prefix = '' #'augmented_imgs'
77  save_format = 'jpeg'
78  crop_coords = 0. #(10, 40, 10, 40)
79  gaussian_noise_sigma = 0.
80  do_ninety_rotations = True
81
82  #LOADING MODEL
83  model_no = 3
84  network_activation = 'relu'# 'sigmoid'
85  loss = 'mse'  if nb_classes==1 else 'categorical_crossentropy'
86  #'binary_crossentropy' 'rmse', 'mae', 'mape', 'msle', 'squared_hinge', 'mse
        ', 'binary_crossentropy'
87  final_activation = 'sigmoid' if nb_classes==1 else 'softmax'# 'relu'
88  and_search_img = True if data_path == '
        Train_Valid_data_andSearch_allclasses.bin' else False
89
90  dropout_p = 0.
91  batch_Norm = True
92  optimizer = SGD(lr = 0.005, decay = 1e-6, momentum = 0.9, nesterov = True)
93  W_regularizer = Regularizers.l1l2(l1 = 0.00,
94                    l2 = 0.00)
95          #l1=0.01:LASSO, l2=0.01 weight decay/Ridge, l1=0.01 & l2=0.01
        ElasticNet
96  activity_regularizer = Regularizers.activity_l1l2(l1 = 0.00,
97                             l2 = 0.00)
98  weight_init = 'glorot_uniform'
99  load_weights = False
100
101
102 #Obtaining Results
103 n_runs = 5
104
105 #*********************************************************************
106 #                                IMPORTS                              *
107 #*********************************************************************
108
109 import fits_reading as fitsReader
110 import load_save_test_data as testReader
111 import numpy as np
112 import sys
```

```
113 import os
114 if not os.path.exists('buffering.py'):
115     print('Downloading buffering.py')
116     os.system('wget https://raw.githubusercontent.com/benanne/kaggle-ndsb
    /11a66cdbddee16c69514b9530a727df0ac6e136f/buffering.py')
117 from buffering import buffered_gen_mp, buffered_gen_threaded
118 from sklearn.cross_validation import train_test_split
119 from sklearn.metrics import roc_curve, roc_auc_score
120 from sklearn import preprocessing
121 import model_selection
122 import matplotlib.pyplot as plt
123 import cPickle
124 import time
125 import shutil
126
127 #**************************************************************************
128 #                                   KERAS IMPORTS                         *
129 #**************************************************************************
130
131 import keras.utils.theano_utils as T_utils
132 from keras.preprocessing.image import ImageDataGenerator
133 from keras.utils import np_utils, generic_utils
134 from keras.utils import visualize_util
135 from keras.callbacks import Callback, ModelCheckpoint, EarlyStopping
136
137
138 #**************************************************************************
139 #                                   INITIALIZE                            *
140 #**************************************************************************
141
142 start_time = time.clock()
143 np.random.seed(random_seed)
144
145 #CHECKPOINT SAVING
146 checkpoint_fname = 'model_data/{}_weights_checkpoint.hdf5'.format(
    run_descriptor)
147 best_train_weights = 'model_data/{}_best_Train_weights.hdf5'.format(
    run_descriptor)
148 best_valid_weights = 'model_data/{}_best_Valid_weights.hdf5'.format(
    run_descriptor)
149 best_test_weights = 'model_data/{}_best_Test_weights.hdf5'.format(
    run_descriptor)
150
151 save_best_only = True
152 history_fname = 'history/{}_training_history.bin'.format(run_descriptor)
153
154
155 #SAVING MODEL
```

```
156 weights_load_fname = 'model_data/{}_weights_load_file.hdf5'.format(
        run_descriptor)
157 model_plot_fname = 'model_data/{}_model_plot.png'.format(run_descriptor)
158 weights_save_fname = 'model_data/{}_final_weights.bin'.format(
        run_descriptor)
159 epoch_insurance_fname = 'model_data/{}_weights_backup.hdf5'.format(
        run_descriptor)
160 overwrite = True
161 results_fname = 'results/{}_results.bin'.format(run_descriptor)
162 best_val_model_results_fname = 'results/{}_best_val_test_results.bin'.
        format(run_descriptor)
163 roc_curve_fname = 'results/{}_roc_curve.png'.format(run_descriptor)
164 best_val_roc_curve_fname = 'results/{}_best_val_roc_curve.png'.format(
        run_descriptor)
165
166 #*****************************************************************
167 #                          SAVE EXECUTED SCRIPT                  *
168 #*****************************************************************
169
170 script_run_name = 'executed_scripts/' + run_descriptor + '_fullscript.py'
171 shutil.copyfile(sys.argv[0], script_run_name)
172
173 #*****************************************************************
174 #                             DATA IMPORT                        *
175 #*****************************************************************
176
177 print 'Data Generated/Loaded'
178
179 X_train, X_valid, Y_train, Y_valid, IDList_train, IDList_valid = fitsReader
        .Load_from_binary(data_path, category_change_list)
180 X_test, Y_test, IDList = testReader.Load_from_binary(test_data_path,
        category_change_list)
181
182 if (nb_classes > 1):
183       Y_train = np_utils.to_categorical(Y_train, nb_classes)
184       Y_valid = np_utils.to_categorical(Y_valid, nb_classes)
185       Y_test = np_utils.to_categorical(Y_test, nb_classes)
186
187 print('X_train_data shape:', X_train.shape)
188 print('Y_train_data shape:', Y_train.shape)
189 print('X_valid_data shape:', X_valid.shape)
190 print('Y_valid_data shape:', Y_valid.shape)
191 print('X_test_data shape:', X_test.shape)
192 print('Y_test_data shape:', Y_test.shape)
193
194 #*****************************************************************
195 #                          CALLBACKS DESIGN                      *
196 #*****************************************************************
```

```
197
198 class LossHistory(Callback):
199     def on_train_begin(self, logs = {}):
200       self.train_loss = []
201       self.train_accuracy = []
202       self.val_loss = []
203       self.val_accuracy = []
204       self.epoch_time = []
205       self.epoch_start_time = time.clock()
206
207     def on_epoch_end(self, epoch, logs = {}):
208       self.epoch_time.append(time.clock() - self.epoch_start_time)
209       self.train_loss.append(logs.get('loss'))
210       self.train_accuracy.append(logs.get('accuracy') if show_accuracy else
      None)
211       self.val_loss.append(logs.get('val_loss') if use_validation else None
     )
212       self.val_accuracy.append(logs.get('val_accuracy') if use_validation
     and show_accuracy else None)
213
214 history = LossHistory()
215 checkpointer = ModelCheckpoint(filepath = checkpoint_fname,
216                    verbose = verbose,
217                    save_best_only = save_best_only)
218
219 early_stopping = EarlyStopping(patience = patience,
220                    verbose = verbose)
221
222 #*********************************************************************
223 #                              MODEL DESIGN                           *
224 #*********************************************************************
225
226 print 'Creating Model (takes a little while):'
227 model = model_selection.get_model(model_no,
228                   and_search_img,
229                   loss,
230                   optimizer,
231                   network_activation,
232                   final_activation,
233                   W_regularizer,
234                   activity_regularizer,
235                   weight_init,
236                   dropout_p,
237                   batch_Norm,
238                   nb_classes)
239
240 #*********************************************************************
241 #                    CREATE MODEL GRAPH AND SAVE TO FILE              *
```

```
242 #*****************************************************************************
243
244 print 'Plotting Model'
245 visualize_util.plot(model,
246                     model_plot_fname)
247
248 #*****************************************************************************
249 #                 RESTORING EXISTING MODEL WEIGHTS(IF THEY EXIST)           *
250 #*****************************************************************************
251
252 if load_weights:
253     print 'Loading Model Weights from', weights_load_fname
254     model.load_weights(weights_load_fname)
255
256 #*****************************************************************************
257 #                      TRAINING WITHOUT AUGMENTATION   OR                   *
258 #*****************************************************************************
259
260 epoch_start_time = time.clock()
261 if not data_augmentation:
262     print('Not using data augmentation:')
263
264     def crop_in_stack(x, (x_left, x_right, y_bottom, y_top)=(0, -1, 0, -1))
    :
265     return x[:, :, x_left:x_right, y_bottom:y_top]
266
267     X_train = X_train.astype('float32')
268     X_valid = X_valid.astype('float32')
269     X_test = X_test.astype('float32')
270
271     X_train = X_train/(12.847059*2.) + 0.5
272     X_valid = X_valid/(12.847059*2.) + 0.5
273     X_test = X_test/(12.847059*2.) + 0.5
274     validation_data = None
275     if use_validation:
276     if validation_split:
277         X_train, X_valid, Y_train, Y_valid = train_test_split(X_train,
    Y_train,
278                                     test_size = validation_split,
279                                     random_state = random_data_split_seed)
280     validation_data = (X_valid, Y_valid)
281
282     print('X_valid shape:', X_valid.shape)
283     print('Y_valid shape:', Y_valid.shape)
284     print('X_train shape:', X_train.shape)
285     print('Y_train shape:', Y_train.shape)
286     print('X_test shape:', X_test.shape)
287     print('Y_test shape:', Y_test.shape)
```

