Nottingham Trent University
School of Science and Technology

The Mathematics Behind Machine Learning & Neural Networks in Autonomous Vehicles

A dissertation submitted in partial fulfilment of the requirements for MMath (Hons) Mathematics at Nottingham Trent University.

April 2022

Author
Syd PROOM, N0788203

Supervised by
Dr Archontis GIANNAKIDIS

Abstract

The number of companies that are attempting to create a fully autonomous vehicle increases each year with most traditional automotive manufacturers pledging to transition to driverless vehicles, it is no wonder why the computer vision neural network space is constantly being pushed forward. Machine learning and neural networks have become buzzwords in the technology industry, but these terms simply describe a mathematical model that can iterate and adapt to data being fed through it. The aim of this paper is to unveil and discuss the mathematics behind Convolutional Neural Networks (CNNs) which are a subset of Neural Networks (NNs) that focus on 2D and 3D input data. This constant push makes it seem like creating the first autonomous vehicle is like the 21st century equivalent of the 20th century "space race".

By using classification, regression and object detection techniques, a total of four models will be created to showcase how CNNs work by using real-world public data. Binary classification will be used to identify whether a car is within an image and multi-class classification attempts to improve this by adding traffic signs/lights to the possible classes. Explainability using Grad-CAM will be explored to determine if the model performs as expected. The steering angle of a vehicle is one of the most important inputs of a vehicle, so a model that predicts the value of steering required to stay in lane will be created using images that are linked to vehicle telemetry for training. Finally, object detection using the latest iteration of "You Only Look Once" (YOLOv5) will create a model that describes where objects like cars and pedestrians are located in 2D space using bounding boxes. All models described will be evaluated using validation and testing data to determine the performance on new data.

Acknowledgements

I would like to thank the following people who helped me complete this dissertation:

- My project supervisor Dr Archontis Giannakidis for allowing me to complete this project and providing consistent guidance while learning this new topic.
- Tuan Bohoran for his willingness to help with any troubleshooting in Python and informing me of any coding improvements I could make, all while completing his own PhD project, I appreciate this a lot.
- Of course, I would not be writing this had it not been for my parents and grandparents who have showered me with their unconditional support and encouragement throughout university and my academic career.

Contents

Chapter 1

Introduction

1.1 Motivation for the Project

Machine learning and neural networks have seen significant development over the past decade with the improvements in computational ability of PCs. The history of neural networks starts in 1943 where a paper introduced a simplified model of a biological neuron found in animal brains including humans [27]. Despite the hype around this discovery at the time, very little was done to push the methods forward as other techniques were found to be much better at the time. These techniques include Support Vector Machines (SVM) in the 1990s [16]. Before the internet, data was incredibly hard to find without creating custom datasets and having a large amount of data is integral to creating a useful neural network.

Now that public datasets are readily available for a variety of applications and computing power has developed significantly to handle training such datasets, there has been a large interest in neural networks once again. Neural networks are now commonly used in medical, financial and automotive industries which showcases the vast number of applications which they can be used in. Even in 2022, the amount of research going into neural networks and its applications does not seem to be slowing.

Some forward-thinking brands within the automotive industry are currently developing neural networks to replicate human driving behaviour. The hope is that in the future, cars driven by software that have more sensors (for example cameras, radar, LiDAR or ultrasonic) than humans will far outperform vehicles driven by humans in terms of safety. The faster that this transition happens the better for all users of public roads and is the reason why so many companies such as Tesla, Waymo and Comma AI are investing so much into developing self-driving vehicles. The logic behind neural networks uses various topics from mathematics which will be discussed in this paper.

1.2 Outline of Study

This paper will analyse various techniques in machine learning for predicting discrete or continuous values. Predicting a discrete value (such as 0, 1, 2, ...) is known as a classification problem where each integer represents a class in categorical data such as types of dogs, cats or plants. Regression on the other-hand predicts a continuous value such as the price of a house, weight of a person or the stock price of a company.

The performance of a supervised machine learning model is heavily dependent on the quality and size of the dataset and labels used to train it. More training data means that a model has more experience with identifying features that relate to the object/value. The dataset is the largest contributor for how well a model performs on new data. A simple example of a poor dataset is a set of images that contain just vehicles. If this dataset was used to train a model that classifies vehicles or pedestrians, the model would not know what features within an image that means a pedestrian is within an image. Finding a dataset for each of the models in this paper will be discussed in the relevant chapters but Figure 1.1 shows how this data is used to create a model:

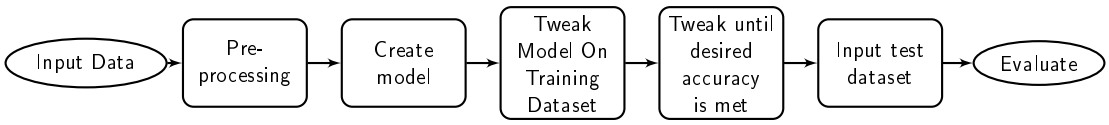


Figure 1.1: Basic machine learning architecture

Since classification and regression are quite different in their approaches, they will be documented in different chapters of this report. In Chapter 3, classification will be used to determine whether there exists a car or not in an image. A problem with two classes is known as a binary classification problem, as the algorithm will simply output the probability that there is a car in the image (p) and thus the probability that there is no car is the opposite probability ($1-p$). This value is typically rounded up or down to 1 or 0 to then give a more clear view of which class the algorithm predicts the image to be part of. This will also be extended to another class by using multi-class classification.

In Chapter 4, regression will be used to predict a value for the most appropriate steering wheel angle in order to stay within the current lane. Rather than output a probability as seen in classification, regression algorithms simply output the value it predicts which could be any rational number. With the appropriate dataset, this technique could be applied to other driving inputs seen in vehicles, such as the accelerator and brake pedal by predicting the percentage that the pedal is being pushed down. However for the

steering angle, the value 0 would be the neutral point in which the car is not steering left or right. Thus, a negative value implies the car needs to steer left and positive for steering right.

Next, in Chapter 5, a state of the art technique known as You Only Look Once (YOLO) [33] will be used as an evolution to traditional classification. The benefit of an object detection technique such as YOLO is that the model can not only predict what an object is, but also the position within the input image. The position is predicted using bounding boxes that encapsulate the entire object within a rectangle. This method is extremely useful for autonomous vehicles, as positioning the objects that are in front (and around) the vehicle gives the car a much better understanding of the environment and what the safest manoeuvre is likely to be.

Finally, Chapter 6 will conclude this paper and the work that was completed. Discussions will be made about the plausibility of self driving vehicles on public roads along with the requirements to achieve this. Future work will be listed that could push forward this project and a conclusion will summarise this papers findings.

Chapter 2

Machine Learning Theory

Machine learning has many applications for many different tasks. Because of this, tools have been created to help create unique machine learning models that work for specific objectives. In this paper the objective is to make tools that could be used for autonomous vehicles, but other examples include handwriting recognition, medical diagnosis or animal classification. Essentially, machine learning tasks involve an input such as an 2D image, vector or value that a model has ideally never seen before and receiving an output in the form of a discrete (classification) or continuous (regression) value. Fundamentally, the model will take the inputs and give a number of probabilities as outputs depending on the task, but how does it find these probabilities? In this chapter, the basis for most machine learning models will be explained before any models are created in the later chapters of this paper.

2.1 Data Preparation

Gathering good data can be the hardest part of creating a machine learning model, especially if the data needs to be created because there is not existing data that is suitable for the problem. At this time, there is no perfect way for labelling to be done without human interaction which is the process of assigning value(s) that relate to each piece of data within a dataset. This process tends to take the longest time out of the entire process of creating a model as the quality and quantity of the labels is a fundamental part of how accurate the model will be.

Data can also be manipulated in order to increase the size of the dataset without needing to create labels for each new image. There are many ways of doing this such as flipping the image in the x or y axis, image cropping and colour adjustments like changing an image to black and white. These techniques are known as data augmentation [45] and are widely used in lots of applications, as the rotation or scale of an

image doesn't tend to change the key features that a model would identify as being a value/class. For example, a model that identifies flowers should not perform differently if an image is input into the model that is upside-down, so it is important to train the model on data with as many possible transformations as possible.

Next, all data used in machine learning must be mathematically transformed to the ensure that the algorithms used in machine learning models perform as expected between different types of datasets [20]. Often for images this is achieved using normalisation, of doing this for 2D 8-bit images is by dividing each RGB value by its maximum value (255). A similar technique should be used to the labels if they fall outside the range of $(0, 1)$.

Finally, the data must be split into a "training" and "test" dataset. The training dataset will also have another split to get a "validation" dataset. The role of each dataset will be discussed further in 2.5, but for now it is worth mentioning that the training data will only be used for training, and the test set will only be used to testing, so each dataset will never be merged together. The model should never see the test data until it has finished training.

2.2 Neural Network Theory

Machine learning has many skews, one of which are Neural Networks (NN). NNs are made of many layers of neurons with independent weights and thresholds [12]. These neurons are similar to how biological neurons work, however neurons found in machine learning output a continuous value between 0 and 1 rather than just 0 or 1 like the biological type would output. Figure 2.1 represents a general neural network layer structure. Where x_n represents the input layer with n input nodes, $h_m^{(n)}$ represents n hidden layers each with m nodes and finally an output layer which typically outputs k probabilities for k classes, or just a single value for regression tasks.

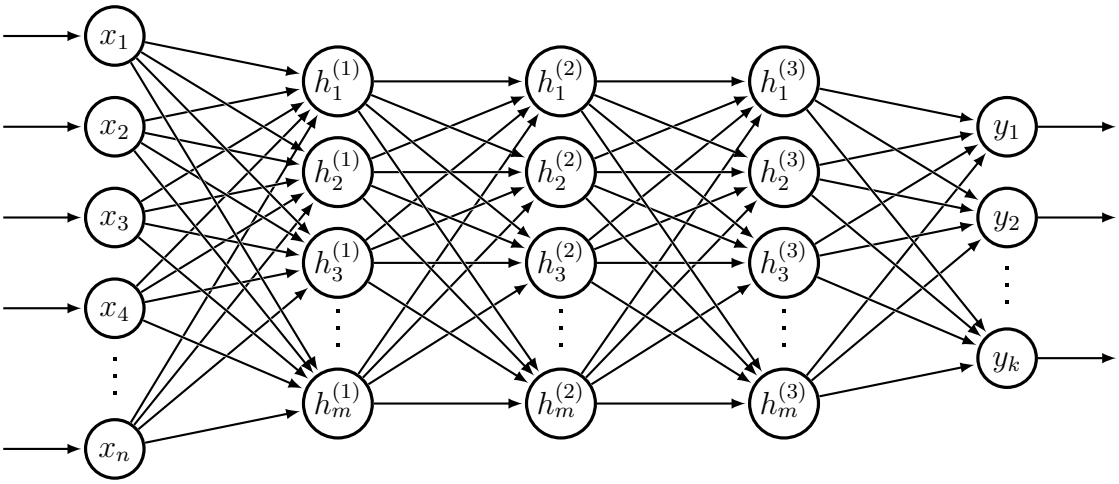


Figure 2.1: General neural network layer structure made of input, hidden and output layers with nodes in green, blue and red respectfully.

When data is put into the neural network, it passes through the hidden layers and a value is calculated at each node. This value is known as the neurons activation and it is calculated at each neuron in each hidden layer of the network, so each layer depends on the layer that came before. The activation can be found using Equation 2.1:

$$z = w_1 x_1 + w_2 x_2 + ... + w_n x_n + b = \sum_{1}^{n} w_i x_i + b = \boldsymbol{x}^T \boldsymbol{w} + b \tag{2.1}$$

Where x_i is the activation of the i^{th} neuron in the previous layer and w_i is the weight of the connection between the i^{th} neuron and the current neuron. However, as z is a weighted sum, it could take any value so each neuron has a bias term b that can be used to change the way each neuron is activated without affecting the inputs. Some applications may need the value of z to be within different constraints, so z is typically put into a function which condenses any number into some given range/discrete value. The following formulae in Equation 2.2 show a few examples of the most common activation functions along with the corresponding graphs in Figure 2.2:

$$\underset{\text{Linear Function}}{h(z) = z} \qquad \underset{\text{Sigmoid Function}}{h(z) = \frac{1}{1+e^{-z}}} \qquad \underset{\text{Softmax Function}}{h(z) = \tanh(z)} \qquad \underset{\text{ReLU Function}}{h(z) = \max(0, z)} \tag{2.2}$$

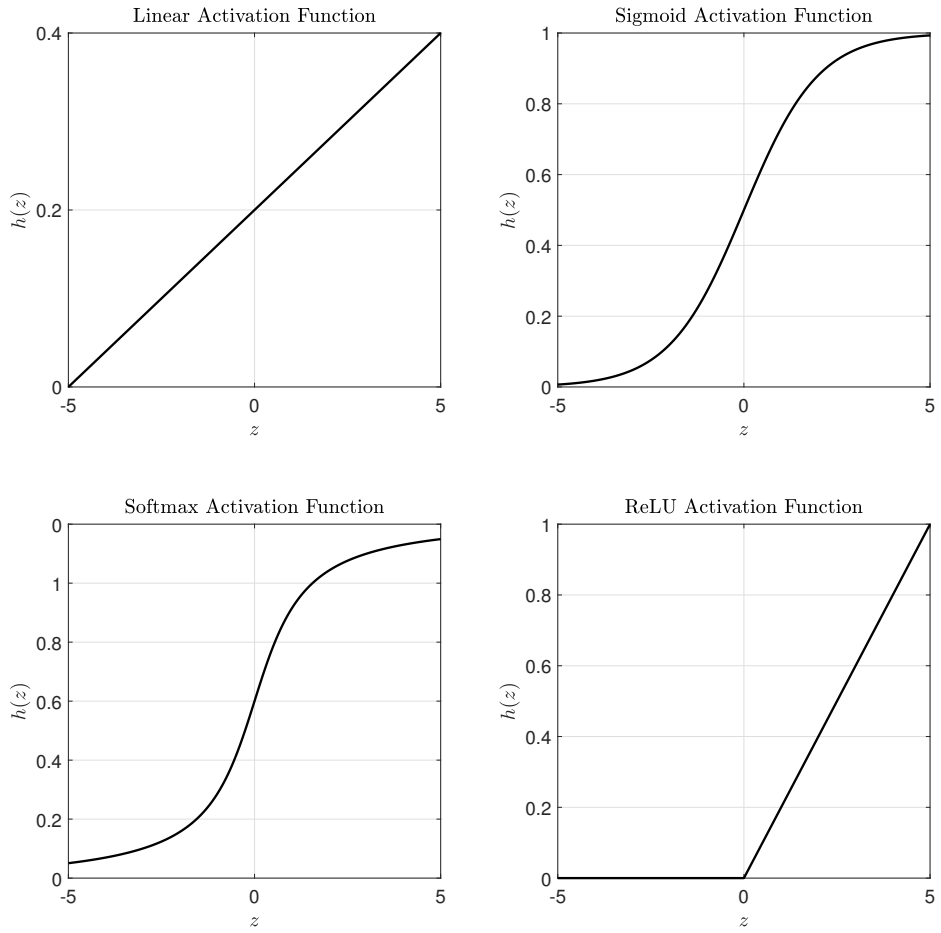


Figure 2.2: Graphs of commonly used activation functions.

Now that the activation has been calculated for each neuron, the model is close to being complete. The network is now filled with neurons that each represent some feature or pattern of features that the network believes the value/class depends upon. The input finally reaches the final layer, which is the output layer. The output layer greatly depends on the type of task that the model is designed for. For classification tasks, the output layer will contain the number of classes that the model can predict (other than for 2 classes, where there would be only one neuron) where each neuron is a probability that the input belongs to the class that the neuron represents.

The overall goal of the neural network is to learn the features that are within some data to correctly predict the value/class it belongs to. Hence, for the algorithm to "learn",

it must correctly tweak the weights w_i and bias terms b for each neuron to minimise a function. This function could take any shape, so it is important that the algorithm tries to find the global minimum of the function rather than get stuck thinking that it has found said minimum but in fact has found a local minimum. Figure 2.3 illustrates the importance of finding the global minimum verses the local minimum.

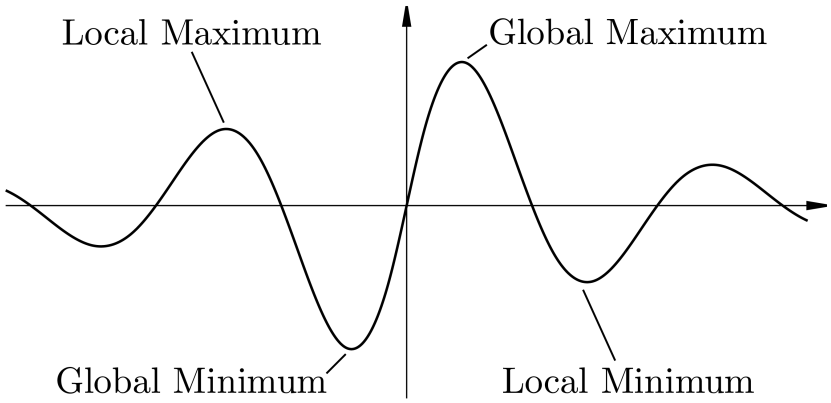


Figure 2.3: Local minimum and maximum vs global minimum and maximum [21].

