Creating Robust Neural Network Generalisation for a Single-Sensor Self-Driving Car

Chara Grant

EERI 474: Final Report

Supervisor: Prof Kenny Uren
Date: November 30, 2020
Student number: 29458447

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction, Problem, and Literature Survey

1.1 Background: Hobbyist Self-Driving

DIYRobocars is a movement started in 2017, aiming at making Self-Driving accessible and cheap. Monthly meet-ups and races are held all over the world, with the aim to develop these tiny cars to outperform a human driver [6].

The flagship system of the DIYRobocars movement is Donkey Car - a python-based platform running on a Raspberry Pi. More recently, it can run using a Jetson Nano (link to product: https://developer.nvidia.com/embedded/jetson-nano-developer-kit). This system is mounted on a scale remote-control car, and consists of a single front-facing camera and a Convolutional Neural Network (CNN) as its stock pilot [7].

Besides the Pilot and Camera, the Donkey Car platform is made up of control scripts to assist with image input and storage, web control for a human driver, and PWM outputs to the scale vehicle's servo motors. These all serve as support structures around the Camera-CNN relationship [8].

The standard Donkey Car currently learns by way of "Behavioural Cloning". Its on-board CNN is trained with image and angle data collected by a human driver. Theoretically, this will result in the CNN Pilot mimicking the driving behaviour of the human driver/s that collected the data.

Figure 1.1 shows a photo taken of the Donkey Car built and used during vacation work periods of 2018-2020, which was fondly dubbed "Littlefoot".

Figure 1.1: "Littlefoot" the Donkey Car

1.2 Problem Definition

Experience working with the Donkey Car system in the past has shown it to be poorly adaptable to changes in its envrionment or new altogether environments in its base configuration. Something as simple as changing the lighting or introducing a new object in the background will cause the Pilot to drive poorly.

In order for self-driving to be useful, a vehicle should obviously be able to perform in environments different from the precise environments that they were trained in. This problem is mostly caused by the models used **overfitting** to their training data. This could be solved by simply training with more datasets. However, this is not always feasible due to great time and resource constraints involved with collecting good, varied, and sufficient datasets. This solution might also pass up the opportunity of truly getting to the bottom of the problem and exploring interesting alternatives.

This considered, the problem can be defined as creating a generalisation solution for an Artificial Neural Network (ANN) based Donkey Car Pilot. This generalisation should be rooted in the design of the ANN Pilot, as well as in the implementation of the ANN portion of the Pilot in symbiosis with a generalisation solution - such as feature engineering or image augmentation.

1.3 Generalisation: A Literature Survey

Generalisation in ANNs has long been a topic of study, with some studies dating back to the 1990's and beyond. Some generalisation practices explored over the years include, but are not limited to [1, 9, 10]:

- Regularization
- Dropout
- Feature Engineering
- Network "Pruning"
- Alternating Training and Test Datasets

Regularization

Regularization is the act of constraining neural network **weight** values to a certain range. This means that when a network trains, it is forced to keep its model simple. Regularization is limiting the entropy and stochasticity of the weights by restricting them to updating only within small increments [1].

Weight Regularization is accomplished by adding a penalty value to each weight as it is updated. This penalty value is determined by the value of the weight [1, 10].

There are two main methods used to accomplish Weight Regularization [1]:

1. L1 regularization, where the penalty value is proportional to the absolute value of the weight.
2. L2 regularization, where the penalty value is proportional to the square of the weight value.

For example, let's take a general example of minimizing the **cost function** of a feed-forward network [10]:

$$E(w) = \alpha\Sigma(y(w, u_i) - d_i)^2, \tag{1.1}$$

where w represents the weights, u_i the data input, and d_i the expected output. This function is defined on the training set τ [10]:

$$\tau = (u_i, d_i), i \in 1, N_v, \tag{1.2}$$

where N_v is the size of the training dataset, and $y(w, u_i)$ represents the input training vector [10].

When (as an example of L1 regularization), a penalty value is added to the cost function as a function of the weight values, the result is [10]:

$$\tilde{E}(w) = E(w) + \lambda E_R(w), \tag{1.3}$$

where λ is some scalar and $E_R(w)$ is a quadratic function of the (linear) model weights [10]:

$$E_R(w) = w^T K w. \tag{1.4}$$

Model weights will be thoroughly defined in Chapter 2. The above set of equations are merely an example of the concept of Weight Regularization.

Dropout

A major cause of poor model generalisation is overfitting. This can either be caused by the network being too large (i.e. too many hidden neurons/layers), or the data provided having too much noise or being too specialized [1, 10].

Dropout is one way to limit the representations of the data that the network can learn, thereby reducing the chance of the network learning representations that do not exist.

Dropout is performed by randomly culling a percentage of the neurons of a specific hidden layer at each training step. This is usually accomplished by multiplying the outputs of randomly selected nodes by 0, at a probability rate of $1 - dp$, where dp can usually range anywhere between 0.5 and 1 [11].

Dropout has also been employed in the hardening of Neural Networks against adversarial attacks, as is in the case of Wang et al [12]. This was done by applying dropout not only during training, but during the testing and deployment phases, hardening the Network towards adversarial examples [12].

Mathematically defining dropout, if the **network feedforward function** were to be described as [12]:

$$x_i^{(l+1)} = w_i^{(l)} \boldsymbol{y}^{(l)} + b_i^{(l)}, \tag{1.5}$$

$$y_i^{(l+1)} = f(x_i^{(l+1)}), \tag{1.6}$$

where x_i represents an input to the next layer, w_i forms part of the weights matrix $\boldsymbol{W}$, b_i forms part of the biases vector $\boldsymbol{b}$ [12].

Dropout in the network feedforward function can be defined as [12]:

$$x_i^{(l+1)} = w_i^{(l)} \tilde{\boldsymbol{y}}^{(l)} + b_i^{(l)} \tag{1.7}$$

Where $\tilde{\boldsymbol{y}}$ is the augmented output of the previous layer resulting from the elementwise product between a probability vector $\boldsymbol{r}^{(l)}$ and the original layer output $\boldsymbol{y}$ [12]:

$$r_j = 1 - Bernoulli(dp). \tag{1.8}$$

dp represents the inverse dropout probability distribution. Thus [12]:

$$\tilde{\boldsymbol{y}} = \boldsymbol{y}. * \boldsymbol{r}. \tag{1.9}$$

Feature Engineering

Feature Engineering refers to the act of actively highlighting or tweaking features present in the dataset to enable certain desired connections to be made.

Feature Engineering is mostly employed with shallower, perceptron networks. When this process is developed far enough in certain problem spaces, it often negates the need for a neural network altogether [1].

In the scope of Deep Neural Networks on the other hand, Feature Engineering has heavily been employed in areas where expert knowledge is paramount. Such as detection of brain shunts in MRI's, prediction of power generated by Building-Integrated Photovoltaics, and Sentiment Analysis [13–15].

Network "Pruning"

Network "Pruning" is another method by which to achieve generalisation through training [9, 12]. Similar in spirit to dropout, pruning is the act of stochastically removing neurons in each hidden layer (by some probability) and then adjusting the weights of the remaining neurons according to this probability. This usually occurs at each training step [12].

Known as the method of Stochastic Activation Pruning (SAP), the probability that the output of a given hidden neuron (h_j^i) will be "pruned" is given by the relation [12]:

$$p_j^i = \frac{|h_j^i|}{\Sigma_{k=1}^{a^i}|h_k^i|}. \tag{1.10}$$

The remaining hidden unit's outputs are scaled up proportionality to this probability using the reweighting factor [12]:

$$q_j^i = 1 - (1 - p_j^i)^{r_p^i}, \tag{1.11}$$

with r_p^i being the number of hidden neuron outputs sampled.

Similar network pruning methods have been developed for Recurrent Neural Networks specifically, with promising generalisation results (only 6.27% of never-before-seen language strings misclassified) [16].

An interesting use for pruning is in the compression of Convolutional Neural Networks and Deep Neural Networks - in order to enable these architectures to be implemented more cost-effectively and on more computationally sparse platforms (such as embedded systems) [17–19].