```
288
289     if crop_coords:
290     X_train = crop_in_stack(X_train,
291                 crop_coords)
292     if use_validation:
293         X_valid= crop_in_stack(X_valid,
294                     crop_coords)
295     X_test = crop_in_stack(X_test,
296                 crop_coords)
297
298     model.fit(X_train, Y_train,
299           batch_size = batch_size,
300           nb_epoch = nb_epoch,
301           verbose = verbose,
302           callbacks = [checkpointer, history, early_stopping],
303           validation_data = validation_data,
304           shuffle = shuffle,
305           show_accuracy = show_accuracy)
306
307     result = model.evaluate(X_test, Y_test,
308                 batch_size = predict_batch_size,
309                 verbose = verbose,
310                 show_accuracy = show_accuracy)
311     test_loss, test_accuracy = result if show_accuracy else [result, None]
312     print('Test Loss/Accuracy:', test_loss, '/', test_accuracy)
313
314     #Save History Information to file
315     hist_file = open(history_fname, 'wb')
316     cPickle.dump(history.epoch_time, hist_file)
317     cPickle.dump(history.train_loss, hist_file)
318     cPickle.dump(history.train_accuracy, hist_file)
319     cPickle.dump(history.val_loss, hist_file)
320     cPickle.dump(history.val_accuracy, hist_file)
321     cPickle.dump(test_loss, hist_file)
322     cPickle.dump(test_accuracy, hist_file)
323     hist_file.close()
324 else:
325
326 #*****************************************************************
327 #                       AUGMENTATION TRAINING                    *
328 #*****************************************************************
329
330     print('Using real time data augmentation with buffer_size = %i :' %
    buffer_size)
331     if validation_split:
332       X_train, X_valid, Y_train, Y_valid = train_test_split(X_train,
    Y_train,
333                                 test_size = validation_split,
```

```
                                    random_state = random_data_split_seed)
    augmentation_datagen = ImageDataGenerator(featurewise_center =
    featurewise_center,
                  samplewise_center = samplewise_center,
                  featurewise_std_normalization =
    featurewise_std_normalization,
                  samplewise_std_normalization =
    samplewise_std_normalization,
                  zca_whitening = zca_whitening,
                  rotation_range = rotation_range,
                  width_shift_range = width_shift_range,
                  height_shift_range = height_shift_range,
                  horizontal_flip = horizontal_flip,
                  vertical_flip = vertical_flip,
                  noise_sigma = gaussian_noise_sigma,
                  crop_coords = crop_coords,
                  do_ninety_rotations=do_ninety_rotations)
    augmentation_datagen.fit(X_train)
    test_datagen = ImageDataGenerator(featurewise_center =
    featurewise_center,
                  samplewise_center = False,
                  featurewise_std_normalization =
    featurewise_std_normalization,
                  samplewise_std_normalization = False,
                  zca_whitening = zca_whitening,
                  rotation_range = rotation_range,
                  width_shift_range = width_shift_range,
                  height_shift_range = height_shift_range,
                  horizontal_flip = horizontal_flip,
                  vertical_flip = vertical_flip,
                  noise_sigma = 0.,
                  crop_coords = crop_coords,
                  do_ninety_rotations = do_ninety_rotations)

    test_datagen.fit(X_train)
    no_train_batches = X_train.shape[0]/batch_size
    no_valid_batches = X_valid.shape[0]/predict_batch_size
    no_test_batches = X_test.shape[0]/predict_batch_size
    print 'Training batches: ', no_train_batches
    print 'Validation batches(Prediction): ', no_valid_batches
    print 'Testing batches(Prediction): ', no_test_batches

    epoch_time = []
    train_loss_history = []
    train_accuracy_history = []
    valid_loss_history = []
    valid_accuracy_history = []
    test_loss_history = []
```

```
377     test_accuracy_history = []
378
379     best_train_loss = np.infty
380     best_validation_loss = np.infty
381     best_test_loss = np.infty
382
383     countdown = patience
384     epoch = 0
385     done_looping = False
386     while (epoch < nb_epoch) and (not done_looping):
387       epoch += 1
388       epoch_time.append(time.clock() - epoch_start_time)
389
390       print('-'*40)
391       print('Epoch', epoch)
392       print('-'*40)
393       #TRAINING
394       print('Training...')
395
396       train_loss = 0.
397       train_accuracy = 0
398       progbar = generic_utils.Progbar(X_train.shape[0])
399       for X_batch, Y_batch in buffered_gen_threaded(augmentation_datagen.
    flow(X_train, Y_train,
400                             batch_size,
401                             shuffle,
402                             save_to_dir,
403                             save_prefix,
404                             save_format), buffer_size=buffer_size):
405
406         train_results = model.train_on_batch(X_batch, Y_batch,
407                             accuracy = show_accuracy)
408         batch_loss, batch_accuracy = train_results if show_accuracy else [
    train_results, None]
409         train_loss += batch_loss
410         train_accuracy += batch_accuracy
411         progbar.add(X_batch.shape[0], values = [('train loss', batch_loss)
    ])
412
413       train_loss, train_accuracy = train_loss/(no_train_batches + 1.0),
    train_accuracy/(no_train_batches + 1.0)
414       print 'Training loss (calc):', train_loss
415       print 'Train Accuracy: ', train_accuracy
416       train_loss_history.append(train_loss)
417       train_accuracy_history.append(train_accuracy)
418       model.save_weights(epoch_insurance_fname,
419                   overwrite = overwrite)
420       if (train_loss < best_train_loss):
```

```
421         print 'Best train score. Saving weights to ', best_train_weights
422         model.save_weights(best_train_weights,
423                            overwrite = overwrite)
424         best_train_loss = train_loss
425       #VALIDATING
426       if (epoch < patience) and (epoch % validation_frequency == 0) and
    use_validation:
427       print('Validating...')
428       progbar = generic_utils.Progbar(X_valid.shape[0])
429       valid_loss = 0.
430       valid_accuracy = 0.
431       for X_batch, Y_batch in buffered_gen_threaded(augmentation_datagen.
    flow(X_valid, Y_valid,
432                                     predict_batch_size,
433                                     shuffle= False,
434                                     save_to_dir=save_to_dir,
435                                     save_prefix=save_prefix,
436                                     save_format=save_format), buffer_size=
    buffer_size):
437         valid_results = model.test_on_batch(X_batch, Y_batch,
438                                 accuracy = show_accuracy)
439         batch_loss, batch_accuracy = valid_results if show_accuracy else [
    valid_results, None]
440         valid_loss += batch_loss
441         valid_accuracy += batch_accuracy
442         progbar.add(X_batch.shape[0], values = [('valid loss', batch_loss)
    ])
443       valid_loss, valid_accuracy = valid_loss/(no_valid_batches + 1.0),
    valid_accuracy/(no_valid_batches + 1.0)
444       valid_loss_history.append(valid_loss)
445       valid_accuracy_history.append(valid_accuracy)
446       print 'Validation Loss (calc): ', valid_loss
447       print 'Validation Accuracy: ', valid_accuracy
448
449       if (valid_loss < best_validation_loss):
450
451           print 'Best validation score. Saving weights to {}'.format(
    best_valid_weights)
452           model.save_weights(best_valid_weights,
453                              overwrite = overwrite)
454           best_validation_loss = valid_loss
455       if (epoch < patience) and (epoch % validation_frequency == 0):
456       #TESTING
457       print('Testing...')
458       progbar = generic_utils.Progbar(X_test.shape[0])
459       test_loss = 0.
460       test_accuracy = 0.
```