Now, the function that needs to be minimised is the cost/loss function which gives a numerical value for how the model is performing [44]. The loss function also aids with punishing the model for incorrect judgement/classification. This function changes depending on the task that the model is solving (classification or regression) and so will be discussed mathematically in detail in the relevant chapter.

2.3 Backpropagation & Optimisers

When the model is being trained using the training set using an API such as Keras/Pytorch, each neurons activation is tweaked to best fit the dataset. Hence, the model must adapt the weights of each neuron after viewing how it performs. Until now, the network has always fed information forward through the network.

In practice, each piece of training data should affect the weights and bias for each neuron by increasing/decreasing by a relative amount. The technique of tweaking the values for individual neurons to minimise the loss function is called backpropagation. While it would be ideal for this to be done for every training example, it is typically the case that the average changes from a batch of training data is calculated and used to tweak the neurons [39].

The formulae for backpropagation to the cost function C_0 on a single training example (which is the reason for the 0 subscript) for the activation $h_k^{(L-1)}$ of the k^{th} neuron in the layer $(L-1)$ is in Equation 2.3:

$$\frac{\partial C_0}{\partial h_k^{(L-1)}} = \underbrace{\sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial h_k^{(L-1)}} \frac{\partial h_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial h_j^{(L)}}}_{\text{Sum over the entire layer } L} \tag{2.3}$$

This equation appears to be quite complicated but it is simply the chain rule applied that is summed over the the layer L. These terms mean the following:

- $\frac{\partial C_0}{\partial h_k^{(L-1)}}$ represents the ratio between two objects: a small change in the cost function ∂C_0 on a single training example and a small change in the activation $\partial h_j^{(L)}$ of the j^{th} neuron in the previous layer $(L-1)$.
- $\frac{\partial z_j^{(L)}}{\partial h_k^{(L-1)}}$ represents the effect small changes to the activation function $\partial h_k^{(L-1)}$ in the previous layer $(L-1)$ (k represents the index of the neuron in the $(L-1)^{\text{th}}$) layer) has on small adjustments to the $\partial z_j^{(L)}$ activation of the neuron in the current layer (L).
- $\frac{\partial h_j^{(L)}}{\partial z_j^{(L)}}$ represents the derivative of the activation function, $h_j^{(L)} = h\left(z_j^{(L)}\right)$, in the current layer on the j^{th} neuron which of course differs depending on the chosen function.
- $\frac{\partial C_0}{\partial h_j^{(L)}}$ represents the derivative of the cost function for one training example with respect the activation of the neuron in the L^{th} layer.

This showcases the sensitivity of the cost function compared to the activation function for a specific training example on a specific neuron [40]. To find the cost of the entire network, Equation 2.3 can be averaged across all training examples. It is important to mention that although the original paper for backpropagation [38] uses C to represent the cost function, this is the same as describing the loss function ℓ as the cost and loss function are the same thing. These equations like will take time to fully understand as there are so many variables.

This data is then used in the optimiser algorithms to change the neurons weights and biases as the gradient of the cost function is typically used to find the global minimum of the cost function itself. The following will describe each optimiser and how it works to minimise the loss function of the neural network.

Stochastic Gradient Descent

What was first an idea published in 1951 by Robbins and Monro [35] was later developed into the stochastic gradient descent (SGD) method known today by Kiefer and Wolfowitz in 1952 [23]. The method aims to minimise an objective function by making small, smooth iterations toward a minima. From the name of the algorithm, it is clear that the algorithm involves some sort of randomness (stochastic) and it does this to help with reducing the number of operations required to find a minima. Equation 2.4 shows the formal definition:

$$\text{SGD}(H_j, \eta_j): \quad \theta_{j+1} = \theta_j + \eta_j \nabla \ell(\theta_j) \tag{2.4}$$

Where $\ell : \mathbb{R}^m \to \mathbb{R}$ is the differentiable loss function, $\eta_j : \mathbb{N} \to (0, \infty)$ is the learning rate (hyperparameter) which affects how much the parameters of the model are changed. Also, $H_j = \{\theta_i, \nabla\ell(\theta_i), \ell(\theta_i)\}_{i=1}^{j}$ is the history of previous iterations which includes the gradient of the loss, alongside the loss function at a sequence of parameters θ_j [10].

SGD can often find a minima faster than other methods because of it's randomness, but this can also make it perform worse. For example, if the learning rate is too high the algorithm is more likely to fluctuate and could even jump to a different minima that is not the global minima. Additionally, each step contains the previous history, which can get grow in size and complexity quickly on large datasets [19].

Momentum

Over a decade later, Polyak would publish a paper [30] describing a generalised approach to SGD. This approach like the name suggests takes into account the momentum, which in this case would be the gradient, to find the minimum point more quickly. This method can be used alongside SGD to improve performance. Equation 2.5 gives the definition of the Momentum optimiser:

$$\begin{aligned} \text{Momentum}(H_j, \eta_j, \gamma): \quad & \theta_{j+1} = \theta_j - \eta_j v_{j+1} \\ & v_{j+1} = \gamma v_j + \nabla \ell(\theta_j) \\ & v_0 = 0 \end{aligned} \tag{2.5}$$

Where all previous variables/parameters from the SGD remain the same in the Momentum formulae, but Momentum introduces a new input parameter γ. This new parameter is known as the momentum parameter and is defined such that $\gamma \in \mathbb{R}^+$. Gradients can be noisy depending on the function, so γ can be tweaked to allow the optimiser to react to sudden jumps in the gradient more smoothly.

Using Momentum as an optimiser typically improves the convergence toward an optimal solution by reducing unnecessary oscillations. The disadvantage of using Momentum is that there is another hyperparameter that needs to be adjusted, which adds more time to the total amount spent tweaking the model to optimise accuracy and reduce overfitting.

Root Mean Squared Propagation

While there is no official paper that introduced this optimiser, Root Mean Squared Propagation (RMSProp) was pitched by Hinton in a lecture in 2012 of which the slides from the lecture were published for free online [17]. Just like Momentum was to SGD, RMSProp is a new improvement of Momentum. First, see Equation 2.6:

$$\begin{aligned} \text{RMSProp}(H_j, \eta_j, \gamma, \rho, \epsilon): \quad \theta_{j+1} &= \theta_j - m_{j+1} \\ m_{j+1} &= \gamma m_j + \frac{\eta_j}{\sqrt{v_{j+1} + \epsilon}} \nabla \ell(\theta_j) \\ v_{j+1} &= \rho v_j + (1-\rho) \nabla \ell(\theta_j)^2 \\ m_0 &= 0 \\ v_0 &= 1 \end{aligned} \tag{2.6}$$

Notice the addition of two new hyperparameter inputs, ρ and ϵ. ρ is a "discounting factor for the history/coming gradient" and ϵ is used to account for "numerical stability" in the gradient [50]. RMSProp differs from Momentum also by using an adaptive learning rate which tweaks itself based on the data the model is trained on. The learning rate is divided by the average squared previous gradients, where the initial value is recommended to be 0.9 [37].

RMSProp generally converges much faster than SGD even with Momentum meaning it is very powerful for most applications such as for larger datasets, whereas SGD falls apart quickly. Despite this, there is still the learning rate parameter which needs to be tweaked.

Adam

Just two years after RMSProp was shown in a lecture, Kingma and Ba would introduce their paper [24] which defined a new optimiser called Adam which boasted efficiency in both computation and memory load. The name Adam comes from the three words "adaptive moment estimation" and similar to RMSProp, Adam is based on SGD.

Equation 2.7 shows the equations for the Adam optimiser:

$$\begin{aligned}
\text{Adam}(H_j, \alpha_j, \beta_1, \beta_2, \epsilon): \quad \theta_{j+1} &= \theta_j - \alpha_j \frac{m_{j+1}}{\sqrt{v_{j+1}} + \epsilon} b_{j+1} \\
b_{j+1} &= \frac{\sqrt{1 - \beta_2^{j+1}}}{1 - \beta_1^{t+1}} \\
v_{j+1} &= \beta_2 v_j + (1 - \beta_2) \nabla \ell(\theta_j)^2 \\
m_{j+1} &= \beta_1 m_j + (1 - \beta_1) \nabla \ell(\theta_j) \\
v_0 &= 0 \\
m_0 &= 0
\end{aligned} \tag{2.7}$$

Here there are three different hyperparameters compared to previous methods. α_j is the learning rate instead of η from previous methods with α_0 is the initial learning rate and keep in mind that the value of ϵ is inversely proportional to α_j. There is also the β_1 and β_2 hyperparameters that represent the decay of the first and second momentum estimates respectively [46].

The advantages of using Adam include its efficient memory usage, meaning large datasets will not be a problem for this optimiser and in most cases outperforms SGD and Momentum [10]. In some cases, Adam struggles to find the global minimum and gets stuck on a local minimum if ϵ is not tweaked for optimal performance.

Nesterov-Accelerated Adam

Finally, in 2016, Dozat [14] would propose an improvement to the Adam that groups it together with Nesterov Momentum and let it be called Nesterov-Accelerated Adam (NAdam). Nesterov Momentum is another improvement to SGD and Momentum which takes into account the gradient of the function in-between steps to assure it is still on track to the optimal point [42]. The combined algorithms produce NAdams which Equation 2.8 shows:

$$\begin{aligned}
\text{NAdam}(H_j, \alpha_j, \beta_1, \beta_2, \epsilon): \quad \theta_{j+1} &= \theta_j - \alpha_j \frac{\beta_1 m_{j+1} + (1 - \beta_1) \nabla \ell(\theta_j)}{\sqrt{v_{j+1}} + \epsilon} b_{j+1} \\
b_{j+1} &= \frac{\sqrt{1 - \beta_2^{j+1}}}{1 - \beta_1^{t+1}} \\
v_{j+1} &= \beta_2 v_j + (1 - \beta_2) \nabla \ell(\theta_j)^2 \\
m_{j+1} &= \beta_1 m_j + (1 - \beta_1) \nabla \ell(\theta_j) \\
v_0 &= 0 \\
m_0 &= 0
\end{aligned} \tag{2.8}$$

All the hyperparameters are the same as those found in the Adams optimiser. The structure is also similar to that of Adam, with the only change being to the numerator of the fraction in the θ_{j+1} equation which represents the Nesterov modification.

Since this was made to improve Adam directly, it is typical that NAdam outperforms Adam for any case that Adam can be used for. Despite this, NAdam is not always the better optimiser to use and so it should be tested, especially advanced optimisers like RMSProp as they can often perform better if NAdam doesn't perform as expected.

2.4 CNN Architecture

Convolutional Neural Networks (CNN) is a type of neural network which focusses on 2-dimensional (2D) or 3-dimensional (3D) arrays of data [2]. Images are fundamentally 2D arrays of values, typically with 3 values per pixel (red, green and blue value) but they could have more or less (for example, black and white requires only one value and additional depth information requires more). Because of the volume of individual values each piece of data could take, pre-processing is even more important with images for use in CNNs. There are even more techniques that are used by CNNs to aid with extracting as much of the useful information from the dataset as possible, a big part being the general structure of a CNN verses that discussed previously of a regular NN.

The layers that make up a CNN are combined using a Sequential model in the Keras API which allows users to create their own "stack of layers" that feed into each other. The most common layers will be discussed in detail for future reference in the following subsections. It is important to mention that it is possible to use an existing CNN architecture in which the layers are pre-defined along with their respective hyperparameters, however it is typically best to create bespoke models for different applications to maximise accuracy.

Figure 2.4 showcases the convolutional layers in a CNN model, before it reaches 1D layers similar to those in regular neural networks:

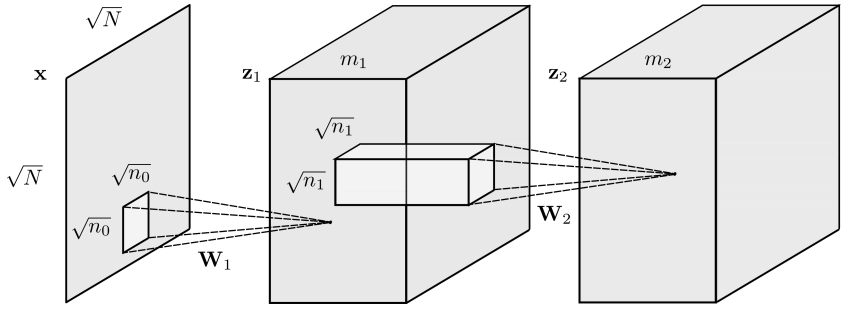


Figure 2.4: CNN layer architecture before the layers are flattened [34].

Convolutional Layer

Like the name suggests, a convolutional layer performs convolution to arrays between 1D and 3D. Since this paper will focus on images, only the 2D case will be discussed. This layer takes an input array (image) and takes a subsection grid of pixels and computes element-wise multiplication using a pre-defined kernel matrix of same size to give a feature value for that grid of pixels. This process is repeated for each subsection of pixels and the matrix of feature values is known as a feature map. Equation 2.9 and Figure 2.5 give a mathematical and visual representation of an element of the feature map:

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \quad \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases} \tag{2.9}$$

$$\begin{bmatrix} 1 & 2 & 5 & 2 & 1 \\ 0 & 6 & 4 & 4 & 1 \\ 0 & 3 & 4 & 3 & 2 \\ 4 & 2 & 2 & 6 & 3 \\ 1 & 0 & 4 & 5 & 3 \end{bmatrix} \bullet \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 29 & 46 & 37 & 20 \\ 24 & 39 & 36 & 23 \\ 26 & 25 & 40 & 36 \\ 11 & 22 & 46 & 39 \end{bmatrix}$$

Figure 2.5: Example of a convolution layer acting on an (5×5) input array (left) with a (2×2) kernel (center) and output (4×4) feature map (right) using a (1×1) stride.

where:

- $z_{i,j,k}$ is the output feature value in position (i, j) in feature map k of the l^{th} convolution layer.
- s_h and s_w are strides in the horizontal and vertical direction respectfully. f_h and f_w are the height and width of the kernel and s_h is the number of feature maps in the previous layer, $(l-1)$.
- $x_{i',j',k'}$ is the output of the (i', j') neuron in the previous layer, $(l-1)$.
- b_k is the bias term related to the feature map k in layer l.
- $w_{u,v,k',k}$ is the weight between neurons in feature map k (layer l) and its input row in location (u, v) and feature map k' [16].

Notice that when the stride is greater than 1, there is potential that the selection of neurons may stretch to outside the input layer. There are a number of techniques that account for this issue, which boil down to whether to add padding or not. Padding increases the size of the input layer on the outer edges and assigns the extra neurons a value depending on the type of padding used including same, casual and valid padding [53]:

- Same Padding: Sets value of outside edges to 0.
- Casual Padding: Used for 1D convolution layers, so will be ignored.
- Valid Padding: No padding

Now that the fundamental concepts behind the convolution layer have been explained, Equation 2.10 shows the code to add a 2D convolution layer to a sequential model, including the most useful hyperparameters set to their respective default values:

```
Conv2D(filters, kernel_size, strides = (1, 1), padding = 'valid',
                          activation = None, use_bias = True)   (2.10)
```

Where the hyperparameters represent the following [47]:

- `filters`: A value that represents the total output filters.
- `kernel_size`: The shape of the kernel, represented as $(n \times m)$.
- `strides`: The amount of stride both horizontally and vertically.
- `padding`: Type of padding (same or valid).
- `activation`: State an activation function (if required) to apply to the layer.

- `use_bias`: Like the name suggests, tell Keras whether to include a bias vector.

It is clear that convolution layers are great for reducing the quantity of data in a model, while not reducing the quality by ensuring that promising features remain in future layers. This layer is best used at the start of the model to reduce the number of parameters in the future.

Pooling Layer

Similarly to a convolution layer, a pooling layer takes a group of values from an input and outputs a single value. Again, with images, only 2D arrays will be considered but this method works for data in 1 to 3-dimensions. Rather than using a kernel that affects the value in which the model will compute against a the group of input values, pooling simply downscales the input layer. This means less hyperparameters will be needed (as will be seen later) and so the layer should be easier to implement without much tinkering. There two different versions of pooling, each with a normal and global version.

The mathematics behind this layer is trivial. For max pooling, select the maximum value within the window of values and use that as the output, repeat this for all windows by moving along the input by the stride amount. This is the same for average pooling, but instead of taking the maximum value, use the average of all the values within the window. Figure 2.6(a)/(b) show the visual representation of max and average pooling respectively:

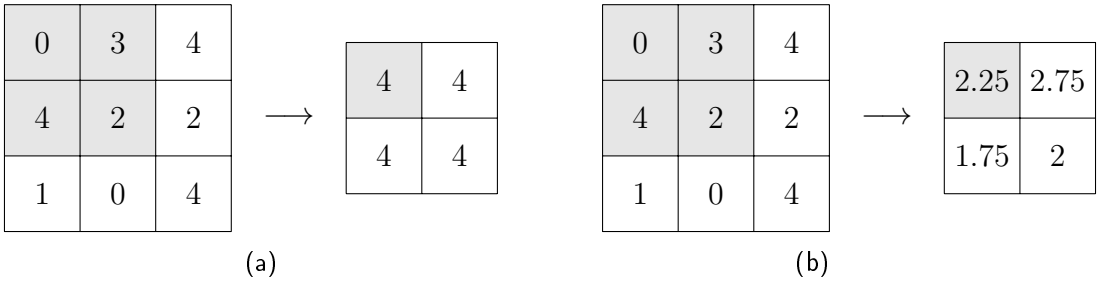


Figure 2.6: (a) shows max pooling on a (3×3) input with a (2×2) pool size and (1×1) stride resulting in a (2×2) output. (b) shows average pooling on a (3×3) input with a (2×2) pool size and (1×1) stride resulting in a (2×2) output.

