Object Detection in Road Images

UNIVERSITY *of* LIMERICK

OLLSCOIL LUIMNIGH

Final Year Project 2019

Author: Rory Egan

ID: 15178668

Bachelor of Science in Computer Systems (LM051)

Supervised by: J.J. Collins

Abstract

This project will be a research-oriented project centred around the use of Convolutional Neural Networks (CNN's) for object detection in images from the perspective of a self-driving car. The area of self-driving cars is a rapidly expanding field, with significant investments having taken place over the past several years. Many companies have chosen to use cameras as one of the primary sensors deployed on their prototype cars in conjunction with several other sensors to allow a vehicle to understand its environment and orient itself appropriately. This project will focus on the way these cameras can allow the vehicle to detect different objects in its environment. An important challenge within the field of machine vision in self-driving cars is reliably classifying multiple objects at once within an image. The current standard Machine Learning class of algorithms for object classification are CNNs. This project will investigate the use of CNN's to analyse colourised 2D images that depict typical scenes presented to a self-driving car and classify the different objects within them. The initial goal is to research and outline the basics of how CNN's operate, then choose a particular CNN paradigm and train an object detector using this paradigm. The CNN should be able to reliably classify major obstacles that are important to the function of a self-driving car such as other vehicles, pedestrians etc. The classifier should aim to be robust to adverse conditions creating variation within the images such as rain, snow etc. This project will also investigate several different types of CNN and document the differences in the ensuing results.

Acknowledgements

First and foremost, I would like to thank my project supervisor, J.J. Collins. I initially came to him in 3rd year with a plan to base my final year project around the subject of Object Detection and Convolutional Neural Networks, and his guidance and knowledge have been instrumental in shaping this project ever since. He has been an exemplary project supervisor, making himself available whenever possible to help.

Secondly, I would like to thank Jamie Mac Manus for the four years of university we have spent together. Without his unwavering discipline and work ethic serving as an example to myself, I can honestly say that my grades would be nowhere near the levels they are at now.

Special thanks are also due to Tom Barrett. His final year project served as the inspiration for this project, and it was him who first set me on the path of investigating self-driving car related datasets.

I would also like to thank Kevin Moynihan and Dan Noonan for all the projects we have worked on together, with them proving to be exemplary teammates.

I have many thanks due to my family and girlfriend, without their continued support and backing I would never have come to the end of my degree. Particular thanks are due to my father for first setting me on the path of Software Development.

Lastly, I would like to thank all of the CSIS staff who have provided guidance and support to me during my time at UL. Without them, none of this would have been possible.

Contents

List of Figures

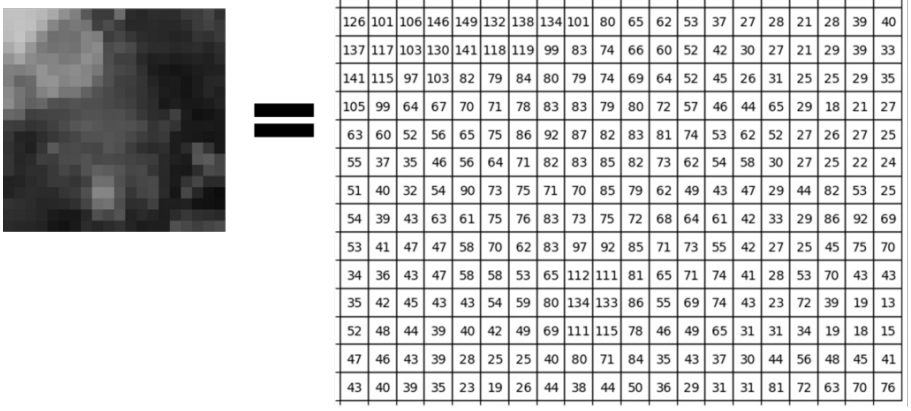
List of Tables

Chapter 1

Introduction

1.1 Overview of Problem Area

Object detection is the process of locating and identifying different types of objects within images (Verschae and Ruiz-del-Solar, 2015), and is an area of much research in the field of Computer Vision. To gain an understanding of what makes this a difficult task for a computer, one must first understand how images are interpreted by computers. Images are represented as matrices of values, with each value corresponding to a pixel in the image (Learned-Miller, 2011). Images can be broadly broken down into two groups, grayscale and colour. A grayscale image will be represented as a 2D array of values ranging from 0 to 255, with each value representing the intensity of that pixel, as seen in Figure 1.1. A value of 0 represents black and a value of 255 represents white. In grayscale images, each pixel therefore represents one colour channel (gray). In colour images however we have multiple different channels for red, green and blue. In order to represent this, a colour image will be represented as a 3D array of pixel values, with each pixel having 3 values between 0 and 255 associated with it. These three values represent the intensity of the three colour channels at that pixel.



193	194	190	184	169	144	128	89	60	60	109	44	40	46	45	45	58	61	72	50
191	195	192	173	134	114	121	116	64	77	60	41	41	43	41	41	46	60	67	57
187	196	178	139	110	113	112	132	126	61	70	55	61	42	40	37	39	53	50	59
170	186	151	122	114	117	114	131	139	76	83	74	52	45	45	41	43	46	39	47
147	163	139	131	132	121	125	143	132	78	64	64	42	33	35	30	32	36	33	43
129	126	132	148	134	136	141	133	121	81	72	67	49	30	24	21	25	30	32	34
126	101	106	146	149	132	138	134	101	80	65	62	53	37	27	28	21	28	39	40
137	117	103	130	141	118	119	99	83	74	66	60	52	42	30	27	21	29	39	33
141	115	97	103	82	79	84	80	79	74	69	64	52	45	26	31	25	25	29	35
105	99	64	67	70	71	78	83	83	79	80	72	57	46	44	65	29	18	21	27
63	60	52	56	65	75	86	92	87	82	83	81	74	53	62	52	27	26	27	25
55	37	35	46	56	64	71	82	83	85	82	73	62	54	58	30	27	25	22	24
51	40	32	54	90	73	75	71	70	85	79	62	49	43	47	29	44	82	53	25
54	39	43	63	61	75	76	83	73	75	72	68	64	61	42	33	29	86	92	69
53	41	47	47	58	70	62	83	97	92	85	71	73	55	42	27	25	45	75	70
34	36	43	47	58	58	53	65	112	111	81	65	71	74	41	28	53	70	43	43
35	42	45	43	43	54	59	80	134	133	86	55	69	74	43	23	72	39	19	13
52	48	44	39	40	42	49	69	111	115	78	46	49	65	31	31	34	19	18	15
47	46	43	39	28	25	25	40	80	71	84	35	43	37	30	44	56	48	45	41
43	40	39	35	23	19	26	44	38	44	50	36	29	31	31	81	72	63	70	76

Figure 1.1: Pixel Representation of Grayscale Image

The overall goal for object detection algorithms is that they should be able to receive these image representations as input data and return values for the probabilities of different classes existing within the image, as well as provide some information as to where the algorithm predicts that the objects are located, typically in the form of a bounding box (Amit, 2002). This can be seen visualised in Fig 1.2. The overall aim of this project is to train an object detection algorithm that works on a set of images extracted from cameras mounted on cars, containing multiple different objects such as pedestrians, vehicles and traffic lights.

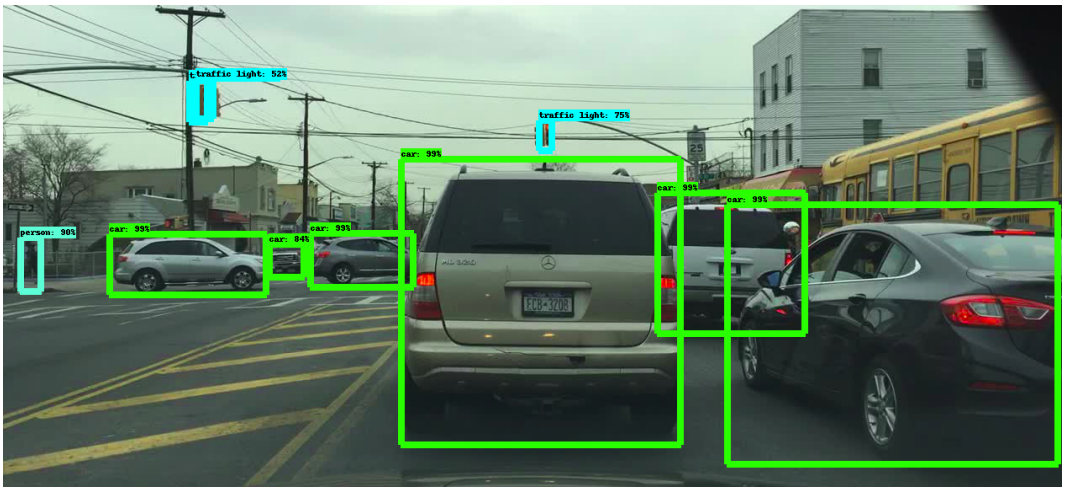


Figure 1.2: Object Detector in action

There are many factors that influence the degree to which an object detector is successful. When dealing with images that are extracted from real-life scenarios issues such as illumination and camera focus can cause an image to appear blurry or distorted. As can be seen in Fig 1.2 above, objects within images can become partially or fully obscured by other objects. Objects can also change depending on their distance from the camera, further compunding these issues. This means that it can become very difficult to perform object detection to a high degree of certainty upon real-life data (Z.-Q. Zhao et al., 2019). Thankfully, a relatively new form of Machine Learning algorithm known as the Convolutional Neural Network

(CNN) has recently risen to prominance that has proven to be quite robust to these complex image-based tasks. This project is based upon the investigation of how CNN's perform upon real-life image data in the context of scenes such as those presented to a self-driving car.

1.2 Objectives

1.2.1 Primary Objectives

Utilise a CNN for Object Classification

The core concept of this project is the use of CNN's for the detection of certain objects present within driving-based images. As such, the primary objective for this project is to research and implement a CNN that will return an acceptable level of accuracy on the testing data. The Berkeley Deep Drive dataset utilised by this project has a related paper in which object detection experiments are run on the dataset and performance metrics from these experiments are recorded (Yu et al., 2018). A major objective for this project therefore is to achieve similar results to those achieved in this paper. As an understanding of how CNN's are evaluated is required in order to properly expand upon the results obtained in the paper, the results will be discussed further in a later section.

Provide a Visual Interaction With The Trained CNN

Once CNN's have been trained to acceptable accuracy levels during the Empirical Studies section, a basic application should be implemented in order for users to observe the CNN in action upon testing data. To this end, a Flask application is to be developed that will allow users to upload images upon which object detection will take place. As the Berkeley Deep Drive dataset contains videos as well as images, object detection should take place on a small number of test videos in order to provide users an example of how a CNN operates in real-time.

1.2.2 Secondary Objectives

Understanding of CNN's

As this is a primarily research based project, one of the main objectives is to gain a deep understanding of how CNN's operate. The following sections of this report should demonstrate a depth of knowledge around the theory of how CNN's work, as well as demonstrate an ability to put these theories into practice. The section on Empirical Studies in particular should demonstrate an ability to forensically examine the results produced from CNN's and provide insight into how these results are achieved.

1.3 Contribution

Although results of applying a CNN to the Berkeley Deep Drive dataset have been published in (Yu et al., 2018), the authors of this study only used one particular architecture of CNN to obtain their results. This report outlines results obtained using two different types of CNN architecture across a range of different types of experiment. As the Berkeley Deep Drive dataset is a newly released dataset from 2018, a large body of research has yet to be carried out on the dataset. As such, this report may be of some small use to aid in avoiding some of the problems encountered during this project.

1.4 Methodologies

1. Define the Research Area: The first step in this project was to define which problem domain to base this FYP around. As an avid SCUBA diver, the initial area I investigated was the field of fish classification in images and videos collected from Irish waters. This area appealed to me as I wanted to investigate the possibility of collecting the dataset myself using underwater camera equipment. However I quickly abandoned this area for several reasons:

    (a) Equipment: Underwater camera equipment that is capable of taking high resolution images and videos is extremely expensive. Due to the strenuous nature of diving in rough Irish waters,

equipment with excellent stabilisation software would be required in order to reliably obtain non-blurred images.

(b) Domain Knowledge: As I am not an expert in the field of fish identification, I would be unable to reliably annotate species present in the training images. This would require enlisting the help of a third-party expert in the field of fish species.

(c) Time Constraints: There are no datasets currently available that contain annotated fish species in underwater images or videos in Northern Atlantic waters. As such, the full dataset would need to be collated and annotated myself. With time constraints present for this project, it would not be feasible to carry out all this work in time to reach project deadlines.

With these issues present for my initial choice of problem domain, I decided to instead focus on the area of road images as there have been several large-scale annotated datasets released in recent years that I could leverage. Moreover, after receiving a graduate offer from Jaguar Land Rover I decided that this project could serve as an excellent introduction into the problems faced by automotive companies in the field of Computer Vision.

2. Background Research: The next step was to carry out a literature review around the topic of Neural Computing, specifically CNN's. Papers dealing specifically with the challenges presented during object detection were the main focus of the literature review. In an attempt to demonstrate an understanding of how CNN's operate, the literature review has been summarised in the following sections of this report.

3. Gain a hands-on knowledge of tensorflow: As tensorflow was the tool utilised to implement CNN's during this project, a Udemy tutorial was first followed in an attempt to gain an understanding of using tensorflow in a hands-on manner (*Complete Guide to Tensorflow for Deep Learning with Python* 2018).

4. Carry out Empirical Studies: Empirical Studies were a major point of focus for this project. Experimentation was carried out on the Berkeley Deep Drive dataset using a range of different types of CNN. Results from these experiments were documented and investigated in

order to gain an understanding of what techniques work best for the dataset. The results from the experiments were then compared to results carried out in (Yu et al., 2018).

5. Build an application: Once Empirical Studies had been carried out and CNN's had been trained on the Berkeley Deep Drive dataset to a satisfactory degree, a prototype application was built to allow users to observe the Object Detection taking place.

1.5 Motivations

The main motivation behind this project for me is working within the field of self-driving cars. I find this particular area fascinating due to the broad range of technologies present within these cars. I have previously worked a summer internship at Jaguar Land Rover in Shannon working within the ADAS (driver assistance) team. I was exposed to a broad range of different technologies, from Computer Vision and Machine Learning based teams to Big Data pipelines concerned with offboarding data from test cars. This internship and the receipt of an offer to return to the company as a graduate has motivated me to further my study within this field. I have found that these technologies are much more interesting to me than many more conventional potential areas of work. With ever increasing amounts of automotive manufacturers investigating this field, I feel like a final year project focusing on self-driving cars could be very beneficial to my career going forward.
I am also very interested in self-driving cars due to the far-reaching safety implications of the technology. Hundreds of people die on Irish roads every year, with driver error being the primary cause of fatalities. Driver distraction has been reported as the cause of one in seven accidents that take place in Ireland (Burns, 2014). Self-driving cars have the potential to significantly increase road safety once fully autonomous capability has been reached (Milakis, Van Arem, and Van Wee, 2017), and I find it very motivating to work within a field that genuinely has the potential to save lives. Outside of the direct safety implications of self-driving cars, they have also shown the potential to have positive environmental effects through a reduction of fuel wastage (Greenblatt and Saxena, 2015). I believe that this technology is certainly a field in which I can leave a

positive and lasting impression on the world.

Chapter 2

Background Research

2.1 Autonomous Vehicles Overview

Before the areas of Machine Learning and CNN's are discussed, it is important to first put into context why these areas matter to an autonomous vehicle. Although driver assistance features such as lane assist, parking assist etc. are prevalent in much of the luxury car market today, a truly self-driving car has yet to be brought to market. A truly self-driving car is a wheeled, autonomous robot that is capable of traveling between destinations without human intervention based on information received from automotive sensors, regardless of road or weather conditions (J. Zhao, Liang, and Chen, 2018). Autonomous vehicles are discussed in the context of levels of autonomy, with each level representing a step closer to fully autonomous capabilities. The most widely accepted definition for the differing levels of autonomy was published by SAE international, with six distinct levels being distinguised by the level of driver attentiveness and involvement required (international, 2016). This publication breaks down the levels of autonomy as follows:

- Level 0: System may issue warnings, however no sustained vehicle control is present.
- Level 1: Control of the vehicle is shared between the driver and the system - for example adaptive cruise control.
- Level 2: System is capable of taking full control of the vehicle under certain conditions such as highway driving, however the driver must

be prepared to intervene at any moment.

- Level 3: Driver is capable of taking their attention away from driving, however they must be capable of intervening when the system alerts.
- Level 4: Similar to level 3, however less driver attention is required. Inside of specific "geofenced" areas and favourable weather conditions, the vehicle is capable of driving in an autonomous manner. However, the vehicle must be able to abort the trip if driver does not take control when the system alerts them - ie. park the car at the side of the road.
- Level 5: Full autonomy, no human intervention required at any point during the trip.

The technologies present within a self-driving car can be split into four distinct categories - environment perception, car navigation, path planning, and car control (J. Zhao, Liang, and Chen, 2018). The technologies focused on in this project relate to the perception module, which handles how the vehicle perceives its surrounding environment, allowing the vehicle to make informed control and planning decisions. The perception module generally consists of several different types of sensors with the main types consisting of radar, LIDAR and camera systems. These sensors operate as follows:

- Camera: Standard camera units, typically producing 30 or 60 frames per second.
- Radar: Standing for Radio Detection and Ranging, it works by generating high-frequency radio waves which bounce back off objects and are then processed. The Doppler effect allows radar units to measure the speed of objects, allowing a self-driving car to calculate the relative velocities of other vehicles (Kocić, Jovičić, and Drndarević, 2018).
- LIDAR: Standing for Light Detection and Ranging, LIDAR is a relatively new form of sensor. Operates in a broadly similar manner to a radar unit, however it uses an infrared laser beam in the place of radio waves. Most LIDAR units consist of a swivelling device, allowing them to create a 3D map of objects around the vehicle.