```
461       for X_batch, Y_batch in buffered_gen_threaded(test_datagen.flow(
    X_test, Y_test,
462                         predict_batch_size,
463                         shuffle = False,
464                         save_to_dir = save_to_dir,
465                         save_prefix = save_prefix,
466                         save_format = save_format), buffer_size =
    buffer_size):
467
468           test_results = model.test_on_batch(X_batch, Y_batch,
469                               accuracy = True)
470           batch_loss, batch_accuracy = test_results if show_accuracy else [
    test_results, None]
471           test_loss += batch_loss
472           test_accuracy += batch_accuracy
473           progbar.add(X_batch.shape[0], values = [('test loss', batch_loss)])
474       test_loss, test_accuracy = test_loss/(no_test_batches + 1.0),
    test_accuracy/(no_test_batches + 1.0)
475       test_loss_history.append(test_loss)
476       test_accuracy_history.append(test_accuracy)
477       print 'Test Loss (calc): {}'.format(test_loss)
478       print 'Test Accuracy: {}'.format(test_accuracy)
479
480       if (test_loss < best_test_loss):
481           print 'Best test score. Saving weights to {}'.format(
    best_test_weights)
482           model.save_weights(best_test_weights ,
483                     overwrite = overwrite)
484           best_test_loss = test_loss
485
486
487       if (epoch > patience) and use_validation:
488       print('Validating...')
489       progbar = generic_utils.Progbar(X_valid.shape[0])
490       valid_loss = 0.
491       valid_accuracy = 0.
492       for X_batch, Y_batch in buffered_gen_threaded(augmentation_datagen.
    flow(X_valid, Y_valid,
493                                 predict_batch_size,
494                                 shuffle= False,
495                                 save_to_dir=save_to_dir,
496                                 save_prefix=save_prefix,
497                                 save_format=save_format), buffer_size=
    buffer_size):
498           valid_results = model.test_on_batch(X_batch, Y_batch,
499                               accuracy = show_accuracy)
500           batch_loss, batch_accuracy = valid_results if show_accuracy else [
    valid_results, None]
```

```
501         valid_loss += batch_loss
502         valid_accuracy += batch_accuracy
503         progbar.add(X_batch.shape[0], values = [('valid loss', batch_loss)
    ])
504       valid_loss, valid_accuracy = valid_loss/(no_valid_batches + 1.0),
    valid_accuracy/(no_valid_batches + 1.0)
505       valid_loss_history.append(valid_loss)
506       valid_accuracy_history.append(valid_accuracy)
507       print 'Validation Loss (calc): ', valid_loss
508       print 'Validation Accuracy: ', valid_accuracy
509       print 'valid_loss:', valid_loss
510       print 'to beat', best_validation_loss*factor_improvement
511       if (valid_loss < best_validation_loss*factor_improvement):
512         print 'validation performance improved'
513         countdown = patience
514       else:
515         countdown -= 1
516
517       if (valid_loss < best_validation_loss):
518         print 'Best validation score. Saving weights to {}'.format(
    best_valid_weights)
519         model.save_weights(best_valid_weights,
520                   overwrite = overwrite)
521         best_validation_loss = valid_loss
522
523       if countdown == 0:
524         print 'Early stopping interrupt.'
525         done_looping = True
526         break
527
528     #TESTING
529     print('Testing...')
530     progbar = generic_utils.Progbar(X_test.shape[0])
531     test_loss = 0.
532     test_accuracy = 0.
533     for X_batch, Y_batch in buffered_gen_threaded(test_datagen.flow(X_test,
     Y_test,
534                         predict_batch_size,
535                         shuffle=False,
536                         save_to_dir=save_to_dir,
537                         save_prefix=save_prefix,
538                         save_format=save_format), buffer_size=buffer_size):
539     test_results = model.test_on_batch(X_batch, Y_batch,
540                         accuracy = True)
541     batch_loss, batch_accuracy = test_results if show_accuracy else [
    test_results, None]
542     test_loss += batch_loss
543     test_accuracy += batch_accuracy
```

```
544     progbar.add(X_batch.shape[0], values = [('test loss', batch_loss)])
545     test_loss, test_accuracy = test_loss/(no_test_batches + 1.0),
        test_accuracy/(no_test_batches+1.0)
546     test_loss_history.append(test_loss)
547     test_accuracy_history.append(test_accuracy)
548     print 'Test Loss (calc):', test_loss
549     print 'Test Accuracy: ', test_accuracy
550
551     #Save History Information to file
552     print 'Saving training history to ', history_fname
553     hist_file = open(history_fname, 'wb')
554     cPickle.dump(epoch_time, hist_file)
555     cPickle.dump(train_loss_history, hist_file)
556     cPickle.dump(train_accuracy_history, hist_file)
557     cPickle.dump(valid_loss_history, hist_file)
558     cPickle.dump(valid_accuracy_history, hist_file)
559     cPickle.dump(test_loss_history, hist_file)
560     cPickle.dump(test_accuracy_history, hist_file)
561     hist_file.close()
562
563 #*************************************************************************
564 #                              SAVING WEIGHTS                             *
565 #*************************************************************************
566
567 print 'Final weights saved to ', weights_save_fname
568 model.save_weights(weights_save_fname,
569             overwrite = overwrite) #Saves to hdf5 file
570
571 #*************************************************************************
572 #                              OBTAIN RESULTS                             *
573 #*************************************************************************
574
575 def diff_scores(pred_class, Y):
576   TP = np.array([pred_class[Y==1]==1]).sum()
577   TN = np.array([pred_class[Y==0]==0]).sum()
578   FP = np.array([pred_class[Y==0]==1]).sum()
579   FN = np.array([pred_class[Y==1]==0]).sum()
580
581   Accuracy = (TP + TN)/float(TP + TN + FP + FN)
582   Precision= TP/float(TP + FP)
583   Recall = TP/float(TP + FN)
584   F1 = 2*TP/float(2*TP + FP + FN)
585   return TP, TN, FP, FN, Accuracy, Precision, Recall, F1
586
587
588 def avg_predict_with_Aug(X_predict,
589               Y_predict=False,
590               n=2,
```

```
            batch_size=predict_batch_size,
            verbose=0,
            test_generator=True):
      if Y_predict==False:

    generator = test_datagen if test_generator else augmentation_datagen

    Y_predict = np.zeros(shape=(X_predict.shape[0]))
    avg_results = np.zeros(shape=(X_predict.shape[0], n))
    for i in range(n):
      x = 0
      progbar = generic_utils.Progbar(X_predict.shape[0])
      for X_batch, Y_batch in buffered_gen_threaded(generator.flow(
    X_predict, Y_predict,
                          batch_size = batch_size,
                          shuffle=False,
                          save_to_dir=False),
                            buffer_size=buffer_size):
        batch_pred = model.predict_proba(X_batch,
                        batch_size,
                        verbose=0)
        avg_results[x:batch_size + x, i] = np.squeeze(batch_pred)
        x += batch_size
        progbar.add(batch_size)
    return np.average(avg_results, axis=1)
      else:
    print 'function not complete'

if nb_classes==1:

      if data_augmentation:
        train_pred = avg_predict_with_Aug(X_train,
                    Y_predict = False,
                    n = n_runs,
                    batch_size = predict_batch_size,
                    verbose = verbose,
                    test_generator = False)
      else:
        train_pred = model.predict_proba(X_train,
                    batch_size,
                    verbose = verbose)
      train_pred_classes = (train_pred > 0.5).astype('int32')
      train_scores = diff_scores(train_pred_classes,
                Y_train)

      if use_validation:
      if data_augmentation:
        valid_pred = avg_predict_with_Aug(X_valid,
```

```
                        Y_predict = False,
                        n = n_runs,
                        batch_size = predict_batch_size,
                        verbose = verbose,
                        test_generator=False)
    else:
      valid_pred = model.predict_proba(X_valid,
                        batch_size,
                        verbose = verbose)
    valid_pred_classes = (valid_pred > 0.5).astype('int32')
    valid_scores = diff_scores(valid_pred_classes,
                    Y_valid)
    if data_augmentation:
  test_pred = avg_predict_with_Aug(X_test,
                    Y_predict = False,
                    n = n_runs,
                    batch_size = predict_batch_size,
                    verbose = verbose,
                    test_generator = True)
    else:
      test_pred = model.predict_proba(X_test,
                        batch_size,
                        verbose = verbose)
    test_pred_classes = (test_pred > 0.5).astype('int32')
    test_scores = diff_scores(test_pred_classes,
              Y_test)

    print 'TP, TN, FP, FN, Accuracy, Precision, Recall, F1'
    print train_scores
    if use_validation:
  print valid_scores
    print test_scores

    fpr, tpr, thresholds = roc_curve(Y_test,
                    test_pred,
                    pos_label=1)

    auc_score = roc_auc_score(Y_test,
                test_pred)
    results_file = open(results_fname, 'wb')
    cPickle.dump(train_scores, results_file)
    if use_validation:
  cPickle.dump(valid_scores, results_file)
    cPickle.dump(test_scores, results_file)

    cPickle.dump(train_pred, results_file)
    if use_validation:
  cPickle.dump(valid_pred, results_file)
```