When using the global versions of both max and average pooling, there is no need to state the pool size or stride. This is because global pooling finds the max/average of the entire input array and outputs said value, meaning the output array will be one value. This can be useful toward the end of the model, as reducing large 2D arrays

down to a single value will obviously remove most of the useful features within the array.

The fundamental mathematics behind the layer have now been described. Equation 2.11/2.12 shows the code to add a 2D max/average pooling layer to a sequential model respectively, including the most useful hyperparameters set to their respective default values:

```
MaxPooling2D(pool_size = (2, 2), strides = None,
                                        padding = 'valid')  (2.11)
AveragePooling2D(pool_size = (2, 2), strides = None,
                                        padding = 'valid')  (2.12)
```

Where the hyperparameters represent the following:

- `pool_size`: The shape of the group of pixels to be pooled together, represented as $(n \times m)$.
- `strides`: The amount of stride both horizontally and vertically.
- `padding`: Type of padding (same or valid).

Pooling layers perform well when used after a convolution layer to further reduce the size of the array that is being transferred through the rest of the layers in the sequential model. It also requires less hyperparameters than convolution, but of course offers less control.

Activation Layer

An activation layer is a fully connected layer, meaning each neuron in the input maps to a neuron in the output. This means that the layer does not change the quantity of information, but rather it seeks to amplify promising features that may contribute to the output label and reduce features that are less meaningful. Activation can be used as its own layer like will be shown later, or it can be appended to other layers like what was seen for convolution in Equation 2.10. Traditionally, activation is used on neurons in a regular NN, but in CNNs this layer can be used wherever necessary. The equations and graphs for a selection of activation functions can be seen previously in Equation 2.2 and Figure 2.2.

As the mathematics have already been discussed earlier in this chapter, the code can be introduced and explained now. Equation 2.13 shows the code to add a separate

activation layer, but note again that this is not always necessary if previous layers can include an activation layer using a hyperparameter:

```
Activation('relu'))
```
(2.13)

There is only one hyperparameter, which is simply the activation function that is needed. This can of course be changed for other functions stated previously. Activation should be used regularly in a sequential model to ensure prominent features are more likely to be passed forward though future layers.

Flatten Layer

Flatten is one of the most simple layers in a CNN as it simply reshapes an input, typically 2 or 3-dimension inputs and turns it into a 1D vector. For 2-dimensions, the input shape $(n \times m)$ is transformed into $(1 \times nm)$. A basic example of this can be seen in Figure 2.7:

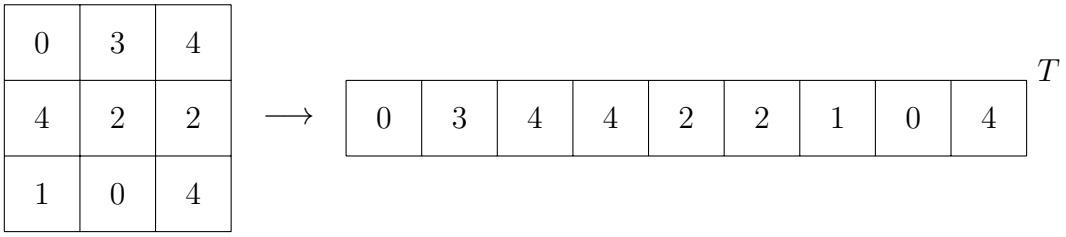


Figure 2.7: shows a flatten layer applied to a (3×3) input on the left which transforms it into the (1×9) output on the right (after transpose).

As there are no configurable hyperparameters, Equation 2.14 shows the basic code to add a flatten layer to a sequential model:

```
Flatten()
```
(2.14)

Also, it is important to note that it is typical to only find one flatten layer in a model, as the layer is usually found later on in the model for final manipulation and there is no reason to undo this action.

Dense Layer

Next, a dense layer is used after passing data through the flatten layer and is the most common layer used in a CNN. Fundamentally, a dense layer is a fully connected layer that has less neurons than the previous layer. The terms "fully connected" and "dense" can be used interchangeably.

This layer works similar to hidden layers found in regular neural networks in that it uses the same weighted sum formula [5] seen in before in Equation 2.1. As the layer is fully connected, it is largely used as the final layer such that it outputs a value for each class in multi-class classification or a single value for regression/binary classification but dense layers are not exclusively final layers. With this in mind, Equation 2.15 shows the code for adding a dense layer to a sequential model including the most useful hyperparameters:

```
Dense(units, activation = None, use_bias = True)
```
(2.15)

Where the new hyperparameter, `units`, is the number of elements in the output. If this layer is the final layer, this hyperparameter should be set to the number of possible classes or set to 1 for binary classification/regression. Of course, the dense layer can also have activation applied to it but by default it does not. Dropout is often required for most models to combine promising features and so is well established as a go-to for reducing the feature map.

Dropout

Dropout layers are used to counter overfitting by randomly removing neurons. Without this layer, the model is likely to select incorrect features that decide the label. There is no mathematics behind the removal of neurons other than randomness, which could mean that some neurons that are removed could actually be useful. See Equation 2.16 for the code and hyperparameters for dropout:

```
Dropout(rate, seed = None)
```
(2.16)

Where the hyperparameters mean the following:

- `rate`: Value between (0,1) which is the fraction of neurons that will be removed [48].
- `seed`: Define the random seed to ensure results can be reproducible.

Hence, there is only one hyperparameter that needs to be tweaked so when combating overfitting makes it relatively simple to reduce. Typically, only one dropout layer is required.

Using Pre-Trained Models

Alternatively, pre-trained models can be used instead of a custom sequential model. This will of course save time and gives another option for a solution to a problem. There are two ways of using a pre-trained model:

- Re-use the entire model that was trained on a different dataset, without changing anything.
- Only re-use the "top half" of the model, where the neurons are finding more general features. Then append layers that are also used in sequential models to create a custom pre-trained model.

With this knowledge, custom models can be created and can be trained using a dataset as will be seen in upcoming chapters since driving is such a dynamic environment and using pre-trained model helps with filling in the gaps between the dataset and reality [29] provided the task is not too different to the dataset used for the original dataset.

2.5 Training the CNN

Now that the main layers used for CNNs have been explained, layers can be combined using a sequential model. Once the sequential model layers have been listed, the API needs to combine them together. This can be done using the following command and its corresponding default hyperparameters within the argument of Equation 2.17:

```
model.compile(optimizer = 'rmsprop', loss = None,
                                              metrics = None)  (2.17)
```

Where each hyperparameter corresponds to the following:

- `optimizer`: This gives the user control of which optimiser is used out the potential methods discussed in 2.3. By default, it used root mean squared propagation but this should be changed and tested to decide whether it is best to change from RMSProp to another method.
- `loss`: Specifies the loss function to be minimised (required to be changed from `None`) The functions that are used here will be detailed further in the respective chapters for each model.
- `metrics`: This allows the user to configure the metrics displayed once the model has started training. It is typically changed from `None` to `['accuracy']` which displays important statistics like the accuracy of both the training and validation data, along with the value of the loss function which is trying to be minimised.

It is now time to train it using training data. It will become obvious that having more training data will help improve the accuracy of the model in a wider range of conditions (depending on the range of training data). The amount of data available will vary depending on the dataset used, but it is common to assign anywhere from

10%-20% of the training data for validation. Validation data is unseen by the model (similar to testing data) but is mainly used as a statistic for how well a model performs using unseen data [8]. As previously mentioned, when the model is compiled using the accuracy metric, information about the validation data accuracy will be given which gives a good indication as to whether or not the model is overfitting on the training data. Once the model has been tweaked to get within the required accuracy, the test dataset should be passed through the model to once again check how well the model performs however once the test set has been used, no further tweaks should be made. This said test set should have enough data such that the accuracy result should roughly stay the same if more data was added.

Now that the data has been separated into training, validation and test sets, it is time to fit the model to the training data. Equation 2.18 shows the command to do this and the hyperparameters that can/must be changed:

```
model.fit(x = None, y = None, batch_size = None, epochs = 1,
            validation_split = 0.0, validation_data = None)  (2.18)
```

Where each hyperparameter means the following:

- `x`: State the pre-processed training data.
- `y`: State the corresponding labels for the training data.
- `batch_size`: Define the number of data samples that the model works with before changing the parameters within the model [7].
- `epochs`: State the number of times that the model use the training data (in batches of size defined in the previous hyperparameter).
- `validation_split`: State the percentage of data to be used for validation (if the validation dataset has not been created).
- `validation_data`: State the validation dataset (if the validation dataset has been created).

2.6 Evaluation

Evaluation of models will be discussed on a more granular level in the each models chapter. The main problem with evaluating CNNs is that is incredibly difficult to inspect the hidden layers within a CNN (convolutional, max pooling, dense layers etc). Knowing whether the layers within a network are working as intended would be a great

way of understanding any issues that are occurring when finding predictions. This is where the Gradient-weighted Class Activation Mapping (Grad-CAM) algorithm can help solve this explainability problem.

As the name suggests, Grad-CAM can only be used on classification models and uses the last convolutional layer to create a coarse feature map which visualises the important areas of an input image that relate to the classification [43]. For example, Figure 2.8 shows a generated heatmap that used Grad-CAM on a binary classification model which predicts the species of elephant in an image. Without the heatmap, it would be impossible to truly understand how the model selects features for identifying the species of elephant. However the heatmap shows that the model is focussing on the ears/facial features of the elephant to make the prediction that the elephant in Figure 2.8 is an African elephant. This interpretability was simply impossible before Grad-CAM was available.

Figure 2.8: Grad-CAM heatmap on a binary classification model predicting the species of elephant featured in an image [11].

Since Grad-CAM uses the last convolutional layer to ensure it showcases only the most promising features (after filtering in previous layers), models with more convolutional layers will output a more useful heatmap. Less convolutional layers will produce a heatmap with larger hotspots due to the network forming less connections between features whereas a suitable number of convolutional layers will make the useful features obvious in the heatmap.

Chapter 3

Binary & Multi-Class Classification

When starting a machine learning project, it is important to have an idea of the steps that need to be followed to stay on track. This is done so that the end goal of a functioning model is never out of sight, as there are many steps that require more time and effort than others. In basic terms, the following list gives a structure for how the machine learning problem will be structured and solved:

1. Describe the problem that needs to be solved.
2. Gather dataset(s).
3. Prepare the dataset for use in a machine learning model.
4. Create and tweak the model appropriate for the task.
5. Evaluate the performance of the model on new data.

3.1 Describing the Problem

While driving, it is trivial for humans to recognise objects like cars, traffic signs and traffic lights. These objects are just a subset of the most important objects to pay attention to while driving in order to be safe. For example, knowing if a car is in front of a autonomous vehicle would be a great indicator for whether or not it is safe to speed up, slow down or maintain the current speed. A similar response could be used if the autonomous vehicle is approaching a traffic sign or traffic light, but this would require a dataset including labels for the type of traffic sign (such as stop signs, give way signs or one-way signs).

Another use for a model similar to this is for more general advanced driver-assistance systems (ADAS) in normal, non-autonomous vehicles. Systems such as adaptive cruise control use sensors including radar or LiDAR to constantly check for objects in front of the vehicle, but this could be simplified such that the radar only switches on when a camera sensor sees if there is in fact an object in front of the vehicle. This has the possibility of improving the efficiency of the vehicle, especially as cars transition to electric drivetrains and batteries where the range of the vehicle needs to be maximised. No human driver is perfect, so systems that alert/correct the driver if it senses something wrong are key to reducing the number of accidents on the road as we slowly transition to a world of autonomous vehicles.

There will be two models created in this section: a binary classifier and a multi-class classifier with 2 and 3 classes to predict respectively. Binary classification, is one of the most basic models to create as there is only one output layer, so an accuracy of at least 90% would be ideal. As previously stated, knowing if there is a car in front of the autonomous vehicle is fundamental, so a high accuracy must be found and in a model used on public roads it must be much higher. The multi-class model will add an additional class for both traffic lights and signs.

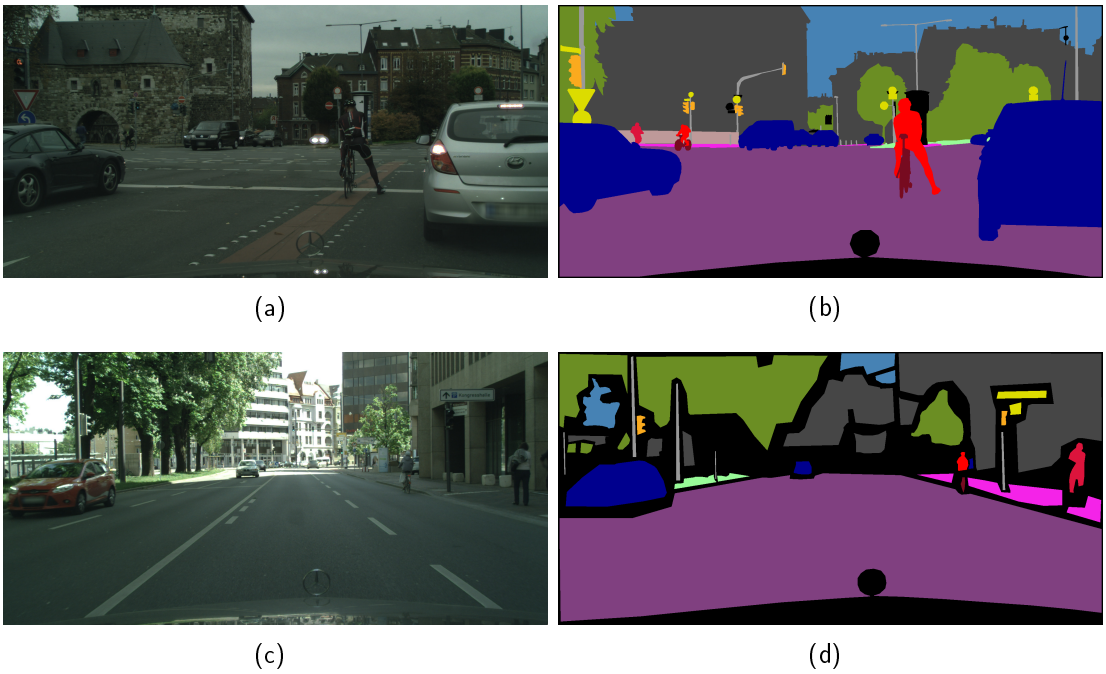
Like most machine learning classification problems, the problem seems trivial for humans, but using a more traditional coding approach would be impossible as there are too many variations of images that a autonomous car could possibly see. It is much easier to use a machine learning approach such as a neural network which closely mimics how human brains recognise patterns using neurons in a network as the name suggests.

3.2 The Dataset

A good dataset is one that is has a large enough size, good quality labels and is reliable [1]. Cars, signs and weather conditions can be different all across the world. For example, a dataset that took images in North America will likely contain different traffic signs and lights to those found in Europe, or in Australia. Because of this, training an algorithm for each part of the world would be more efficient.

The Cityscapes Dataset [13] features a good range of labelled objects that it groups into categories including vehicles (such as cars, trucks and buses), objects (including traffic signs and traffic signs) and a selection of other groups (buildings, vegetation and sky). Each image in the dataset is labelled using semantic segmentation, which is a form of clustering all of the pixels in an image and giving each cluster a label [52]. Figure 3.1(a) shows an example of an image from the training dataset that will be

used in the model created in this chapter. The labels for this image can be seen in Figure 3.1(b) where each colour represents a label like the objects listed earlier. This dataset featuring "fine" annotations (more accurate) includes 2975 training images, 500 validation images and 1525 test images. Cityscapes also offers an additional 19998 training images that feature "coarse" annotations, meaning the clusters of pixels are less precise and more general. An example of this is in some images with lots of cars, the annotation will be one group of pixels with the label "cargroup". For classification, the quality of annotations does not affect how the model performs because as long as the annotations are accurate enough (for example, if a car is in the image, then there is an annotation for it and the preciseness of the location does not matter). Figure 3.1(c) and (d) show another comparison of the raw image sent from the dashcam of the car and the coarse labels respectfully. Clearly, there are more gaps in the coarse annotations which are represented by the black colour. The areas in black were decided to not be included likely because there is nothing important in that part of the raw image. This is a more efficient way of labelling but of course sacrifices accuracy, as labelling takes a great amount of time and precision to ensure each pixel is correctly assigned to a label.