Alternating Training and Test Datasets

When limited data is available, or the data that is available is similar, K-Fold training can often be used to avoid overfitting [1]. Another alternative training method is swapping the training and test datasets during training, as was explored by Zhang Jie at the University of Newcaste [9]. These two approaches shall be referred to as the K-fold Method and the Swap Method, respectively.

The K-fold Method, or K-fold crossvalidation, is used most often when a small dataset and likelihood of a high variance in the validation error when testing the model is present [1]. The K-fold method consists of splitting the dataset into K partitions, randomly selecting $K-1$ of those data partitions for training, and reserving the K_{th} one for testing and validation. The $K-1$ data partitions are used to train $K-1$ identical versions of a model, and test each of these models on the validation set.

This is used to assist in the tracking of the effect changes made while training the model - in order to tell whether or not the regualrization method applied in any way improved the validation performance of the model. Once all changes have been made and assessed appropriately, a final model can then be trained on the full dataset [1].

The K-fold method not only eases the testing and experimentation of different regularization techniques, but it also increases the network's exposure to data representations that it would not have otherwise seen if the dataset is not well distributed. Cycling through the k-folds of trainingn and validation data during training may also help to prevent overfitting on one sort of data.

The Swap Method is quite similar to this approach to the K-fold method [9]. At the end of each training epoch, the error on both training and validation datasets are calculated, and, if the errors on either can be improved, the two datasets swap roles and training resumes. This process is repeated until **earlystopping** ends the process [9].

Chapter 2

Literature Study

2.1 Introduction

In 1783, there was great excitement about a particular chess match. This chess match, which featured the grand master of the time, was unlike any other. The opponent was not a human challenger. Instead, it was a Machine: the Mechanical Turk. A clockwork automaton able to move chess pieces with his mechanical arm, and, some say, make decisions [20].

However, things were not as spectacular as it seemed. This automaton, invented by Wolfgang Von Kempelen to impress the Archduchess of Austria, was in fact an elaborate hoax. Hidden in a chamber within the machine was a human chess master observing pieces on a magnetic chess board and manuevering the machine to do his bidding [20].

Despite the hoax and disappointment, the display was not for nothing. Charles Babbage, the inventor of the first programmable machine, lost to the Mechanical Turk twice. This, in turn, reportedly helped inspire his invention of the Difference Engine in 1819 [20].

During the Second World War, Alan Turing and his team were recruited to help crack the Enigma Machine - leading to the development of the Bombe, an ancestor of the modern computer [21]. This led to the train of thought: what if this machine could think for itself?

In his paper, Computer Machinery and Intellegince, Turing proposed the Imitation Game. This is a means by which it can be determined whether or not a machine can "think", or if it is "self-aware". Later on in the same paper, he discusses the possibility of Computer Machines' ability to learn. Later on in his paper, he addresses Lady Ada's objection: "The Analytical Engine has no pretentions to originate anything. It can do whatever we know how to order it to perform", by setting up the concept that computers may, in fact, with sufficient resources be able to learn [22].

The concept of modern Machine Learning is partially based on a brain cell interaction model published in "The Organisation of Behaviour" by Donald Hebb in 1949. This was inspired by his research and theories of neuron excitement and communication [23]. In keeping with the anthropomorphization of computers, IBM released the IBM 704 in 1957. This was the first ever perceptron - a physical node that was capable of learning, and was touted as

a means to provide Computer Vision. Many years was poured into the development of this machine - however, the single perceptron never truly lived up to the expectations of Computer Vision as it was advertised to, and research into this concept eventually died down, which very well might mark the first of many 'AI Winters' [23].

In the 1960's, the concept of Neural Networks (or multi-layered perceptrons) with **feed-forward** and **backpropogation** algorithms began to take hold. Backpropogation, invented in the early 1970's is essential to modern **Deep Learning**. In the 1980's, **hidden layers** propelled the development of Artificial Neural Networks, which were now capable of learning and detecting representations and patterns of data that humans would not be able to teach it [23].

Now, Machine Learing Algorithms are applied in several areas: such as Fraud Detection, Natural Language Processing, Computer Vision, and, of course, Self Driving Vehicles [23].

2.2 Machine Learning and Deep Learning

"Machine Learning" differs from "Symbolic AI" in that instead of having rules developed and given to it on how to deal with data and inputs, the "rules" are learned and developed by the mahine itself. This takes the form of a "black box" wherein data representations and patterns are randomly generated, tested against the truth, and updated to better "fit" the behaviour required. The rules that are developed from this are then applied to new data in new contexts, and it is expected that the machine is then able to perform as expected [1].

"Training" refers to the process of using statistical methods to extract representations of the data, and "learning" is applying those rules and representations successfully to new datasets [1].

Besides this explanation, machine learning is divided into three main categories: Supervised Learning, Unsupervised Learning and Semisupervised Learning [24].

Unsupervised learning takes place when the data that is provided to the model is unlabeled - i.e. there is no clear structure to the data in the first place. The task of unsupervised learning is to process the data and detect patterns in it - forming clusters that can can later be utilized in useful ways [24].

Supervised Learning, which is what will be dealt with in this problem scope, is training models using expert, labeled data. This is typically used in regression and classification tasks. The neural network, or model, is trained using labeled examples known as the training dataset. Then, it is given unlabeled inputs and is tested on the true labels [24].

The goal of supervised learning is to minimize false positives and false negatives in the predictions made by the Model on the data. This is accomplished by having an error function that compares the output to a truth label, and updating the weights in the model to reduce this error [24].

Deep Learning, while being a form of Machine Learning, augments this process [1]. Instead of having just as many "neuron layers" as is needed to make predictions on the data, the

"Deep" in Deep Learning refers to including far more neuron layers - in order to extract different representations of the input data. These representations and patterns are better used to make accurate predictions than what would be made on the raw input data itself [1].

Deep Learning can also be referred to as "Hierarchical Representation Learning", where each neural layer learns a representation via continuous exposure to the data [1]. Neural Networks, as a major mechanism of Learning Models, will be explored in Section 2.3 below.

2.3 Neural Networks and Learning Models

A Neural Network is a Mathematical model inspired by the Brain. However, the actual implementation of this is not completely similar to how a real Biological Neural Network would operate, and it is often unhelpful to think of it in this way. A Neuron, or in some cases a **perceptron** can be thought of as a mathematical 'node' or placeholder for input data representations. The 'dendrites' between these nodes can be thought of Connections transporting data representations between Neurons [1]. Let it be assumed, for the time being, that **Densely Connected Layers** are being worked with in this example.

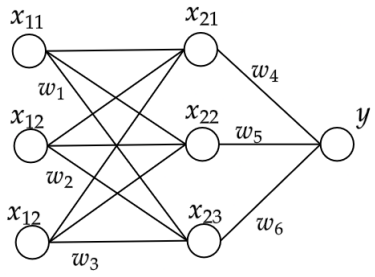


Figure 2.1: Basic Illustration of an example Densely Connected Network

In Figure 2.1 a practical example illustration of a Densely Connected Neural Network is shown. This network contains an input layer, a densely connected layer, and one output node. The x values represent the data values exported from each node, and the w values represent the weights linked to each connection. In the above image, the weights are not indicated on each connection line, but each connection line has it's own weight that it applies to the data output x_{in} of the neuron before it. The y value represents the final output value of the network.