```
686       cPickle.dump(test_pred, results_file)
687
688       cPickle.dump(train_pred_classes, results_file)
689       if use_validation:
690     cPickle.dump(valid_pred_classes, results_file)
691       cPickle.dump(test_pred_classes, results_file)
692
693       cPickle.dump(fpr, results_file)
694       cPickle.dump(tpr, results_file)
695       cPickle.dump(thresholds, results_file)
696       cPickle.dump(auc_score, results_file)
697       results_file.close()
698       print 'Results file saved'
699       plt.switch_backend('agg')
700       print 'Plotting ROC'
701       plt.plot(fpr, tpr)
702       plt.title('ROC AUC Score: ' + str(auc_score))
703       plt.xlim(xmin=0, xmax=1)
704       plt.ylim(ymin=0, ymax=1)
705       plt.xlabel('False positive rate')
706       plt.ylabel('True positive rate')
707       print 'Saving ROC'
708       plt.savefig(roc_curve_fname)
709       plt.close()
710
711       print 'Loading Model Weights from', best_valid_weights
712
713       if data_augmentation:
714           model.load_weights(best_valid_weights)
715           val_best_test_pred = avg_predict_with_Aug(X_test,
716                       Y_predict = False,
717                       n = n_runs,
718                       batch_size = predict_batch_size,
719                       verbose = verbose,
720                       test_generator = True)
721       else:
722           model.load_weights(checkpoint_fname)
723           val_best_test_pred = model.predict_proba(X_test,
724                           batch_size,
725                           verbose = verbose)
726       val_best_test_pred_classes = (val_best_test_pred > 0.5).astype('int32
    ')
727       val_best_test_scores = diff_scores(val_best_test_pred_classes,
728                       Y_test)
729       fpr, tpr, thresholds = roc_curve(Y_test,
730                       val_best_test_pred,
731                       pos_label=1)
732
```

```
733         auc_score = roc_auc_score(Y_test,
734                                   val_best_test_pred)
735         print 'best validation model Test Scores: ', val_best_test_scores
736         best_val_model_results_file = open(best_val_model_results_fname, 'wb'
    )
737         cPickle.dump(val_best_test_scores, best_val_model_results_file)
738         cPickle.dump(val_best_test_pred, best_val_model_results_file)
739         cPickle.dump(val_best_test_pred_classes, best_val_model_results_file)
740         cPickle.dump(fpr, best_val_model_results_file)
741         cPickle.dump(tpr, best_val_model_results_file)
742         cPickle.dump(thresholds, best_val_model_results_file)
743         cPickle.dump(auc_score, best_val_model_results_file)
744         best_val_model_results_file.close()
745         print 'Best Val Results file saved'
746         plt.switch_backend('agg')
747         print 'Plotting Best Val ROC'
748         plt.plot(fpr, tpr)
749         plt.title('ROC AUC Score: ' + str(auc_score))
750         plt.xlim(xmin=0, xmax=1)
751         plt.ylim(ymin=0, ymax=1)
752         plt.xlabel('False positive rate')
753         plt.ylabel('True positive rate')
754         print 'Saving ROC'
755         plt.savefig(best_val_roc_curve_fname)
756         plt.close()
757
758 else:
759         print  'Multi Class Analysis, requires extra code.'
760       #train_pred = model.predict(X_train, batch_size=predict_batch_size,
    verbose=verbose)
761       #train_pred_classes = train_pred > 0.5
762       #train_scores = diff_scores(train_pred_classes, Y_train)
763
764       #if validation_split:
765     #if data_augmentation:
766       #valid_pred=avg_predict_results(X_valid, Y_predict=False, n=n_runs,
    batch_size=predict_batch_size, verbose=0)
767     #else:
768       #valid_pred = model.predict(X_valid, batch_size=predict_batch_size,
    verbose=verbose)
769     #valid_pred_classes = valid_pred > 0.5
770     #valid_scores = diff_scores(valid_pred_classes, Y_valid)
771
772       #if data_augmentation:
773     #test_pred=avg_predict_results(X_test, Y_predict=False, n=n_runs,
    batch_size=predict_batch_size, verbose=0)
774       #else:
```

```
775     #test_pred = model.predict(X_test, batch_size=predict_batch_size,
        verbose=verbose)
776       #test_pred_classes = test_pred > 0.5
777       #test_scores = diff_scores(test_pred_classes, Y_test)
778
779 #*********************************************************************
780 #                              SAVING RESULTS                         *
781 #*********************************************************************
782
783       results_file = open(results_fname, 'wb')
784       cPickle.dump(train_scores, results_file)
785
786       if validation_split:
787         cPickle.dump(valid_scores, results_file)
788       cPickle.dump(test_scores, results_file)
789       cPickle.dump(train_pred, results_file)
790
791       if validation_split:
792         cPickle.dump(valid_pred, results_file)
793       cPickle.dump(test_pred, results_file)
794       cPickle.dump(train_pred_classes, results_file)
795
796       if validation_split:
797         cPickle.dump(valid_pred_classes, results_file)
798       cPickle.dump(test_pred_classes, results_file)
799       results_file.close()
800
801 #*********************************************************************
802 #                                  END                                *
803 #*********************************************************************
804
805 end_time = time.clock()
806 print >> sys.stderr, ('The code for file ' + os.path.split(__file__)[1] +
807                 ' ran for %.2fm' % ((end_time - start_time) / 60.))
808
809 #*********************************************************************
810 #                              EMAIL MY MAKER                         *
811 #*********************************************************************
812
813 import smtplib
814 server = smtplib.SMTP('smtp.gmail.com', 587)
815 server.ehlo()
816 server.starttls()
817 server.login(<username>, <password>)
818
819 #Send the mail
820 msg = '\nRun {} completed'.format(run_descriptor) # The /n separates the
    message from the headers
```

```
821 server.sendmail(<source email>, <destination email>, msg)
```

A.2 Model Designer Script

```
1  from keras.models import Sequential
2  from keras.layers.core import Dense, Dropout, Activation, Flatten, Merge
3  from keras.layers.convolutional import Convolution1D, Convolution2D,
       MaxPooling1D, MaxPooling2D
4  from keras.layers.advanced_activations import LeakyReLU, PReLU
5  from keras.layers.normalization import BatchNormalization
6  from keras.optimizers import SGD, Adadelta, Adagrad, RMSprop, Adam
7  import keras.regularizers as Regularizers
8
9  W_regularizer = Regularizers.l1l2(l1 = 0., l2 = 0.) #l1=0.01:LASSO, l2=0.01
       weight decay/Ridge, l1=0.01 & l2=0.01 ElasticNet
10 activity_regularizer = Regularizers.ActivityRegularizer(l1 = 0., l2 = 0.)
11
12 def get_model(model_no = None,
13           and_search_img = False,
14           loss = 'mse',
15           optimizer = 'sgd',
16           network_activation = 'relu',
17           final_activation = 'relu',
18           W_regularizer = W_regularizer,
19           activity_regularizer = activity_regularizer,
20           weight_init = 'glorot_uniform',
21           dropout_p = 0.5,
22           batch_Norm = True,
23           nb_classes =1):
24
25   channels = 6 if and_search_img else 3
26   class_mode = 'binary' if nb_classes==1 else 'categorical'
27
28   if model_no==0:
29     print('Build model...')
30     model = Sequential()
31
32     model.add(Convolution2D(64, 2, 2, input_shape=(channels, 51, 51),
     border_mode = 'valid', activation = network_activation, W_regularizer =
      W_regularizer, activity_regularizer = activity_regularizer, init =
     weight_init))
33     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
34     if batch_Norm: model.add(BatchNormalization())
35     model.add(Dropout(dropout_p))
36
```