Although each of these sensors has their own distinct advantages and disadvantages, this project focuses on the information created by camera sensors. The reason for this is due to the fact that cameras are widely accepted as a major part of any autonomous car system currently being developed (Kocić, Jovičić, and Drndarević, 2018). The reasons for camera sensors being so prevalent within the industry are as follows:

- Cost: Cameras are among the least expensive sensors available to automotive manufacturers. A major concern with other sensors such as LIDAR is the availability of sensors that are cheap enough to feasibly be mounted on production cars.
- Colour: Cameras are capable of returning colour information, which is an important part of many detection tasks.
- Power: Cameras typically require very little power input in order to run.

Their cost effectiveness and colourised output ensure that cameras are likely to remain an integral part of autonomous vehicle systems for years to come. Due to the immaturity and cost of LIDAR technologies, Tesla have controversially decided to eschew LIDAR sensors altogether in favour of camera, radar and ultrasonic sensors, with certain arguments being put forth that cameras will replace LIDAR entirely in the future (Harris, 2015). Regardless of which particular configuration of sensors are utilised in different autonomous vehicle solutions, cameras remain the main sensor utilised by most manufacturers (Kocić, Jovičić, and Drndarević, 2018). As such, it was decided that this project would focus on image data that has been generated from cameras.

Once data has been generated by a sensor, it must be acted upon in some way by the autonomous system. The different ways in which the system will act upon the received data is determined through the "learned" experience available to the system in the form of some type of Machine Learning algorithm (Surden and M.-A. Williams, 2016). The standard type of Machine Learning algorithm that is utilised for image based problems is the Neural Network, a class of Machine Learning algorithm modelled off the

neural pathways of the brain. Within this class of algorithm, a CNN's have become increasingly popular due to their suitability for tackling image based problems. Both Neural Networks and CNN's will be explained further in this paper, however it should now be apparent what role a CNN performs within the overall architecture of an autonomous vehicle.

2.2 Introduction to Machine Learning

A general definition of Machine Learning: "[Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed" - Arthur Samuel, 1959.
One of the goals for Machine Learning algorithms is automatically observing structures in data and fitting these structures to a model in order to allow people to interact with the data in a way that is humanly intuitive. Development within the field has progressed to the point where people interact with Machine Learning algorithms multiple times in their daily lives without noticing. Areas such as email spam detection, Facebook image tagging suggestions and voice-to-text are all examples of a broad range of Machine Learning algorithms that are commonly used.
Machine Learning can be roughly divided into two categories - Supervised and Unsupervised.

2.2.1 Supervised Learning

In Supervised Learning, algorithms are provided with some labelled input data which they attempt to learn patterns from. The algorithms will then attempt apply this learned experience to new unseen data and attempt to create their own labels for the data, with varying degrees of success. Prominent examples of the Supervised category of Machine Learning include Support Vector Machines, Linear Regression and the focus of this project, Convolutional Neural Networks (O'Shea and Nash, 2015).

2.2.2 Unsupervised Learning

Unsupervised Learning algorithms differ from Supervised Learning algorithms in that they are given data with no labels. They must then attempt to find some structure in this input data themselves with no given

direction or explicit programming. Popular Unsupervised Learning examples include clustering algorithms.

2.2.3 History of Neural Computing

Neural Networks are not a new concept - they were first proposed in 1943 by neurophysicists in the form of a primitive electrical circuit. The concept was studied up until the 1960's until it fell out of favour with researchers. Bold claims had been made by many researchers about the vast potential of the field, however a failure to back up these claims led to widespread skepticism about the true potential of Neural Computing. The area was in part hampered by the technology of the time - processing power available to researchers was very low. Additionally, in 1969 a paper was published introducing the XOR problem, which will be explained in later section. The paper stated that the research being carried out at the time on Neural Networks was fundamentally flawed and that the field would not experience any major successes (Minsky and Papert, 1969). A revival was seen in the 1980's, when the concept of using multiple layers of neurons to create a network began to emerge, and the issues presented by the XOR problem were solved. From 1989 to 1994 Yann LeCun developed the LeNet architecture, one of the first examples of a CNN, which was used to recognise handwritten postal addresses. In around the 2010 the field of Deep Learning (explained below) experienced a surge in popularity, primarily due to the increase in processing power made available through GPU's. In 2012, Alex Krizevsky won the ImageNet competition, a popular image recognition competition. The architecture he used, known as AlexNet, achieved an error rate of 15%, which was approximately 10% better than the closest competition at the time. His architecture essentially scaled up the LeNet architecture into a larger, more complex network (Krizhevsky, Sutskever, and Hinton, 2012). This architecture caused a widespread adoption of CNN's within image based Machine Learning, and CNN's are now the de facto standard for many image processing tasks (O'Shea and Nash, 2015).

2.3 Introduction to Neural Networks and Deep Learning

Artificial Neural Networks (ANN's) are a particular Supervised Machine Learning paradigm modelled after the neural pathways present in the brain. The building blocks for every Neural Network are called perceptrons or nodes, and can be thought of as artificial neurons. A perceptron receives inputs with weights associated with them that show the importance of each input relative to the others. The perceptron applies a particular activation function with a bias attached to the weighted sums of the input, and an output is generated, as shown in Figure 2.1 (Géron, 2017). The Perceptron Training Algorithm determines what weights are selected in order to produce the correct output.

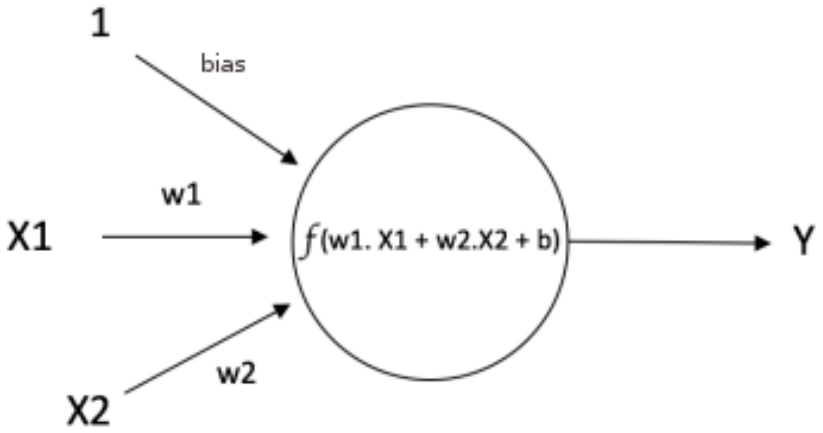


Figure 2.1: An Artificial Neuron

A single Perceptron is only able to classify linearly separable data (Kotsiantis, Zaharakis, and Pintelas, 2007). In order to classify data that is not linearly separable, techniques such as Multilayer Perceptrons must be used. These will be explained in a following section. Linear separability

refers to the ability for a single line to separate all members of a given class A from all members of a given class B. This is illustrated in Figure 2.2. The two classes present in the diagram can be easily seperated by finding an ideal line between the two classes.

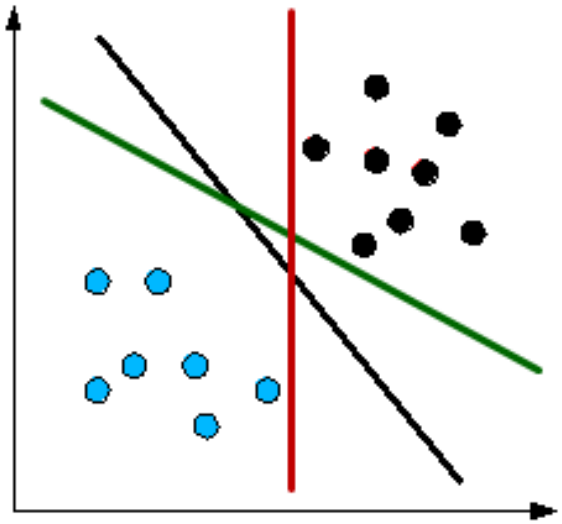


Figure 2.2: Linearly Separable Data

There are several different types of activation function, however every activation function takes in a single number as input and performs a certain mathematical operation on the number. The three most common activation functions generally encountered are ReLU, tanh and sigmoid. Rectified Linear Unit, or ReLU, takes in an input and replaces negative numbers with zero. The tanh function squashes the input to between the range between -1 and 1. Finally the sigmoid activation function takes the input and squashes it to the range between 0 and 1. In the field of CNN's ReLU is commonly used, as training times are significantly better when using this activation function. (Krizhevsky, Sutskever, and Hinton, 2012). These different types of activation function are visualised in Figure 2.3.

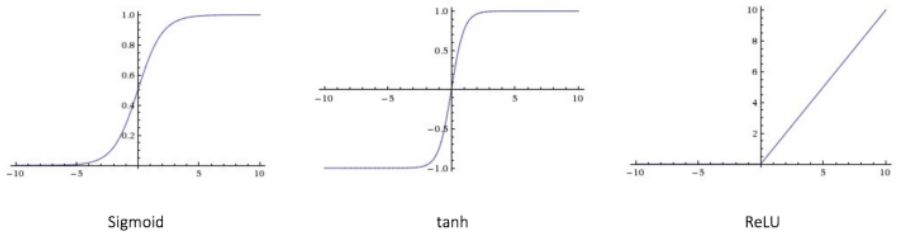


Figure 2.3: Activation Functions Visualised

A Neural Network consists of a series of interconnected layers of these artificial neurons, with the output of each layer of neurons serving as input for the next layer of neurons. The simplest and most common type of Neural Network is the feedforward neural network. It consists of multiple layers of neurons, with connections to all of the neurons in the preceding layer. Each connection or edge has a weight associated with it. There are three types of nodes - input nodes, hidden nodes and output nodes. As the name suggests, data in a feedforward neural network only moves forward through the network - into the input layer, through the hidden layers and then on to the output layer.

2.3.1 Input Nodes

Input nodes take in the input data fed into the network without performing any computation and pass the data on to the hidden nodes. These nodes make up the input layer of a neural network.

2.3.2 Hidden Nodes

The hidden nodes perform the actual computation, taking data from the input nodes and providing information to the output nodes. There can be multiple or zero hidden layers in a network, whereas there will only ever be one input and output layer.

2.3.3 Output Nodes

Like the input nodes, the output nodes do not perform any computation on the data - they simple take information from the hidden layers and expose

this to the outside. This will generally consist of a prediction.

2.3.4 XOR Problem

In the paper mentioned above, "Perceptrons: An Introduction to Computational Geometry", the Xor problem was first introduced. Xor is a function that given two binary inputs returns 0 if the inputs are equal and 1 if they are not. Xor is a classification problem with known expected results, therefore it is appropriate to utilise a Supervised Learning approach to solving it. However, the Xor problem is an example of a problem that is not linearly seperable. As mentioned previously, a single perceptron is not capable of predicting data that is not linearly seperable, therefore a single layered architecture of perceptrons is simply not capable of solving this problem. The only way for a Neural Network to solve this problem is to expand the number of layers in the architecture, adding a hidden layer. This type of architecture is known as the Multilayer Perceptron.

2.3.5 Multilayer Perceptron

A Multilayer Perceptron (MLP) consists of an input layer, an output layer and one or more hidden layers, as seen in 2.4. MLP's are feedforward neural networks, with the output of each node in the network serving as the input for every single node in the next layer. This concept of having each node taking inputs from every single node in the preceding layer is referred to as "fully-connected" layers.
The MLP architecture solves the Xor problem by introducing linear seperability (Singh and Pandey, 2016).
Deep Learning is when there is more than one hidden layer present in a network (O'Shea and Nash, 2015).
Learning takes place in a MLP through changing the connection weights for each perceptron in the network based on the error between the expected and true output of each perceptron. This is carried out through a concept known as backpropagation.

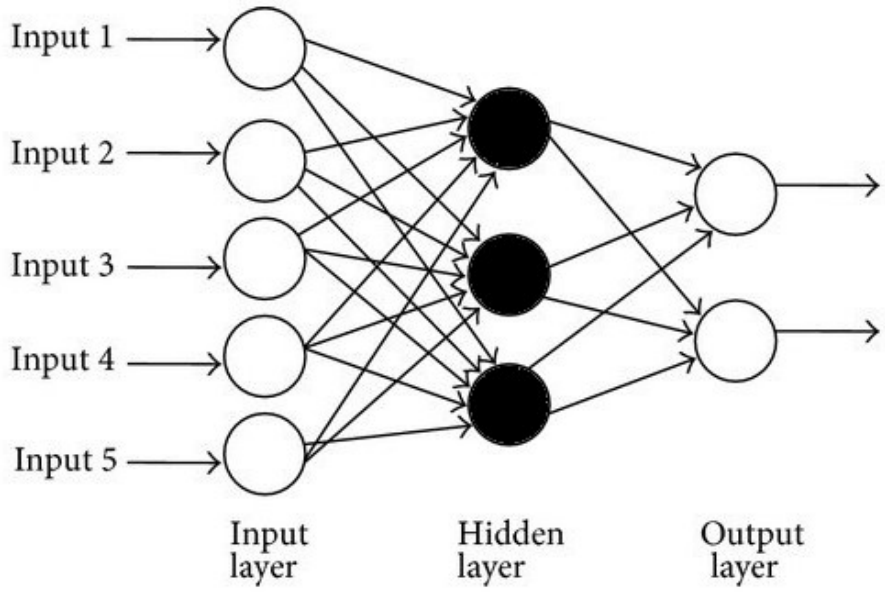


Figure 2.4: Multilayer Perceptron Architecture

2.3.6 Gradient Descent and Backpropagation

Gradient descent is an algorithm used to optimise the weights between neurons in such a manner that creates the least possible amount of error. When the network is training, a cost function is used to keep track of how the network is performing. The cost function looks at the discrepancies between the training output and the true values to determine an error. When the network trains, the goal is to therefore get this cost function as low as possible to ensure the lowest possible error. The way the cost function is minimised in an MLP is through gradient descent and backpropagation.

Every time the weights must be updated, the derivative of the cost function with regards to the weight itself scaled to a learning rate is subtracted. As the network trains the derivative should decrease with each training iteration. This is known as gradient descent. When the cost function cannot be reduced any more, it has converged.

The backpropagation algorithm works in conjunction with this by starting at the output layer of the network and stepping back through all the layers,

updating the weights for each neuron as it goes (Rumelhart, Hinton, and R. J. Williams, 1985). This is in an attempt to minimise the overall error. These steps should allow the network to reduce its error and converge on an optimal solution over time as the network trains.

2.4 Computer Vision

2.4.1 Noise and Occlusion

Within any image dataset that contains images extracted from real-life scenarios, two commonly used terms are noise and occlusion. Image noise is the presence of random variation in pixel colour and intensity within images, typically produced by the sensor taking the image. Within the Berkeley Deep Drive dataset, a common source of noise is motion blur produced by the camera taking a picture while the test car is in motion. This type of image noise can be observed in Fig 2.5. A proliferance of noisy images within a dataset should be avoided if possible, as an overly noisy dataset will cause classifier performance to suffer (Xiao et al., 2015).

Figure 2.5: Image noise produced by motion blur

Occlusions within an image are when a particular object is covered in some way, making it only partially visible. This is problematic for classification as it hides some features of the object, making it more difficult for the classify to determine what class the object belongs to. In the Berkeley Deep Drive dataset this is typically present in long rows of cars occluding each other as seen in Fig 2.6.

Figure 2.6: Image containing occluded cars

2.4.2 Edge Detection

Edge detection is the ability to recognise object boundaries within an image, with most edge detection techniques using changes in pixel intensities to identify potential edges (Arbelaez et al., 2011). Although edge detection may appear straightforward in concept, in practice it can be a difficult task. Input images typically suffer from noise due to focal blur, as well as blur caused by shadows. This can cause smoothing in the variations in pixel intensity at edge points, and make it difficult to define what actually is an edge (Ziou and Tabbone, 1998). The magnitude of a gradient will determine if a point in the image is an edge or not - a high gradient implies that an edge is likely present. The direction of a gradient shows how the edge is oriented within the image.

Approaches

There is a multitude of different techniques used for edge detection that can be split into two categories, search based and zero-crossing based. Search based techniques work by computing gradient edge strength, then searching for the direction of the edge by computing the gradient orientation. The Sobel operator is an example of this type of technique (Gupta and Mazumdar, 2013). Zero-crossing based techniques differ in that they search for zero-crossing points on a second-order derivative function calculated from the image. Generally smoothing is applied to the image prior to any of these technques (Ziou and Tabbone, 1998).

2.4.3 Multilayer Perceptron for Image Tasks

Prior to the popularisation of the CNN, MLP's were the standard for image based problems. However, there are several issues that make the MLP quite unsuitable for this type of problem.

The main issue with MLP's is the manner in which they receive input data. As mentioned above, pixel representations of colour images consist of 3D arrays of pixel values. In order for this data to be passed into an MLP, it must be "flattened" into a 2D vector - as each input node in the MLP accepts a single pixel value (Ben-Yacoub, Fasel, and Luettin, 1999). This disregards the spatial characteristics of the image, as the image has been

transformed into a large vector of pixel values. This is a very inefficient way of processing images, as it stands to reason that the spatial relationships between pixels are important ie. if two pixels are close together it is likely that they are related in some way (LeCun, Boser, et al., 1989). This lack of consideration for spatial relationships means that an MLP must be given training data that consists of labelled classes in a range of different positions and levels of occlusion within the training images in order for it to reliably identify the classes when it is applied to unseen data, rather than being able to identify a class regardless of its position within an image. Although an MLP may perform adequately upon very simple target domains, more complex tasks will require a large amount of training data in order to produce a reliable classifier (Al-Qudah, 2009).