Let's look at the feed-forward propogation of this network. Let's consider the output of the node x_{2_1} as an example of how the feed-forward propogation of a typical neural network operates (without **activation functions**) [12]:

$$x_{2_1} = \begin{bmatrix} w_{1_1} & w_{1_2} & w_{1_3} \end{bmatrix} * \begin{bmatrix} x_{1_1} \\ x_{1_2} \\ x_{1_3} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}. \tag{2.1}$$

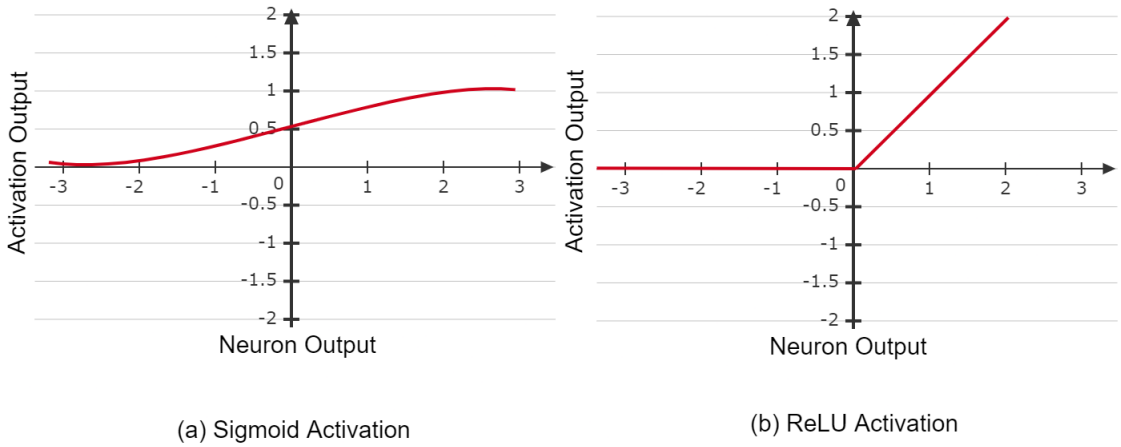
This process is similar for all of the neurons in the densely connected layer. Similarly, the output, y, is finally given by (once again, without activation functions):

$$y = \begin{bmatrix} w_4 & w_5 & w_6 \end{bmatrix} * \begin{bmatrix} x_{2_1} \\ x_{2_2} \\ x_{2_3} \end{bmatrix} + \begin{bmatrix} b_4 & b_5 & b_6 \end{bmatrix}. \tag{2.2}$$

This equation illustrates the basic inner workings of a neuron. The b-values indicate biases, which are randomly initialised along with the weights. These biases are there to compensate for very small results from the matrix multiplication of weights and previous neuron outputs [1].

For this network to be able to make reliable and useful statistical predictions, it must be able to learn the ideal weight and bias values that will produce these predictions. Besides the ability to learn, a network must be able to increase its probability space - since a linear probability space is quite constrained.

These two ends can be achieved by two means - firstly, activiation functions must be utilized to achieve non-linearity in the model. These activations take whatever output from the neuron and pass it through a function. There are several activation functions, but for the purposes of this illustration, let's focus on the **Sigmoid** and **Rectified Linear Unit (ReLU)** functions [1], which are illustrated in Figure 2.2



(a) Sigmoid Activation

(b) ReLU Activation

Figure 2.2: Graph (a): Sigmoid, Graph (b): ReLU

Sigmoid is the activation function that will be used in the output node, it maps every value to a relative (probability) value between 0 and 1 [1]:

$$\tilde{y} = \frac{1}{1 + e^{-y}}. \tag{2.3}$$

where y is the output of the network, and $\tilde{y}$ is the resulting prediction output from the sigmoid activation applied at the output node. (Note that this not the only sort of activation function used at output nodes, for some regression tasks, no activation function is used at the output at all.)

The Rectified Linear Unit activation function is commonly used as an activation function for the rest of the nodes in a network. The ReLU function zeros out all negative neuron outputs, and only allows positive outputs to pass through to the following layer. The ReLU function is defined as thus [1]:

$$y_{1_1} = max(0, x_{1_1}), \tag{2.4}$$

where y_{1_1} nonlinear output of the node producing x_{11}. Incorporating the activation functions into the example representation of a neural network results in Figure 2.3 below:

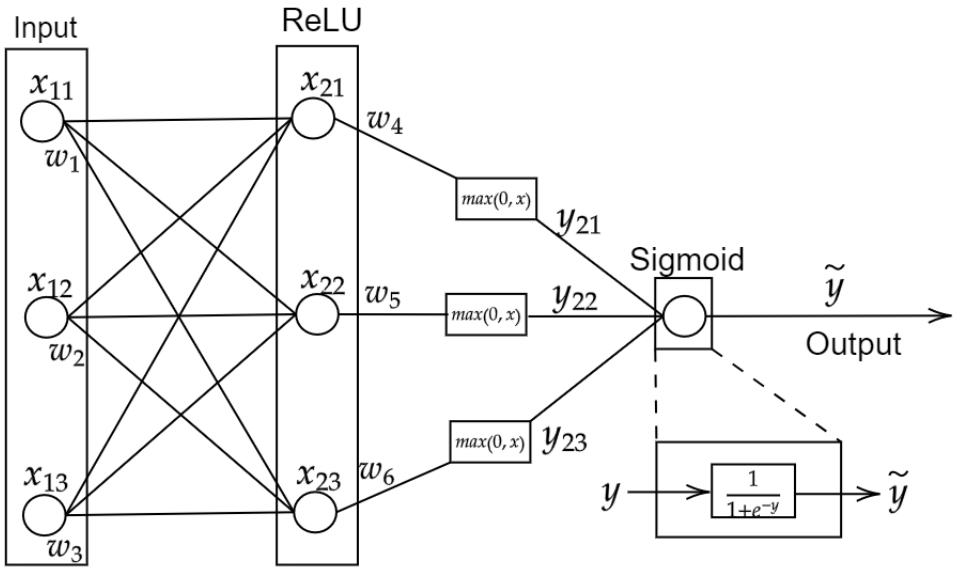


Figure 2.3: Example Densely Connected Network with Activation Functions

Now that activation functions have been incorporated into the example, the next ingredient of the Learning Model can be moved onto: A "means to learn".

Neural Networks learn by means of backpropogation: a method that was developed in the 1980's [23]. However, in order to use **backpropogation**, a feedback signal with which to adjust the values of the network's weights and biases is required. This is accomplished by way of a **loss function**. A loss function computes the error (can be mean absolute error, mean squared error, or anything relevant to that problem's data context) between the network's output, $\tilde{y}$, and the expected output, $\tilde{y}'$ [1]. Let the error function below be used for the example network [10]:

$$E(w) = \alpha\Sigma(\tilde{y}(w, x_i) - y_i')^2, \tag{2.5}$$

with $\tilde{y}'$ represents the expected output, w represents the weight values, x_i represents the previous node outputs, and α is a randomly selected scaling factor [10].

Figure 2.4 shows the result of incorporating the error function into the example network:

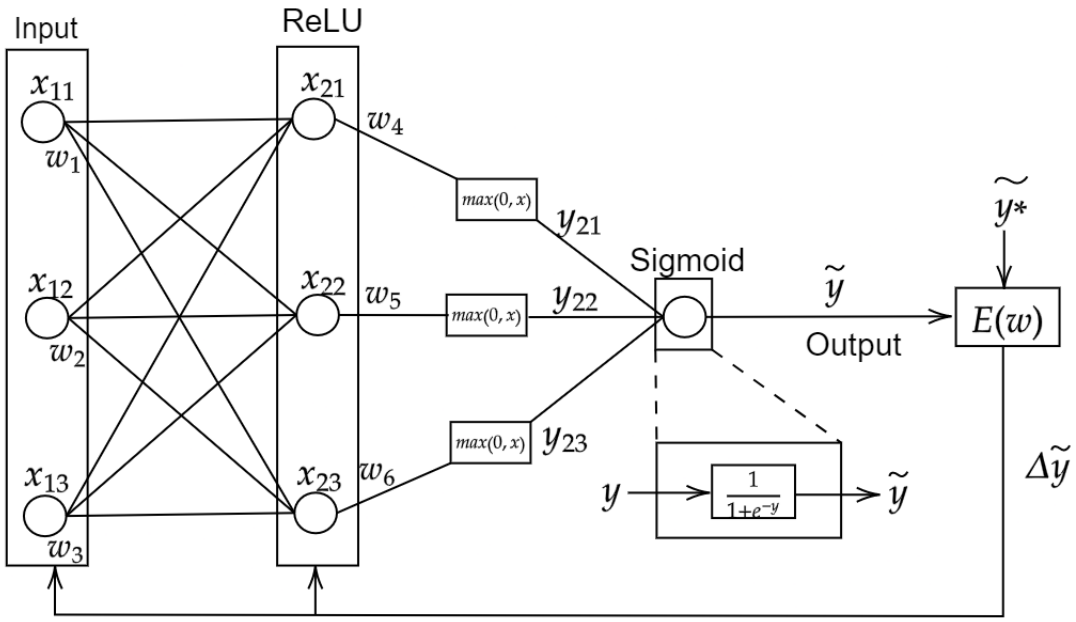


Figure 2.4: Example Densely Connected Neural Network incorporating Loss Function

The error value generated from the cost function is then fed back into the neural network to update the weight values. Once this is done, the cycle is repeated with the next set of inputs and labeled outputs - resulting in the network's weights being updated again. With enought training cycles, the Model should be able to make reliable predictions [1].