```
37      model.add(Convolution2D(128, 2, 2, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
38      model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
39      if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
40      model.add(Dropout(dropout_p))
41
42      model.add(Convolution2D(256, 2, 2, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
43      model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
44      if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
45      model.add(Dropout(dropout_p))
46
47      model.add(Flatten())
48      model.add(Dense(2048, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
49      model.add(Activation(network_activation))
50      if batch_Norm: model.add(BatchNormalization(input_shape=(1024, )))
51      model.add(Dropout(dropout_p))
52
53      model.add(Dense(nb_classes, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
54      model.add(Activation(final_activation))
55
56      model.compile(loss = loss,
57                    optimizer = optimizer,
58                    class_mode=class_mode)
59      return model
60
61  elif model_no==1:
62      '''
63      Subtypes: None,1
64      '''
65      model = Sequential()
66      model.add(Convolution2D(64, 3, 3, input_shape=(channels, 51, 51),
        border_mode = 'valid', activation = network_activation, W_regularizer =
         W_regularizer, activity_regularizer = activity_regularizer, init =
        weight_init))
67      model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
68      if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
69      model.add(Dropout(dropout_p))
70
```

```
71      model.add(Convolution2D(128, 3, 3, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
72      model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
73      if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
74      model.add(Dropout(dropout_p))
75
76      model.add(Convolution2D(256, 2, 2, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
77      model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
78      if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
79      model.add(Dropout(dropout_p))
80
81      model.add(Flatten())
82      model.add(Dense(2048, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
83      model.add(Activation(network_activation))
84      if batch_Norm: model.add(BatchNormalization())
85      model.add(Dropout(dropout_p))
86
87      model.add(Dense(nb_classes, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
88      model.add(Activation(final_activation))
89
90      model.compile(loss = loss,
91                    optimizer = optimizer,
92                    class_mode=class_mode)
93      return model
94
95    elif model_no==2:
96      '''
97      Subtypes: None or 1,2,3,4,5,6,7.
98      '''
99      model = Sequential()
100     model.add(Convolution2D(64, 5, 5, input_shape=(channels, 51, 51),
        border_mode = 'valid', activation = network_activation, W_regularizer =
         W_regularizer, activity_regularizer = activity_regularizer, init =
        weight_init))
101     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
102     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
103     model.add(Dropout(dropout_p))
104
```

```
105     model.add(Convolution2D(128, 5, 5, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
106     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
107     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
108     model.add(Dropout(dropout_p))
109
110     model.add(Convolution2D(256, 5, 5, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
111     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
112     model.add(Dropout(dropout_p))
113
114     model.add(Flatten())
115     model.add(Dense(2048, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
116     model.add(Activation(network_activation))
117     if batch_Norm: model.add(BatchNormalization())
118     model.add(Dropout(dropout_p))
119
120     model.add(Dense(nb_classes, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
121     model.add(Activation(final_activation))
122
123     model.compile(loss = loss,
124                   optimizer = optimizer,
125                   class_mode = class_mode)
126     return model
127
128   if model_no==3:
129     print('Build model...')
130     model = Sequential()
131
132     model.add(Convolution2D(64, 2, 2, input_shape=(channels, 51, 51),
        border_mode = 'valid', activation = network_activation, W_regularizer =
         W_regularizer, activity_regularizer = activity_regularizer, init =
        weight_init))
133     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
134     if batch_Norm: model.add(BatchNormalization())
135
136     model.add(Convolution2D(128, 2, 2, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
137     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
138     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
```

```
139
140     model.add(Convolution2D(256, 2, 2, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
141     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
142     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
143
144     model.add(Flatten())
145     model.add(Dense(2048, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
146     model.add(Activation(network_activation))
147     if batch_Norm: model.add(BatchNormalization(input_shape=(1024, )))
148     model.add(Dropout(dropout_p))
149
150     model.add(Dense(1024, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
151     model.add(Activation(network_activation))
152     if batch_Norm: model.add(BatchNormalization())
153     model.add(Dropout(dropout_p))
154
155     model.add(Dense(nb_classes, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
156     model.add(Activation(final_activation))
157
158     model.compile(loss = loss,
159                   optimizer = optimizer,
160                   class_mode=class_mode)
161     return model
162
163
164   elif model_no==4:
165
166     model = Sequential()
167     model.add(Convolution2D(64, 3, 3, input_shape=(channels, 51, 51),
      border_mode = 'valid', activation = network_activation, W_regularizer =
       W_regularizer, activity_regularizer = activity_regularizer, init =
      weight_init))
168     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
169     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
170
171     model.add(Convolution2D(128, 3, 3, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
172     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
173     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
```

```
174
175     model.add(Convolution2D(256, 2, 2, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
176     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = False))
177     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
178
179     model.add(Flatten())
180     model.add(Dense(2048, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
181     model.add(Activation(network_activation))
182     if batch_Norm: model.add(BatchNormalization())
183     model.add(Dropout(dropout_p))
184
185     model.add(Dense(1024, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
186     model.add(Activation(network_activation))
187     if batch_Norm: model.add(BatchNormalization())
188     model.add(Dropout(dropout_p))
189
190     model.add(Dense(nb_classes, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
191     model.add(Activation(final_activation))
192
193     model.compile(loss = loss,
194                   optimizer = optimizer,
195                   class_mode=class_mode)
196     return model
197
198   elif model_no==5:
199     '''
200     Subtypes: None or 1,2,3,4,5,6,7.
201     '''
202     model = Sequential()
203     model.add(Convolution2D(64, 5, 5, input_shape=(channels, 51, 51),
        border_mode = 'valid', activation = network_activation, W_regularizer =
         W_regularizer, activity_regularizer = activity_regularizer, init =
        weight_init))
204     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
205     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
206
207     model.add(Convolution2D(128, 5, 5, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
208     model.add(MaxPooling2D(pool_size=(2, 2), ignore_border = True))
```

```
209     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
210
211     model.add(Convolution2D(256, 5, 5, border_mode = 'valid', activation =
        network_activation, W_regularizer = W_regularizer, activity_regularizer
        = activity_regularizer, init = weight_init))
212     if batch_Norm: model.add(BatchNormalization(epsilon=1e-6, weights=None)
        )
213
214     model.add(Flatten())
215     model.add(Dense(2048, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
216     model.add(Activation(network_activation))
217     if batch_Norm: model.add(BatchNormalization())
218     model.add(Dropout(dropout_p))
219
220     model.add(Dense(1024, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
221     model.add(Activation(network_activation))
222     if batch_Norm: model.add(BatchNormalization())
223     model.add(Dropout(dropout_p))
224
225     model.add(Dense(nb_classes, W_regularizer = W_regularizer,
        activity_regularizer = activity_regularizer, init = weight_init))
226     model.add(Activation(final_activation))
227
228     model.compile(loss = loss,
229                   optimizer = optimizer,
230                   class_mode = class_mode)
231     return model
232
233   else:
234     print 'no such model'
```

A.3 FITS Reading Script

```
1  import pyfits
2  import os
3  import numpy as np
4  import cPickle
5  from sklearn.cross_validation import train_test_split
6  import skimage
7  import theano
8
9  def floatX(X):
10   return np.asarray(X, dtype=theano.config.floatX)
```

```
11
12
13 def Build_img(folder, file_id, and_search_img=False, normalise=True,
       inner_folder=''):
14   #print 'file_id: ', file_id
15
16   img_g = pyfits.getdata(folder + '/' + 'cutObj-' + 'g-' + file_id + '.diff
       .fit', 0)
17   img_i = pyfits.getdata(folder + '/' + 'cutObj-' + 'i-' + file_id + '.diff
       .fit', 0)
18   img_r = pyfits.getdata(folder + '/' + 'cutObj-' + 'r-' + file_id + '.diff
       .fit', 0)
19   if and_search_img==True:
20     search_img_g = pyfits.getdata(folder + '/' + 'cutObj-' + 'g-' + file_id
       + '.diff.im.fit', 0)
21     search_img_i = pyfits.getdata(folder + '/' + 'cutObj-' + 'i-' + file_id
       + '.diff.im.fit', 0)
22     search_img_r = pyfits.getdata(folder + '/' + 'cutObj-' + 'r-' + file_id
       + '.diff.im.fit', 0)
23     img_out = np.dstack((img_g, img_i, img_r, search_img_g, search_img_i,
       search_img_r))
24   else:
25     img_out = np.dstack((img_g, img_i, img_r))
26
27   if normalise:
28     img_out = img_out.astype('float32') / 255.0 # normalise and convert to
       float
29   return img_out
30
31
32 def Make_4tensor_from_folder(folder, file_ids, and_search_img=False):
33   print '4tensor', folder
34   four_tensor = None
35   for file_id in file_ids:
36     if and_search_img==False:
37       tensor_img = Build_img(folder, file_id).transpose(2, 0, 1).reshape(1,
       3, 51, 51)
38     else:
39       tensor_img = Build_img(folder, file_id, True).transpose(2, 0, 1).
       reshape(1, 6, 51, 51)
40     if four_tensor==None:
41       four_tensor = tensor_img
42     else:
43       four_tensor = np.vstack((four_tensor, tensor_img))
44   return four_tensor
45
46
47 def Extract_file_id_list(file_name):
```