MLP's also encounter other issues when attempting to carry out classification tasks on large input images. As mentioned, each pixel of an input image must correspond to an input node in the network. As every node is connected to every other node in its preceding layer, large images quickly require a large amount of nodes in the network. A small image of size 28x28 pixels such as the images found in the MNIST dataset will require a network with 784 input nodes. Images sourced from the Berkeley Deep Drive dataset are of size 1280x720 pixels, which would require an input layer consisting of 928080 nodes. This means that an MLP that takes in large input images requires a huge number of parameters. Computing all of these parameters naturally leads to extremely long training times.

So, it should now be clear that the manner in which MLP's handle image related tasks could be much more efficient. It is with these issues in mind that CNN's were developed (LeCun, Boser, et al., 1989).

2.5 Introduction to Convolutional Neural Networks

When humans look at an image of say, a dog, we automatically extract the things we have learned that make a dog unique in order to allow us to recognise it. For example we may see a tail, fur and a snout and recognise that we are looking at a dog. At an extremely high level this what a CNN will do - it will look for certain low-level aspects of the object known as

"features" such as curves and edges, building these up into more abstract concepts such as a leg or a tail in order to determine what features the object is comprised of, thus allowing it classify the object (LeCun and Bengio, 1995).

Interestingly enough, CNN's do take some inspiration from how the brain processes images. The visual cortex of the brain contains many fields that are sensitive to different specific elements of the input. Some groups of neurons will only respond when certain elements of the input are present, for example certain vertical edges, and other groups of neurons will respond to different elements, such as horizontal edges (Hubel and Wiesel, 2011). The concept of distinct groups of neurons looking for certain features is one that has translated very well into Computer Vision through the use of CNN's. The most important aspect of CNN's that improve their suitability for image related problems is the manner in which they detect classes regardless of their position within an image (Albawi, Mohammed, and Al-Zawi, 2017) ie. a CNN trained to detect faces does not pay any regard to where instances of faces are located within an image, it simply recognises them. The main point of note when trying to understand what distinguises CNN's from traditional Machine Learning approaches is that they preserve the matrices that the images are represented as, preserving the relationships between pixels rather than simply unpacking the images into long vectors (LeCun and Bengio, 1995). Instead, they observe groups of pixel values within the matrices in order to process the data contained within them. They do this through special types of layers known as Convolution and Pooling layers.

2.5.1 Convolution Layer

There are three important points of note regarding the Convolution layer. These are the input image, the feature detector and the feature map. The input image is the image which the CNN is given. The feature detector is a matrix (usually 3x3 or 7x7), also called the kernel or filter, which is multiplied against matrix values of regions within the input image to create what is known as a feature map, with the aim of capturing only important features of the input image. The way this works is the feature detector slides or "convolves" over the whole image (O'Shea and Nash, 2015). The particular point on the image that the feature detector is analysing is

known as the receptive field.

The parameter known as the stride is the value of the number of pixels by which the feature detector convolves over the input image. For example, a stride of 2 will convolve the feature detector over the input image jumping 2 pixels at a time. As the feature detector convolves around the input image, the pixel values of the feature detector are multiplied against the pixel values of the receptive field. These multiplications are all summed to give a single value, as shown in Figure 2.7. The higher the value, the higher the likelihood that the particular feature represented in the feature detector is present in the receptive field. The feature detector then continues to carry out this process for every section of the input image. The output of this process is a range of new 2D arrays of potentially important features of the input image known as feature maps (Z.-Q. Zhao et al., 2019). As multiple filters are applied at each layers, images are "deepened" ie. an image that is 32x32x32 (a 32x32 image with 3 colour channels) that has 9 filters applied will result in a matrix of dimensions 32x32x9. Each of these 9 layers represents the original image, however they now contain feature information rather than colour values (LeCun and Bengio, 1995).

A benefit of this process is that the layers of a CNN do not need to be fully-connected as in an MLP, rather they need only be connected to neurons close to them in a principle known as local connectivity - the input of a neuron is received from a small local group of pixels (LeCun and Bengio, 1995). Local connectivity is made possible due to the fact that the CNN does not alter the structure of the image as in an MLP, instead related pixels remain close together. Due to the fact that neurons are only connected to a small number of other neurons, the number of parameters required to be computed by the network is reduced, and therefore the overall computation time is reduced.

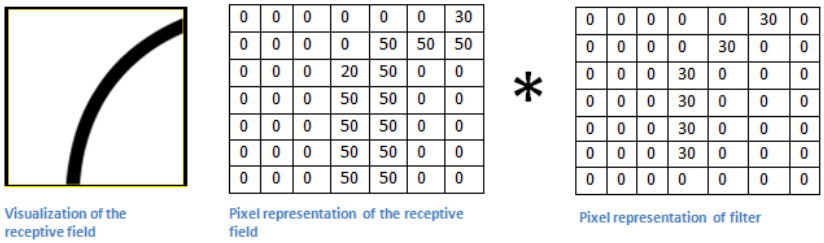


Figure 2.7: Convolution taking place

2.5.2 Pooling Layer

The pooling layer performs downsampling to extract key important features and reducing the number of parameters in the network. The pooling layer operates over every feature map by placing a matrix over the feature map and selecting the min value, max value or mean value, depending on the type of pooling being utilised, as shown in Figure 2.8. These extracted values form the pooled feature map (O'Shea and Nash, 2015). Thus, pooling layers serve to reduce the width and height of an image while preserving only the most important features that have been extracted by the filters in the convolution layer (Z.-Q. Zhao et al., 2019).

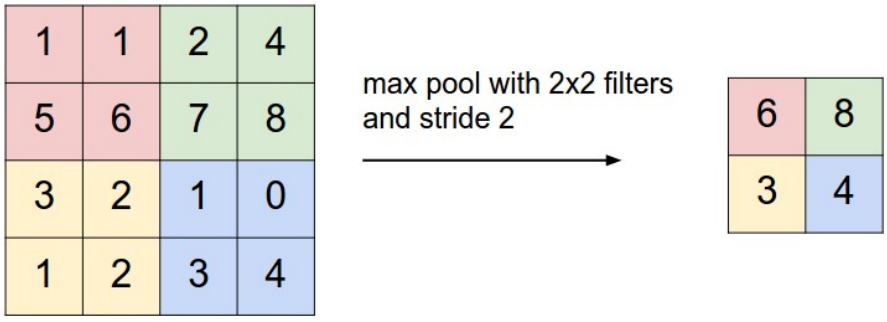


Figure 2.8: Pooling

The architecture of a CNN follows essentially the same architecture as a standard MLP, with the addition of these extra layers of Convolution and

Pooling. When these layers of Convolution and Pooling are stacked with a fully-connected layer, a CNN architecture has been formed. Once all of the Convolution and Pooling steps have been performed, a fully-connected layer will compute the overall class scores, resulting in a final output of 1x1xn, where n is the number of possible classes. The fully-connected layer does this by looking at the output of the Convolution and Pooling layers, which takes the form of activation maps of features, and attempts to determine which features represent a particular class.

CNN's learn in the same manner as any other Neural Network - initially the filter values are randomised, but after each training iteration an error is calculated and backpropagation is used to step back through the network and adjust weights until an optimal solution is reached and the filters represent features that correspond to the correct classes. Fig 2.9 shows a basic view of how these unique layers are combined to form a CNN architecture.

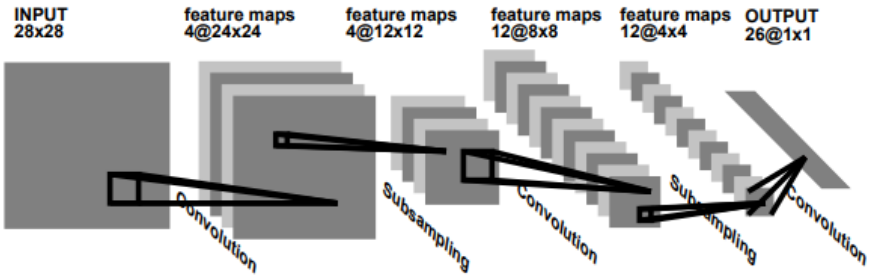


Figure 2.9: CNN Architecture source: LeCun and Bengio, 1995

2.5.3 Relevant Architectures

There are many different ways by which to structure the different layers of a CNN, and this section will serve to outline some important CNN architectures that are commonly used. These architectures will later be implemented and investigated during the Empirical Studies section.

MobileNet

Following the success of AlexNet in 2012, CNN's followed a general trend towards deeper and more computationally expensive architectures. However, increased network complexity does not guarantee increased network performance (Szegedy et al., 2016), and in computationally limited platforms such as smartphones and self-driving cars tasks need to be carried out quickly with a low computational overhead. The MobileNet architecture arose out of this demand for more lightweight and efficient architectures (Howard et al., 2017). MobileNet differs from conventional CNN architectures in that it makes use of depthwise seperable convolutions in the place of regular convolutions. A depthwise seperable convolution consists of two operations - a depthwise convolution that filters through the input image followed by a pointwise (simply 1x1) convolution which combines the outputs of the former. This depthwise seperable convolution serves the same purpose as a traditional convolution, however it is much faster (Howard et al., 2017).

Inception

In a similar vein to the MobileNet architecture, the Inception architecture arose from a desire to create more complex networks to increase performance, without leading to the issues presented by simply stacking more layers into a network. At the current time there are four main versions of the Inception architecture, with each improving slightly over the preceding version. The core concept behind the creation of this network was that important objects or regions within a given image can vary greatly depending on the differing distances between the object instances and the camera. Due to this variance, selection of the correct kernel size is difficult (Szegedy et al., 2016). A larger kernel size is suitable for the larger objects, however this will not suffice for the smaller objects. Rather than simply adding extra layers, a possible solution is to have multiple filters operate on

the same level, creating a "wider" rather than "deeper" architecture (Szegedy et al., 2016). This was the driving concept behind the creation of the first Inception architecture (also known as GoogLeNet).

2.5.4 Detection Algorithms

An integral part of the performance of any CNN is the particular detection algorithm that the CNN utilises. These algorithms are necessary in order to reduce target images into regions that likely contain instances of objects, rather than attempt object detection upon all regions of the images (Girshick et al., 2014). This is required in order to reduce the computational requirements of the CNN. Base networks such as Inception or MobileNet perform high-level detection of objects, while detection algorithms are used to find potential occurences of objects to be detected - thus the two work in tandem.

R-CNN

To avoid the problem of simply selecting a huge number of regions (Girshick et al., 2014) proposed a method whereby 2000 distinct regions of interest (ROI's) were identified in the input image in what are known as "region proposals", with these being identified based on a selective search algorithm. The CNN then acts upon these ROI's in order to detect objects within the region proposals.

Faster R-CNN

Following advancements made upon the R-CNN algorithm, the Faster-RCNN algorithm was proposed by (Ren et al., 2015). This algorithm involves the target image being provided to a CNN which produces a feature map. The predicted ROI's are then reshaped using a particular pooling layer, following which object detection can take place upon the ROI's. This advancement upon earlier R-CNN algorithms produced an algorithm much faster than previous versions of the algorithm.

YOLO

YOLO (You Only Look Once) does not utilise regions in an attempt to localise objects. Instead, this algorithm utilises a network that attempts to

predict classes and their bounding boxes in a single evaluation (Redmon et al., 2016), creating a search algorithm that is extremely fast, albeit one that struggles with objects at smaller scales.

SSD

SSD (Single Shot Detection) operates in a similar manner to YOLO, with ROI's being avoided in favour of a single evaluation. SSD has six feature maps at different sizes, making it more capable than YOLO of detecting objects at differing levels of scale.

2.5.5 Evaluating the Classifier

There are various techniques which can be used to evaluate the operation of a classifier. Some of the more prevalent ones, many of which will be seen throughout the rest of this report, are explained below.

Top-1 and Top-5 Accuracy

First, the CNN makes a classification. Then in the case of Top-1, the class of the highest probability is checked against the target label. In the case of Top-5, the target label is compared with the top five highest probability predictions. Following this, in both cases the classifier score is calculated as the number of times that the prediction matched in either Top-1 or Top-5, divided by the total number of data points.

Accuracy Issues

Generally speaking using accuracy alone (number of correctly labelled classes) is not a very reliable metric for an object detection algorithm. For example, if there were 95 dogs and 5 cats to be classified a classifier may label all objects present as dogs. This would lead to an overall accuracy score of 95%, however this classifier is obviously quite flawed. This is known as class imbalance, and in real life scenarios tends to be the norm. The decisions reached by the classifier can be explained as follows:

1. True positive: a positive example classified as positive
2. True negative: a negative example classified as negative

3. False positive: a negative example classified as positive

4. False negative: a positive example classified as negative

False negatives are particularly an issue for the field of self-driving cars, where this result could have potentially fatal ramifications. An example of a false negative classification was seen during a fatal crash of a Tesla model S in May 2016, whereby the autopilot system failed to recognise a white truck against a clear, brightly lit sky. Thus, we need to do more than just measure accuracy in order to properly evaluate the classifier (Rogers and Girolami, 2016).

Confusion Matrix

A Confusion Matrix, also referred to as an Error Matrix, is a relatively simple technique for classifier evaluation. It is essentially a table that plots the number of correct and incorrect predictions for all of the different classes. The false positives, false negatives, true positives and true negatives are plotted then the average value for all classes combined is calculated, as demonstrated in the confusion matrix below.

		Actual	
		+	-
Predicted	Y	True Positives	False Positives
	N	False Negatives	True Negatives

Table 2.1: Confusion matrix visualised

Other Evaluation Metrics

There are a number of different metrics of evaluation that can be used other than accuracy alone, with the major ones being summarised in the section. An important point to note is that each of these metrics is derived from the confusion matrix explained above, and can be thought of as different ways of summarising the matrix.

Precision
The precision is the number of correct positive predictions compared to the actual number of positive examples. It can be represented by the formula $precision = \frac{\#TP}{\#TP+\#FP}$. This is a good metric to evaluate when there is a high cost associated with a false positive. For example, an email spam detector must have high precision in order to avoid mistakenly classifying legitimate emails as spam.

Recall
The recall is the ratio of correct positive examples to the number of positive predictions, and can be represented as $recall = \frac{\#TP}{\#TP+\#FN}$. Recall is an important metric when there is a high cost associated with false negatives and as mentioned above are an extremely important metric for this project.

F-score
Another metric can be derived from precision and recall, known as F-score. F-score is the harmonic mean of precision and recall, and is defined as $F = 2.\frac{precision-recall}{precision+recall}$. The F-score can be thought of as representing the balance between the precision and the recall. Although this is an important metric overall, it does not take precedence in this project, where a good recall score is of utmost importance.

AUC-ROC Curve

The AUC (Area Under Curve) ROC (Receiver Operating Characteristics) curve is another method of classifier evaluation, enabling the visualisation of results from a classification problem. The ROC curve is used to plot the true positive rate against the false positive rate, resulting in a probability curve, while the AUC represents how well the model distinguishes between classes. The higher the AUC, the better the model is. This is visualised in Fig 2.10.

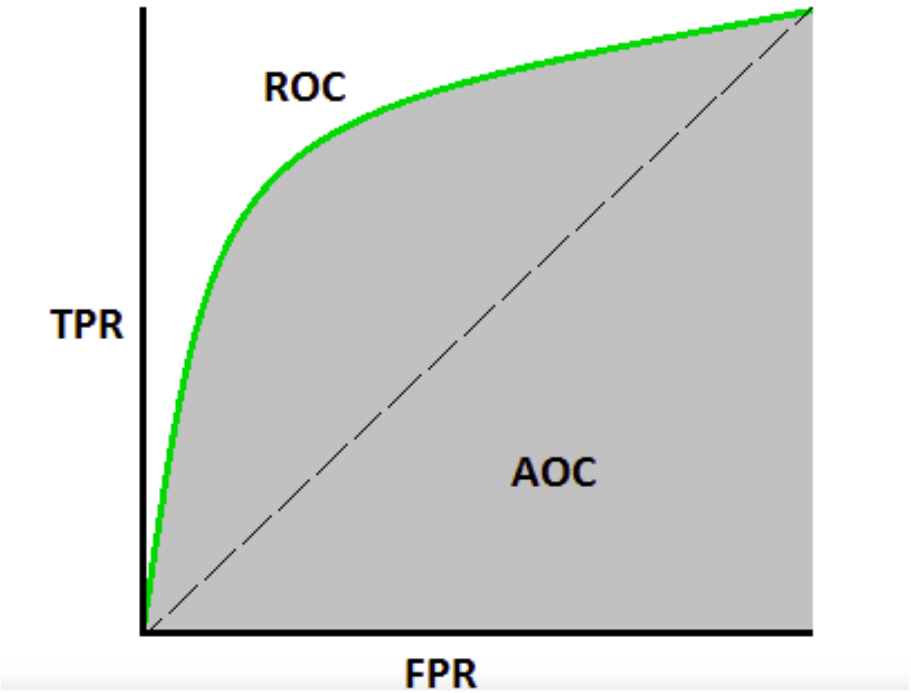


Figure 2.10: AUC-ROC Visualised

Loss

As mentioned above, Neural Networks utilise a loss function in order to evaluate the predictions the network makes in relation the known values in the training data. The loss of a network can be graphed to infer how well the model is adjusting to learn the parameters of the training data, with the loss generally lowering over each training iteration.

2.5.6 Transfer Learning

Transfer learning is a popular technique in Machine Learning whereby an already trained model is retrained on some limited set of new data whilst keeping most of the learned parameters from the original training data. As mentioned earlier, CNN's use their lower layers to extract primitive features such as edges, the middle layers detect shapes from these edges and the final layers learn the task specific abstractions of these shapes. Thus, it is only the final layers that are specific to the classes of the training dataset, with the preceding layers being more general (Yosinski et al., 2014). In transfer learning a base network is trained on a base set of data, after which the learned features of this network are repurposed to a second network which is then trained on the target dataset. In essence, the initial layers of a neural network can be viewed of as feature extractors, and these feature extractors tend to be quite versatile across different datasets (Tan et al., 2018). In order for transfer learning to be effective, the original and target datasets must have some degree of similarity in order for the originally learned features to be effective when applied to the target dataset. Transfer learning is a very popular field within Machine Learning, as it can help drastically cut down on training times, as well as enable better results on smaller datasets (Tan et al., 2018). The bulk of the experiments carried out in the Empirical Studies section of this report leverage transfer learning in order to cut down on training times.