(a) (b)

(c) (d)

Figure 3.1: (a), (c) shows un-processed image from Cityscapes dataset training dataset and (b), (d) shows the corresponding semantic segmentation labels.

The Cityscapes dataset like the name suggests is a dataset full of images of a driver driving around various German cities in decent weather conditions. Being in a city, there are more likely to be more infrequent objects such as tall buildings and bike riders that are less likely to be seen in images taken from a motorway for example. It is likely that the model created in this chapter will over-fit to the city, meaning it will be less accurate in situations where there are for example no buildings on either side of the frame or no other objects visible at all. The same can be said for the weather conditions, if an image is given to the model where visibility is poor, there is snow on the ground or rain hitting the camera then it is less likely to correctly classify the image.

As previously mentioned, there are approximately 25,000 8-bit colour images that each have a resolution of 2048×1024. This means that the images alone take up around 55GB of storage and the semantic segmentation labels use around 2.5GB of space. This amount of data is manageable on a high end PC so it can be stored locally on the machine creating the model. Of course, when dealing with images taken from public areas like roads, care must be taken for privacy of those features within images. Unfortunately, there is no current way of training a model on a dataset with privacy features such as blurred faces or number plates without heavily sacrificing the accuracy. This is because in the real world, peoples faces/number plates will obviously not be blurred, so training a model using blurred parts will make the model think that all pedestrians/vehicles have blurred areas which is not true and could lead to inaccurate classification. Hence, when training the model, the dataset without blurred faces/number plates will be used. However in this paper, any images shown for visualisation will be blurred either manually or by using the blurred dataset provided by the Cityscapes team themselves. On the Cityscapes website, it is stated that "This dataset is for non-commercial use only. However, if you find yourself or your personal belongings in the data, please contact us, and we will immediately remove the respective images from our servers." [13] so privacy has been heavily considered when creating this dataset. To avoid possible privacy issues, it is important to use the most up-to-date version of the dataset.

For the binary classification model, the data must be filtered into images that contain a car or not and for a multi-class classification model, a similar filter would sort the dataset into images with cars, traffic signs or neither. A way of doing this is to iterate over each label in the training set and check for whether there is a cluster for each label respectively. Using this approach will disregard the quality of the annotations (fine or coarse) so both types of training data can be used. This process will be used in the next section.

3.3 Pre-Processing the Dataset

Now that the dataset has been chosen for this task, it can be imported into a Jupyter notebook and be manipulated ready for use in a CNN. The Jupyter notebook for this chapter can be found in Appendix A (binary) and Appendix B (multi-class). As mentioned before, the data uses semantic segmentation so label each image, so first it must be decided which labels will be used for the classification task.

- For binary classification, all types vehicles must be included in the "1" class (there is a vehicle in the image). If there is no label for any type of vehicle in the image, then the label must be "0".

- For multi-class classification, if there is a car but not a sign, then append it to the "1" class. If there is a traffic sign but not a car, then add it to the "2" class. If there are no cars or traffic signs then assign the image to the "0" class. Otherwise, class the image as "3" and remove these images from the dataset later.

The reason that there cannot be images with a mixture of cars and signs in the image at once is because the model will only assign one class per image. Hence, if there are two of the objects that the model is trained to see then it will only assign the image to one of them objects which is incorrect.

By giving each image a new label, it is important to make sure that there is a equal balance of each class before training the model on the dataset. For example in the dataset for binary classification, approximately 90% of the images contain a vehicle. This means that if the model was trained on this dataset without making the number of images in each class equal, the model could theoretically predict all images in the test set to be under the "1" class and it would be 90% accurate. This of course is not true if an equal dataset was to be predicted using the model, so pre-processing is needed to ensure the classes have roughly the same amount of images for each class.

Since the dataset will have a large amount of its images removed for weighting purposes, it would be best if the training (fine annotations), extra training (coarse annotations) and validation sets were to be combined to increase the size of the dataset, then make a validation set later on when it comes to training the model. Now that the dataset is significantly smaller than before it would be ideal for it to be increased in size so that the model has enough data to be trained correctly. As discussed in Section 2.1, a simple way of doing this is by making adjustments to each image and use that image as a new image in the dataset. The technique used to expand the dataset in this chapter will be flipping the images in the x axis, as this will allow all the labels to be kept the same which saves a lot of time without creating data that would

be impossible in the real world (for example, flipping the image upside down). By flipping all the images in the binary classification set, the dataset has doubled in size. Unfortunately, due to the weights of each class for the multi-class data, only the "0" class will be flipped since the data would not be equally distributed between each class.

After the pre-processing steps that have been completed thus far, the dataset for each task has the following amounts of images shown in Table 3.1 and Table 3.2 for binary and multi-class classification respectfully:

Binary Classification	Number of Images					
	Before Pre-Processing				After Pre-Processing	
Dataset	Training	Extra Training	Validation	Test	Training	Test
"0" Class	128	2608	18	1525	4989	519
"1" Class	2847	17390	482	0	5027	481
Total Images	2975	19998	500	1525	10016	1000

Table 3.1: Number of images for each class before and after pre-processing has completed for the binary classification task.

Multi-Class Classification	Number of Images					
	Before Pre-Processing				After Pre-Processing	
Dataset	Training	Extra Training	Validation	Test	Training	Test
"0" Class	9	651	3	1525	1161	165
"1" Class	121	3367	19	0	1153	173
"2" Class	119	1957	15	0	1164	162
Not Applicable	2726	14023	463	0	0	0
Total Images	2975	19998	500	1525	3478	500

Table 3.2: Number of images for each class before and after pre-processing has completed for the multi-class classification task.

Each class now has an equal number of images, other than for the test dataset. This may seem odd at first, but this occurs due to the creators of the Cityscapes dataset redacting the labels for the test dataset and requiring use of their test server to receive evaluation data. As the dataset is not designed for the task that is being completed in this chapter, using the test server will not yield any useful information and so it will not be used. Instead, the test dataset will be ignored as labelling over 3000 images will be extremely time consuming. Therefore, the combined training and validation set will be split by removing 1000 images for binary classification, and 500 for the multi-class

classification to be used for testing and evaluation later.

All images within each dataset will be scaled down to reduce the complexity. Binary classification will be scaled to 132×132 and multi-class classification will be scaled to 196×196. Multi-class will use a high resolution to ensure that the model has the best chance of picking up on small signs that may be in the image. This will increase the size of the model, but since the model has less training data this is a suitable trade-off.

Now that the data has been pre-processed, the images can be converted into Numpy arrays and randomised to prevent bias and stop the model from taking advantage of patterns of consecutive classes in the dataset [18]. Finally, the image must be normalised. Normalising an image is transforming the values of each RGB pixel from the range $(0, 255)$ to $(0, 1)$ by simply dividing each value by 255. The model is now ready to be constructed using layers and trained on the training dataset.

3.4 Building & Compiling the Model

Now that the training data is ready to be used, it is time to create a model to use the data on. This step of creating a solution involves the most trial and error in terms of tweaking the hyperparameters of each layer. Both binary classification and multi-class classification share the same layers but differ when the models are compiled.

Table 3.4 below shows the final model that was chosen to produce the highest accuracy and with the lowest overfitting. These models are the product of many changes to a base model featured in the code in Appendix A and B.

Binary Classification			Multi-Class Classification		
Layer	Output Shape	Number of Parameters	Layer	Output Shape	Number of Parameters
Conv2D(64, (5, 5))	(None, 128, 128, 64)	4,864	Conv2D(64, (5, 5))	(None, 192, 192, 64)	4,864
Activation('relu')	(None, 128, 128, 64)	0	Activation('relu')	(None, 192, 192, 64)	0
MaxPooling2D(pool_size = (5, 5))	(None, 25, 25, 64)	0	MaxPooling2D(pool_size = (5, 5))	(None, 38, 38, 64)	0
Conv2D(128, (5, 5))	(None, 21, 21, 128)	204,928	Conv2D(128, (5, 5))	(None, 34, 34, 128)	204,928
Activation('relu')	(None, 21, 21, 128)	0	Activation('relu')	(None, 34, 34, 128)	0
MaxPooling2D(pool_size = (2, 2))	(None, 10, 10, 128)	0	MaxPooling2D(pool_size = (2, 2))	(None, 17, 17, 128)	0
Conv2D(256, (3, 3))	(None, 8, 8, 256)	295,168	Conv2D(256, (3, 3))	(None, 15, 15, 256)	295,168
Activation('relu')	(None, 8, 8, 256)	0	Activation('relu')	(None, 15, 15, 256)	0
MaxPooling2D(pool_size = (2, 2))	(None, 4, 4, 256)	0	MaxPooling2D(pool_size = (2, 2))	(None, 7, 7, 256)	0
Flatten()	(None, 4096)	0	Flatten()	(None, 12544)	0
Dense(1024)	(None, 1024)	4,195,328	Dense(1024)	(None, 1024)	12,846,080
Activation('relu')	(None, 1024)	0	Activation('relu')	(None, 1024)	0
Dense(64)	(None, 64)	65,600	Dense(64)	(None, 64)	65,600
Activation('relu')	(None, 64)	0	Activation('relu')	(None, 64)	0
Dropout(0.5)	(None, 64)	0	Dropout(0.5)	(None, 64)	0
Dense(1, activation = 'sigmoid')	(None, 1)	65	Dense(3, activation = 'sigmoid')	(None, 3)	195
Total Trainable Parameters		4,765,953	Total Trainable Parameters		13,416,835

Table 3.3: Summary of both binary and multi-class classification models featuring the layers and parameters for each.

The initial model only featured two groups of 2D convolution, activation and max pooling layers which was found to not select the appropriate number features of each image to correctly identify the labels so more were added to solve this. After this, the model was still experiencing overfitting whereby the accuracy on the training data was far greater than the accuracy on the validation (unseen) data. This is likely due to the extremely large number of parameters involved in the model, especially in the Dense(1024) layer found in both models. To prevent overfitting, simply adding a dropout layer toward the end to eliminate potentially meaningless features reduced the accuracy on the training dataset, but made it similar to the accuracy of both validation and test datasets. Overall, the shape of the output for each model progressively decreases in a way where important features are unlikely to be lost which is ideal. However, the multi-class model has a large issue that will be discussed in Section 3.5 that is the reason why the accuracy of both models is so drastically different. Continuing with the layer discussion, it may be noticed that the activation function used each time was ReLU. ReLU was used as the model struggled to improve its accuracy over 0.5 for binary and 0.33 for multi-class classification. These accuracy numbers represent the same value as if the model were to guess randomly which class an image belonged to or alternately guessed the same label for every image which of course is incorrect.

Both models were compiled using different optimisers due to the structure of the labels for each. First, the code in Equation 3.1 is used to compile the binary classification model:

```
model.compile(optimizer = 'SGD', loss = 'binary_crossentropy',
                                        metrics = ['accuracy']))  (3.1)
```

The hyperparameter that requires further explanation is the loss function that was

chosen. Binary cross-entropy is a loss function which, as the name suggests, is only used for binary classification tasks. The loss function is the function that is to be minimised by the network, so it is equally important to punish the network for incorrectly classifying an image. Equation 3.2 shows binary cross-entropy loss function that is required to be minimised where $y_i \in \{0, 1\}$ is the given label and $p_i = 0$ is the model predicted value.

$$\ell_i(y_i, p_i) = -\left(y_i \log(p_i) + (1 - y_i) \log(1 - p_i)\right) \tag{3.2}$$

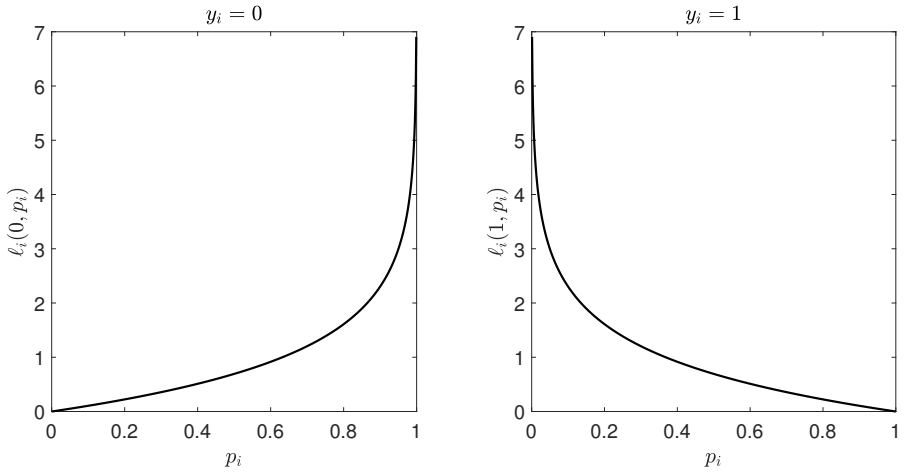


Figure 3.2: Two graphs showing the binary cross-entropy loss function with the left predicting label $y_i = 0$ and the right predicting label $y_i = 1$.

Consider an image that is passed through the model and this images has a true label that belongs to the "0" class. This means that $y_i = 0$ and presents the function on the left in Figure 3.2. It is then up to the model to decide what class the image belongs to by outputting a probability between 0 and 1 of it being in the "1" class. As the loss function is being minimised, it is clear that if the predicted value p_i is below 0.5 then the function is close to the global minimum of the loss function. However, if it incorrectly classes the image being within the "1" class, the loss function blows up as $p_i \to 1$ and so the model is severely punished for incorrect predictions. The same can be said for the opposite case when an image with given label of $y_i = 1$ is given to the model for prediction, as the formula will cancel the other logarithm term that was used previously.

For multi-class, the code in Equation 3.3 is used to compile the model:

```
model.compile(optimizer = 'SGD', loss =
  'sparse_categorical_crossentropy', metrics = ['accuracy']))
```
(3.3)

A similar technique for the loss function is used for multi-class classification, which instead uses a "one-hot" method that makes other labels $y_i = 0$. Essentially, it turns more than two classes into binary data by setting the actual class to "1" and the rest to "0" [54]. Sparse categorical cross-entropy can be seen in Equation 3.4 and the plot works similar to that in Figure 3.2, where N represents the number of classes ($N = 3$ for multi-class model).

$$\ell_i(y_i, p_i) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i) \tag{3.4}$$

Both models use stochastic gradient descent as the optimiser as it was found to work best on both datasets. Further information on SGD can be found in Section 2.3.

Fitting the models to the training dataset is done using the following commands in Equation 3.5 and 3.6 for binary and multi-class models respectively:

```
model.fit(X_train_norm, y_train, batch_size = 16, epochs = 20,
                                        validation_split = 0.2)
```
(3.5)

```
model.fit(X_train_norm, y_train, batch_size = 32, epochs = 20,
                                        validation_split = 0.2)
```
(3.6)

Most of the hyperparameters in this code are dependent on the system that the model is trained on. In this example, a computer with a GPU including 10GB of RAM (random-access memory) so the entire dataset cannot be used at once without filling the memory up. The values for both batch size and epochs are affected by system specification, however these numbers were found to not get stuck on a local minimum which is ideal. The validation data is also created in this code, where it takes 20% of both of the training datasets which for the binary model is around 2000 images and for multi-class was around 700 images.

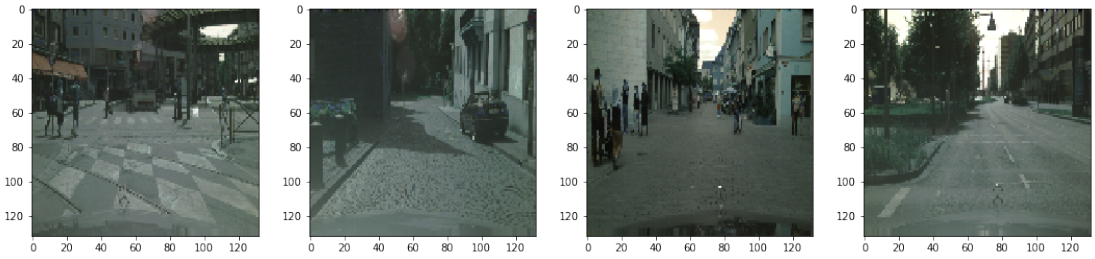
3.5 Evaluation

Now that the model has been created, compiled and fit to the training dataset, the accuracy of the model can be discussed. The results for both models will be explained separately Table 3.4 presents the results of the binary classification model:

Dataset	Accuracy
Training	90.55%
Validation	89.12%
Test	88.10%

Table 3.4: Accuracy results of binary classification model on each dataset.