In practice, the training process is broken down into epochs and steps, and an **Optimization Function** is used to backpropogate the error through the model (addressed in Section 2.4). A training process can have any number of epochs (which incompass a set of training steps), and each epoch contains any number of training steps. The training steps are dependent on how the data is split: for example, there may be 1000 datapoints that are split into 10 batches of 100. Thus, each epoch will have 10 steps - each with their feedforward, error calculation, and backpropogation phases. At the end of each epoch, a set of validation data is fed through the network in order to test its performance, and its accuracy and error value is calculated [1]. Figure 2.5 illustrates the high-level NNA training flow discussed here.

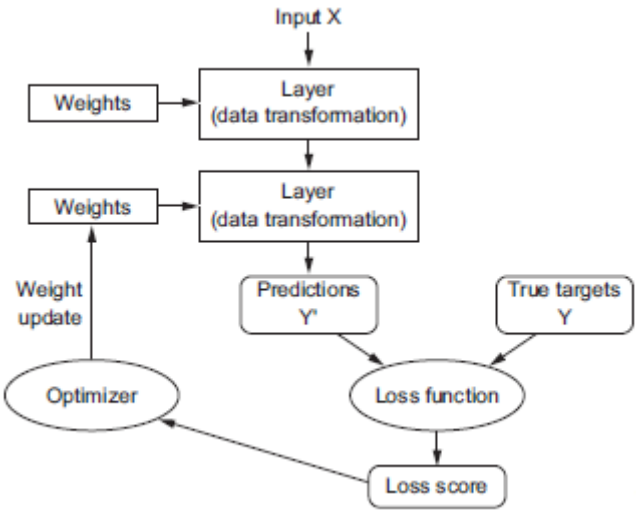


Figure 2.5: High Level Network Architecture and Training Flow [1]

2.4 Backpropogation and Optimization Functions

The cost function, or error, of a Neural Network is computed as a function of the Network's weights [1,10,25]. In keeping with the example from before, let us for the sake of the argument assume that the curve of this cost function at the end of the first epoch is as shown in Figure 2.6 below:

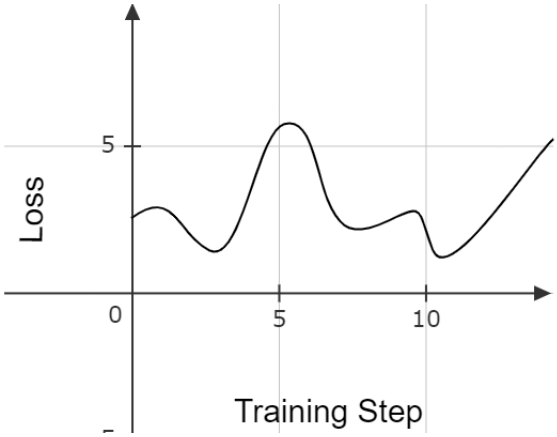


Figure 2.6: Example Cost Function Curve

The in order to update the weights, the gradient of the cost function must be determined. The partial derivative of the function at each weight is determined - in order to ascertain the effect of changing that weight would have on the entirety of the cost function [25]. Based on this, certain weights are updated in the opposite direction of and relative to their gradients [1]. This is referred to as Gradient Descent, and the goal of this exercise is to find the global minimum of the cost function, and to find the combination of weights that, when updated, will minimise the total error [1].

This process is highly computationally intensive, and complex. Therefore, several optimization functions have been developed to handle this process. This includes, but is not limited to:

1. Stochastic Gradient Descent [1, 26]
2. Mini-Batch Stochastic Gradient Descent [1]
3. ADAGRAD [26, 27]
4. RMSProp [26, 27]
5. Adam [1, 26]

Stochastic Gradient Descent works by selecting a weight point at random, and computing the gradient. If the gradient is not zero, it updates the weight in the opposite direction of the gradient, and moves on to stochastically select another weight, until it reaches the global minimum of the cost function. **Mini-Batch Stochastic Gradient Descent** works in much the same way, where mini-batches of data are selected at random. This mini-batch is then run through the neural network, and its own cost function is determined. The gradients of the weights in this cost function is determined, and the weights are updated to minimize this function. Then the algorithm moves on to the next stochastically selected mini-batch. This method is less computationally intensive than traditional stochastic gradient descent [1].

ADAGRAD (Adaptive Gradient Algorithm) is the first of many adaptive optimizers to come after it. The goal of ADAGRAD is to take into consideration the previous gradients and adjusts the learning rate and step sizes of future gradients accordingly [27]. At each timestep, the algoritm obtains a gradient of the lost function in the position of a stochastically selected weight value. Instead of simply updating the weight in the opposite direction of the gradient, each weight is updated according to the following function [28]:

$$w_{t+1} = \Pi_X^{diagG_t}(w_t - \alpha diag(G_t)^{\frac{1}{2}} g_t), \tag{2.6}$$

where w_t is the currently stochastically selected weight, X is the probability space of the selected weights, G_t is the matrix of the squares of all previously calculated gradients, g_t is the gradient of the currently selected weight, and α is the learing rate [28].

RMSProp (Root-Mean-Square Propagation) is another adaptive optimization algorithm based on Stochastic Gradient Descent [27]. A basic explanation of RMSProp would be to say that it divides the current gradient by the running average of the previously calculated gradients, and then the weight is updated as in stochastic gradient descent. [28]

Adam is an adaptive optimization algorithm that updates its learning rate by means of the first and second moments of its gradients [27,29]. This is done by means of cyclically updating the gradient moments based on selected estimators and gradient values, and then using these values to update the weight value until the cost function converges [29]. This method has been seen to take up far less computational power and memory than other methods.

However, in many problem contexts, Adaptive Learning Optimization Functions have been shown to both converge and generalise poorly [26].

For that reason, Stochastic Gradient Descent with some acceleration will be tried for this project, and compared with adaptive learning algorithms.

2.5 Activations

Activation functions are used to port Neural Network layer outputs to a non-linear space, in order to increase the probability space and predictive power of the network [1]. An activation function can also be understood to be a transfer function [25]. Three of the most commonly used activation functions (especially within Convolutional Neural Networks - The focus of this generalisation exercise) are [25]:

1. Sigmoid
2. Softmax
3. ReLU (Rectified Linear Unit)
4. leaky ReLU [1]

The **Sigmoid** activation function maps all output values of a neuron/layer to between 0 and 1 continuously - instead of 'jumping' values to either 0 or 1 like a unit step function would [1, 25]. The sigmoid activation function is described mathematically by [1, 25]:

$$\tilde{y} = \frac{1}{1 + e^{-y}}, \tag{2.7}$$

where y is the raw output of the node.

The sigmoid function is typically used in binary classification problems, since it will provide a 'probability' that a prediction is either 'yes' or 'no', or 'option 1' or 'option 2' [1].

Another typical output activation for classification problems is **Softmax Activation**. Softmax Activation is also known as Softmax Regression or a Maximum Entropy Classifier in the statistics world. Softmax Activation is applied when an output layer - or a vector of logits - has more than 1 entry (or non-binary options). The outputs of the neurons in the output layer, or the logit entries in the logit vector, are each converted to probability values that sum up to 1 [30].