2.6 Berkeley Deep Drive

Dataset

As mentioned previously in this report, the dataset utilised for this project is the Berkeley Deep Drive dataset, a "large-scale diverse driving video

dataset", also known as the BDD100K dataset. Released in 2018, the dataset consists of 100,000 1280x720 annotated images and 100,000 720p 30fps video sequences (Yu et al., 2018). The video was collected from a number of locations throughout the United States and consists of a range of weather conditions including sunny, overcast, and rainy. The data consists of a mixture of both daytime and nighttime driving conditions. The images section of the dataset has been created by selecting and annotating the frame at the 10th second of every video. Annotated objects range across 10 different classes - bus, light, sign, person, bike, truck, motor, car, train and rider. The number of instances of each of these annotations are displayed in Fig 2.11.

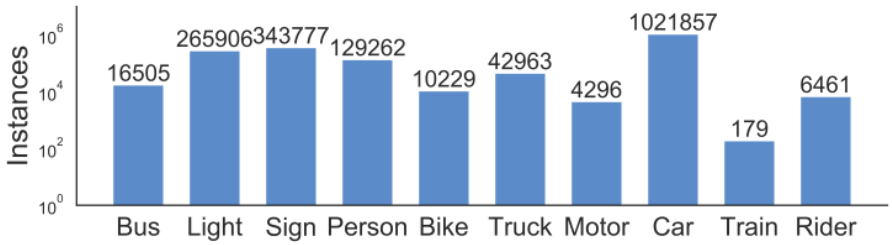


Figure 2.11: source: https://bair.berkeley.edu/blog/2018/05/30/bdd/

Related Results

The paper released with the BDD100K dataset contains a list of results from experiments carried out on the dataset. The experiments focused on the performance of a CNN on different domains within the dataset. For example, a Faster-RCNN was trained on daytime images and then its performance on nighttime images was tested and the discrepancies recorded. The performance metrics recorded from these experiments allow some basis upon which the CNN's trained in this project can be evaluated against. Across the three main classes that will be focused upon in later sections of this report - car, traffic light and person - Yu et al. achieved a mean average precision (mAP) of 0.53. Unfortunately, this is the only metric by which their model was evaluated, however this still provides a rough estimation of how classifiers implemented in later sections should aim to perform.

2.7 Overfitting

An important point to consider when training any Machine Learning model is how well the model generalises to new data (Domingos, 2012). An algorithm should be able to apply concepts learned from the training data to any previously unseen data in the problem domain in order to make accurate predictions. Overfitting is the concept of a model learning the details and noise in the training domain too well, and thus failing to provide accurate predictions on new data. Overfitting tends to occur when a model learns noise and randomness in the training data as concepts that it attempts to apply to data in the problem domain. These concepts will not apply to the new data however, and poor accuracy generally results. Although a noisy dataset will increase the severity of overfitting, it is not simply a result of a noisy dataset and can occur in any dataset (Domingos, 2012). Within the field of Neural Networks, a common approach to avoid severe overfitting is to keep the network architecture relatively simple. The less parameters required to train the model, the lower the overall chance that the network will overfit and fail to generalise (O'Shea and Nash, 2015). However, this approach is not always feasible when dealing with large and complex datasets, and other approaches must also be taken. Another popular method to reduce overfitting is the introduction of dropout in the network. Dropout is when nodes are randomly dropped from the network along with their connections (Srivastava et al., 2014). This serves to constrain the adaptation of the network while it trains in order to prevent the model from over-learning the training data.

2.8 Tensorflow Introduction and Environment Setup

This section is dedicated to the background work done in order to gain an understanding of how to implement CNN's using tensorflow. The experiments illustrated in this section were carried out during the completion of the Complete Guide to TensorFlow for Deep Learning with Python course on Udemy (*Complete Guide to Tensorflow for Deep Learning with Python* 2018). This section consists of two experiments carried out on the MNIST and CIFAR-10 datasets. The first experiment is sample code provided as part of the course to illustrate the concepts learned during the course. The second experiment is an exercise for the course carried out to implement an understanding of these concepts.

2.8.1 MNIST Experiment

The first experiment carried out was the implementation of a very simple CNN. The architecture of the CNN consisted of two layers of Convolution, two pooling layers and a fully-connected layer. After carrying out 5000 training steps with a batch size of 50, a final Top-1 accuracy of 98.75% was achieved.

Code Breakdown

The functions displayed in Figure 2.12 are all helper functions. The init_weights function initialises the random weights for fully connected or convolution layers, with the shape of the layer passed in as a parameter. The init_bias function performs the same operation for the bias. The conv2d function creates a convolution using a built in function from tensorflow. The max_pool_2by2 function creates a max pooling layer, also using built in tensorflow functions. The convolutional_layer function uses the conv2d function to return an actual convolutional layer with a ReLu activation function. Lastly, the normal_full_layer returns a normal fully connected layer.

The functions displayed in Figure 2.13 are used to create the convolution and pooling layers of the network. In creating convo_1, we can see that a

```
def init_weights(shape):
    init_random_dist = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(init_random_dist)

def init_bias(shape):
    init_bias_vals = tf.constant(0.1, shape=shape)
    return tf.Variable(init_bias_vals)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2by2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')

def convolutional_layer(input_x, shape):
    W = init_weights(shape)
    b = init_bias([shape[3]])
    return tf.nn.relu(conv2d(input_x, W) + b)

def normal_full_layer(input_layer, size):
    input_size = int(input_layer.get_shape()[1])
    W = init_weights([input_size, size])
    b = init_bias([size])
    return tf.matmul(input_layer, W) + b
```

Figure 2.12: Creating Helper Functions

6x6 filter is used from the parameters. An ouput value of 32 is used to represent the number of filters used. The parameter of 1 represents the original input of the image. This carries down to the other layers, with convo_2 taking in an input image of 32 from convo_1.

The code displayed in Figure 2.14 shows the model being trained. A tensorflow session is created and data is read in from the dataset. The session is ran to begin model training. The current training step and current error calculated from a loss function is displayed as output as the model trains.

```
convo_1 = convolutional_layer(x_image,shape=[6,6,1,32])
convo_1_pooling = max_pool_2by2(convo_1)
```

```
convo_2 = convolutional_layer(convo_1_pooling,shape=[6,6,32,64])
convo_2_pooling = max_pool_2by2(convo_2)
```

```
convo_2_flat = tf.reshape(convo_2_pooling,[-1,7*7*64])
full_layer_one = tf.nn.relu(normal_full_layer(convo_2_flat,1024))
```

```
# NOTE THE PLACEHOLDER HERE!
hold_prob = tf.placeholder(tf.float32)
full_one_dropout = tf.nn.dropout(full_layer_one,keep_prob=hold_prob)
```

```
y_pred = normal_full_layer(full_one_dropout,10)
```

Figure 2.13: Creating Layers

```
steps = 5000

with tf.Session() as sess:

    sess.run(init)

    for i in range(steps):

        batch_x , batch_y = mnist.train.next_batch(50)

        sess.run(train,feed_dict={x:batch_x,y_true:batch_y,hold_prob:0.5})

        # PRINT OUT A MESSAGE EVERY 100 STEPS
        if i%100 == 0:

            print('Currently on step {}'.format(i))
            print('Accuracy is:')
            # Test the Train Model
            matches = tf.equal(tf.argmax(y_pred,1),tf.argmax(y_true,1))

            acc = tf.reduce_mean(tf.cast(matches,tf.float32))

            print(sess.run(acc,feed_dict={x:mnist.test.images,y_true:mnist.test.labels,hold_prob:1.0}))
            print('\n')
```

Figure 2.14: Training the Model

2.8.2 CIFAR-10 Experiment

The second experiment was carried out on the CIFAR-10 dataset, and the architecture used was consistent with the first experiment, consisting of two layers of Convolution and Pooling. After 5000 training steps with a batch size of 100, a final Top-1 accuracy of 72% was reached.

2.8.3 Initial Findings

The first two experiments were carried out using CNN's of the same architecture, however an accuracy discrepancy of 26.75% was observed. Both datasets contain images labelled into 10 classes. The differences arise in the dataset size and image complexity. Firstly, the CIFAR-10 dataset contains 50,000 training images, whereas the MNSIT dataset contains 60,000 training images. The images in the MNSIT dataset are grayscale, whereas images in the CIFAR-10 dataset consist of 3 colour channels. The images in the CIFAR-10 dataset are also of a higher complexity compared to the handwritten digits in the MNIST dataset. The CIFAR-10 images contain more complex real-world images with noisy backgrounds. These disprepancies between the datasets may explain the disprepancies in the accuracy achieved by the classifiers.

2.8.4 Environment Setup and Issues

Initial environment setup to carry out the above experiments proved quite challenging. During the first attempts at carrying out the experiments, tensorflow was erroneously installed to run using the training machines CPU. This lead to untenable training times for the experiments. Although the training times were not recorded precisely, each experiment took over a day to finish training, during which time the training machine was rendered essentially unusable due to the high load on its CPU. There were two main take-aways from the initial iterations:

1. Tensorflow should be installed properly to use the GPU in order to manage training times.

2. Training times should be recorded for better understanding of the experiments.

Tensorflow Install

The initial attempts to setup tensorflow were carried out on a machine running Windows 7. Several versioning issues were encountered due to Bazel, the build tool used to compile tensorflow. These issues were solved via downgrading to an earlier version of Bazel, however new versioning issues were then encountered due to CUDNN, Nvidias GPU-accelerated

library for deep learning. Several different versions were installed without success. Due to these issues seemingly being exclusive to Windows, it was decided to attempt an install on a different machine running Ubuntu. The tensorflow install for the Ubuntu machine was carried out without issue. All neccessary Nvidia drivers were installed on the system, and tensorflow was then installed and the experiments were rerun. The training times for running the two experiments with GPU acceleration were 12 and 16 minutes respectively, a vast improvement over the initial times. The GPU on the Ubuntu machine was an Nvidia Geforce GTX 650 with 2GB memory.

Memory Issues

Although the first two experiments ran without issues, problems were quickly encountered when modifications were made to the Cifar-10 experiment. In an attempt to improve accuracy results, two further layers of convolution and pooling were added. This led the out of memory (OOM) errors being thrown during training. This was caused by the graphical memory on the training machine being maxed out at the full 2GB. Several attempts at rectifying the issues were carried out. The first attempt tried was reduction of batch size from 100 images, causing less of the dataset being loaded into memory. At a batch size of 4 images, the three layer network successfully trained. The issue with this approach however is that a smaller batch size causes the gradient estimate of the network to be less accurate, and Top-1 accuracy of the network dropped to 60.46%. The second approach was to simply keep the number of layers low, resulting in a simpler network with less memory requirements. Both of these solutions were not viable for this project. Poor accuracy scores and an inability to train complex networks created an obvious demand for a more powerful training machine. Another point to note is image size disparities between the Cifar-10 dataset and the overall target dataset for this project, the Berkeley Deep Drive dataset. A machine struggling with the 32x32 low resolution images of the Cifar-10 dataset would surely run into a host of issues with the 1280x720 images of the target dataset. As such, it was decided that a much more advanced machine would be required. It was decided that utilising Amazon Web Services (AWS) products was the only way to quickly and easily access the required computational power demanded by this project.

AWS Setup

One of the many services offered by the AWS platform are Amazon Machine Images (AMI) for Deep Learning. These are virtual machine instances that launch with pre-installed pip deep learning frameworks such as tensorflow, keras etc. in different environments, as well as GPU acceleration. This allows development of models remotely from a client machine which can be then trained using the available GPU resources on the deep learning instance. These instances can be deployed and the desired environment (tensorflow in this case) activated. Jupyter notebooks can be run as normal on the AWS instance, and a rule is set up on the client machine to forward all requests on a certain port to the AWS instance. This allows the user to write code in a Jupyter notebook remotely on the AWS instance. The instance type chosen for this project was the p2.xlarge instance which contains 122GiB of memory, 16 virtual CPU cores and a Nvidia K80 GPU. This instance is the cheapest GPU accelerated instance type available, as this project has been funded out of pocket on a tight budget. Costs for these machines are based on usage, so every hour that the machine is running costs are incurred. Much more computationally powerful instances are available, however the costs for these increase dramatically and were not within budget for this project. The steps followed for setting up the instance were from the official AWS documentation (*Launching and Configuring a DLAMI* 2018).

Chapter 3

Cifar-10 Further Experimentation

Once an AWS Deep Learning machine had been set up to enable investigation into more complex CNN architectures, further experiments were carried out on the Cifar-10 dataset in an attempt to improve the disappointing initial results from the experiments carried out as part of the (*Complete Guide to Tensorflow for Deep Learning with Python* 2018) tutorial. The base code provided as part of the tutorial was modified in a range of different ways in order to maximise results. In order to evaluate the network properly the following metrics were recorded:

1. Peak Top-1 accuracy
2. Precision
3. Recall
4. Training time

The baseline of these metrics achieved by the simple network provided from the tutorial are shown in the table below, with the accuracy, precision and recall being graphed in Fig 3.1. Interestingly enough, the accuracy and recall values for this experiment were identical throughout all of the training steps. This does not necessarily mean that there is an issue, however it is very unusual. A focus for the following experiments was to investigate what changes, if any, would cause these values to diverge.

Peak Top-1 accuracy	Training Time	Precision	Recall
70.16%	220s	71.67%	70.16%

Table 3.1: Results from Initial Experiment

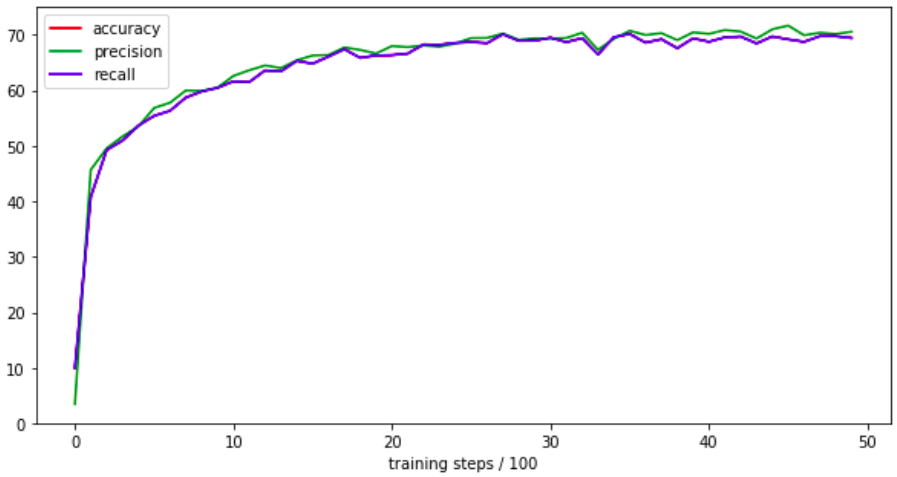


Figure 3.1: Baseline Plot

3.1 Experiment 1: Training Steps

Before investigating the effects of altering the network architecture, the optimal number of training steps was investigated. This parameter was investigated first as it has a large effect on training time. A network that does not have enough training steps will underlearn, whereby it does not learn all possible parameters. However having too many training steps will lead to increased training time with no increase in accuracy. The initial experiment carried out 5000 training steps, so this was doubled to 10,000 steps in an attempt to see if this would boost accuracy, precision and recall. The results are displayed in the table below and plotted in Fig 3.2. All three metrics levelled off at around the 5000 step mark, with no significant increases reported. It would therefore appear that 5000 training steps is the optimal value.

Peak Top-1 accuracy	Training Time	Precision	Recall
70.46%	466s	70.71%	70.46%

Table 3.2: Results from Experiment 1

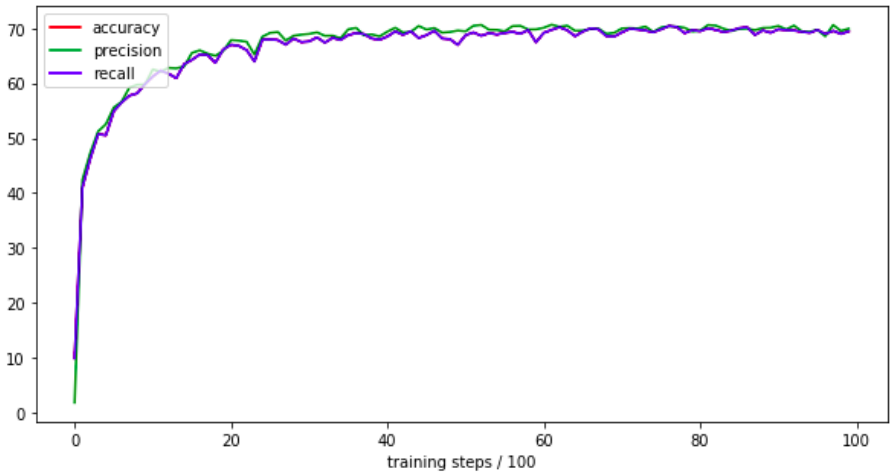


Figure 3.2: 10,000 Training Steps

3.2 Experiment 2: Convolution and Pooling Layers

The next experiment undertaken was to increase the number of convolution and pooling layers. For this experiment only one extra layer of convolution and pooling was added. As can be shown in the results displayed in the table below as well in Fig 3.3, this did not yield any significant results.