```
48    #print 'extracting ID list'
49    fileObj = open(file_name, 'r')
50    file_list = fileObj.read().split('\n')
51    fileObj.close()
52    return file_list
53
54
55  def Redefine_categories(input_array, list_of_categories,
        corresponding_list_of_changes):
56    print 'Changed categories'
57    new_array = np.zeros(input_array.size)
58    for old, new in zip(list_of_categories, corresponding_list_of_changes):
59      new_array[input_array==old] = new
60    return new_array
61
62
63  def Save_to_binary(list_of_np_arrays, file_path='save_file'):
64    save_file = open(file_path, 'wb') #will overwrite file
65    for item in list_of_np_arrays:
66      cPickle.dump(item, save_file, -1)
67    save_file.close
68
69
70  def Load_from_binary(file_path='save_file', category_change_list = None):
71    print 'Loading data from file'
72    fileObj = open(file_path, 'rb')
73    train_x = cPickle.load(fileObj)
74    test_x = cPickle.load(fileObj)
75    train_y = cPickle.load(fileObj)
76    test_y = cPickle.load(fileObj)
77    IDList_train = cPickle.load(fileObj)
78    IDList_valid = cPickle.load(fileObj)
79    fileObj.close()
80    if category_change_list is not None:
81      sorted_current_categories = np.sort(np.unique(test_y))
82      train_y = Redefine_categories(train_y, sorted_current_categories.tolist
        (), category_change_list)
83      test_y = Redefine_categories(test_y, sorted_current_categories.tolist()
        , category_change_list)
84
85    return floatX(train_x), floatX(test_x), floatX(train_y), floatX(test_y),
        IDList_train, IDList_valid
86
87
88  def Load_all_data(category_change_list=None, and_search_img=False,
        save_as_binary=False):
89    print 'Loading all data...'
90    train_x, test_x, train_y, test_y = None, None, None, None
```

```
91  IDList_train = list()
92  IDList_valid = list()
93  file_split_lists = next(os.walk('70_30_splits'))[2]
94  print 'File Split lists', file_split_lists
95
96  for txt_file in file_split_lists:
97    folder_number, tr_or_tst_type = txt_file.split('_')[0], txt_file.split(
    '_')[2].split('.')[0]
98    print 'folder_number', folder_number
99    print 'type', tr_or_tst_type
100   id_contents = Extract_file_id_list('70_30_splits/' + txt_file)
101
102   image_stack = Make_4tensor_from_folder(folder_number, id_contents,
    and_search_img)
103   print 'stack shape', image_stack.shape
104
105   if tr_or_tst_type=='train':
106     IDList_train.extend(id_contents)
107     if train_x==None:
108   train_x = image_stack
109   train_y = np.array([int(folder_number)]*len(id_contents))
110     else:
111   train_x = np.vstack((train_x, image_stack))
112   train_y = np.hstack((train_y, np.array([int(folder_number)]*len(
    id_contents))))
113
114
115   elif tr_or_tst_type == 'test':
116     IDList_valid.extend(id_contents)
117     if test_x==None:
118   test_x = image_stack
119   test_y = np.array([int(folder_number)]*len(id_contents))
120     else:
121   test_x = np.vstack((test_x, image_stack))
122   test_y = np.hstack((test_y, np.array([int(folder_number)]*len(
    id_contents))))
123
124 print train_x.shape
125 print train_y.shape
126 print test_x.shape
127 print test_y.shape
128
129 if category_change_list is not None:
130   #print train_y
131   sorted_current_categories = np.sort(np.unique(train_y))
132   #print sorted_current_categories
133   #assert len(category_change_list) == len(sorted_current_categories)
```