This is accomplished by the following [31]:

$$\tilde{y_i} = \frac{e^{y_i}}{\Sigma_{j=0}^{n} e^{y_j}}, \tag{2.8}$$

where $\tilde{y_i}$ is the activation of the output y_i of the i_{th} of n neurons.

The most commonly used activation function, **ReLU**, and its sibling, **leaky ReLU** are now discussed. These two activation functions essentially 'mute' neurons that give negative outputs, and only pass outputs that are positive. In the case of leaky ReLU, the over-muting of some neurons would cause some network architectures to not easily learn representations - making errors take longer to converge. Leaky ReLU then only scales negative output values, while fully passing positive values, so that not too much of the network's predictive power is lost [1, 25].

The ReLU activation function is mathematically represented by [32]:

$$\tilde{y} = \begin{cases} 0 & \text{if } y < 0 \\ y & \text{if } y >= 0 \end{cases}. \tag{2.9}$$

The leaky ReLU activation function is represented by [32]:

$$\tilde{y} = \begin{cases} \alpha y & \text{if } y < 0 \\ y & \text{if } y >= 0 \end{cases}, \tag{2.10}$$

where α is some scaling constant smaller than 0.1.

2.6 Typical Layer Types

When constructing a neural network, it is important to know that there are several different layer types. These types include, but are not limited to [1, 2]:

1. Input Layers
2. Output Layers
3. Dense Layers
4. Convolutional Layers
5. Pooling Layers

Input Layers contain placeholders, or Tensors, for input data. These layers are of a fixed size, and expect inputs of a certain type. Tensors are a special matrix-like datatype for Neural Network experimentation software like Tensorflow and Pytorch, and are how the data in neural networks are represented. In Python, a Tensor expecting batches of 120 datapoints, with a rank of 3, would be represented by the tuple [1]:

$$(120, 15, 20, 10), \tag{2.11}$$

with each rank having dimensions 15, 20 and 10 respectively. However, in many machine learning circles, Tensor rank is referred to as dimension as well. However, in this report, this will be referred to as rank from now on in order to prevent confusion [1].

An **Output Layer** contains the raw values for the final prediction of the model. Depending on the application, this layer may or may not have an activation function attached to it. If it is a regression application, it will most likely not have an activation function attached to the (mostly single) output node. On the other hand, if it is a binary classification task, it will most likely have a sigmoid activation. If it were a multi-class classification task, it might have have a softmax activation [1].

Dense layers, or fully-connected layers are those layers where every hidden unit or node is coupled to each node in the following layer. These layers are typically used to either extract representations of data, or to generate predictions that will be fed into the output layers. These layers are also typically activated using ReLU activation [1, 2].

Default neural network layers are often poorly equipped to deal with data in grid format - such as images. Images can be flattened into a $1 \times n$ vector and passed into a dense network - however, most of the power of image recognition lies in analysing the arrangements of the pixels themselves, instead of the pixel values contained in the image, as a standard densely connected layer will do. Therefore, **Convolutional Layers** are used in networks used for image processing [1, 2].

Convolutional layers are optimized for use on data that is of a grid-like format. The internal structure of a Convolutional layer operates differently to that of a standard hidden unit - instead of matrix multiplication with inputs and weights, a convolutional operation between the grid data and a **filter** kernel [2] takes place.

The convolutional operation (for discrete signals) is represented by [2]:

$$y[k] = \Sigma_{n=-\infty}^{\infty} x[n]h[k-n], \tag{2.12}$$

where y represents the vector resulting from the convolution of a digital signal x and impulse function h. In the context of a convolutional layer, the input signal would be an $n \times m$ image or feature map, the impulse function would be an $n \times n$ filter matrix, and the output would be a feature map. The filter matrix, like the weight matrix in regular hidden units, is made up of randomly initialized filter values (or, weights). These filter values are learned via backpropogation - and the network is trained to pick up on certain features [2].

A **pooling layer** is another layer often used in convolutional network architectures. This layer transforms a feature map that is output by a convolutional layer by either transferring only the maximum values of this feature map (i.e., the most common features) to a new feature map, or, by averaging the values in a certain radius within the feature map. This is known as **max pooling** and **average pooling** respectively. The purpose of the pooling operation is to reduce the variance in feature maps that would be caused by a feature being in different locations in the image - thereby hardening the convolutional neural network to overfitting features to a specific location [2].

2.7 Architectures of Interest

Two of the main problems that neural networks can be used for is regression and classification problems. The regression and classification problems that will be found in this project's scope are those that are applied on image data, and directions drawn from that image data.

There are many different well-used and documented neural network architectures. This section deals with the architectures of interest for this project context.

2.7.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are some of the most powerful networks to implement on problems involving images, or multi-dimensional data. A typical CNN is made up of

convolutional layers, pooling layers and densely connected layers.

The convolutional layers of a CNN are essentially the same as a regular hidden layer - meaning that there are weights, inputs and biases. However, instead of there being a simple matrix multiplication between the inputs to the layer and the layer's weights, the incoming data is convolved with a weights matrix - this matrix is known as the kernel, or filter. The output of this convolution operation, as with conventional networks, is passed through an activation function such as ReLU. The resulting output matrix is known as a feature map [1,2].

Many neural network libraries do not use convolution in their layers as is strictly mathematically understood. Rather, many of these libraries used cross-correlation [2]:

$$S(i,j) = \Sigma_m \Sigma_n I(i+m, j+n) K(m,n), \tag{2.13}$$

where S is the resulting feature map matrix, I represents the input matrix, and K represents the kernel.

Figure 2.7 below shows an illustration of how such a convolution operation may take place:

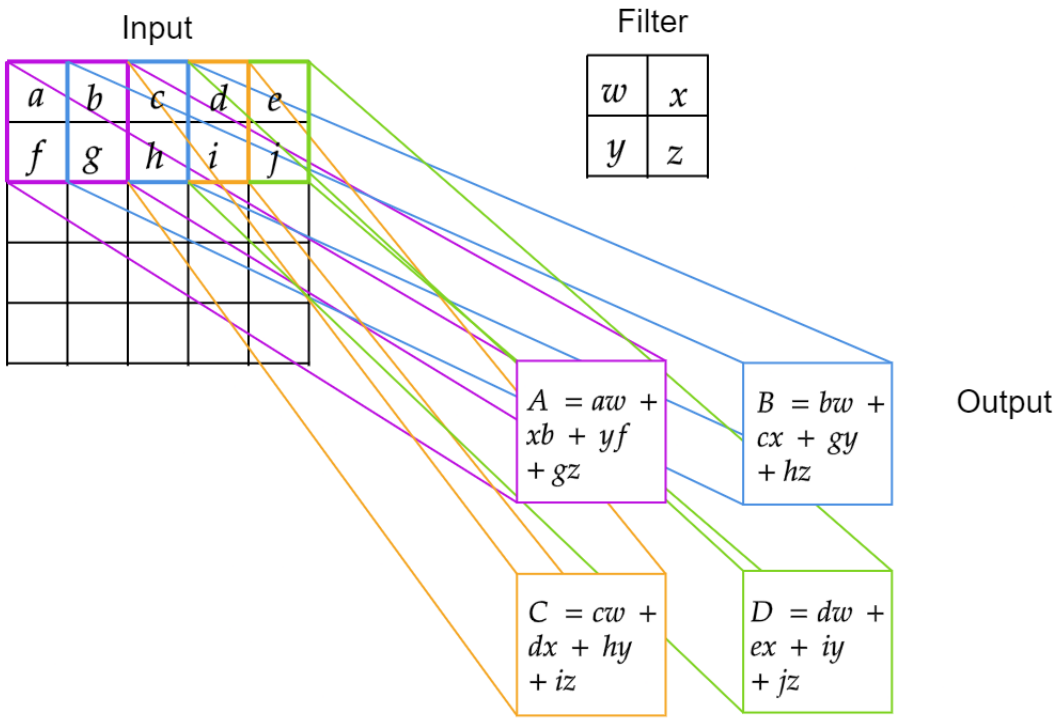


Figure 2.7: Visual Representation of Neural Network Convolution [2]

Cross-correlation peforms the same basic function as convolution, but does not flip the kernel. This does not have any major influence on the learning power of a Convolutional Neural Network, as convolutional layers are rarely implemented on their own, and the same features will be learned either way (even if it is in a reverse order) [2].