Peak Top-1 accuracy	Training Time	Precision	Recall
70.38%	192s	70.82%	70.38%

Table 3.3: Results from Experiment 2

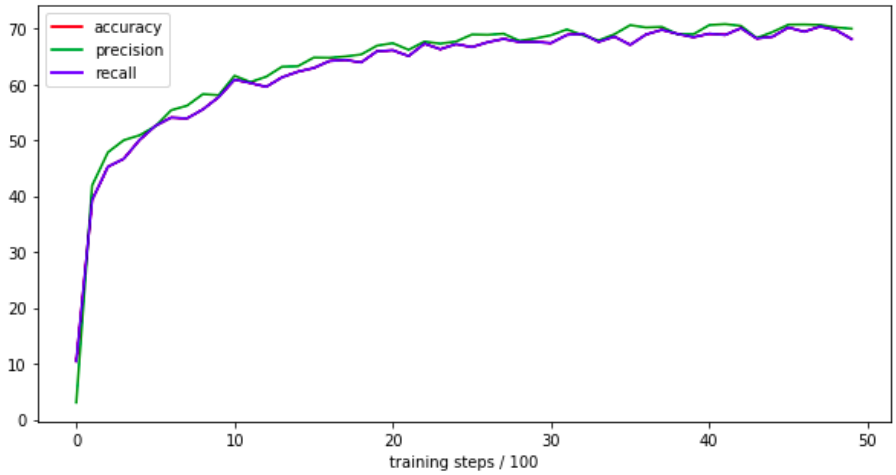


Figure 3.3: Experiment 2

3.3 Experiment 3: Fully Connected Layers

As the extra layers of convolution and pooling had failed to yield any benefits, the next step taken was to increase the number of fully connected layers in the network. One extra fully connected layer was added as well as a dropout layer. The reasoning behind this experiment was to investigate the possibility that the network was performing poorly with the class predictions. The images in the Cifar-10 dataset consist of non-occluded blurred objects with one class of object present per image. As such it was reasoned that only a small number of convolution and pooling layers were required to extract feature maps from the images, and increased numbers of fully-connected layers would be more important. Dissapointingly, the results of this experiment were a marginal drop in accuracy, precision and recall.

Peak Top-1 accuracy	Training Time	Precision	Recall
69.52%	161s	69.67%	69.52%

Table 3.4: Results from Experiment 3

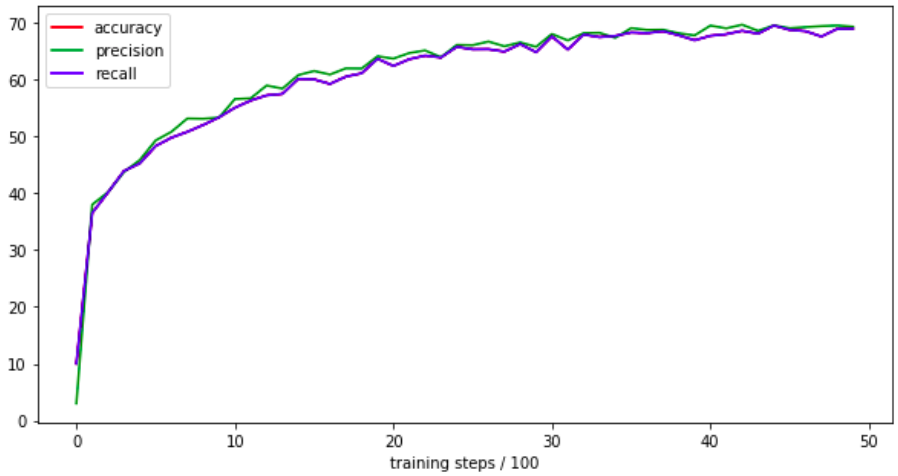


Figure 3.4: Experiment 3

3.4 Experiment 4: Convolution, Pooling and Fully Connected Layers

As Experiments 2 and 3 had failed to yield better results, 2 extra layers of convolution and pooling were added in conjunction with an extra fully-connected layer. As can be seen from the results table and Fig 3.5, results actually went down marginally.

Peak Top-1 accuracy	Training Time	Precision	Recall
67.3%	276s	68.56%	67.3%

Table 3.5: Results from Experiment 4

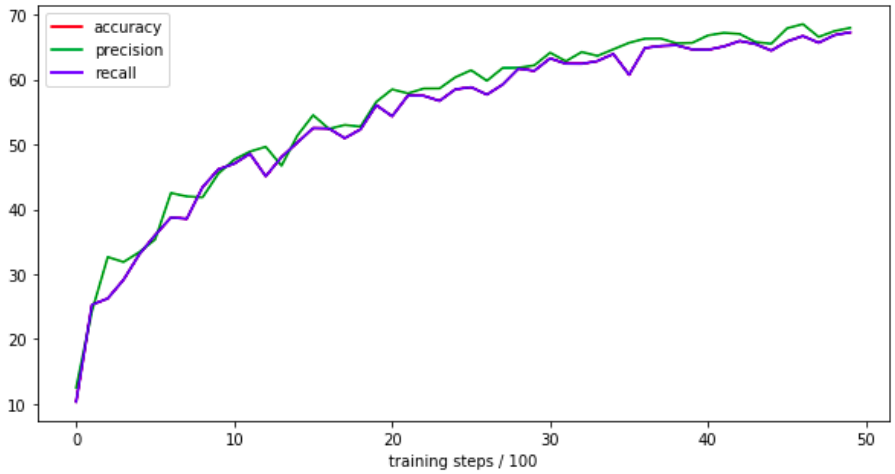


Figure 3.5: Experiment 4

3.5 Experiment 5: Extra Dropout Layers

Following the failure of experiment 4 to improve results, 2 extra layers of dropout with a probability of 0.25 were added to the experiment 4 architecture after the 2nd and 4th pooling layers in an attempt to tale any potential overfitting. Although the results summarised in the table below and Fig 3.6 do not show any meaningful increase in accuracy either, the accuracy curve from this experiment as well as experiment 4 suggest that the network is underlearning as the curves do not level off as much as in previous results. This implied that the network may simply need more time to train. Interestingly, during this experiment the accuracy and recall metrics diverged from one another, as can be observed in Fig 3.6. This would seem to imply that the network may have been overfitting slightly.

Peak Top-1 accuracy	Training Time	Precision	Recall
60.47%	425s	60.60%	60.57%

Table 3.6: Results from Experiment 5

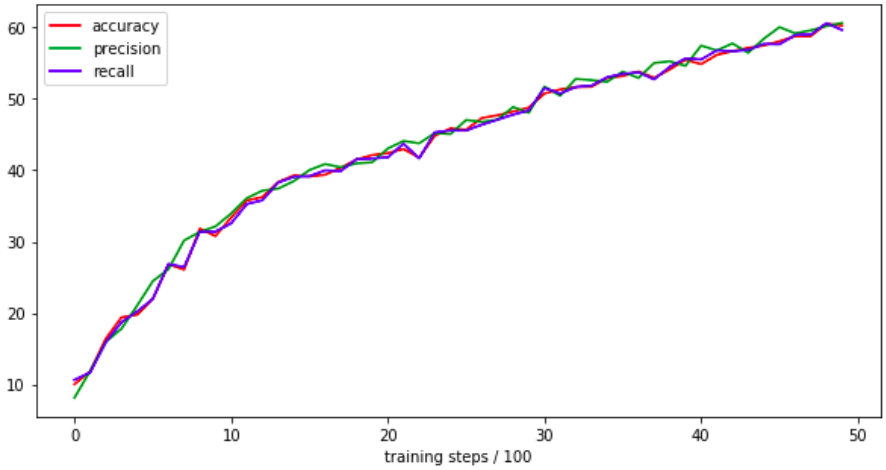


Figure 3.6: Experiment 5

3.6 Experiment 6: Increased Training Steps

The observations in experiments 4 and 5 suggested that the network might simply need more time to train. In experiment 1 it was concluded that 5000 training steps would be sufficient, however this experiment was carried out using the simple baseline network architecture. It was therefore a mistake to assume that 5000 training steps would be sufficient for all following experiments. The number of training steps was increased drastically to 25,000, with the architecture remaining the same as in experiment 5. Although training time was significantly increased, all metrics gained a slight increase with accuracy peaking at 73.25%, a decent improvement from the initial results.

Peak Top-1 accuracy	Training Time	Precision	Recall
73.54%	1778s	74.03%	73.48%

Table 3.7: Results from Experiment 6

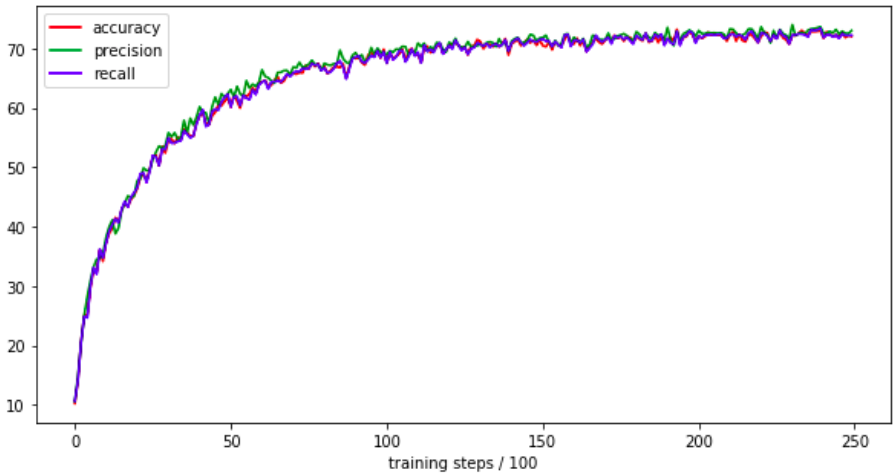


Figure 3.7: Experiment 6

3.7 Conclusions

These experiments conclude with a small yet significant increase in accuracy, precision and recall scores from the final experiment, with the results being collated in Table 3.8. These experiments were focused mainly upon the effects of network architecture rather than the effects of parameter tuning. Empirical studies in later sections will focus more on the effects of parameter tuning. Adding just extra layers of convolution and pooling without adding extra fully-connected layers did not result in improvements, and vice versa. Upon creation of a more complex network architecture with dropout layers results actually went down slightly. However, observation of the metric graphs showed that the network was possibly underlearning, and a drastic increase in training steps led to positive results. Accuracy, precision and recall all seemed to increase in proportion with one another for these experiments, however this is generally not guaranteed for all networks. Datasets containing more classes or high levels of class imbalance would typically not result in such regular accuracy, precision and recall metrics. If there is anything to take from these experiments moving forward with this project, it is that simply adding one or two extra layers is not sufficient to gain satisfactory results - adding multiple layers of convolution, pooling and fully-connected layers should be investigated in conjunction with each other in future experiments. As the unusual accuracy and recall results also demonstrate, it may also be beneficial to introduce dropout or some other form of regularisation in networks that are not obviously overfitting. Although the increases in performance shown are slight, I am confident that further increasing the complexity of the network and increasing training times can lead to further increases.

Experiment no.	Peak Top-1 Accuracy	Training Time	Precision	Recall
1	70.46%	466s	70.71%	70.46%
2	70.38%	192s	70.82%	70.38%
3	69.52%	161s	69.67%	69.52%
4	67.3%	276s	68.56%	67.3%
5	60.47%	425s	60.6%	60.57%
6	73.54%	1778s	74.03%	73.48%

Table 3.8: Collated results from CIFAR-10 experiments

Chapter 4

Empirical Studies

The following sections will primarily investigate the effects of retraining various different network architectures using transfer learning. Models were retrained using the Tensorflow Object Detection API, with the steps followed being from a tutorial made available on GitHub (*TensorFlow Object Detection Model Training* 2018). The main point of note for the Tensorflow Object Detection API is the main config file for each of the experiments. This file defines what type of architecture is used, what pretrained model (if any) is loaded in for retraining and what dataset the model is to be retrained upon. All of the pretrained models utilised in the following sections have been trained on the COCO (Common Objects in Context) dataset, a dataset provided by Microsoft. Although the COCO dataset is updated every year, it currently consists of 164,000 images populated with 90 classes of objects. The dataset was chosen for this project as it contains 8 different classes of vehicle and displays objects in real-life contexts, with potential for occlusion and general variance in the images (Lin et al., 2014). Many other datasets instead provide an "iconic" view of objects, whereby the object appears centrally and unobstructed in the image. As the BDD100K dataset images are drawn from real-world scenarios from the perspective of a car, using the COCO dataset as the initial training dataset should allow the retrained models to generalise better to the BDD100K data than datasets with less variance in the images. As mentioned above in the section related to transfer learning above, it is important that the original training dataset is as close as possible to the target dataset (Tan et al., 2018).

4.1 SSD MobileNet V1 Experiments

The SSD MobileNet V1 architecture was selected for the first experiments, with the pretrained model available from tensorflow.

4.1.1 Experiment 1: Full Dataset Retraining

Objectives

Initial attempts at retraining the SSD MobileNet V1 were carried out using the full 10k subset of the bdd100k dataset, containing 10,000 images annotated with 10 classes. The objective for this experiment was to retrain the MobileNet model with the new BDD100K data. With training times being a concern when dealing with large datasets, the MobileNet architecture was selected to be retrained first due to its quick training times. This main objective for this first experiment was to investigate how models would handle the large amount of data presented by the BDD100K dataset and to gain some understanding of how the object detection API works.

Setup

The tensorflow object detection API only accepts datasets in the tfrecord format, a binary file format for data storage. This requires parsing the images and their labels and writing them into tfrecord format. Luckily, an open source parser for the BDD100K dataset has been made available on GitHub (*Convert the Berkeley Deepdrive dataset to a TFRecord file* 2018). This tool was used to parse all images within the 10k section of the BDD100K dataset, resulting in a tfrecord file containing the data for the full 10,000 images and their annotations.

Results

After 5000 training steps, the results for this experiment were disappointing albeit not unexpected. As can be seen in Fig 4.2, loss values were extremely high, and did not follow any steady downward trend. Total training time to 10,000 steps took 2 hours and 48 minutes. These poor results were expected due to the very large and highly complex dataset. Fig 4.1 provides an example of this - the nighttime conditions combined with slight motion blur creates a highly variable image on which to make predictions

with the classes present within the image being very hard to make out clearly. Notwithstanding dataset variance created by time of day or weather conditions, many of the images in the dataset are complex simply due to the scenes depicted within them. For example, in city scenes rows of cars cause occlusion and people are hard to pick out between the cars. With the dataset consisting in large part of images with similar levels of variance, training times for the full dataset would likely be untenable if a CNN was to be trained to a point where it performed close to the results presented in (Yu et al., 2018). A more powerful GPU accelerated machine may have potentially cut down on these training times drastically, however these machines were simply out of budget for this project. As such, investigation into a different approach was required.

Figure 4.1: Example of an image that is very hard to annotate

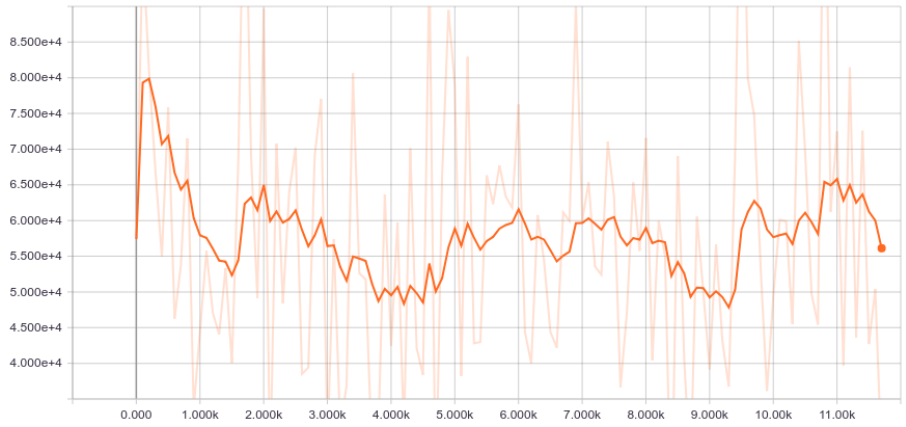


Figure 4.2: Loss values for full dataset

4.1.2 Experiment 2: Partial Dataset Retraining

Objectives

Following the failure to produce adequate results on the full dataset, it was determined that an approach of drastic data reduction was required in order to investigate training on a problem domain of reduced complexity. The objective for this experiment was therefore to retrain the MobileNet V1 architecture again, this time with a much simpler dataset and with less classes required for detection. In order to achieve this, a subgroup of images would need to be extracted from the dataset and labelled manually.

Setup

It was determined that 200 total images would be selected to be manually labelled from the bdd100k dataset, with 160 being utilised for training and 40 for testing. Classes labelled for detection were reduced in number to just 3 - cars, traffic lights and people. These classes were chosen as they are commonly present in most images in the dataset and are all unrelated to each other. The images selected were all images taken with good lighting, no motion blur and no inclement weather present. The scenes depicted by the selected images were semi-urban - all three classes were present albeit in lesser numbers for traffic lights and people. Fully urban scenes were avoided due to the potential for extreme levels of class occlusion within the images.

The images were annotated using the open source LabelImg tool, before being converted to tfrecord format. Training was then carried out exactly the same as the above experiment, this time with the reduced dataset.

Results

Although far from perfect, results from this experiment were much more encouraging. As evidenced in Fig 4.3, loss quickly declined before settling to values just below 2, with the minimum value being 1.788 and a total training time of just 1hr 25min. Mean average precision (4.4) and average recall (4.5), although slighly erratic, appeared to settle into an upward trend as training progressed. Maximum precision reached was 0.1995 and maximum recall reached was 0.2812. The model can be seen in action at 500 training steps in Fig 4.6 and then at 5000 training steps in Fig 4.7. The improvement in the model is clearly apparent in these two images. Please note however that these images appear blurred due to the fact that this MobileNet architecture was initially trained on images of size 300x300 pixels in order to reduce training times. This means that the BDD100K images that it has been retrained on have been scaled down to size 300x300. Scaling the images back up to full size results in the distortion present in the example images. These results are far from perfect however, and will need to be improved upon. Although the model at 5000 steps detects far more than it did at 500 steps, there are still several of the more occluded cars present in the image not being detected. As mentioned, mean average precision and recall did seem to be improving, however neither value increased to above 0.3, which is comparatively poor results. As such, further experimentation is required.