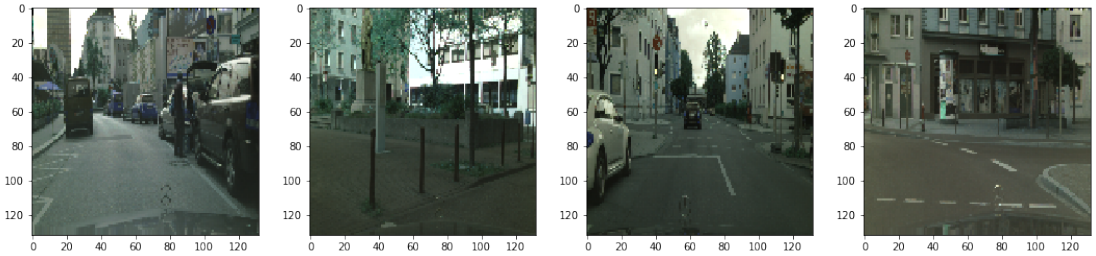
These accuracies, while not perfect of course, show that 9 out of 10 images will be classified correctly. This result is not fit for an autonomous vehicle, but more training data will only increase the accuracy. It is also important to note that the images in the dataset were all taken in main cities, so the results will likely not be the same on highways or country roads. To better understand the potential problems with the model, Figure 3.3 show some examples of images that have been incorrectly predicted by the model:



(a) (b) (c) (d)

Figure 3.3: Examples from test dataset of incorrect labels, where (a) predicted "1", (b) predicted "0", (c) predicted "1", (d) predicted "0".

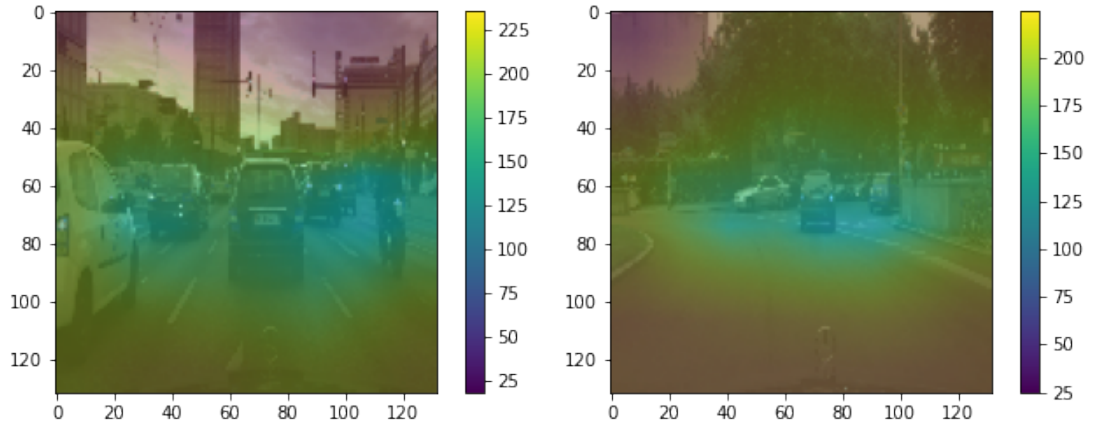
It is difficult to cipher which features a model uses to make a prediction, but guesses can be made that make logical sense. For example, in Figure 3.3(a) the model is likely mistaking the stall in the centre of the frame for a vehicle, as it does look very similar to the shape of a rear of a truck. For (b), it is unusual that it incorrectly labelled the image as the car is quite obviously there, a possible reason for the label is that the car could look similar to a group of objects like bins or poles, however this is still a bad label. (c) shows many pedestrians, but perhaps the planters outside of the shops in this street are throwing the model off track, as cars are typically this colour and shape (only larger). Lastly, (d) on first inspection looks like a correct label but there is actually a vehicle far in the distance that due to the pre-processing of the image (scaling down) is small enough to be invisible to the model. There are of course more images that the model has incorrectly labelled which can be seen viewed using the code, but Figure 3.4 shows some a small subset of the 90% of correctly labelled images:



(a) (b) (c) (d)

Figure 3.4: Examples from test dataset of correct labels, where (a) predicted "1", (b) predicted "0", (c) predicted "1", (d) predicted "0".

Finally for binary classification, there exists an algorithm called Grad-CAM as discussed in Section 2.6 which seeks to showcase what features a model is looking at in the form of a heatmap. It does this by using the gradients of the final convolutional layer which in this chapter is labelled the "visual_layer" in the model. Figure 3.5 shows some examples of the heatmap on some test data:



(a) (b)

Figure 3.5: Heatmap examples on test dataset.

It is clear that the model is looking at the correct areas of the images in both Figure 3.5(a) and (b). The green-blue areas clearly contain most/all vehicles within the image. This is great news and shows that the accuracy is not just a coincidence that the model is finding other features that relate to the label. Overall, this model performs great, and with more data (specifically outside of city environments) would only improve the

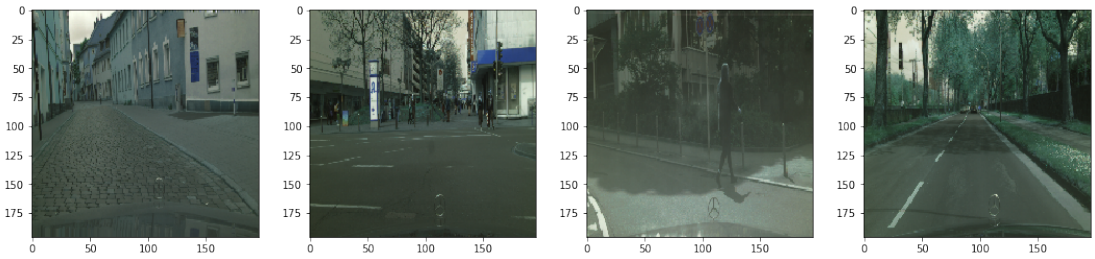
quality of the model.

Next, Table 3.5 shows the accuracy of the multi-class classification model:

Dataset	Accuracy
Training	68.66%
Validation	67.53%
Test	66.00%

Table 3.5: Accuracy results of multi-class classification model on each dataset.

Clearly, the accuracy of this model is far less than the binary model. Tweaking this model did not make the accuracy improve (without overfitting) and after viewing the predictions that the model makes, it will be clear how the model is working. With that said, Figure 3.6 show some examples of incorrect predictions by the multi-class model.



(a) (b) (c) (d)

Figure 3.6: Examples from test dataset of correct labels, where (a) predicted "0", (b) predicted "0", (c) predicted "0", (d) predicted "2".

In Figure 3.6(a), the pre-processing of the image has destroyed the detail in the traffic sign(s) that according to the labels are in the image. (b) would appear that the model is not familiar with sideways traffic lights (seen on the right). (c) features traffic signs that are high up in the view of the dashcam, which is typically where the sky/trees are located in most images which could be a reason why this label is incorrect. Finally, (d) could seem that the model is viewing the black vertical line on the left (between the two trees) as a traffic light. Clearly there is a pattern here, the model confuses traffic sign/light images with images with no objects (background data). This is due to how small these signs tend to be within the image. This is confirmed using the heatmap technique discussed previous and shown for test examples in Figure 3.7:

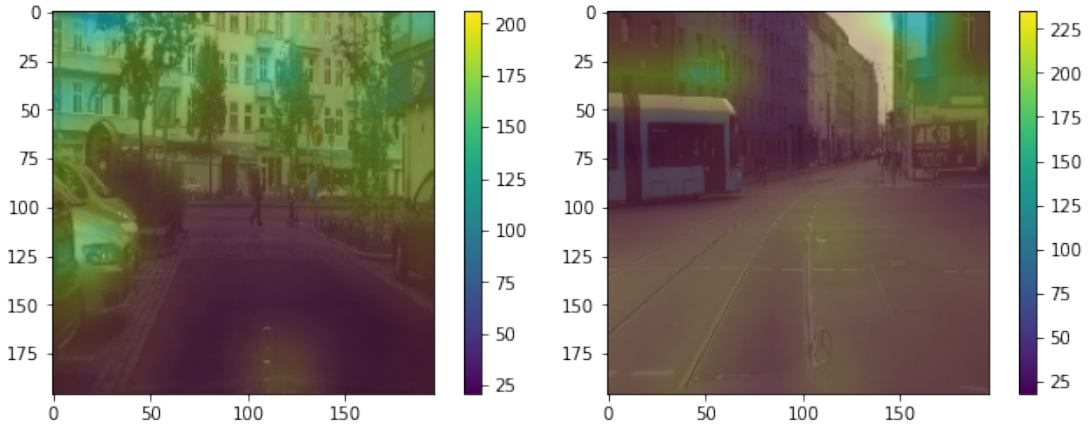


Figure 3.7: Heatmap examples on test dataset.

Both Figure 3.7(a) and (b) show the model struggling to find relevant features that relate to the label. To improve the accuracy of this model and reduce the issue of the model guessing the background class, a much larger and training dataset would be required.

Chapter 4

Regression

Regression models share similar techniques to binary classification, as they both predict a single value. Because of this, the general structure for going from data to a model from the previous task will be reused for this regression task.

4.1 Describing the Problem

A vehicle only has a handful of inputs which a human uses for the fundamentals of driving, the main inputs being the accelerator and brake pedal along with the steering wheel. The steering input is often determined by the lane lines (if available) in front of the vehicle, so it would be ideal that a model were to predict the angle of steering in order to stay within the current lane using a front facing camera. A system similar to this could both be used in autonomous vehicles with no human input, or as a fail-safe for human driving to ensure the driver is paying attention and not drifting out of the lane. Most new cars that are sold currently feature this technology within the other suite of ADAS technology.

However, the steering angle of a car sometimes is not related to lane lines. An example of this is when a vehicle is exiting a T-junction. It is likely that the front facing camera would only have the view of the perpendicular road, which a model would struggle to predict to drive without left or right direction. Situations like these will try to be avoided, along with other situations where there are multiple outcomes.

Now that the uses of such a model have been discussed, most of which are shared between this model and the models created in the previous chapter, a suitable dataset can be found for use in training and testing such a model.

4.2 The Dataset

An ideal dataset for this task would be much simpler than the previous dataset used in the last chapter. This is because manual labelling is not necessary, as the vehicles telematics can simply be recorded along with the respective image from a front facing dashcam.

A company called Comma AI, who is also a well-known player in the self-driving research space, develop systems that can be retrofitted into existing car models to improve the driver assistance systems of these cars. The "Comma Three" is a forward facing camera connected to a mobile device that links to the existing cruise control systems found in specific cars that enables adaptive cruise control with steering and brake control. Comma AI decided to open source the data that they collected for creating the device to help others learn to make systems themselves.

The dataset [41] in question features a total of 11 drives around a random area in the USA totalling roughly 7 hours of driving data. The data includes a view from a front facing camera along with almost 50 different telematics including speed, brake, and gear choice to name a few. Of course, the most important information for this task is the steering angle which it provides. This label data comes straight from the cars sensors and so pre-processing is critical to ensure the data makes sense to humans. Figure 4.1 shows two examples of the images from the dataset with the corresponding steering angle input from the telematics data:

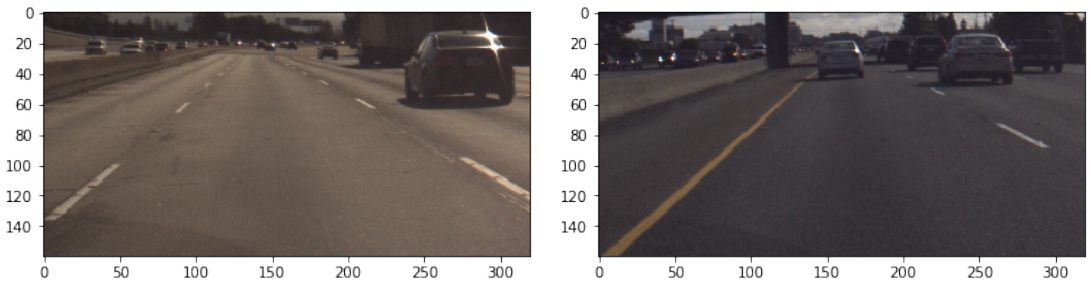


Figure 4.1: Examples from Comma AI dataset, where (a) has steering angle "74" and and (b) has steering angle "19".

Clearly from the labels of Figure 4.1 the steering angles must be incorrect, as the car clearly is not steering $74°$ or 74 radians, so clearly some pre-processing is required to get the correct steering angle. To work out what transformation is done to the steering

angle, it is useful to know the maximum of all the values which could give a clue as to how the angle is scaled. It turns out the the maximum value is around ± 5000 for both steering directions. After some research into how cars steer, it is clear that this steering value is around 10 times the true value of the steering wheel itself (not the wheels). So, dividing all the steering values by 10 will give more meaningful data but a model will not care how the number is scaled as long as it is consistent. Typically, vehicles have a steering ratio (from steering wheel to tyre) is between 12:1 and 20:1 [56], so this should be considered when viewing the labels in this chapter.

It is important to discuss the size of the dataset. As stated previously, there is roughly 7 hours of driving data which equates to roughly 250,000 8-bit full colour images (at 100Hz) which is around 75GB of data. This is significantly less space than if there were the same number of images in the Cityscapes dataset because the images in the Comma AI set only have a resolution of 320×160 (2048×1024 in Cityscapes) so scaling these down later is likely not necessary. Privacy is a valid concern, but unlike Cityscapes there is no blurred version of the dataset. A privacy issue starts with the first and last images in each drive, where the dataset includes the driver exiting and arriving to their house. Reversing will likely confuse the model, especially when inside a garage with no view outside so these images will be removed (6000 images from the start and end of the dataset).

The conditions of the environment are good in all the data and in most images the lane lines are clearly visible. Another area that the dataset differs from Cityscapes is the inclusion of data from driving during the night. This will be useful for improving the accuracy of the model for a wider range of conditions, as roughly half of the 24-hour day will experience these conditions. Importantly, training a model using this dataset will create a system that at best is as safe as a human since the dataset is of a human driving. Therefore, the driver of the car that was recorded for this dataset is assumed to be driving safely and legally.

4.3 Pre-Processing the Dataset

The images and vehicle telematics come from two different sources, each sources captures its data differently. Importantly, the rate at which images are taken verses the telematics is very different. Images are taken at a 100Hz rate (10 every second) and telematics are taken at a 20Hz rate (50 every second). To combat this difference, the average of every group of 5 labels will be calculated to get an equal number of images and labels.

Given the label values can be so drastically different, they should be scaled to en-

sure the model remains stable for all values that the label could take. All label values will be scaled down to have values between 0 and 1 to combat this. This will be done by dividing by twice the maximum value which will ensure the labels are within $(-0.5, 0.5)$ then add 0.5 to all the values. Of course, this will mean that the opposite operation should be done when the model predicts a value to get it back to the original form. While on the topic of scaling, all images will be normalised by dividing each pixel value by 255 to ensure all values are between 0 and 1.

It is also possible to use the same trick of reflecting each image in the y-axis to double the number of images like with the classification dataset. This would also require the label to be flipped by multiplying by -1 to get the opposite steering angle. However, with the number of images within this dataset it is unlikely that more will be needed, but if so then this technique can be used.

The Comma AI dataset does not come with splits for training, validation and testing so the data must be split manually. Validation data can be easily defined when fitting the model to the training set by using the training split hyperparameter. However, the testing dataset typically does not have a feature like this and so much be defined manually. It is common to use around a 70:15:15 split for training, validation and testing respectively [15]. For these reasons around 13% of the data will be used for validation and a further 13% for testing (which creates round numbers as will be seen shortly). As the dataset is so large compared to the previous task, the accuracy results will also be more useful as they will represent a much wider range of data.

As the dataset contains full drives from start to end, there is data that should be removed. For example, when the car is behind another vehicle the view will be limited and as such, it will be difficult for the model to predict an appropriate steering angle. To combat this, any data where the speed of the car is less than 10mph will be removed (telematics data features speed data). This will remove any slow moving, stationary and reversing data that may trick the model.

Table 4.1 shows the splits of training, validation and test data used for training the CNN in the upcoming section. It also shows some key statistics for the labels of the images including average, minimum and maximum values.

Dataset	Average Steering Angle	Standard Deviation of Steering Angle	Minimum Steering Angle	Maximum Steering Angle	Total Images
Training	0.50036	0.02806	0	0.9995	55,000
Validation	0.50069	0.02990	0	0.9995	10,000
Testing	0.50051	0.02555	0	0.9791	10,000
Total Images:					75,000

Table 4.1: Number of images for each dataset for regression model alongside important statistics about each dataset after pre-processing.

It is clear that the data within each respective split features the same wide range of images which will make validation using the validation and test dataset accurate and includes a mix of day and night driving scenarios. Only around half the dataset will be used to create the model in the upcoming section to save on computation time and space.

4.4 Building & Compiling the Model

It is now time to use the data found in the previous section to train a convolutional neural network. Table 4.2 showcases the final model that will be used for evaluation along with the dimension of each layer and parameters to be trained. This model was created using the code within Appendix C:

Regression		
Layer	**Output Shape**	**Number of Parameters**
Conv2D(64, (5, 5))	(None, 156, 316, 64)	4,864
Activation('relu')	(None, 156, 316, 64)	0
MaxPooling2D(pool_size = (5, 5))	(None, 31, 63, 64)	0
Conv2D(64, (5, 5))	(None, 27, 59, 64)	102,464
Activation('relu')	(None, 27, 59, 64)	0
MaxPooling2D(pool_size = (2, 2))	(None, 13, 29, 64)	0
Dropout(0.4)	(None, 13, 29, 64)	0
Conv2D(256, (3, 3))	(None, 11, 27, 256)	147,712
Activation('relu')	(None, 11, 27, 256)	0
MaxPooling2D(pool_size = (2, 2))	(None, 5, 13, 256)	0
Conv2D(256, (3, 3))	(None, 3, 11, 256)	590,080
Activation('relu')	(None, 3, 11, 256)	0
MaxPooling2D(pool_size = (2, 2))	(None, 1, 5, 256)	0
Flatten()	(None, 1280)	0
Dense(512)	(None, 512)	655,872
Activation('relu')	(None, 512)	0
Dense(64)	(None, 64)	32,832
Activation('relu')	(None, 64)	0
Dropout(0.5)	(None, 64)	0
Dense(1, activation = 'linear')	(None, 1)	65
Total Trainable Parameters		1,533,889

Table 4.2: Summary of the regression model featuring the layers and parameters for each.