Figure 2.8 shows an image taken from Francois Chollet's book [1] which illustrates possible feature extraction operations of a CNN.

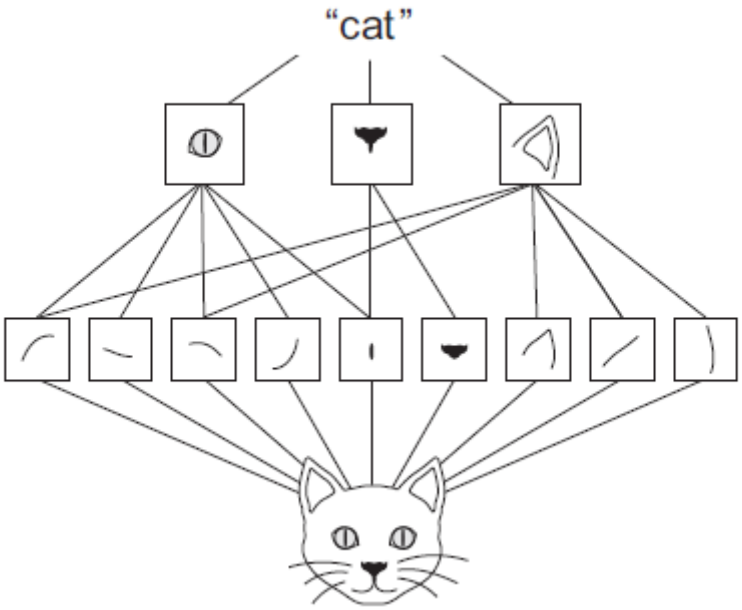


Figure 2.8: Illustration of Feature Extraction [1]

The feature map may then be passed to a following layer - whether it be another convolutional layer, a pooling layer, or a **flattening** layer.

The pooling layer reduces the feature map by either averaging the activations, or getting the maximum activations from a certain surrounding area. This hardens the network to variations in location of the features in the input data, increasing the effectiveness of the network. This is especially useful when used in detection of objects. Another useful effect of pooling is that over time, the network learns which variations in the data it must become invariant to [2].

The output of a pooling layer or convolutional layer, depending on the goals of the network, can then be **flattened** and fed into a Dense Layer. This dense layer then takes the flattened feature maps, and applies traditional neural network operations to this data, allowing the network to make statistical predictions. The output layer of a convolutional neural network is then typically Dense Layer with a sigmoid or softmax activation for a classification problem, or an activation appropriate for a regression problem [1].

This CNN Architecture is shown in Figure 2.9.

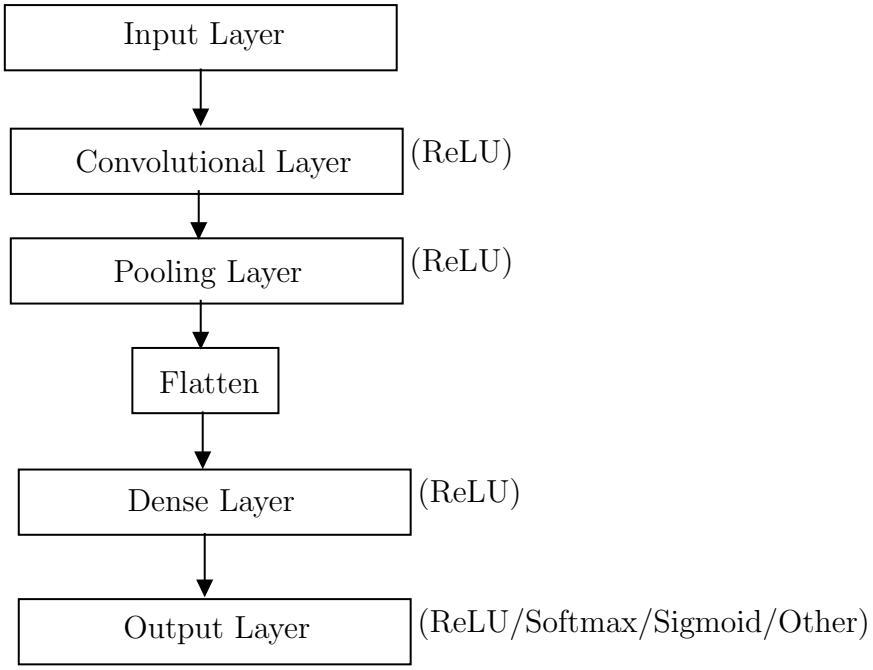


Figure 2.9: Typical Convolutional Neural Network Architecture [1] [2]

2.7.2 U-Nets

An interesting, and highly useful topic of research in ANNs is that of Semantic Segmentation. Besides the detection of objects and the locations of those objects in an image, as is made possible by Convolutional Neural Networks, often it is useful to know which pixels in an image belong to a certain object, and have those pixels highlighted. An architecture that has proven highly effective in this area is a U-Net [3, 33].

Figure 2.10 shows an image of a kitten that has been passed through such a network.

Figure 2.10: Semantic Segmentation: Pixels Belonging to a Kitten [3]

This network architecture is a Fully Convolutional Network - Meaning that its architecture is made up entirely of Convolutional, Pooling and such related layers. U-Nets have been used to great success in areas of biomedicine, agriculture, geography and, most pertinently,

autonomous driving [33].

First brought forward by Ronneburger, Fischer and Brox in 2015, the U-Net was proposed as a more elegant solution to the Image Segmentation problem than a more typical solution: the sliding window CNN. Their main argument was that a U-Net would take fewer images to train, be able to work through the data faster, and is able to both obtain good localisation and make use of context simultaneously - all of which the sliding CNN solution was not capable of [4].

Figure 2.11 shows the typical architecture of a U-Net. A U-Net consists of three main phases: the convolution phase, bottleneck phase, and deconvolution stage. These phases can also be portrayed as the "Downsampling Path" and "Upsampling Path" - which gives the network its u-like architecture, and hence its name [4].

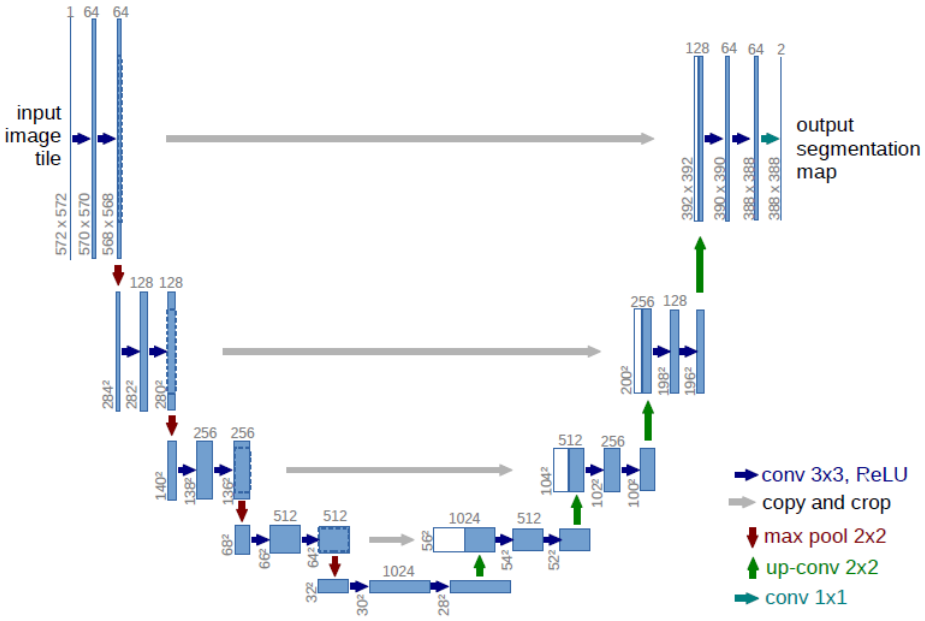


Figure 2.11: U-Net Architecture [4]

The "Downsampling Path" consists of a few "convolutional blocks". Each convolutional block contains two convolutions with kernels of size 3×3. These two convolutions are then followed by a pooling layer of filter size 2×2 in order to downsample the output of these convolutions. The corresponding feature maps are fed as input to the next convolutional block, and the input to the current convolutional block is copied to the correspondig "Upsampling Path" convolutional block. At each new convolutional block, the number of feature channels specified for the convolutional layers are doubled [4].