```
134     train_y = Redefine_categories(train_y, sorted_current_categories.tolist
        (), category_change_list)
135     test_y = Redefine_categories(test_y, sorted_current_categories.tolist()
        , category_change_list)
136
137   if save_as_binary:
138     Save_to_binary((train_x, test_x, train_y, test_y, IDList_train,
        IDList_valid), file_path = save_as_binary)
139   return train_x, test_x, train_y, test_y, IDList_train, IDList_valid
140
141
142 def Train_valid_test_split(X, Y, train_frac=0.6, valid_frac=0.2, test_frac
        =0.2, random_state=7):
143   assert train_frac + valid_frac + test_frac == 1.
144
145   x_temp, x_test, y_temp, y_test = train_test_split(X, Y, test_size =
        test_frac, random_state=random_state)
146   not_copied_seed = 100 + random_state
147   x_train, x_valid, y_train, y_valid = train_test_split(x_temp, y_temp,
        test_size=valid_frac, random_state=not_copied_seed)
148   return floatX(x_train), floatX(x_valid), floatX(x_test), floatX(y_train),
         floatX(y_valid), floatX(y_test)
```

Bibliography

[1] P. A. Abell, J. Allison, S. F. Anderson, J. R. Andrew, J. R. P. Angel, L. Armus, D. Arnett, S. J. Asztalos, T. S. Axelrod, S. Bailey, et al. LSST Science Book, Version 2.0. *arXiv Preprint:0912.0201*, 2009.

[2] M. Jurić, J. Kantor, K. T. Lim, R. H. Lupton, G. Dubois-Felsmann, T. Jenness, T. S. Axelrod, J. Aleksić, R. A. Allsman, Y. AlSayyad, et al. The LSST Data Management System. *arXiv Preprint:1512.07914*, 2015.

[3] K. D. Borne. A Machine Learning Classification Broker for the LSST Transient Database. *Astronomische Nachrichten*, 329:255–258, 2008.

[4] Siri, 2014. URL https://www.apple.com/ios/siri/.

[5] M. M. Waldrop. No Drivers Required. *Nature*, 518:20, 2015.

[6] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the Gap to Human-level Performance in Face Verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[7] L. du Buisson, N. Sivanandam, B. A. Bassett, and M. Smith. Machine Classification of Transient Images. *Proceedings of the International Astronomical Union*, 10:288–291, 2014.

[8] P. Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012.

[9] P. Domingos. A Few Useful Things to Know About Machine Learning. *Communications of the ACM*, 55:78–87, 2012.

[10] C. Sammut and G. I. Webb. *Encyclopaedia of Machine Learning*. Springer, 2011.

[11] Linear Regression, 2016. URL https://en.wikipedia.org/wiki/Simple_linear_regression#/media/File:Linear_regression.svg.

[12] E. Alpaydin. *Introduction to Machine Learning*. MIT press, 2004.

[13] Classification, 2016. URL https://en.wikipedia.org/wiki/Kernel_method#/media/File:Kernel_Machine.svg.

[14] Y. LeCun and C. Cortes. The MNIST Database. *Tomado en octubre de*, 2007.

[15] D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[16] Clustering, 2016. URL https://en.wiktionary.org/wiki/cluster_analysis#/media/File:Cluster-2.svg.

[17] Y. Bengio. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2:1–127, 2009.

[18] B. Schölkopf. Artificial Intelligence: Learning to See and Act. *Nature*, 518:486–487, 2015.

[19] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[20] Tin Kam Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE, 1995.

[21] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[22] M. Caudill. Neural Nets Primer, Part VI. *AI Expert*, 4:61–67, 1989.

[23] Neuron, 2016. URL http://www.wpclipart.com/medical/anatomy/cells/neuron/neuron.png.html.

[24] W. S. McCulloch and W. Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[25] Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, A. Culotta, et al. Advances in Neural Information Processing Systems 22. 2009.

[26] R. Kurzweil. *The Singularity is Near: When Humans Transcend Biology*. Penguin, 2005.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Representations by Back-propagating Errors. *Cognitive Modeling*, 5:1, 1988.

[28] D. Mo. A Survey on Deep Learning: One Small Step Toward AI. *Dept. Computer Science, Univ. of New Mexico, USA*, 2012.

[29] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Springer Science & Business Media, 2013.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[31] M. Minsky and S. Papert. *Perceptrons*. MIT press, 1988.

[32] Y. Bar-Yam. *Dynamics of Complex Systems*, volume 213. Addison-Wesley Reading, MA, 1997.

[33] C. Gershenson. Artificial Neural Networks for Beginners. *arXiv Preprint:0308031*, 2003.

[34] H. Wang and B. Raj. A Survey: Time Travel in Deep Learning Space: An Introduction to Deep Learning Models and How Deep Learning Models Evolved from the Initial Ideas. *arXiv Preprint:1510.04781*, 2015.

[35] B. Kröse and P. van der Smagt. *An Introduction to Neural Networks*. Citeseer, 1993.

[36] K. Hornik, M. Stinchcombe, and H. White. Multilayer Feed-forward Networks are Universal Approximators. *Neural Networks*, 2:359–366, 1989.

[37] D. E. Rumelhart and J. L. McClelland. The PDP Research Group: Parallel Distributed Processing: Explorations in the Microstructure of Cognition. *Foundations*, 1, 1986.

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

[39] Back-propagation, 2014. URL https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks

[40] Neural Net, 2016. URL https://theclevermachine.files.wordpress.com/2014/09/perceptron2.png.

[41] M. A. Nielsen. Neural Networks and Deep Learning, 2015. URL http://neuralnetworksanddeeplearning.com/chap2.html.

[42] Y. A. LeCun, L. Bottou, G. B. Orr, and K. Müller. Efficient Backprop. In *Neural Networks: Tricks of the Trade*, pages 9–48. Springer, 2012.

[43] Y. Bengio. Practical Recommendations for Gradient-based Training of Deep Architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[44] P. J. Werbos. Back-propagation Through Time: What it Does and How to Do It. *Proceedings of the IEEE*, 78:1550–1560, 1990.

[45] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[46] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[47] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k2)$. In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[48] M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv Preprint:1212.5701*, 2012.

[49] D. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv Preprint:1412.6980*, 2014.

[50] Feature Scaling, 2016. URL http://openclassroom.stanford.edu/MainFolder/VideoPage.php?course=MachineLearning&video=03.1-LinearRegressionII-FeatureScaling&speed=100/.

[51] A. Karpathy. Preprocessing 1, 2016. URL https://cs231n.github.io/assets/nn2/prepro1.jpeg.

[52] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv Preprint:1502.03167*, 2015.

[53] V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[54] A. Karpathy. Sigmoid, 2016. URL https://cs231n.github.io/assets/nn1/sigmoid.jpeg.

[55] T. M. Breuel. The Effects of Hyper-parameters on SGD Training of Neural Networks. *arXiv Preprint:1508.02788*, 2015.

[56] A. Krizhevsky and G. E. Hinton. Learning Multiple Layers of Features from Tiny Images. 2009.

[57] Neural Network Initialization, 2014. URL https://cs231n.github.io/neural-networks-2/.

[58] X. Glorot and Y. Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[59] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2013.

[60] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column Deep Neural Networks for Image Classification. In *Computer Vision and Pattern Recognition 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[61] CUDA Nvidia. C Programming Guide Version 4.0. *Nvidia Corporation*, 2011.

[62] X. Chen and X. Lin. Big Data Deep Learning: Challenges and Perspectives. *IEEE Access*, 2:514–525, 2014.

[63] W. Ding, R. Wang, F. Mao, and G. Taylor. Theano-based Large-scale Visual Recognition with Multiple GPUs. *arXiv Preprint:1412.2302*, 2014.

[64] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv Preprint:1605.02688*, 2016.

[65] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2016.

[66] Y. LeCun, Y. Bengio, and G. E. Hinton. Deep Learning. *Nature*, 521:436–444, 2015.

[67] D. H. Wolpert. The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation*, 8:1341–1390, 1996.

[68] V. Cherkassky and F. M. Mulier. *Learning from Data: Concepts, Theory, and Methods*. John Wiley & Sons, 2007.

[69] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the Devil in the Details: Delving Deep into Convolutional Nets. *arXiv Preprint:1405.3531*, 2014.

[70] S. Goyal and P. Benjamin. Object Recognition Using Deep Neural Networks: A Survey. *arXiv Preprint:1412.3684*, 2014.

[71] K. Chatfield, R. Arandjelović, O. Parkhi, and A. Zisserman. On-the-fly Learning for Visual Search of Large-scale Image and Video Datasets. *International Journal of Multimedia Information Retrieval*, 4:75–93, 2015.

[72] C. J. Lintott, K. Schawinski, A. Slosar, K. Land, S. Bamford, D. Thomas, M. J. Raddick, R. C. Nichol, A. Szalay, D. Andreescu, et al. Galaxy Zoo: Morphologies Derived from Visual Inspection of Galaxies from the Sloan Digital Sky Survey. *Monthly Notices of the Royal Astronomical Society*, 389:1179–1189, 2008.

[73] J. Deng, A. Berg, S. Satheesh, H. Su, A. Khosla, and L. Fei-Fei. Imagenet Large Scale Visual Recognition Competition 2012 (ILSVRC2012), 2012.

[74] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving Neural Networks by Preventing Co-adaptation of Feature Detectors. *arXiv Preprint:1207.0580*, 2012.

[75] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Drop-out: A Simple Way to Prevent Neural Networks from Over-fitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[76] H. Wu and X. Gu. Towards Drop-out Training for Convolutional Neural Networks. *Neural Networks*, 71:1–10, 2015.

[77] S. Marsland. *Machine Learning: An Algorithmic Perspective*. CRC press, 2015.

[78] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A CPU and GPU Math Compiler in Python. In *Proceedings 9th Python in Science Conference*, pages 1–7, 2010.

[79] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio. Theano: New Features and Speed Improvements. *arXiv Preprint:1211.5590*, 2012.