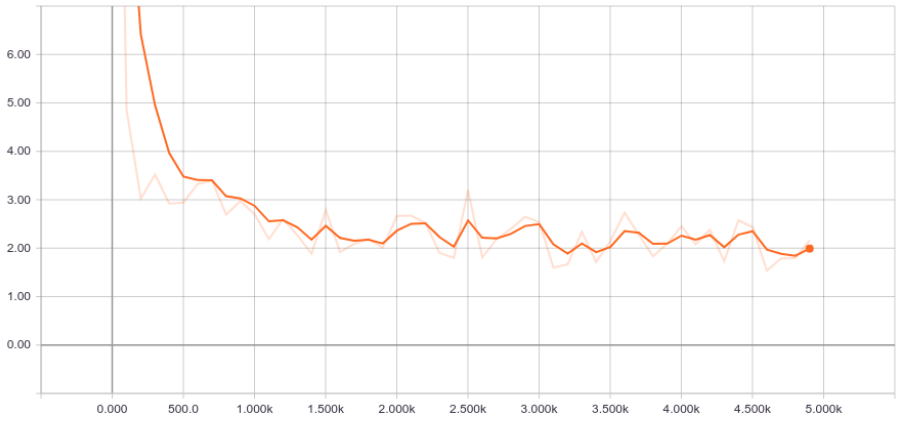


Figure 4.3: Loss values for reduced dataset

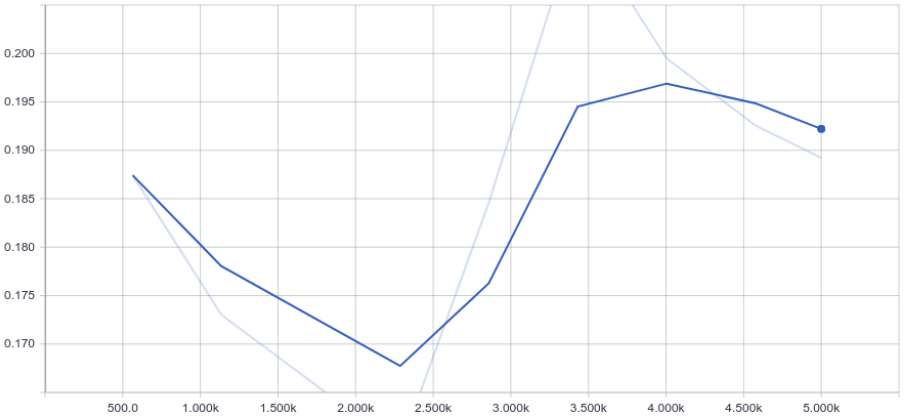


Figure 4.4: Mean average precision (mAP) values for reduced dataset

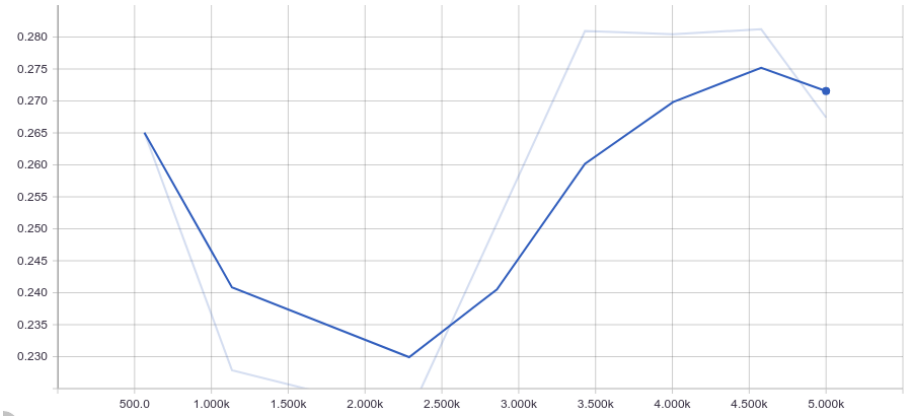


Figure 4.5: Average recall (ar) values for reduced dataset

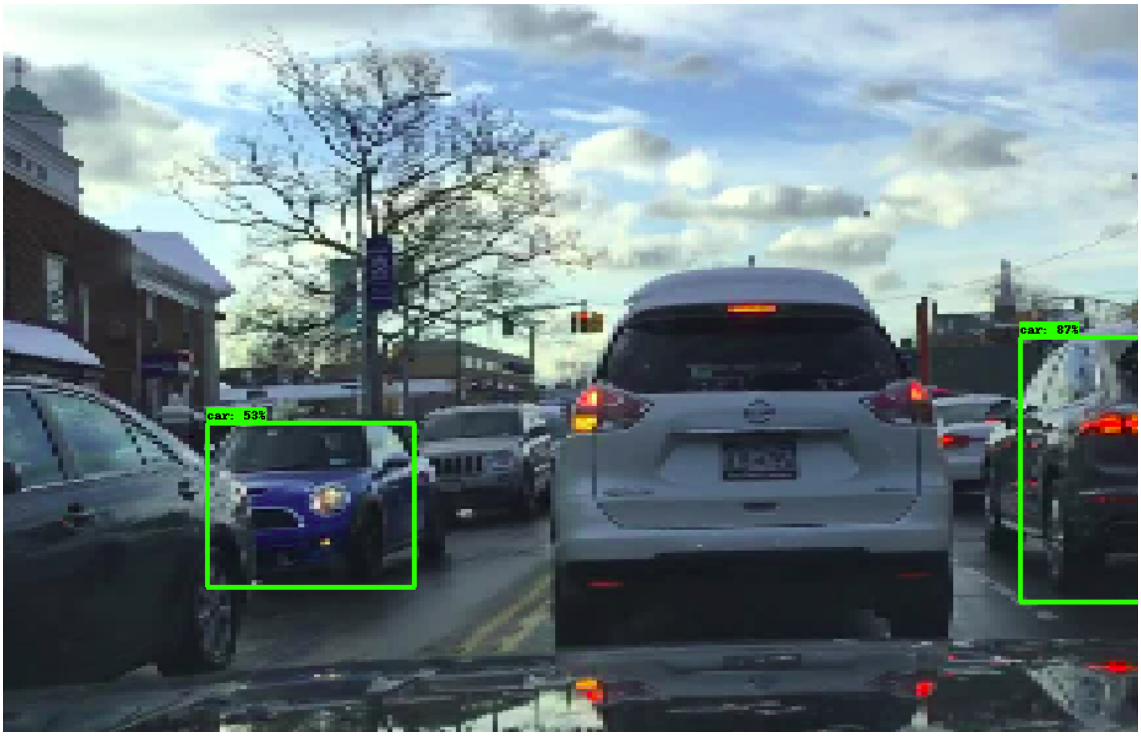


Figure 4.6: Object detector in action at 500 steps

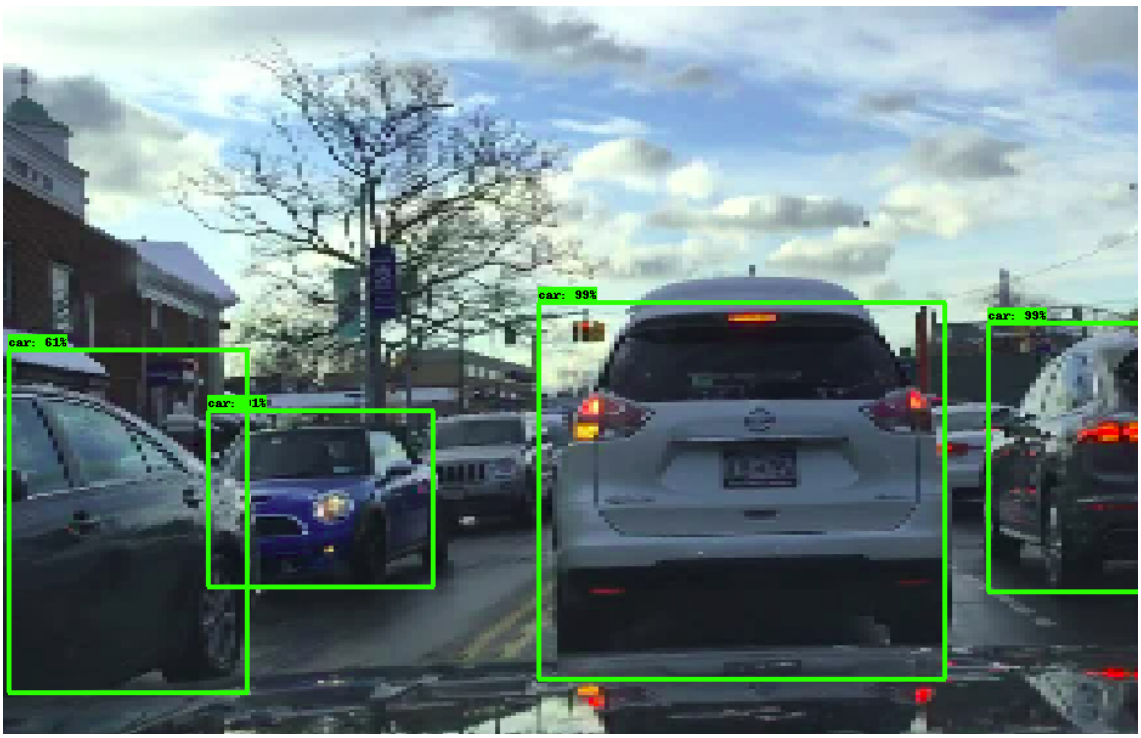


Figure 4.7: Object detector in action at 5000 steps

4.1.3 Experiment 3: Expanded Partial Dataset Retraining

Objectives

Following the more encouraging results from the previous experiment, it was decided that the easiest way to boost performance was to simply increase the amount of training data. A further 200 images were hand annotated, with 120 being added to the training data and the final 80 being added to the validation data. This brought the size of the manually annotated dataset to 400 images. Although this is a comparitively small amount, the relatively close relationship between the COCO dataset and the BDD100K dataset was a mitigating factor that allowed a smaller dataset.

Setup

Setup was carried out almost exactly the same as the previous experiment. 200 total images were annotated manually using LabelImg across the three classes - car, traffic light and person. Images were selected from the same semi-urban scenes without inclement weather or poor lighting. As the dataset had now doubled in size, the number of training steps was increased. The CNN was trained for 14,000 steps, at which point the loss values appeared to not be dropping signicantly.

Results

Results over the previous experiment were much improved. Over a total training time of 3hr and 50mins to reach 14,000 steps, the loss value fell to a lowest value of 1.356. Precision reached a maximum value of 0.2389 and recall reached a maximum value of 0.3402. Although these results are a significant improvement over the initial experiments, they are still inadequate. Although the MobileNet architecture proved valuable to carry out some initial experiments and highlight the issues presented by the BDD100K dataset it is still unsuitable for this project for several reasons. The requirement to resize the images down to 300x300 pixels in order to provide training images of the same dimensions of the initial COCO training data means that the pretrained MobileNet models provided by tensorflow will never be suitable for achieving acceptable results on the larger BDD100K images, although the MobileNet architecture itself does

achieve similar performance to other more complex networks (Howard et al., 2017). As such, a different CNN architecture will need to be investigated in order to obtain a trained model that can reliably carry out object detection on the BDD100K dataset.

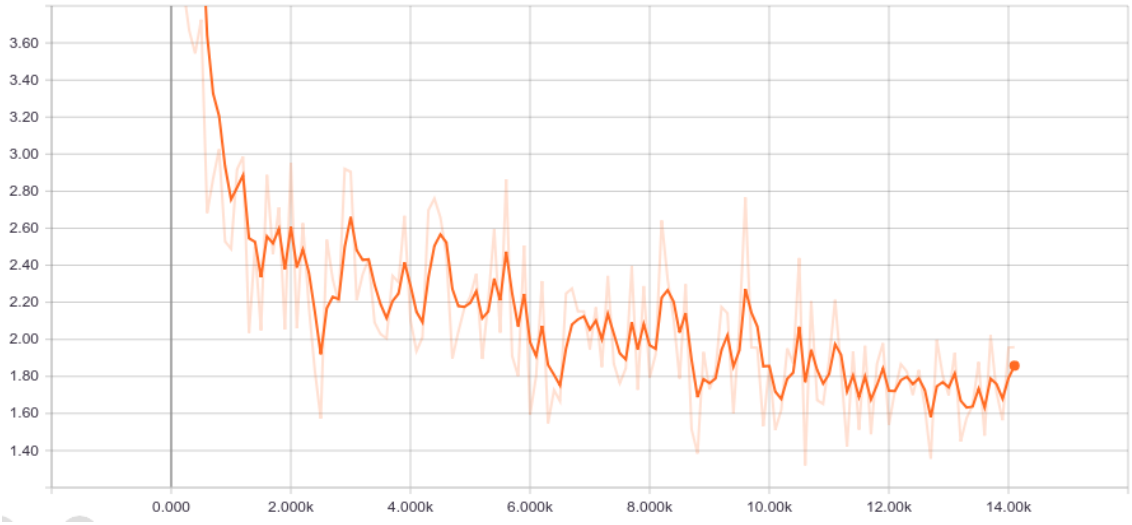


Figure 4.8: Loss values for MobileNet with 400 training images

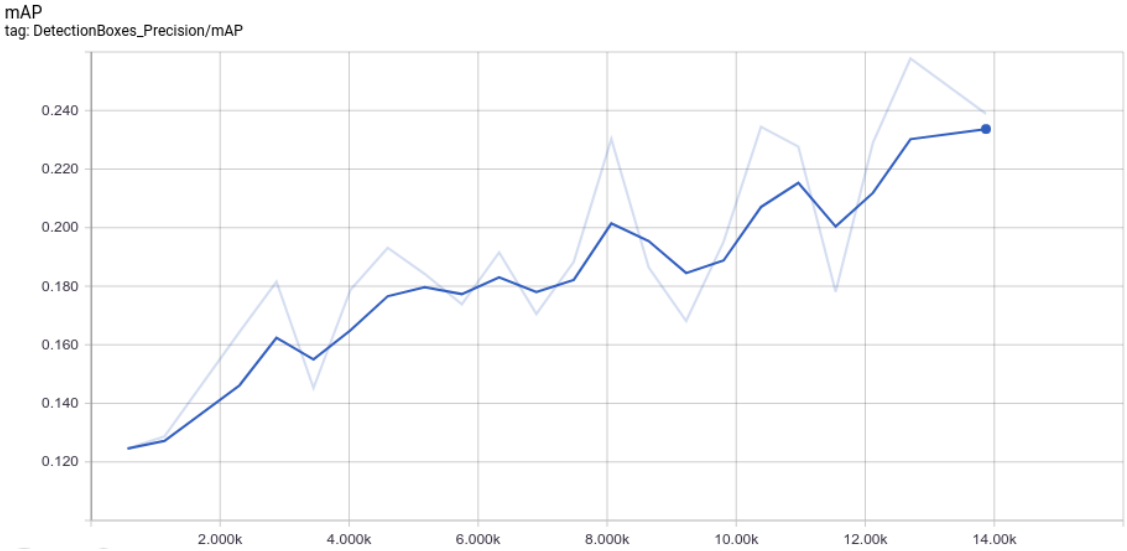


Figure 4.9: Precision values for MobileNet with 400 training images

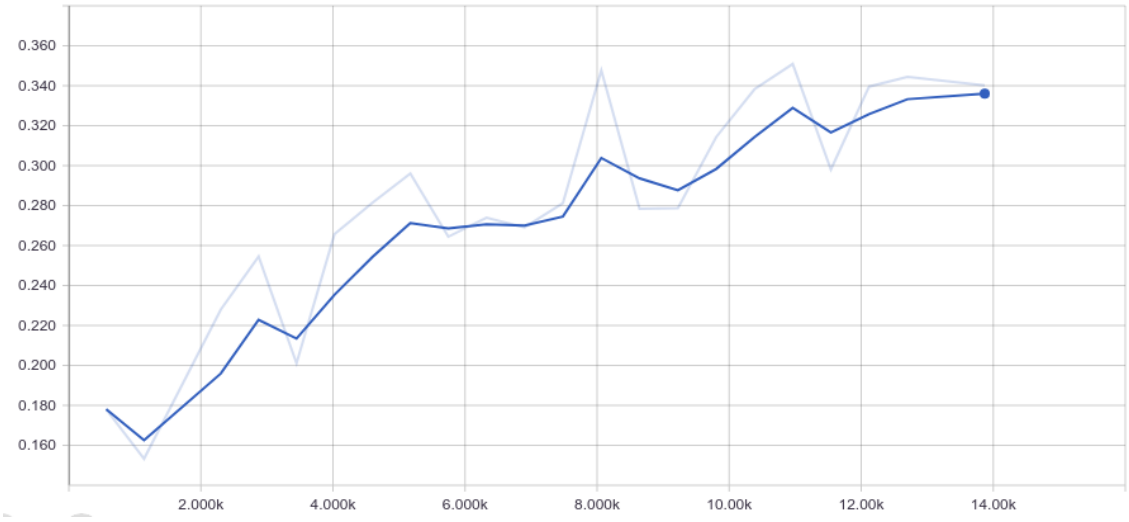


Figure 4.10: Recall values for MobileNet with 400 training images

4.1.4 Real-time Detection

In attempt to discover how suitable the SSD MobileNet model was for detecting objects in real time, the model was applied to video sequences from the BDD100K dataset. As Experiment 3 of this section had produced the best results, the model produced from this experiment was selected. Video sequences were selected from driving conditions that matched those present in the training data. Video sequences were fed into the model frame by frame, where annotation took place frame by frame, producing an output video with objects annotated. Annotation for the model took approximately 0.5 seconds per frame.