Similarly, the "Upsampling Path" also contains so-called convolutional blocks. The data copied from their corresponding "Downsampling Path" blocks' input serves as context inputs to these convolutional blocks. These blocks consist of two 3×3 kernel convolutional layers, followed by a 2×2 "up-sampling" convolutional layer. After each block, the number of feature map channels for the convolutional layers are halved [4].

The "up-sampling" convolutional layers are in fact **transposed convolutional** layers. Transposed Convolution takes a lower resolution input volume and up-samples it to a higher-resolution output volume. A transposed convolution takes place when the output of a convolution operation, as well as the kernel used to achive that output, is reused. The convolved output, or feature matrix, is flattened to a $1 \times n$ vector, and the kernel is padded to become an $n \times m$ matrix, and then transposed. This transposed matrix is then multiplied with the flattened feature map vector, and the resulting input is the expanded approximation of the original input [33, 34].

2.7.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs), and their cousins, Long Short Term Memory Units, are Artificial Neural Networks that take into consideration time and sequence of data that is input. The Tensors of these networks, besides having dimensions for batches and the data dimensions, these networks also have a temporal dimension. Recurrent Neural Networks can be applied to image data - with the images presented as "data patches" occuring through time [35].

In order for a time-capable ANN to be able to make temporally influenced predictions, it has to take as input both the current input data, as well as previous inputs. Jeffrey Elman, in his 1990 paper proposing an RNN for the first time, described a previously suggested solution to the time-dimension problem. This solution involved a neural network having recurrent connections from a static pattern to a serially ordered output pattern in order to connect the "Plan" (the static pattern) to the "Action" (the output pattern). This meant that a network could have access to its previous outputs and predictions on the current input dataset, and allow that to influence its current predictions. Elman [36] took it a step further - by suggesting that a second input layer, consisting of "context units", would be used to allow the network to temporally associate data [36].

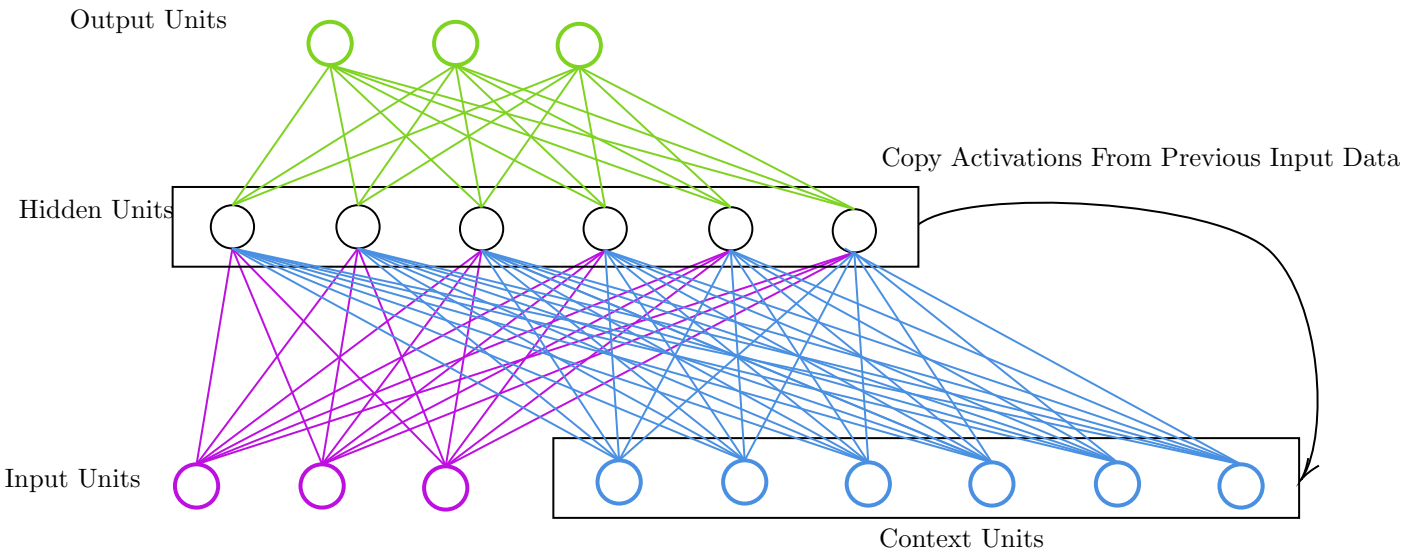


Figure 2.12: Simple Illustration of RNN Flow

Figure 2.12 is an illustration of a simple two-layer densely connected network with an RNN flow. The output, or predictions made by, and RNN are affected by both the present and the recent past. This means that a prediction or an activation generated at timestep $t-1$ will influence the prediction or activations generated at time t. The **activations**, or internal representations of data input to the network just previously, are copied to the context layer as-is. This is then fed in again to the network along with the current input, and representations and predictions are generated using these inputs. Once again, at the same time, these newly generated representations are copied to the context layer to be fed in with the next set of input data [35, 36].

Since RNNs are designed to make predictions on a temporal array of input values $x^{(1)}...x^{(T)}$ [2], it stands to reason that RNNs can be used with an array of sequential image data. Therefore, this network is considered to be of interest to this problem.

2.7.4 3-D Convolutional Neural Networks

Another way to incorporate the temporal dimension with Image processing in Convolutional Neural Networks is by way of a 3-Dimensional Convolutional Neural Network.

A 3-Dimensional Convolutional Neural Network, or 3D-CNN, is similar in structure and function to the 2D-CNN, or standard Convolutional Neural Network previously discussed, in that it consists of convolutional layers, pooling layers, and densely connected layers. This network architecture similarly uses the convolutional operation in its convolutional layers - in 3 dimensions [37]:

$$S(i,j,k) = \Sigma_m \Sigma_n \Sigma_l I(i+m, j+n, k+l) K(m,n,l), \tag{2.14}$$

where S represents the output of this convolution operation, I the input data, and K the 3-D Kernel.

Typically, in video processing, the temporal image data is processed on a bag-by-bag basis (this terminology is used in Natural Language processing, where a set of randomly selected words or phrases to be vectorized is known as a **bag**). Each bag is a short clip of video, or a small set of temporally linked images. These bags can be processed by way of 3DCNNs in one of three main ways, each with different stages of **fusion** [5].

Fusion refers to the joining of image bags into one processing stream - so that each image subset shares the same parameters. There are three main ways of doing this, each with their own benefits and drawbacks for their relevant context [5]:

Single-Frame Architecture

Single-Frame Architectures are dicussed as a comparison point. Single-Frame 3DCNNs process temporal data frame by frame, with the data not being split into bags before hand. This architecture is a standard CNN architecture - with convolutional layers, pooling layers and a dense output layer with some activation [5].

Early Fusion

Early Fusion combines the temporal information immediately on the pixel level - i.e. the bag of images is treated as one single, massive, image. This is done by modifying the input units (Tensors) of an otherwise normal 3DCNN by extending them to receive inputs of size $mxnxT$ pixels (colour channels are disregarded for now), with T being equal to some elapse of time. T is often measured in milliseconds [5].

The benefit of Early Fusion is that precision and processing speed is increased - since the bags of images are not processed separately (i.e., not more than one convolution is taking place simultaneously), and the linear processing of the bags allows for a focus on precise local detections of motion and other features [5].

Late Fusion

Late Fusion occurs when two or more bags of images are processed separately, and then have the resulting activations fused together at the output layer. This method means that two or more convolutional operations are taking place simultaneously. While this greatly slows down processing times, this does provide global context to the analysis of the images in the clip/set - meaning that more context-driven classifications and decisions can be made [5].