[80] L. Deng. Three Classes of Deep Learning Architectures and Their Applications: A Tutorial Survey. *APSIPA Transactions on Signal and Information Processing*, 2012.

[81] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep Learning Face Attributes in the Wild. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3730–3738, 2015.

[82] A. Karpathy. Classification, 2016. URL https://cs231n.github.io/assets/challenges.jpeg.

[83] M. Elawady. Sparse Coral Classification Using Deep Convolutional Neural Networks. *arXiv Preprint:1511.09067*, 2015.

[84] R. E. Baker and A. R. Bradlow. Variability in Word Duration as a Function of Probability, Speech Style, and Prosody. *Language and Speech*, 52:391–413, 2009.

[85] Y. LeCun, K. Kavukcuoglu, and C. Farabet. Convolutional Networks and Applications in Vision. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 253–256. IEEE, 2010.

[86] Y. LeCun and Y. Bengio. Convolutional Networks for Images, Speech, and Time Series. *The Handbook of Brain Theory and Neural Networks*, 3361:310, 1995.

[87] R. R. Salakhutdinov and G. E. Hinton. Deep Boltzmann Machines. In *AISTATS*, volume 1, page 3, 2009.

[88] R. R. Salakhutdinov. *Learning Deep Generative Models*. PhD thesis, University of Toronto, 2009.

[89] Y. Bengio, Y. LeCun, et al. Scaling Learning Algorithms Towards AI. *Large-scale Kernel Machines*, 34, 2007.

[90] P. Y. Simard, D. Steinkraus, and J. C. Platt. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *2013 12th International Conference on Document Analysis and Recognition*, volume 2, pages 958–958. IEEE Computer Society, 2003.

[91] T. P. Karnowski, I. Arel, and D. Rose. Deep Spatio-temporal Feature Learning with Application to Image Classification. In *ICML-10*, pages 883–888. IEEE, 2010.

[92] Q. V. Le et al. Building High-level Features Using Large Scale Unsupervised Learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[93] S. S. Farfade, M. J. Saberian, and L. Li. Multi-view Face Detection Using Deep Convolutional Neural Networks. In *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, pages 643–650. ACM, 2015.

[94] S. Ji, W. Xu, M. Yang, and K. Yu. 3D Convolutional Neural Networks for Human Action Recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35:221–231, 2013.

[95] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Large Vocabulary Continuous Speech Recognition with Context-dependent DBN-HMMs. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4688–4691. IEEE, 2011.

[96] D. Ferrucci, A. Levas, S. Bagchi, D. Gondek, and E. T. Mueller. Watson: Beyond Jeopardy! *Artificial Intelligence*, 199:93–105, 2013.

[97] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for Thin Deep Nets. *arXiv Preprint:1412.6550*, 2014.

[98] J. Ba and R. Caruana. Do Deep Nets Really Need to be Deep? In *Advances in Neural Information Processing Systems*, pages 2654–2662, 2014.

[99] D. Gabor. Theory of Communication. Part 1: The Analysis of Information. *Journal of the Institution of Electrical Engineers-Part III: Radio and Communication Engineering*, 93:429–441, 1946.

[100] P. Haffner, P. G. Howard, P. Simard, Y. Bengio, Y. LeCun, et al. High Quality Document Image Compression with "DjVu". *Journal of Electronic Imaging*, 7: 410–425, 1998.

[101] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. In *ICML-14*, pages 647–655, 2014.

[102] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv Preprint:1409.1556*, 2014.

[103] O. Abdel-Hamid, A. Mohamed, H. Jiang, and G. Penn. Applying Convolutional Neural Networks Concepts to Hybrid NN-HMM Model for Speech Recognition. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4277–4280. IEEE, 2012.

[104] T. N. Sainath, B. Kingsbury, A. Mohamed, and B. Ramabhadran. Learning Filter Banks Within a Deep Neural Network Framework. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 297–302. IEEE, 2013.

[105] Y. Anzai. *Pattern Recognition & Machine Learning*. Elsevier, 2012.

[106] V. Dumoulin and F. Visin. A Guide to Convolution Arithmetic for Deep Learning. *arXiv Preprint:1603.07285*, 2016.

[107] T. Wang, D. J. Wu, A. Coates, and A. Y. Ng. End-to-end Text Recognition with Convolutional Neural Networks. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 3304–3308. IEEE, 2012.

[108] N. M. Ball and R. J. Brunner. Data Mining and Machine Learning in Astronomy. *International Journal of Modern Physics D*, 19:1049–1106, 2010.

[109] M. Sokolova, N. Japkowicz, and S. Szpakowicz. Beyond Accuracy, F-score and ROC: A Family of Discriminant Measures for Performance Evaluation. In *Australasian Joint Conference on Artificial Intelligence*, pages 1015–1021. Springer, 2006.

[110] J. Davis and M. Goadrich. The Relationship Between Precision-Recall and ROC Curves. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 233–240. ACM, 2006.

[111] T. Fawcett. An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27: 861–874, 2006.

[112] W. Baade and F. Zwicky. Photographic Light-Curves of the Two Supernovae in IC 4182 and NGC 1003. *The Astrophysical Journal*, 88:411, 1938.

[113] L. Du Buisson. *Machine Learning in Astronomy.* PhD thesis, University of Cape Town, 2015.

[114] A. G. Riess, A. V. Filippenko, P. Challis, A. Clocchiatti, A. Diercks, P. M. Garnavich, R. L. Gilliland, C. J. Hogan, S. Jha, R. P. Kirshner, et al. Observational Evidence from Supernovae for an Accelerating Universe and a Cosmological Constant. *The Astronomical Journal*, 116:1009, 1998.

[115] S. Perlmutter, G. Aldering, G. Goldhaber, R. A. Knop, P. Nugent, P. G. Castro, S. Deustua, S. Fabbro, A. Goobar, D. E. Groom, et al. Measurements of Ω and Λ from 42 High-redshift Supernovae. *The Astrophysical Journal*, 517:565, 1999.

[116] D. Branch and G. A. Tammann. Type Ia Supernovae as Standard Candles. *Annual Review of Astronomy and Astrophysics*, 30:359–389, 1992.

[117] N. Takanashi, M. Doi, and N. Yasuda. Light-curve Studies of Nearby Type Ia Supernovae with a Multi-band Stretch Method. *Monthly Notices of the Royal Astronomical Society*, 389:1577–1592, 2008.

[118] K. N. Abazajian, J. K. Adelman-McCarthy, M. A. Agüeros, S. S. Allam, C. A. Prieto, D. An, K. S. J. Anderson, S. F. Anderson, J. Annis, N. A. Bahcall, et al. The Seventh Data Release of the Sloan Digital Sky Survey. *The Astrophysical Journal Supplement Series*, 182:543, 2009.

[119] C. Stoughton, R. H. Lupton, M. Bernardi, M. R. Blanton, S. Burles, F. J. Castander, A. J. Connolly, D. J. Eisenstein, J. A. Frieman, G. S. Hennessy, et al. Sloan Digital Sky Survey: Early Data Release. *The Astronomical Journal*, 123: 485, 2002.

[120] C. Alard and R. H. Lupton. A Method for Optimal Image Subtraction. *The Astrophysical Journal*, 503:325, 1998.

[121] R. Kessler, J. Marriner, M. Childress, R. Covarrubias, C. B. D'Andrea, D. A. Finley, J. Fischer, R. J. Foley, D. Goldstein, R. R. Gupta, et al. The Difference Imaging Pipeline for the Transient Search in the Dark Energy Survey. *The Astronomical Journal*, 150:172, 2015.

[122] J. K. Adelman-McCarthy, M. A. Agüeros, S. S. Allam, C. A. Prieto, K. S. J. Anderson, S. F. Anderson, J. Annis, N. A. Bahcall, C. A. L. Bailer-Jones, I. K. Baldry, et al. The Sixth Data Release of the Sloan Digital Sky Survey. *The Astrophysical Journal Supplement Series*, 175:297, 2008.

[123] J. A. Frieman, B. A. Bassett, A. Becker, C. Choi, D. Cinabro, F. DeJongh, D. L. Depoy, B. Dilday, M. Doi, P. M. Garnavich, et al. The Sloan Digital Sky Survey-II Supernova Survey: Technical Summary. *The Astronomical Journal*, 135:338, 2007.

[124] B. Dilday, R. Kessler, J. A. Frieman, J. Holtzman, J. Marriner, G. Miknaitis, R. C. Nichol, R. Romani, M. Sako, B. A. Bassett, et al. A Measurement of the Rate of Type Ia Supernovae at Redshift $z \approx 0.1$ from the First Season of the SDSS-II Supernova Survey. *The Astrophysical Journal*, 682:262, 2008.

[125] W. Skidmore, I. Dell'Antonio, M. Fukugawa, A. Goswami, L. Hao, D. Jewitt, G. Laughlin, C. Steidel, P. Hickson, L. Simard, et al. Thirty Meter Telescope Detailed Science Case: 2015. *Research in Astronomy and Astrophysics*, 15:1945, 2015.

[126] A. M. Mickaelian. Astronomical Surveys and Big Data. *Baltic Astronomy*, 24, 2015.

[127] S. Bailey, C. Aragon, R. Romano, R. C. Thomas, B. A. Weaver, and D. Wong. How to Find More Supernovae with Less Work: Object Classification Techniques for Difference Imaging. *The Astrophysical Journal*, 665:1246, 2007.

[128] R. A. Romano, C. R. Aragon, and C. Ding. Supernova Recognition Using Support Vector Machines. In *2006 5th International Conference on Machine Learning and Applications (ICMLA-06)*, pages 77–82. IEEE, 2006.

[129] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[130] S. Russell and P. Norvig. A Modern Approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

[131] I. Jolliffe. *Principal Component Analysis*. Wiley Online Library, 2002.

[132] A. J. Izenman. Linear Discriminant Analysis. In *Modern Multivariate Statistical Techniques*, pages 237–280. 2013.

[133] L. Maaten and G. E. Hinton. Visualizing Data Using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[134] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, et al. API Design for Machine Learning Software: Experiences from the Scikit-learn Project. In *European Conference on Machine Learning and Principles and Practices of Knowledge Discovery in Databases*, 2013.

[135] M. Wattenberg, F. Vigas, and I. Johnson. How to Use t-SNE Effectively. *Distill*, 2016. URL http://distill.pub/2016/misread-tsne.

[136] F. Chollet. Keras, 2015. URL https://github.com/fchollet/keras.

[137] R. J. Hanisch, A. Farris, E. W. Greisen, W. D. Pence, B. M. Schlesinger, P. J. Teuben, R. W. Thompson, and A. Warnock III. Definition of the Flexible Image Transport System (FITS). *Astronomy & Astrophysics*, 376:359–380, 2001.

[138] P. E. Barrett and W. T. Bridgman. PyFITS, a FITS Module for Python. In *Astronomical Data Analysis Software and Systems VIII*, volume 172, page 483, 1999.

[139] T. E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9:10–20, 2007.

[140] A. Liaw and M. Wiener. Classification and Regression by Random Forest. *R News*, 2:18–22, 2002.