4.2 Faster RCNN Inception V2 Experiments

Following the relatively poor results of the MobileNet experiments, the Faster RCNN Inception V2 architecture was utilised.

4.2.1 Experiment 1: Pretrained Model Experiment

Objectives

The objective for this experiment was to retrain the Inception V2 architecture using all of the subsampled data from the BDD100K dataset. As the Inception V2 architecture is a much more complex architecture than the MobileNet architecture, it was expected that the results obtained from this experiment would be much closer to those presented in (Yu et al., 2018).

Setup

Setup for this experiment was minimal as the training data for this experiment had already been annotated. The pretrained Inception V2 model was simply downloaded and the steps presented by (*TensorFlow Object Detection Model Training* 2018) were once again followed, this time for the new model.

Results

Results for this experiment were far better than the MobileNet experiments. Over 10,000 training steps taking a total time of 1d 3hr and 25min, loss fell to a minimum value of 0.1253. Precision reached a maximum value of 0.4392 and recall reached a maximum value of 0.2170. The precision score of 0.4392 was a very encouraging result, as this is not drastically different from the results obtained in (Yu et al., 2018). These metrics are graphed in Fig 4.11, Fig 4.12 and Fig 4.13. As these metrics did not seem to be improving significantly at the 10,000 steps mark, training was halted at this point.

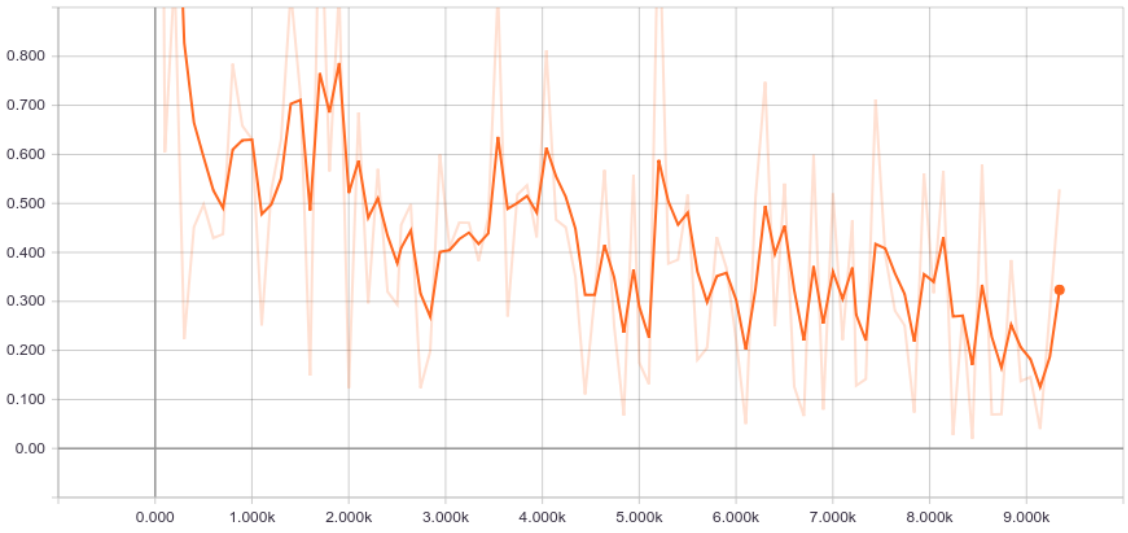


Figure 4.11: Loss values for Inception V2

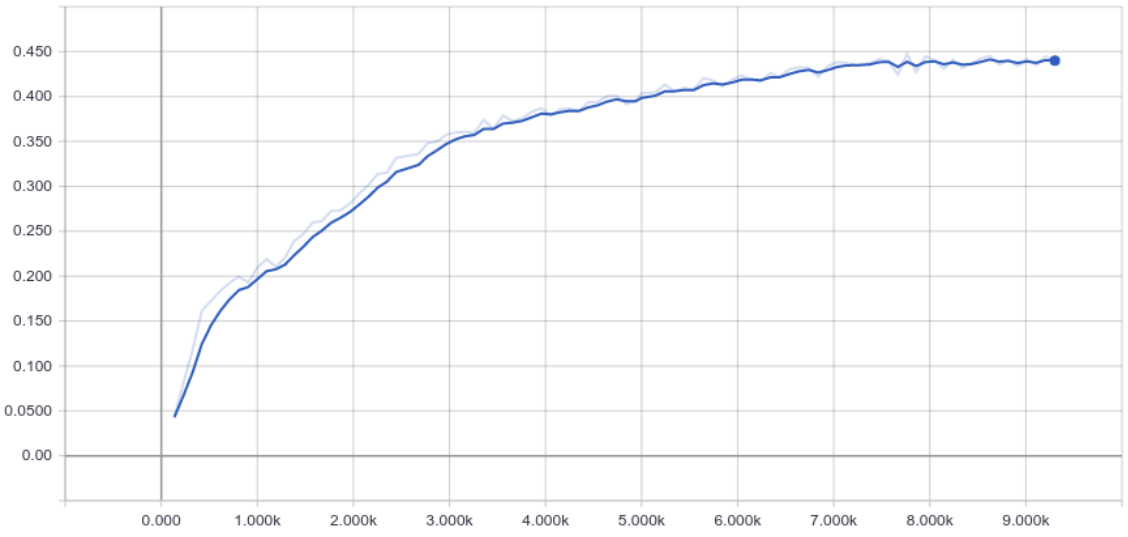


Figure 4.12: Precision values for Inception V2

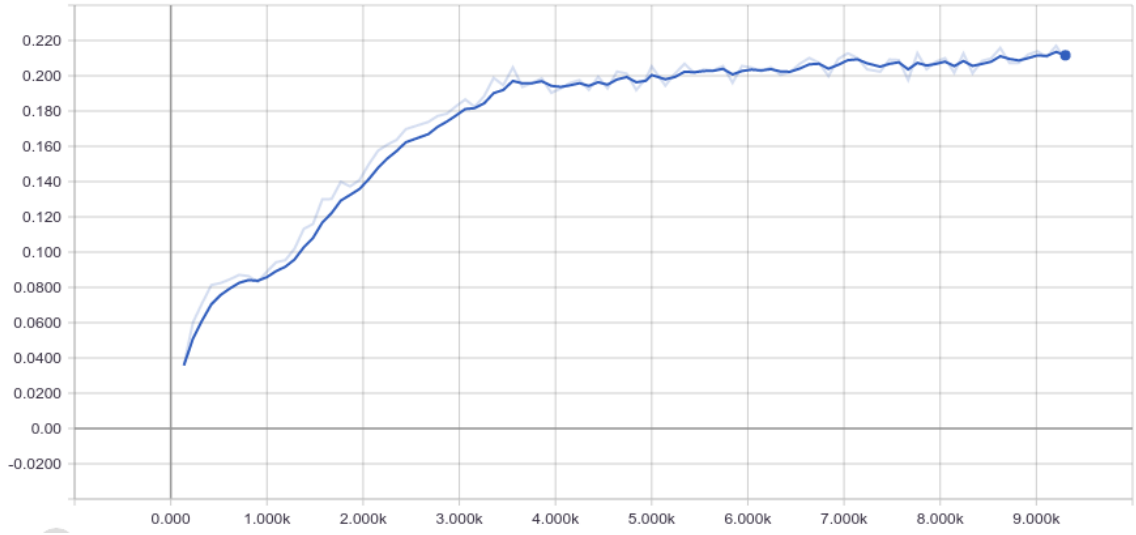


Figure 4.13: Recall values for Inception V2

4.2.2 Experiment 2: No Pretraining Experiment

Objectives

As every experiment prior to this experiment was carried out on a pretrained model provided by tensorflow, it was decided to train a model using the Inception V2 architecture that had received no pretraining whatsoever. Results for this experiment were expected to be extremely poor - with a dataset consisting of a meager 400 images overfitting was almost guaranteed.

Setup

Setup for this experiment was consistent with the previous experiment with one exception - no pretrained model checkpoint was loaded in at the beginning of training.

Results

As expected, results for this experiment were extremely poor over 3500 training steps totalling 5hr 35 minutes. Although Fig 4.14 may appear to be indicative of positive results, note the fact that this metric is simply the *training* loss. Training any CNN will lead to a lower training loss over time as the CNN adapts and learns the parameters of the training dataset. Observation of the validation loss in Fig 4.15, precision in Fig 4.16 and recall in Fig 4.17 clearly illustrate that this model is severely overfitting. The validation loss does not appear to be following any clear downward trend and although the precision and recall values appear to be following an upward trend, their values are extremely low. This implies that while the model is steadily getting better at detecting objects within the training data, it is failing to achieve these results upon the validation data - a sure sign of a model that is overfitting. Training was halted at 3500 steps as it was clearly apparent that overfitting was taking place and further training would simply be a waste of resources.

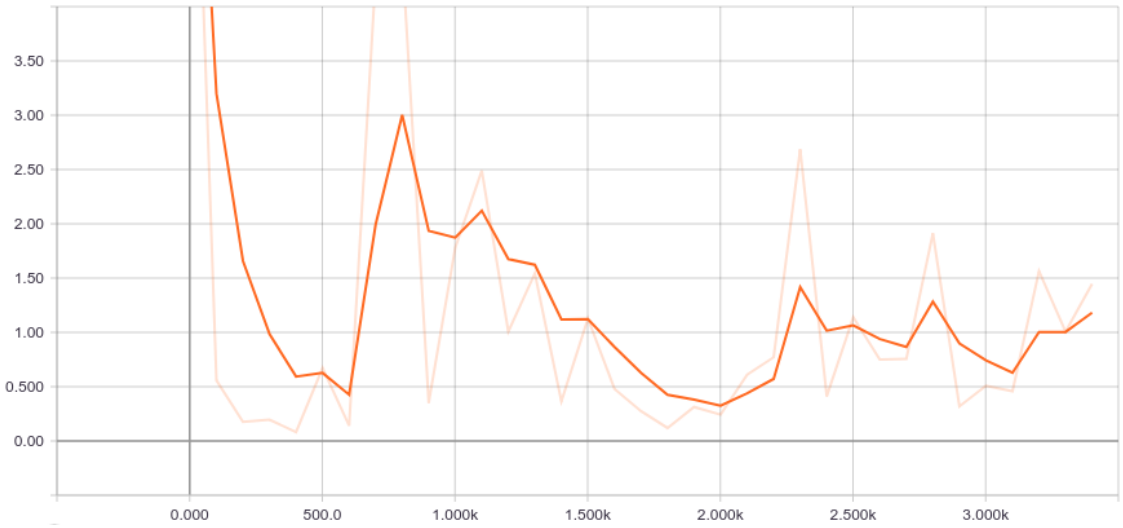


Figure 4.14: Training Loss with no Pretraining

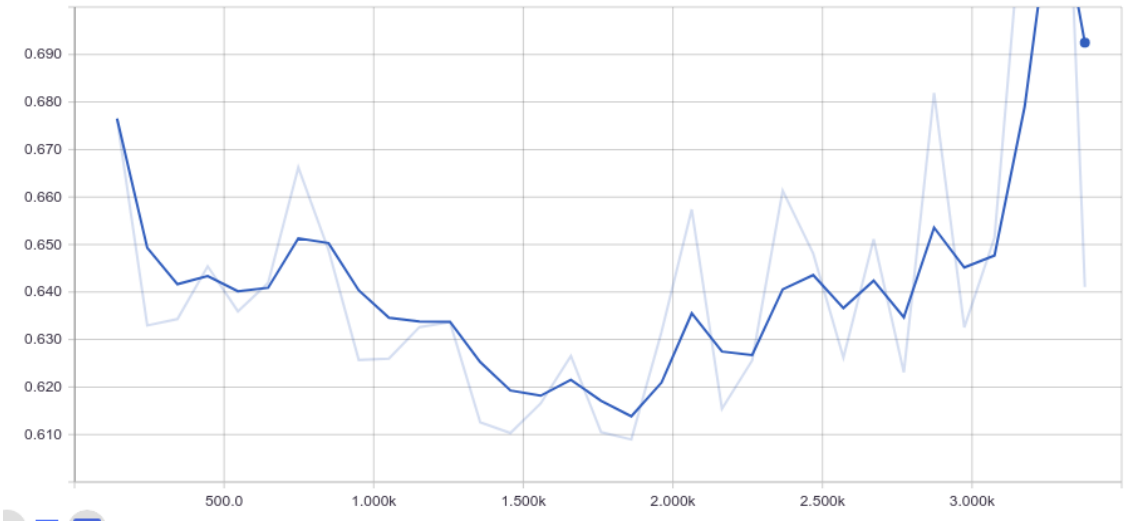


Figure 4.15: Validation Loss with no Pretraining

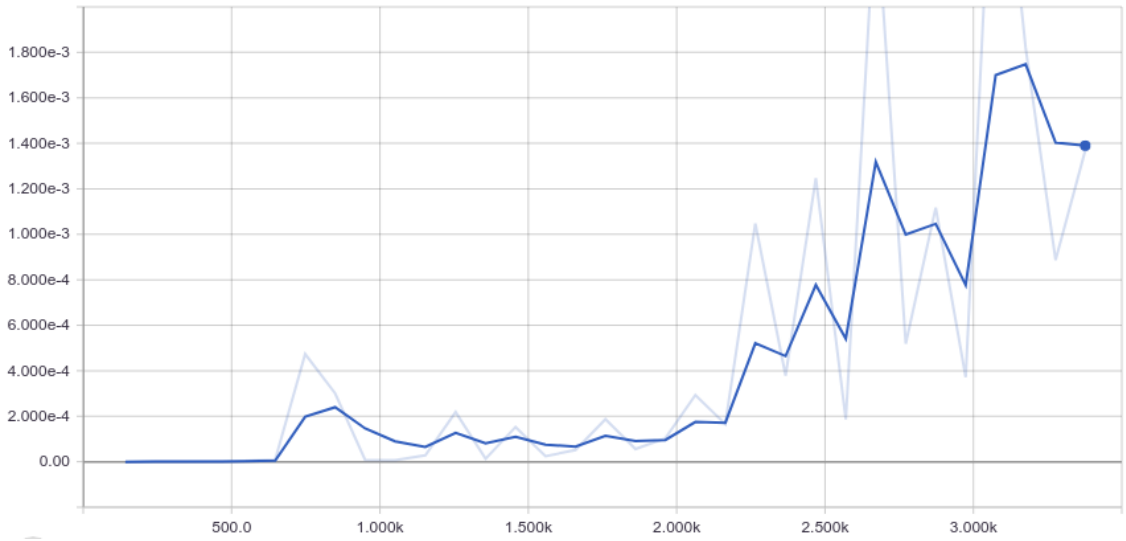


Figure 4.16: Precision with no Pretraining

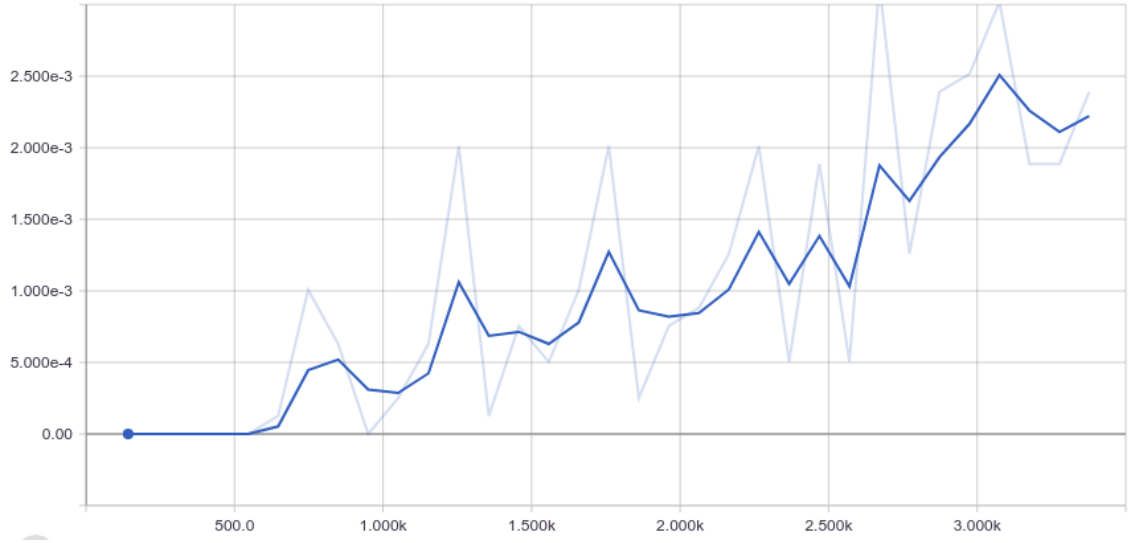


Figure 4.17: Recall with no Pretraining

4.2.3 Real-time Detection

In attempt to discover how suitable the Faster RCNN Inception architecture was for performing real-time detection, the model from Experiment 1 of this section was utilised for performing video annotation. Although this experiment had achieved much better results than any of the MobileNet experiments, annotation took approximately 3 seconds per frame - a significant increase over the MobileNet experiments. This level of performance would be entirely unacceptable for any potential real-world solution, and it appears that a middle-ground should be investigated, with a solution retaining some of the speed of the SSD MobileNet experiments whilst also retaining some of the evaluation metrics of the Faster-RCNN Inception experiments. A sample video has been uploaded to YouTube to demonstrate how the model behaves in the context of a video and can be viewed at: https://youtu.be/vLHHTuWw1Vw

4.3 Empirical Studies Cost and Results

At this point in the Empirical Studies, the bills from AWS had run up to a point where it was no longer feasible for further experimentation to take place. Even though the EC2 instance type being utilised was the cheapest GPU accelerated option, the total bill across the months of February and March totalled $192.53, with a further $12.48 estimated bill for April across a total of 123.530 total hours of active instance time. The bill for March, during which the bulk of training was carried out, is shown in Fig 4.18. Due to this project being funded by a very tight student budget, this regrettably forced a halt to any further work using AWS resources. However, at this point enough experimentation had been carried out to form some final conclusions.