Notice the total number of parameters in this regression model is significantly less than the numbers which were trainable in the classification model in the previous chapter. This difference is due to the additional dropout layer within the list of 2D layers. Removing 40% of the neurons this early on into the model will greatly improve overfitting issues but decreases the accuracy. After careful tweaking, this value for dropout was found to not drastically affect the accuracy of the model.

This regression model features another change from the previous classification model which is the inclusion of a set of 2D convolutional, activation and 2D max pooling layers. Without the inclusion of these layers, the model struggled to identify appropriate features such as lane lines to predict the steering angle. The number of filters in each convolution layer were also tweaked to aid with the models accuracy.

The final layer, which is a dense layer, scales down all the neurons in the previous layer into a single neuron which is the same as binary classification. The activation used for this layer is "linear", which does not change the activation of the neuron which can be seen in Figure 2.2.

To compile the model, the code in Equation 4.1 was used along with a tweaked Adam optimiser described in Equation 4.2:

```
  model.compile(optimizer = opt, loss = 'mean_squared_error',
metrics = [tf.keras.metrics.RootMeanSquaredError(name = 'rmse')]))
```
(4.1)

```
opt = keras.optimizers.Adam(learning_rate = 0.0001)
```
(4.2)

First, the loss function used was mean-square error (MSE) which is the default used for regression tasks. This was found to be the best loss function out of the possible options such as mean absolute error (MAE) or mean-square logarithmic error (MSLE) to name a few. Equation 4.3 showcases MSE, where the value of MSE must be minimised:

$$\ell_i(y_i, \hat{y}_i) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{4.3}$$

Where y_i represents the true value of the label for the i^{th} image and $\hat{y}_i$ represents the predicted label for the same image. N is the number of elements in the dataset. Fundamentally, the larger the error between the real and predicted labels, the larger the loss function.

Next, the Adam optimiser was used (see Equation 2.7) as it was one of the only optimisers that improved the accuracy of the model (as did RMSProp). However, as most of the values for the steering angle were low, the learning rate was adjusted to slow down the training of the model and reduce overfitting. The learning rate was finalised as 0.0001 as lower values was too slow, and higher values adjusted the neurons too much (overfitting).

Finally, the metrics that were displayed during training shows the root mean-square error (RMSE) which is simply the square-root of MSE. This was used to display another statistic during training to ensure the model remains stable.

The model was fit to the training data using the code found in Equation 4.4:

```
model.fit(X_train_norm, y_train, batch_size = 8, epochs = 1000,
        validation_data = (X_val_norm, y_val),callbacks=[
      tf.keras.callbacks.EarlyStopping(monitor = 'val_loss',
                patience = 5, restore_best_weights = True)])  (4.4)
```

The hyperparameters within the fit command are mostly used to configure training for different PCs and the machine used for training this model features a much larger amount of RAM (48GB) than the previous machine used for classification. This machine is able to train at any time, so a large number of epochs was used to maximise training time. However, to ensure the model does not overfit, early stopping was used. Early stopping is a check that occurs after every epoch that measures whether a metric has failed to get better [49]. For this model, training ends when the validation loss function fails to improve after 5 epochs (patience), after this the model restores the weights from the previous epochs to further reduce overfitting.

Finally, the validation data was pre-defined before fitting to reduce the computational load on the system rather than stating a validation split. Otherwise, the system could run out of memory before starting training causing Python to fail. With 25 minutes of training using an NVidia RTX A6000 GPU, the results are to follow in the next section.

4.5 Evaluation

Table 4.3 shows the accuracy results achieved using the model described in Section 4.4. As the model predicts continuous values, the way of describing the accuracy of the model is different to the way technique used for predicting classes. It is typical to compare the root-mean-square error (RMSE), mean-square error (MSE) or mean absolute error (MAE) [22]. For this model, the loss function used was MSE so this is what will be compared:

Dataset	Loss Function Value	Scaled Loss Function Value
Training	9.1080×10^{-5}	364.32
Validation	4.1164×10^{-5}	164.656
Test	4.6059×10^{-5}	184.236

Table 4.3: Loss function (MSE) results of regression model on each dataset. Recall that the scaled data represents the same scaling as the original data, which represents a 10 times increase from the steering wheel input.

At a glance, it appears that the model performs similarly between all three datasets. Importantly, the difference between the training loss and validation/testing loss is roughly half meaning the model is likely not overfitting. Clearly, the difference between the label and prediction is very low, but this will be scaled by 2000^2 (as MSE is the error, squared) to convert back into the original format of the steering angle labels which can be seen in the "Scaled Loss Function Value" column of Table 4.3. Note that this is the squared value of the error, so in reality the average error is between 12-20 between all datasets (which is known as the root-mean-square deviation). These numbers are now scaled the same way the original dataset was, which means that the numbers that the labels represent correspond to $10\times$ the steering wheel input.

To view the potential errors of the model, images with the largest difference between the real and predicted steering angle can be discussed. Figure 4.2 shows a few examples of images with the largest prediction errors with positive steering angles:

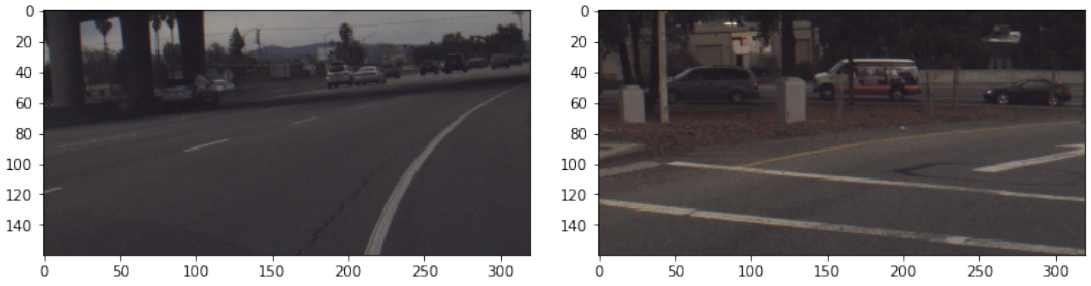


Figure 4.2: Examples from test dataset, where (a) predicted 815.7835 vs real 625.3879 and (b) predicted 492.0963 vs real 708.5258.

Whilst the predictions are in the correct direction in both 4.2 (a) and (b), the intensity of the steering angles are quite different. In (a), it would seem the vehicle is between two lanes and was likely completing a lane change when the image was taken. This would be difficult for a model to predict, as the algorithm is mostly trained on images where the vehicle remains in the same lane and does not change, so asking the model to predict which way the driver wishes to go is a random guess. The image in (b) again predicted the correct direction, but the steering angle is larger than the model predicted. From the lane lines, it could be that the model predicted to position the car more to the left of the lane line than the driver was aiming. Figure 4.4 shows similar examples of images but includes images with true negative steering angles:

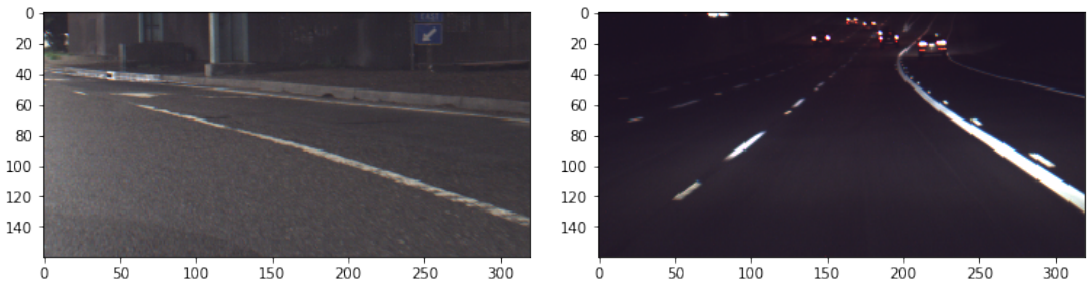


Figure 4.3: Examples from test dataset, where (a) predicted -844.3475 vs real -573.68286 and (b) predicted 1.9472 vs real -141.1080.

Figure 4.4 (a) shows the model predicted the correct direction however for (b) it did not. In (a), it would appear that the model incorrectly chose the left lane which the driver is driving toward. Like in the previous examples, it is random chance which lane the driver wants to drive into. (b) shows another example of the driver changing lanes (to the left) rather than staying in the current lane like the model predicts.

From these results with large errors, it is clear that the model struggles to predict the steering angle on sharp bends or lane changes, the latter of which would be down to randomness. More training data with sharper steering angles would likely improve this accuracy. To showcase that the model works for most of the drive, Figure 4.4 show examples where the model closely matches the true steering angle:

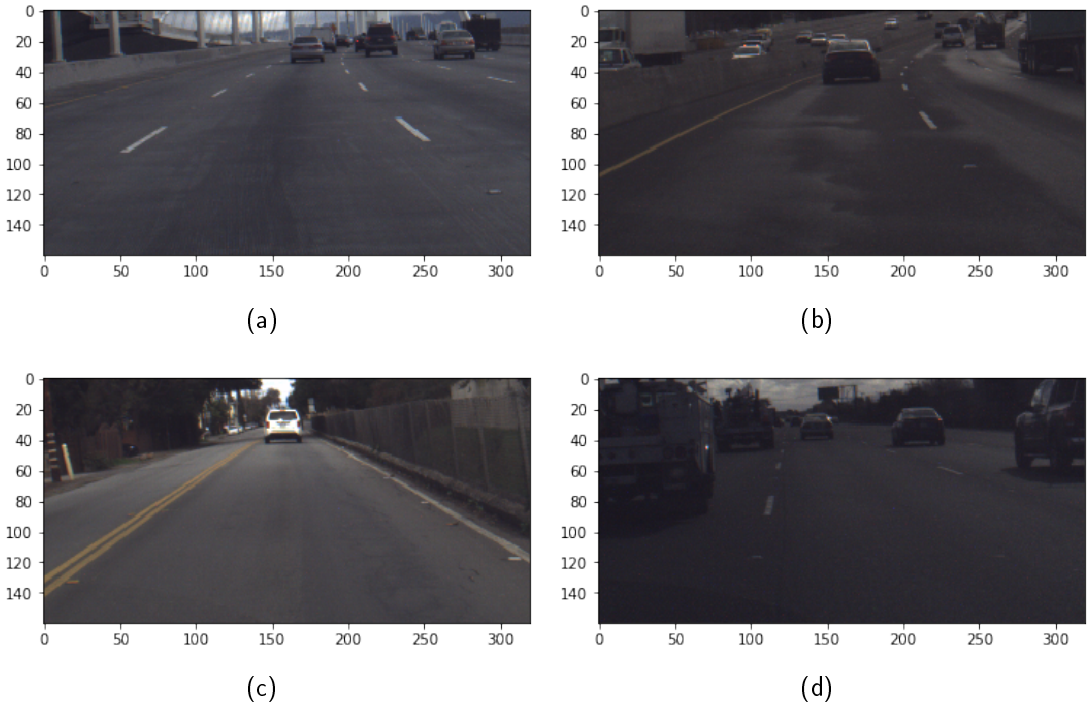


Figure 4.4: Examples from test dataset, where (a) predicted 24.2954 vs real 36.9827, (b) predicted 71.6643 vs real 59.9999, (c) predicted -12.7462 vs real -9.4246 and (d) predicted -15.3850 vs real -12.2234.

Unlike classification, the Grad-CAM technique for creating heatmaps cannot be used for regression. Therefore, the accuracy of the model can only be explained by noting patterns manually when viewing the test dataset images. Overall, the model performs extremely well, and this model was only trained using roughly a third to a half of the full Comma AI dataset.

Chapter 5

Object Detection

Localisation is related to classification as they both aim to identify objects within images. Where localisation differs is that it also tries to locate the object within the image using different labelling techniques. The natural evolution of localisation is object detection, which has the ability to localise multiple objects (including multiple of the same object) within one image input. It is clear that a technique such as this has huge potential for autonomous vehicles.

A technique used for labelling has already been seen in the Cityscapes dataset [13] where all the labels were created using semantic segmentation (labels separated into clusters containing individual objects) but there are other techniques for labelling images for this task. A simpler labelling method is bounding boxes, which uses a simple rectangle that contains the object within its area [36].

Object detection algorithms have recently seen great advancements from more traditional techniques used previously. Before 2015, an input image was split into an $(S \times S)$ grid where a $(n \times n)$ CNN would slide across the image by iterating on the grid. If an object was within the grid it would identify and keep iterating. Due to the varying sizes of objects within images, the $(n \times n)$ CNN region would be changes by increasing/decreasing the size [16]. This technique is clearly very inefficient if most of the grid does not feature any object, so a new algorithm would be proposed that would revolutionise object detection speed and efficiency.

The algorithm in question is named You Only Look Once (YOLO) which was released in 2015 [33] and improved in later versions including the latest YOLOv5 in 2020 (with no corresponding paper as of writing). As the name suggests, the algorithm passes through each image only once through the convolutional layers. Because of this, YOLO can run object detection so fast that it can be done in real time which is a breakthrough for applications like autonomous vehicles. The method works similar to

previous methods by dividing the image into a grid and detecting objects within each grid cell. Unlike previous methods, YOLO was trained to predict both the classification of the object alongside the bounding box values (coordinates and size). Next is a brief history of the some of the main improvements of each YOLO version:

- YOLOv2 in 2016 [31]:
  - Batch normalisation (for inputs).
  - Trained on higher resolution images (for some epochs).
  - Improvement to bounding box localisation using different boxes.
  - Localisation predicts that fall within $(0, 1)$.
- YOLOv3 in 2018 [32]:
  - Completely new architecture called "DarkNet-53" (YOLOv2 uses "DarkNet-19").
  - New architecture features far more layers and allows predictions at three different scales to aid with smaller objects.
  - Increased overall size, but improvement in accuracy.
- YOLOv4 in 2020 [6]:
  - Support for cross mini-batch normalisation (for inputs).
  - New loss function (Discussed later, as it follows in YOLOv5).
  - Support for mosaic data augmentation (stitching multiple images together) [28].
- YOLOv5 in 2020 (no paper):
  - Reduced overall size by 90% from YOLOv4.
  - Improved accuracy of predictions overall despite the lower size.
  - Far fewer convolutional layers.

The structure of YOLO predictions is shown in Equation 5.1 where p_c is the probability that there is an object within the box. The bounding box b has center coordinates b_x and b_y along with height b_h and width b_w. The probability that the object belongs to each class j is represented by c_j.

$$y_i = \begin{bmatrix} p_c & b_x & b_y & b_h & b_w & c_0 & c_1 & c_2 & ... & c_{n-1} \end{bmatrix}^T \tag{5.1}$$

The structure of y_i also depends on the number of objects detected within the input X_i, and so the vector is repeated for each object that is detected. The object class j is

given by the highest value of c_j. Alternatively, the boundary box coordinate prediction structure can also be b_{xmin}, b_{ymin} and b_{xmax}, b_{ymax} for the top left and bottom right corners of the bounding box respectively. More theory behind YOLO will be discussed in Section 5.4 where the model will be created.

5.1 Describing the Problem

In Chapter 3, a model was created to identify whether objects existed in front of a vehicle or not. Specifically, the model predicted the existence of cars and traffic signs which could be useful for some applications, but for an autonomous vehicles this would be quite basic. The natural evolution of a classification model would be to combine it with localisation to predict where a vehicle, traffic sign or even pedestrian is in front of the vehicle. The benefits of a system such as this are clear, as this is typically how humans drive by seeing where objects are in relation to the driver.

This chapter will aim to create a model which predicts the location and classification of objects within a 2D image input. With data on the position of the objects, along with calibration information about the cameras position in relation to the car, it would allow for the creation of an approximate localisation within a 3D space (just like in the real world). This is extremely powerful, and would negate the need for sensors currently found in cars such as radar, ultrasonic or LiDAR sensors that all use techniques that humans do not need to be able to drive a car safely to extend the predictions into 3D space.

A model such as this could be used in passive safety systems such as whether an object has appeared in front of the vehicle unexpectedly (for example a kid running on the road or the car in front stopping randomly) but could also be adapted for full active cruise control where the vehicle controls all driving input when combined with the steering angle model created in Chapter 4. Needless to say, YOLO (or other types of object detection) will be fundamental in solving the self driving problem, and by using the smaller version of YOLOv5 (YOLOv5s) it will prove that a model can be trained for the self driving application where computational power is much less than that of a training machine.