Details

AWS Service Charges		**$182.30**
▾ Data Transfer		$1.04
▸ **EU (Ireland)**		**$1.04**
▾ Elastic Compute Cloud		$147.17
▾ **EU (Ireland)**		**$147.17**
Amazon Elastic Compute Cloud running Linux/UNIX		$109.27
$0.972 per On Demand Linux p2.xlarge Instance Hour	112.420 Hrs	$109.27
EBS		$37.90
$0.05 per GB-Month of snapshot data stored - EU (Ireland)	70.533 GB-Mo	$3.53
$0.11 per GB-month of General Purpose SSD (gp2) provisioned storage - EU (Ireland)	312.452 GB-Mo	$34.37
▸ Key Management Service		$0.00
Taxes		
VAT to be collected		$34.09

Figure 4.18: AWS Bill for March

The results from the Empirical Studies are shown in the below table along with the results provided in (Yu et al., 2018). The results from the MobileNet experiment 1 have been omitted as no meaningful results were gleaned from this experiment.

Experiment	Training Steps	Training Time	Precision	Recall	Per-frame annotation
Yu et al.	-	-	0.53	-	-
MobileNet Experiment 2	5000	1 hr 25 min	0.1995	tbd	0.5s
MobileNet Experiment 3	14000	3 hr 50 min	0.2389	tbd	0.5s
Inception Experiment 1	10000	1 d 3 hr 25 min	0.4392	0.2170	3s
Inception V2 Experiment 2	3500	5 hr 35 min	1.7800 e-3	2.500 e-3	3s

Table 4.1: Results from Empirical Studies

Chapter 5

Application Implementation

In order to provide an interaction with the models trained during the Empirical Studies section, a lightweight Flask application was created. This application was developed to allow users to upload images to a simple public-facing website and observe object detection taking place on the uploaded images.

5.1 AWS Instance

As the AWS Deep Learning AMI that was used to train the models was still in use, it was decided to simply use this virtual machine for the deployment of the application. All AWS EC2 instances are provided with a public-facing IP address by default, so the only work required on the instance was editing its security roles to allow HTTP requests to the instance for all IP addresses.

5.2 Implementation

The first step for the implementation of this prototype was providing some way for the trained model to annotate the provided images. Tutorial code provided by tensorflow to interact with their pre-trained models (tensorflow, 2019) was taken and adapted for use in the context of a Flask application. Modifications to the code were slight - the program was changed to accept files passed as a parameter instead of loading in images from a predefined location on the machine, and images were then converted

to bytes and return after annotation had taken place. Next a simple flat html page was created to be served by the Flask application that contained for image upload. Lastly a simple Flask script was written to accept images sent via POST request, send these images to the object detection code and finally display the newly annotated image. Fig 5.1 shows the design of the simple interface through which users can upload their images.

Figure 5.1: Simple Webpage to allow Image Upload

5.3 Code Snippets

Some interesting snippets of code from the application implementation have been provided below:

```python
# Import method from the object detection script
from object_detection.ObjectDetector import detect_objects
import io

from flask import Flask, render_template, request

from PIL import Image
from flask import send_file

app = Flask(__name__)

# Calls the index.html file present in the "templates" folder
@app.route("/")
def index():
    return render_template('index.html')


# Accept jpg files via POST request, send to be annotated and then display result on-page
@app.route("/", methods=['POST'])
def upload():
    if request.method == 'POST':
        file = Image.open(request.files['file'].stream)
        img = detect_objects(file)
        return send_file(io.BytesIO(img),attachment_filename='image.jpg',mimetype='image/jpg')

# Accept requests through port 5000, this has been opened for HTTP requests for the AWS instance
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

Figure 5.2: Flask Application Code, calls Annotation Code

```python
def detect_objects(image):
    # the array based representation of the image will be used later in order to prepare the
    # result image with boxes and labels on it.
    image_np = load_image_into_numpy_array(image)
    # Expand dimensions since the model expects images to have shape: [1, None, None, 3]
    image_np_expanded = np.expand_dims(image_np, axis=0)
    # Actual detection.
    output_dict = run_inference_for_single_image(image_np, detection_graph)
    # Visualization of the results of a detection.
    vis_util.visualize_boxes_and_labels_on_image_array(
      image_np,
      output_dict['detection_boxes'],
      output_dict['detection_classes'],
      output_dict['detection_scores'],
      category_index,
      instance_masks=output_dict.get('detection_masks'),
      use_normalized_coordinates=True,
      line_thickness=8)

    img = cv2.cvtColor(image_np, cv2.COLOR_RGB2BGR)
    img = cv2.imencode('.jpg', img)[1]
    return(img.tobytes())
```

Figure 5.3: Method called by Flask Application, accepts and returns image

Chapter 6

Final Conclusion and Discussion

6.1 Summary

The primary purpose behind this project has been an investigation into how CNN's perform object detection and the issues encountered during this process. Once the research area for this project had been defined, a comprehsive literature review of the subject material was undertaken. One of the main objectives for the literature review was to develop and document an expansive knowledge of CNN's. A working knowledge of using tensorflow was also developed through a Udemy tutorial (*Complete Guide to Tensorflow for Deep Learning with Python* 2018). Once this knowledge had been gained, several experiments were undertaken in an attempt to replicate the results published in (Yu et al., 2018). Although the full BDD100K dataset quickly proved far too complex to use in its entirety, a subsampled version of the dataset was sufficient for training instead. Although the results provided by Yu et al. were not replicated perfectly, results that were within an acceptable margin of error were obtained. Unfortunately financial constraints related to using AWS services did not allow for further experimentation, however I am satisfied with the results obtained throughout the experimentation phase of this project. Finally, a lightweight Flask application was built and deployed in order to allow users to interact with the models trained during the experimentation phase.

6.2 Reflections

6.3 Future Work

This project has very much served to foster a passion for the field of Computer Vision and object detection tasks specifically. This has been one of the very few projects for me where I have a genuine desire to continue work upon related projects after completion. I have several ideas as to how I am going to go about work upon future personal projects in this field.

This Project

With AWS costs forcing experimentation to stop for this project, there are several potential areas of future work that could be worth pursuing. With the Faster-RCNN Inception architecture producing good results albeit at the cost of annotation speed, I would like to investigate the possibility of using SSD or YOLO based Inception architectures, as I feel that this would help decrease annotation times and create a model more suited to performing accurate object detection in real-time. If I am to continue work on this project in future, opting for a more powerful AWS machine instance would also be a priority in order to aid detection in real-time. I would also like to investigate training a MobileNet architecture myself on the COCO dataset without image resizing, and then retrain this model using the BDD100K dataset. As mentioned above, the MobileNet architecture can produce similar results to other more complex architectures at a fraction of the computational costs (Howard et al., 2017), so I feel that this solution could be a good trade off between performance and computation times.

Future Projects

One avenue that I am definitely going to be exploring is applying my experience gained from this project to my initial choice of project, a fish detector. With time constraints no longer an issue and a graduate salary allowing the purchase of camera equipment, I intend to begin the collation of a dataset for this project. Although the issue still remains of gaining the required domain knowledge to annotate species for training data, I am confident that with enough research I will be able to annotate some basic easily-identifiable species. Thus far, I have not been able to find any models

that have been pretrained on a fish species dataset. I have however found some publically available datasets such as (Phoenix X. Huang and Fisher, 2013), which contains species of fish found in European waters, upon which it may be possible to train a model myself. I believe that then retraining this model using my collated dataset may potentially lead to positive results. This is going to be my primary personal project going forward.

Bibliography

Albawi, Saad, Tareq Abed Mohammed, and Saad Al-Zawi (2017). "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. IEEE, pp. 1–6.

Amit, Yali (2002). *2D object detection and recognition: Models, algorithms, and networks*. MIT Press.

Arbelaez, Pablo et al. (2011). "Contour detection and hierarchical image segmentation". In: *IEEE transactions on pattern analysis and machine intelligence* 33.5, pp. 898–916.

Ben-Yacoub, Souheil, B Fasel, and Juergen Luettin (1999). "Fast face detection using MLP and FFT". In: *Proc. Second International Conference on Audio and Video-based Biometric Person Authentication (AVBPA'99)*. CONF, pp. 31–36.

Burns (2014). "An in-depth exploration of in-car driving distractions from an Irish perspective". In: *RSA International Conference on Driver Distraction March 20th 2014*.

Complete Guide to Tensorflow for Deep Learning with Python (2018). URL: https://www.udemy.com/complete-guide-to-tensorflow-for-deep-learning-with-python/ (visited on 13/12/2018).

Convert the Berkeley Deepdrive dataset to a TFRecord file (2018). URL: https://github.com/meyerjo/deepdrive_dataset_tfrecord (visited on 01/03/2018).

Domingos, Pedro (2012). "A few useful things to know about machine learning". In: *Communications of the ACM* 55.10, pp. 78–87.

Géron, Aurélien (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.".

Girshick, Ross et al. (2014). "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587.

Greenblatt, Jeffery B and Samveg Saxena (2015). "Autonomous taxis could greatly reduce greenhouse-gas emissions of US light-duty vehicles". In: *Nature Climate Change* 5.9, p. 860.

Gupta, Samta and Susmita Ghosh Mazumdar (2013). "Sobel edge detection algorithm". In: *International journal of computer science and management Research* 2.2, pp. 1578–1583.

Harris, Mark (2015). "A cheaper way for robocars to avoid pedestrians". In: *IEEE Spectrum* 52.7, pp. 16–16.

Howard, Andrew G et al. (2017). "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861*.

Hubel, David and Torsten Wiesel (2011). *Hubel & Wiesel - Cortical Neuron - V1*. Youtube. URL: https://www.youtube.com/watch?v=8VdFf3egwfg.

international, SAE (2016). "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles". In: *SAE International,(J3016)*.

Kocić, Jelena, Nenad Jovičić, and Vujo Drndarević (2018). "Sensors and Sensor Fusion in Autonomous Vehicles". In: *2018 26th Telecommunications Forum (TELFOR)*. IEEE, pp. 420–425.

Kotsiantis, Sotiris B, I Zaharakis, and P Pintelas (2007). "Supervised machine learning: A review of classification techniques". In: *Emerging artificial intelligence applications in computer engineering* 160, pp. 3–24.

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

Launching and Configuring a DLAMI (2018). URL: https://docs.aws.amazon.com/dlami/latest/devguide/launch-config.html/ (visited on 05/02/2018).

Learned-Miller, Erik G (2011). "Introduction to computer vision". In: *University of Massachusetts, Amherst*.

LeCun, Yann, Bernhard Boser, et al. (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4, pp. 541–551.

LeCun, Yann, Yoshua Bengio, et al. (1995). "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10, p. 1995.

Lin, Tsung-Yi et al. (2014). "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer, pp. 740–755.

Milakis, Dimitris, Bart Van Arem, and Bert Van Wee (2017). "Policy and society related implications of automated driving: A review of literature and directions for future research". In: *Journal of Intelligent Transportation Systems* 21.4, pp. 324–348.

Minsky, Marvin and Seymour Papert (1969). "An introduction to computational geometry". In: *Cambridge tiass., HIT*.

O'Shea, Keiron and Ryan Nash (2015). "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458*.

Phoenix X. Huang, Bastiaan B. Boom and Robert B. Fisher (2013). *Fish Recognition Ground-Truth data*. Fish4Knowledge project. URL: http://groups.inf.ed.ac.uk/f4k/GROUNDTRUTH/RECOG/.

Al-Qudah, Ghaith Ahmad (2009). "A Multi-Layer Perceptron Neural Network Based-Model for Face Detection". MA thesis. Middle East University for Graduate Studies.

Redmon, Joseph et al. (2016). "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788.

Ren, Shaoqing et al. (2015). "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems*, pp. 91–99.

Rogers, Simon and Mark Girolami (2016). *A first course in machine learning*. CRC Press. Chap. 2.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1985). *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science.

Singh, Vaibhav Kant and Shweta Pandey (2016). "Minimum configuration MLP for solving XOR problem". In: *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 174–179.

Srivastava, Nitish et al. (2014). "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Surden, Harry and Mary-Anne Williams (2016). "Technological opacity, predictability, and self-driving cars". In: *Cardozo L. Rev.* 38, p. 121.

Szegedy, Christian et al. (2016). "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826.

Tan, Chuanqi et al. (2018). "A survey on deep transfer learning". In: *International Conference on Artificial Neural Networks*. Springer, pp. 270–279.

tensorflow (2019). *Object Detection Demo*. tensorflow. URL: https://github.com/tensorflow/models/blob/master/research/object_detection/object_detection_tutorial.ipynb.

TensorFlow Object Detection Model Training (2018). URL: https://gist.github.com/douglasrizzo/c70e186678f126f1b9005ca83d8bd2ce (visited on 01/03/2018).

Verschae, Rodrigo and Javier Ruiz-del-Solar (2015). "Object detection: current and future directions". In: *Frontiers in Robotics and AI* 2, p. 29.

Xiao, Tong et al. (2015). "Learning from massive noisy labeled data for image classification". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2691–2699.

Yosinski, Jason et al. (2014). "How transferable are features in deep neural networks?" In: *Advances in neural information processing systems*, pp. 3320–3328.

Yu, Fisher et al. (2018). "BDD100K: A diverse driving video database with scalable annotation tooling". In: *arXiv preprint arXiv:1805.04687*.

Zhao, Jianfeng, Bodong Liang, and Qiuxia Chen (2018). "The key technology toward the self-driving car". In: *International Journal of Intelligent Unmanned Systems* 6.1, pp. 2–20.

Zhao, Zhong-Qiu et al. (2019). "Object detection with deep learning: A review". In: *IEEE transactions on neural networks and learning systems*.

Ziou, Djemel, Salvatore Tabbone, et al. (1998). "Edge detection techniques-an overview". In: *Pattern Recognition and Image Analysis C/C of Raspoznavaniye Obrazov I Analiz Izobrazhenii* 8, pp. 537–559.

Appendix A

Project Plan

The following table outlines the delivery dates agreed upon with my supervisor for the sections of this report along with the implementation of the demo day product. Athough it was not followed strictly, it provides a general overview of how this project progressed over time.

Chapter	Section	Est. Completion Date
Introduction	Overview of Problem Area	12/10/2018
	Motivations	12/10/2018
	Objectives	05/10/2018
	Contribution	05/04/2019
	Methodologies	05/04/2019
	Project Plan	05/10/2018
Background Research	Intro to ML	19/10/2018
	Intro to Neural Networks	26/10/2018
	Intro to CNN's	26/10/2018
	Intro to Computer Vision	03/11/2018
	Tensorflow Tutorial	19/10/2018
Empirical Studies	MobileNet Architecture	25/01/2019
	Inception Architecture	25/02/2019
Application Impl.	Flask App	28/03/2019
	Object Detector Script	28/03/2019
Final Conclusion	Summary	08/04/2019
	Reflections	08/04/2019
	Future Work	08/04/2019

Table A.1: Delivery dates for project milestones

Appendix B

Poster

OBJECT DETECTION IN ROAD IMAGES

Name: Rory Egan

Supervisor: J.J. Collins

OBJECTIVES

This project explores the use of Convolutional Neural Networks (CNNs) to detect and classify objects in road scenes. The images used throughout this project were taken from the Berkeley Deep Drive dataset. Several CNN architectures have been investigated.

CNNs EXPLAINED

CNNs are a type of deep neural network commonly applied to vision problems. Inspired by the visual cortex of the brain, CNNs identify primitive features of an object which are then mapped to more abstract concepts that make up the classes to be detected.

BERKELEY DEEP DRIVE

The dataset utilised for this project is the Berkeley Deep Drive dataset, consisting of 100,000 images and 100,000 HD video sequences annotated with 10 different classes. Due to the highly complex nature of the dataset it was manually subsampled down to 3 classes - cars, traffic lights and people. An example of an image being annotated is shown below:

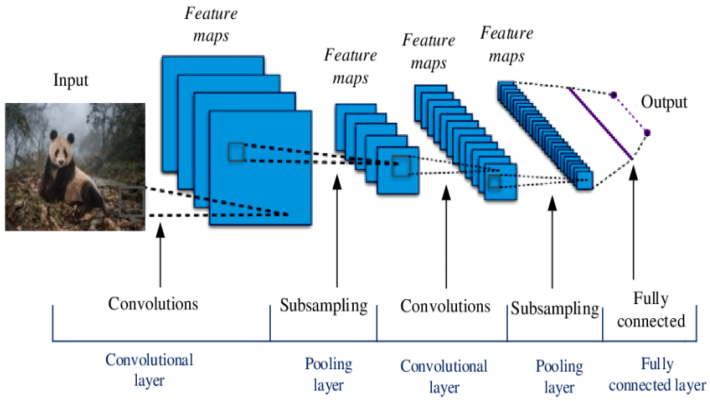


Image sourced from https://www.researchgate.net/figure/An-example-of-a-simple-CNN-architecture_fig6_327260166

AMAZON WEB SERVICES

Training was carried out using an AWS Deep Learning AMI to gain access to more computational power. This VM is also being utilised to serve a Flask application to demo the object detection abilities of the trained model.

INCEPTION ARCHITECTURE

The Inception v2 architecture was utilised for this project. The Inception architectures were developed to handle objects appearing at different focus levels within an image. The architectures are based on the concept of inception blocks. This is a combination of filters concatenated into a single ouput rather than having a layer for each filter. This allows the creation of "wider" rather than "deeper" networks, reducing the computational requirements of the network.