5.2 The Dataset

Unlike the regression model seen earlier, the dataset required to train a YOLO model would be similar to the Cityscapes dataset that was used for the classification model. However, as YOLO predicts bounding boxes rather than clusters, a different dataset

will be required for training such a model. This isn't strictly necessary, as algorithms exist that find the minimum bounding box that fits a group of points (pixels for images) such as "minimum-area enclosing rectangle" method [4]. However, this is not needed as there are plenty of existing YOLO-friendly datasets.

The dataset that will be used to create the YOLO model is a public dataset from Roboflow called "Udacity Self Driving Car Dataset (Fixed)" [51]. Clearly, the of the dataset title implies that there was an original dataset that required improvement which is indeed the case. According to Roboflow, original dataset features missing labels which have been fixed in this release. The fixed version is now suitable for this problem.

The dataset features 15,000 images that are either full resolution (1920×1200) or scaled down (500×500). The scaled down dataset was chosen due to the training time required for images as large as the full resolution, and these images would need to be scaled down regardless so the latter dataset saves time. Roboflow also offers the dataset to feature labels that are formatted for different packages and YOLO versions. As the model will be created using YOLOv5, it also limits the use to PyTorch (which works similar to Keras and is simply another Python framework used within the deep learning space). As such, the code found in Appendix D will be quite different to the previous code but annotations will aid with explanation of each step.

The dataset is made up of images from video clips from a forward facing camera mounted on a vehicle driving in urban areas in California, US. The weather conditions are good and the data only contains daytime drives. This limitation is likely due to the difficulty of labelling data during the night, as the camera may not pick up all objects like pedestrians in poorly lit environments. Despite this, there are a total of almost 100,000 individual labels featuring vehicles by type (car, truck and bikers), pedestrians and traffic lights (which includes the lights on the traffic light that are on). Table 5.1 shows the number of labels per class, and Figure 5.1 shows some examples of the images including the bounding box labels:

Class	Number of Labels
car	64,399
pedestrian	10,806
trafficLight-Red	6,870
trafficLight-Green	5,465
truck	3,623
trafficLight	2,568
biker	1,864
trafficLight-RedLeft	1,751
trafficLight-GreenLeft	310
trafficLight-Yellow	272
trafficLight-YellowLeft	14
Total Labels in 15,000 Images	97,942

Table 5.1: Table showing the total labels for each class found within all 15,000 images in the Udacity Self Driving Car Dataset (Fixed) dataset [51].

(a) (b)

Figure 5.1: Examples of images from the Udacity Self Driving Car Dataset (Fixed) dataset [51] with bounding box labels. Dark blue represents "car", orange represents "rider", pink represents "pedestrian", red represents "trafficLight-RedLeft" and light blue represents "trafficLight-Red".

It is clear that the dataset is lacking images featuring variations of traffic lights and vehicles. This is bound to happen when traffic lights spend most of the time either green or red, with a further subset of these traffic lights direct one direction (left). Hence, the model will likely perform worse when classifying traffic lights compared to other objects.

5.3 Pre-Processing the Dataset

Little pre-processing is needed for this dataset compared to previous datasets used in classification and regression, as the Udacity dataset was designed to be used with YOLO.

As previously mentioned, the dataset is hosted by Roboflow, which hosts a number of public datasets and allows for the user to add multiple pre-processing steps to create fully custom datasets. There is a limitation for free users that means users are unable to upload over 10,000 images which is quite disappointing when there is more than this amount in the dataset. This size can be increased using data augmentation for the training set using the horizontal flip to the training set (limitation of Roboflow) and the totals seen in Table 5.2:

Dataset	Number of Images
Training	10,052
Validation	1,717
Testing	841
Total Images:	12,610

Table 5.2: Number of images for each dataset for object detection model.

These images will all be scaled down to a resolution of (416×416) due to the way YOLO works computationally in multiples of 32. This is all the pre-processing that was required to create a YOLO model.

5.4 Building & Compiling the Model

As there is no paper introducing YOLOv5 yet, there is no formal documentation of the structure of the algorithm. However, YOLO is different to the other methods seen in this paper previously as the algorithm creates the model itself using predetermined (and trained) layers. It is known that YOLOv5 features significantly less convolutional layers compared to previous versions, with YOLOv4 having 53 convolutional layers [6] and YOLOv5s (Where YOLOv5s represents the small version of YOLOv5) having only 9 (1×1) to (3×3) convolutional layers. The smaller version of YOLOv5 will be used in this chapter as autonomous vehicles need to have the least amount of latency possible for safety reasons, and less layers of course means less computation is required for a car to compute.

The code in Appendix D will showcase the model and shows the layers that YOLOv5s

generated to best suit the input images. As there are a much larger number of layers that are connected not just linearly, but also to previous layers in the model, the layers themselves will not be shown in this paper. However, there are a number of difference to previous models that YOLOv5s does that will be further explained.

The activation function used on most layers is an adaptation of the ReLU function seen in Equation 2.2 and Figure 2.2. Traditionally, ReLU takes the maximum of 0 and z, resulting in a function which maps z to 0 for $z < 0$. Leaky-ReLU changes this to allow negative values of z to be mapped to small negative values. Equation 5.2 and 5.2 show the formula and graph of Leaky ReLU function:

$$h(z) = \begin{cases} z & z \geq 0 \\ 0.01z & z < 0 \end{cases} = \max(0.01z, z) \tag{5.2}$$

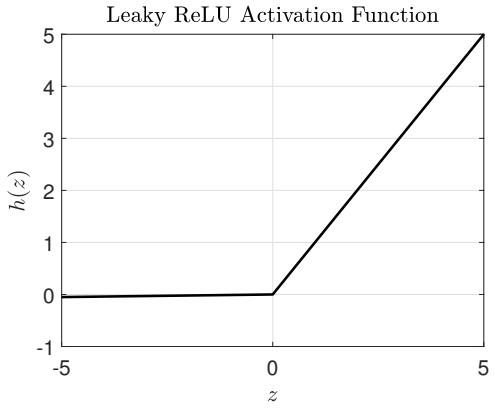


Figure 5.2: Graphs of the Leaky ReLU activation function.

This activation is likely used because of the way backpropagation works when the algorithm is learning. If the value of z is positive, the gradient of $h(z)$ is 1 which will encourage the algorithm to learn. However, if $z < 0$, regular ReLU would give a gradient of 0 and force no learning to occur. Leaky ReLU gives a small value to negative z so learning can still take place. See Section 2.3 for more information on backpropagation.

Now, there are several loss functions that correspond to each prediction within y_i. The classification elements of the predictions p_c and c_j ($j \in \{0, 1, ..., n-1\}$) use the binary cross entropy loss function seen previously in Equation 3.2 for binary predictions. The loss function becomes more complex when considering bounding boxes, as there needs to be a metric which can describe the difference in distance and size that the predicted bounding box compares to the actual labelled bounding box.

Introducing Intersection over Union (IoU), which as the name suggests is simply a function which calculates the ratio of two boxes intersection and union. See Equation 5.3 for the function (where $\hat{B}$ represents the predicted bounding box and B represents the true bounding box):

$$\text{IoU}(\hat{B}, B) = \frac{|\hat{B} \cap B|}{|\hat{B} \cup B|} \tag{5.3}$$

Now, this equation should be adapted to suit a loss function. This created the Complete IoU (CIoU) loss function [57] which is used in YOLOv5s. This function takes into consideration three properties of the prediction: overlap area, center point and aspect ratio. Equation 5.4 shows this loss function:

$$\ell_{\text{CIoU}}(\hat{B}, B) = \underbrace{1 - \text{IoU}(\hat{B}, B)}_{(1)} + \underbrace{\frac{\rho(\hat{B}, B)^2}{c^2}}_{(2)} + \underbrace{\alpha v}_{(3)} \tag{5.4}$$

Where:

(1) is the overlap area of the real and predicted bounding boxes.

(2) is the normalised distance between the center point of the predicted and real bounding boxes where:

$$\rho(\hat{B}, B) = \|\hat{B} - B\|_2$$

(3) is the aspect ratio consistency between bounding boxes where:

$$\alpha = \frac{v}{(1 - \text{IoU}(\hat{B}, B)) + v} \quad , \qquad v = \frac{4}{\pi^2}\left(\tan^{-1}\left(\frac{\hat{w}}{\hat{h}}\right) - \tan^{-1}\left(\frac{w}{h}\right)\right)^2$$

Now the loss function has been explained, most of the remaining key points of YOLOv5 are automatically assigned during training. Equation 5.5 shows the code to train the YOLOv5s model:

```
!python train.py -img 416 -batch 256 -epochs 100 -data
    {dataset.location}/data.yaml -weights yolov5s.pt -cache
```
(5.5)

Not much needs explaining here as the code in Appendix D will make things self-explanatory (such as the dataset.location). The img hyperparameter is simply the input image size (recall that the images are pre-processed to 416×416). Batch and epochs are specific to the memory available on the system that the model is trained on, and in this model each epoch used approximately 27GB of RAM.

5.5 Evaluation

The YOLOv5s model has now been trained on the dataset and so can be evaluated on new data. Before showcasing the results, the structure of the results is again different to the previous models evaluation metrics. The new metrics involved with YOLO are "precision" and "recall" and can be seen in Equations 5.6 and 5.7 using the information from Table 5.3:

		Real Label	
		Positive	Negative
Predicted Label	Positive	TP	FP
	Negative	FN	TN

Table 5.3: Table used to find true/false positive/negatives using real and predicted data.

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{5.6}$$

$$R = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{5.7}$$

Therefore, the model should have a precision and recall value growing closer to 1 meaning the number of FP and FN shrinks to 0. With this knowledge, Figure 5.3 displays the results found after training the YOLOv5s model:

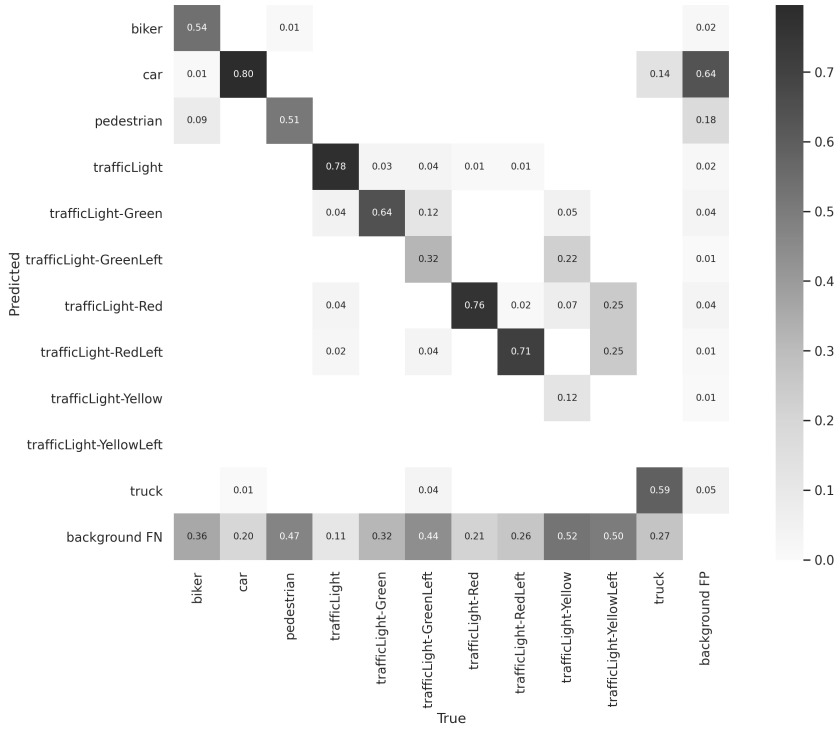


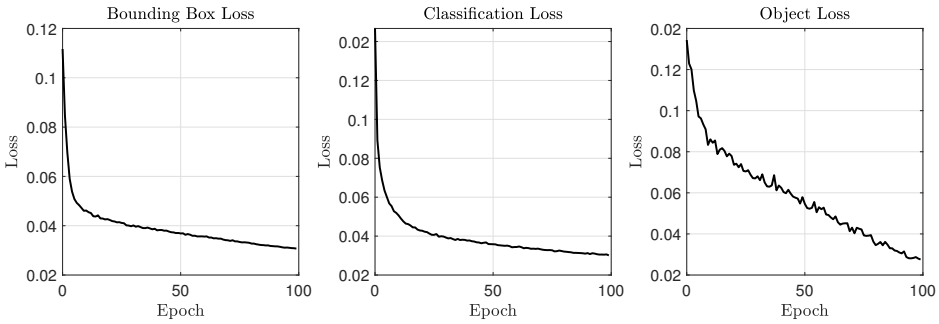
Figure 5.3: Confusion matrix showing the accuracy of predictions vs. real (true) class.

What Figure 5.3 clearly shows is that the model predicts cars, normal traffic lights and red traffic lights (darker squares) quite well in comparison to other classes. The various traffic light states clearly are the main issue with the model, where traffic lights which feature a priority left turn on yellow seemingly never classified correctly. This is due to the low likelihood of a traffic light being on yellow (the colour which traffic lights spend the least time one) with left turn only traffic lights being is even rarer. This issue can also be seen in the green version of this traffic light. Finally, the biker and pedestrian class are lower than should be expected. This lower accuracy could be due to the model struggling to distinguish between the two. Figure 5.4 shows some examples of the models predictions by plotting the bounding boxes on top of the original input images:

Figure 5.4: Examples from test dataset, with predicted bounding box labels and confidence on each image.

Each example within Figure 4.2 contain at least one error. (a), (c) and (d) show the model misclassifying the traffic lights within each image. It is clear that the model is

misbehaving when an object contains a yellow light traffic light, so more data of traffic lights in the yellow state is necessary. (b) suffers from a similar issue, but the prediction is of a traffic light with no light on when there is clearly a red light. Images (e) and (f) feature incorrect classification completely, as the first image does not feature a biker and the latter is mistaking a crane sign for a truck which may be subject to debate. It goes without saying that more data would improve this model (including more residential areas where there are a greater variance of objects like bins, signs and postboxes) just like any model would, but Figure 5.5 showcases how the model is moving toward an optimal solution:



(a)

Figure 5.5: Plots showing the loss function for each subset of predictions with a single image prediction.

These three graphs show that with more epochs (training time), the model should continue to minimise all of the loss functions. The object loss function is minimising the fastest as the problem of determining whether an object exists within a box exists or not is the simplest part of the problem. The bounding box loss is the function that is slowest to find a minimum but continues to get smaller over the epochs that were run while training.

Some examples of correct object detection are featured in Figure 5.6, with a decent spread of images with only a few to lots of objects:

Figure 5.6: Examples from test dataset, with predicted bounding box labels and confidence on each image.

Most of the test images are similar to those seen in Figure 5.6 where the bounding boxes are great. (c) shows how the model has learnt that vehicles can also tow trailers and be considered as one object rather than separate. Secondly, (d) perfectly represents how the predictions are great on objects that are occluded by other objects such as bushes in this example. Lastly, the model can also cope with a range of labels from a large amount seen in (e) and none in (f). Overall, the performance of this model is good given the low number of images required for training compared to the dataset for the regression model in Chapter 4.

Chapter 6

Discussion, Future Work & Conclusion

6.1 Discussion

A total of four models have been created in this paper that predicted a range of types of data. These models have had varying accuracies which mostly came down to the dataset that was used for each. This is the biggest limitation of training neural networks, as gathering and labelling data often takes the largest amount of time compared to creating the model itself. This presents a large hurdle to any individual/group of people that are attempting to solve this problem without using public datasets.

Currently, the companies that are attempting to solve autonomous vehicles are using varying approaches for collecting data for model training. The company Waymo uses human drivers to generate HD-Maps (where the vehicle knows its position within 10cm) for use in its autonomous taxi service available in Phoenix, US. These HD-Maps that are generated will be exponentially larger in size than the maps used in regular satellite mapping services and require far more effort to decode the raw LiDAR data for the road network around the entire world. The car manufacturer Tesla appears to have the upper hand as they have been gathering data from their almost 2,000,000 vehicles worldwide that have been manufactured with most having 8 cameras around the vehicle [3] able to send data back to Tesla. Of course, there are very few companies that have the possibility of doing something similar which again comes back to how the most difficult part of solving this problem is data collection. This also means that Tesla is also able to test out the performance of any models they create by running predictions in the background (known as "shadow mode") of the vehicle without physically controlling the car.

Another limitation of creating CNNs is the requirement for specialist hardware such as high end GPUs and high capacity RAM. GPUs especially are much more efficient compared to using a CPU due to their ability to perform large mathematical tasks by using the fast memory bounded to the GPU cores. CNNs essentially perform a huge number of matrix dot products (multiplying elements of matrices to one another) which quickly scales up when using images of high resolution. The amount of computing power needed to train a model that could outperform the safety of a human driver is infeasible without a data center capable of using multiple systems at once. Not to mention that the dataset would require enough storage to fully account for any possible situation that could arise during driving.

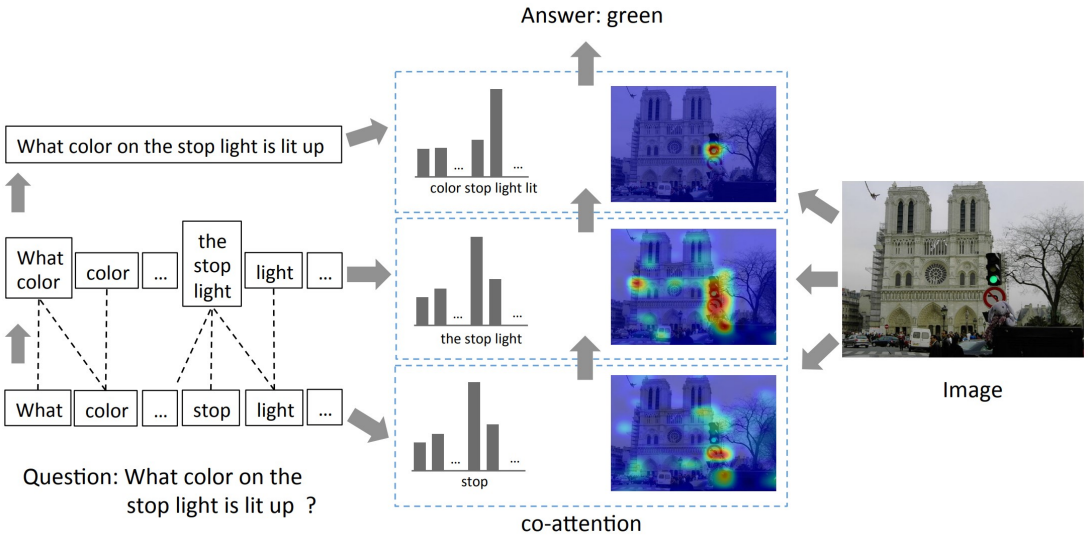
There has been reasonable concern over the use of technology such as object detection in applications like the military, but using object detection for driving to create self driving cars could inevitably save more lives than seat belts and air-bags combined. By using a different dataset, this technology can also have great potential in the medical industry to identify problems on scans or images without the need for a medical professional to spend time doing so.

6.2 Future Work

While the models in this paper can be used for basic predictions in autonomous vehicles, models will need to predict the best driving behaviour more so than the objects within an image. For example, detecting road markings and pedestrian crossings will give more advanced and accurate predictions for safe driving manoeuvres. Further work could involve predicting when to use the brake pedal as the dataset used in Chapter 4 had labels for braking.

More advanced techniques include "Visual Attention", which is an adaptation to CNNs which are similar to those used in this paper. The main aim of visual attention is to get the network to predict the most important part of an image. For example, an image taken from the front of a vehicle where a pedestrian crossing is approaching should be the most important thing that a normal driver focuses on as a pedestrian could appear [25].

This can be done using a technique known as "Image Captioning" which generates a caption related to the input by comparing the input layer to the output caption. This comparison is done by finding the context of the output label from the input image. Figure 6.1 gives an example [26] of the flowchart which a model follows to correctly distinguish what the question is asking and relate that to the input image using heatmaps:



(a)

Figure 6.1: Attention flowchart using driving example. The model is asked what colour the traffic light is lit [26].

Visual attention is another technique that assists with the explainability in neural networks, as Figure 6.1 clearly shows what each word(s) relate to in the image in order to make a correct prediction. The context vector $\mathbf{c}$ that relates the caption the input image is found using the following in Equation 6.1 where the alignment α is given in Equation 6.2:

$$c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j \qquad (6.1) \qquad \alpha_{t,j} = \frac{\exp\left(score(s_{t-1}, h_j)\right)}{\sum_{j'=1}^{T} \exp\left(score(s_{t-1}, h_{j'})\right)} \qquad (6.2)$$

Where h_j is the activation of the j^{th} neuron in the input layer, T represents the length of the input layer, s_t represents the activation of the t^{th} output layer. The $score()$ function is replaced with an alignment score function of the users choosing. Some examples of alignment score functions are additive attention, general attention or dot-product detection [55].

Finally, with the fast rate at which processing power is progressing, training a network using videos rather than static images is becoming possible leading to much more stable predictions. This would allow predictions within 3D space and potentially increase to 4D with the inclusion of time.

6.3 Conclusion

In conclusion, this paper summarises the mathematics behind machine learning and neural networks by completing experimentation that can be used in autonomous vehicles. CNNs contain vast amounts of trainable neurons which are organised into configurable layers when creating a custom model. The neurons value (activation) are then tweaked based on the training dataset to best represent it and any new data that is tested on the model. Using this knowledge, a total of four models were created to showcase the mathematics in practice with varying results.

The binary classification model which used using around 11,000 training images was able to correctly classify whether a car was in the image with 90% accuracy which was consistent with the accuracy on the validation and testing datasets proving there was very little overfitting. The Grad-CAM algorithm was used to determine which features of the input image the model was focusing on the most which aided with the explainability of the model and proved the model was functioning correctly. The applications of such a classifier are limited due to the simplicity of the task but one example could be as redundancy for conventional adaptive cruise control systems found most in vehicles today. Systems like cruise control often use radar to determine whether or not an object is in front of the vehicle and how far away it is, so by using a model such as the binary classification model could be used as confirmation that the radar is correctly identifying if a car is in front.

Despite using the same dataset and CNN layer structure, the roughly 4,000 images used to train the multi-class classification were not able to get the model to correctly identify the features within the input image that are traffic signs and traffic lights. The model performs well when identifying cars but the model confuses the traffic signs class with the background class resulting in an accuracy of around 67%, which was backed up by Grad-CAM that showed the model was struggling to identify probable features of the image which determined the label. A way of solving this problem is to increase the size of the dataset, as this is the current limitation of the models performance and means that is isn't useful for autonomous vehicles in its current state. With more data and more labels, it may be possible to identify specific signs rather than all signs being under one category like the labels currently are in the dataset.

Returning to predict a single value, a regression model was created using 75,000 images and corresponding steering angle data with an average mean-square error of between 160 and 370. These values for the loss function should be scaled down to true steering angle of the steering wheel would be between 16° and 37° which drops further when converting to the steering angle of the wheels themselves. The main reasons that this value could not go lower was two reasons: random chance from the driver of the vehicle

in the dataset and the hardware used for training not having enough memory to use the whole dataset. Despite the relatively high loss values, the values for unseen data (training and test dataset) were in fact lower than the training dataset which shows that the model is unlikely to be overfitting. The values which the model got most incorrect were predicted incorrectly for reasons that mostly come down to when the data shows the driver changing lanes randomly or selecting a different lane to what the model predicts.

Finally, a more advanced technique for object detection known as YOLOv5 was used to correctly annotate bounding boxes for key objects in an image. These objects include vehicles, pedestrians and traffic lights which are the most important objects to detect while driving. Despite YOLOv5 not having a paper which details how it differs from previous versions, some of the main differences that have been found by the community were listed along with a brief history of YOLO versions before the 5^{th}. Results showed that dedicating more data and epochs to the model will likely improve the accuracy by reducing the loss function further for all predictions (bounding box, object detection and classification). The main problem with the model was the classification of traffic light states, as the dataset did not feature a large enough number of images with each respective state.

Overall, the performance of three out of the four models created in this paper were great. The models were generated using public datasets on a mix of standard to specialist hardware for training. With more data and better hardware for training, it seems inevitable that full self-driving is only around the corner. The recent announcement that Tesla has begun rolling out software updates to vehicles in Canada to enable a "beta" version of the "Full Self-Driving" package (which only uses 8 cameras) after the success in the USA [9] shows the incredible amount of progress that real-world artificial intelligence is making.

This generations "space race" is becoming very exiting between the many companies in the autonomous vehicle space including Tesla, Waymo, Comma AI and Uber as continue to develop a self-driving vehicle using various techniques including those mentioned in this paper. The safety of the roads will no doubt improve very quickly when the law accepts that autonomous vehicles are the future.

Bibliography

[1] The size and quality of a data set, data preparation and feature engineering for machine learning, Jul 2019. URL https://developers.google.com/machine-learning/data-prep/construct/collect/data-size-quality.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. *Understanding of a convolutional neural network*. 2017. doi: 10.1109/icengtechnol.2017.8308186.

[3] Julija Andjelic. How many teslas have been sold? 25+ tesla car sales statistics, Jan 2022. URL https://fortunly.com/statistics/tesla-car-sales-statistics/.

[4] Dennis S Arnon and John P Gieselmann. A linear time algorithm for the minimum area rectangle enclosing a convex polygon. 1983.

[5] Sandeep Balachandran. Machine learning - dense layer, Jan 2020. URL https://dev.to/sandeepbalachandran/machine-learning-dense-layer-2m4n.

[6] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020. URL https://arxiv.org/abs/2004.10934.

[7] Jason Brownlee. Difference between a batch and an epoch in a neural network, Oct 2019. URL https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/.

[8] David Canty. Training, validation and testing data explained, Apr 2021. URL https://www.applause.com/blog/training-data-validation-data-vs-test-data.

[9] Jason Cartwright. Tesla fsd beta rolling out to canada, officially expands beyond united states, Mar 2022. URL https://techau.com.au/tesla-fsd-beta-rolling-out-to-canada-officially-expands-beyond-united-states/.

[10] Dami Choi, Christopher J Shallue, Zachary Nado, Jaehoon Lee, Chris J Maddison, and George E Dahl. On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446*, 2019.

[11] François Chollet. Keras documentation: Grad-cam class activation visualization, Mar 2021. URL `https://keras.io/examples/vision/grad_cam/`.

[12] IBM Cloud Education. What are neural networks?, Aug 2020. URL `https://www.ibm.com/cloud/learn/neural-networks`.

[13] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[14] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[15] Rachel Draelos. Best use of train/val/test splits, with tips for medical data, Sep 2019. URL `https://glassboxmedicine.com/2019/09/15/best-use-of-train-val-test-splits-with-tips-for-medical-data/`.

[16] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019. ISBN 9781492032618. URL `https://books.google.co.uk/books?id=HHetDwAAQBAJ`.

[17] Geoffrey Hinton. Lecture 6a: Overview of mini-batch gradient descent.

[18] DSKim (https://stats.stackexchange.com/users/27783/dskim). Why should we shuffle data while training a neural network? Cross Validated. URL `https://stats.stackexchange.com/q/245502`. URL:https://stats.stackexchange.com/q/245502 (version: 2020-03-09).

[19] Cliff AB (https://stats.stackexchange.com/users/76981/cliff ab). Why is gradient descent inefficient for large data set? Cross Validated. URL `https://stats.stackexchange.com/q/315571`. (version: 2017-11-27).

[20] Jianglin Huang, Yan-Fu Li, and Min Xie. An empirical analysis of data preprocessing for machine learning-based software cost estimation. *Information and Software Technology*, 67:108–127, 2015. ISSN 0950-5849. doi: 10.1016/j.infsof.2015.07.004.

[21] Inductiveload. Maxima and minima.svg, Sep 2020. URL `https://commons.wikimedia.org/wiki/File:Maxima_and_Minima.svg`.

[22] Alboukadel Kassambara. Regression model accuracy metrics: R-square, aic, bic, cp and more, Mar 2018. URL http://www.sthda.com/english/articles/38-regression-model-validation/158-regression-model-accuracy-metrics-r-square-aic-bic-cp-and-more/.

[23] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952. ISSN 00034851. URL http://www.jstor.org/stable/2236690.

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[25] Yi-Chieh Liu, Yung-An Hsieh, Min-Hung Chen, Chao-Han Huck Yang, Jesper Tegner, and Yi-Chang James Tsai. Interpretable self-attention temporal reasoning for driving behavior understanding. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, may 2020. doi: 10.1109/icassp40776.2020.9053783. URL https://doi.org/10.1109%2Ficassp40776.2020.9053783.

[26] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. Hierarchical question-image co-attention for visual question answering, 2016. URL https://arxiv.org/abs/1606.00061.

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

[28] Manivannan Murugavel. Yolo v4, May 2020. URL https://manivannan-ai.medium.com/yolo-v4-750cd627064f.

[29] Tim Oosterhuis and Lambert Schomaker. "who is driving around me?" unique vehicle instance classification using deep neural features, 2020. URL https://arxiv.org/abs/2003.08771.

[30] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL https://www.sciencedirect.com/science/article/pii/0041555364901375.

[31] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger, 2016. URL https://arxiv.org/abs/1612.08242.

[32] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018. URL https://arxiv.org/abs/1804.02767.

[33] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2015. URL https://arxiv.org/abs/1506.02640.

[34] Renanar2. Convolutional layers of a convolutional neural network.svg, Dec 2018. URL https://commons.wikimedia.org/wiki/File:Convolutional_Layers_of_a_Convolutional_Neural_Network.svg.

[35] Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 2007.

[36] Adrian Rosebrock. Object detection: Bounding box regression with keras, tensorflow, and deep learning, Oct 2020. URL https://pyimagesearch.com/2020/10/05/object-detection-bounding-box-regression-with-keras-tensorflow-and-deep-learning/.

[37] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.

[38] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[39] Grant Sanderson. What is backpropagation really doing?, Feb 2022. URL https://www.3blue1brown.com/lessons/backpropagation.

[40] Grant Sanderson. What is backpropagation really doing?, Feb 2022. URL https://www.3blue1brown.com/lessons/backpropagation-calculus.

[41] Eder Santana and George Hotz. Learning a driving simulator. *CoRR*, abs/1608.01230, 2016. URL http://arxiv.org/abs/1608.01230.

[42] Dominik Schmidt, Nov 2018. URL https://dominikschmidt.xyz/nesterov-momentum/.

[43] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359, Oct 2019. ISSN 1573-1405. doi: 10.1007/s11263-019-01228-7. URL http://dx.doi.org/10.1007/s11263-019-01228-7.

[44] Saily Shah. Cost function: Types of cost function machine learning, Mar 2021. URL https://www.analyticsvidhya.com/blog/2021/02/cost-function-is-no-rocket-science/.

[45] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1), 2019. ISSN 2196-1115. doi: 10.1186/s40537-019-0197-0.

[46] Keras Team. Keras documentation: Adam, 2020. URL https://keras.io/api/optimizers/adam/.

[47] Keras Team. Keras documentation: Conv2d, 2020. URL https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D.

[48] Keras Team. Keras documentation: Dropout layer, 2020. URL https://keras.io/api/layers/regularization_layers/dropout/.

[49] Keras Team. Keras documentation: Earlystopping, 2020. URL https://keras.io/api/callbacks/early_stopping/.

[50] Keras Team. Keras documentation: Rmsprop, 2020. URL https://keras.io/api/optimizers/rmsprop/.

[51] Roboflow Team. Udacity self driving car object detection dataset - fixed-small, Feb 2020. URL https://public.roboflow.com/object-detection/self-driving-car/3.

[52] Martin Thoma. A survey of semantic segmentation, 2016. URL https://arxiv.org/abs/1602.06541.

[53] Yugesh Verma. Guide to different padding methods for cnn models, Sep 2021. URL https://analyticsindiamag.com/guide-to-different-padding-methods-for-cnn-models/.

[54] Lai Wei. Multi-hot sparse categorical cross-entropy, Oct 2018. URL https://cwiki.apache.org/confluence/display/MXNET/Multi-hot+Sparse+Categorical+Cross-entropy.

[55] Lilian Weng. Attention? attention!, Jun 2018. URL https://lilianweng.github.io/posts/2018-06-24-attention/.

[56] Wikipedia contributors. Steering ratio — Wikipedia, the free encyclopedia, 2022. URL https://en.wikipedia.org/w/index.php?title=Steering_ratio&oldid=1066395316. (Online; accessed 29 March 2022).

[57] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss: Faster and better learning for bounding box regression, 2019. URL https://arxiv.org/abs/1911.08287.

Appendix A

Binary Classification Model Code

Code

The code for binary classification can be found in the GitHub link below:

`https://github.com/SydProom/MMath-Dissertation/blob/main/Binary%20Classification/BinaryClass.ipynb`

Model

The Keras model file can be downloaded below (exceeds GitHub size limit):

`https://drive.google.com/file/d/1Mu5_w_HAcjpfbmYLRl5AUxXbE-RPmtt4/view?usp=sharing`

Appendix B

Multi-Class Classification Model Code

Code

The code for multi-class classification can be found in the GitHub link below:

`https://github.com/SydProom/MMath-Dissertation/blob/main/Multi-Class%20Classification/MultiClass.ipynb`

Model

The Keras model file can be downloaded below (exceeds GitHub size limit):

`https://drive.google.com/file/d/1KH62FNXJJ-LsQVPakpweZutNuxNhTZAp/view?usp=sharing`

Appendix C

Regression Model Code

Code

The code for regression can be found in the GitHub link below:

`https://github.com/SydProom/MMath-Dissertation/blob/main/Regression/SteeringAngle.ipynb`

Model

The Keras model file can be downloaded below (exceeds GitHub size limit):

`https://drive.google.com/file/d/1I8JU5vB20LFUGrY97P2FQCe4nRwv2mjt/view?usp=sharing`

Appendix D

Object Detection Model Code

Code

The code for object detection can be found in the GitHub link below:

https://github.com/SydProom/MMath-Dissertation/blob/main/YOLO%20Object%20Detection/ObjectDetection.ipynb

Model

The PyTorch model weights (best) can be found in the GitHub link below:

https://github.com/SydProom/MMath-Dissertation/blob/main/YOLO%20Object%20Detection/best.pt