Slow Fusion

Slow Fusion is a marriage of Early and Late Fusion. This fusion scheme is carried out so that the later convolutional layers get more and more global context data to work with. The early layers combine the bags at the pixel level, such as early fusion. However, later on, the feature maps are split into two processing streams - which introduces progressively more context into the network's predictions [5].

Figure 2.13 below shows a visual comparison between these three Fusion Methodologies:

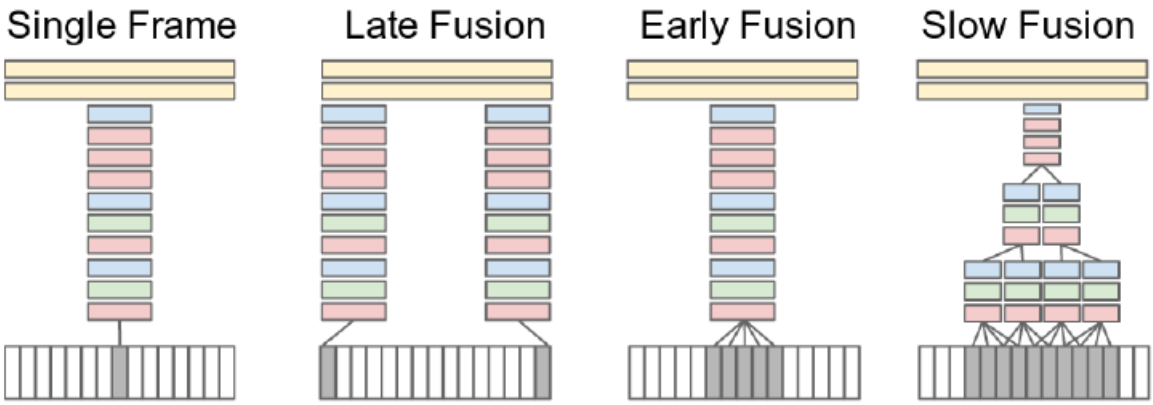


Figure 2.13: Fusion Types Comparison [5]

Chapter 3

Design Problem, Methodology, and Language

3.1 Introduction

This chapter deals with the design process followed and the tests conducted to design and implement a Neural Network Pilot that is capable of generalistion. The evaluation of the final design is qualitative, and will take place in Chapter 5 as part of the Implementation Process. This Pilot will be designed to react to input images (of a track and environment) with driving signals. These driving signals are floating point numbers between the values of -1 and 1, which are interpreted as Pulse Width Modulation (PWM) signals in a physical system. These driving signals are also interpreted as player movement signals in the Unity Engine-based Donkey Car Simulator. Due to restrictions brought about by the ongoing COVID-19 outbreak in 2020, the Donkey Car Simulator environment shall be used as both a simulation environment and a final implementation environment.

The design methodology followed in the development of a Neural Network Pilot is an iterative and experimental process. The design process laid out and explained in Section 3.3. This section lays out an integrated design method for Machine Learning (ML) models that aims to satisfy the Graduate Attribute (G.A.) 3 requirements for this module - Engineering Design.

The main end-goal of the project, as defined in Section 3.2, is to produce a Neural Network Pilot that is able to generalise to and perform well in spite of changes in environment, location, lighting and track topology. To this end, Feature Engineering in the form of extracting lane line features and dulling non-essential features in input images is investigated and experimented with. Additionally, the effects of diversifying data fed to the model (using datasets from different environments) is also explored.

The model architecture of choice for the NNA is a Convolutional Neural Network (CNN), which was a pioneering NNA in image processing [1]. Additionally, the focus of the project is to promote the generalisation of a generic ML pilot. Additional architectures, such as a

3D-CNN, or a Recurrent Neural Network (RNN), which are discussed in Chapter 2 are also valid solutions. However, for the purposes and scope of this project, an end-to-end design of a CNN with an additional generalisation method is sufficient.

3.2 Design Problem Definition

The design problem can be defined as follows: Create and train a Neural Network Architecture (NNA) as part of a Pilot that is able to take as input images of a track and output driving signals to be interpreted by an autonomous car. This Pilot must be robust to changes in lighting, environment, and track shape (referring to the number of corners, the 'aggression', or steepness of said corners, as well as the overall length of straight track portions and the total length of the track itself), with track markings remaining mostly consistent.

This statement can be broken down even further to mean the following:

- Incoming images (of a certain shape/resolution) must be processed and important features extracted - so a means of accomplishing this - typically a CNN - must be designed and implemented.
- These images and their features must be mapped to desired outputs using a Neural Network - so the aforementioned CNN must have dense layers added to it, and the resulting architecture be trained and tested.
- The Pilot in question must be able to robustly adapt to environmental changes. Therefore, means of generalizing the Pilot's response to input data must be developed, tested and adapted.

3.3 Design Method Clarification

Since ML is a unique and emerging field the design of Neural Networks depends heavily on iteration, experimentation and continuous testing. One will need to establish and clarify design methods and conventions that will be used throughout this project. This is in an effort to bring together the iterative 'product-first' and waterfall 'big design up front' methodologies of the ML and engineering worlds. This will also serve as a justification for many of the methods that will need to be used in the development of this solution, as well as a confirmation that there is indeed a firm understanding of Engineering Design (as demanded by G.A. 3) present.

3.3.1 Design Methodology

Neural Networks are mathematical models that have already gone under many years of research and development to deliver a standardised prediction and classification tool. The nature of Neural Networks is also highly experimental - meaning that a traditional, theory- and mathematics-based design would both be redundant and highly time-consuming. Therefore, designing a good solution to a ML problem involves applying known solutions well. Both

Ian Goodfellow, in his book "Deep Learning", and Francois Chollet, in "Deep Learning with Python", propose working methodologies to design well thought-through machine learning solutions [1, 2]. The recommendations of these two authors and developers will be taken into consideration during the design of this solution.

Goodfellow lists the following steps when designing an ML model or NNA [2]:

- Determine the Goal (what performance metrics are important, what performance is required according to these metrics).
- Establish working end-to-end solution process, or pipeline, early on.
- Instrumentalize the pipeline (determine how performance "bottlenecks" will be measured. Set up a method with which to diagnose and fix problems).
- Iterate through a process and incrementally make changes to the model (collecting new data, changing algorithms, adjusting parameters - based on findings from previous step).

This process detailed above closely emulates the methodology that has been developed for use in this project. The design methodology will be better showcased through implementation, but it may be outlined as follows:

- Problem Outset:
  - Problem and Context Definition.
  - Choose Performance Metrics.
  - Create a high-level functional flow diagram of the solution based on problem definition, context, and performance metric. (i.e., how will the model fit into the rest of the system?)
- Model Pre-Preparation:
  - Determine data type and other characteristics of available data.
  - Find typical Model Architectures used in similar problem contexts.
  - Define "functional blocks" consisting of network architectures and layer types that will help solve the problem.
  - Create a set of Functional Unit diagrams based on defined blocks and available information.
  - Create a shortlist of Optimization and Activation Functions that may be used.
- End-To-End Pipeline Establishment:
  - Create experimental networks (end-to-end working pipelines) based on the Functional Unit Diagrams created in the previous step.
  - For each of these experimental networks, create new Functional Unit diagrams in a higher level of detail.
  - Experiment with the base network (add/reduce layers, change layer content, tweak optimization functions) until statistical power is reached (the ML model does better than a random guess).
  - Narrow down on optimization functions and loss functions based on performance.
  - Record experiments and selections made in the final base network architecture.
- Instrumentalisation:

- Continue improving on the networks' performance developed in the Pipeline Establishment step until the model begins to overfit to the data.
- Introduce regularization techniques to correct for this overfitting. Record this process and its results.
- Select/design a final optimization function and loss function based on insights if necessary.
- Narrow down network architectures based on performance metrics and apply these selections in a simulated environment.
- Select best performing networks for testing and implementation in a practical or simulated environment. Justifiy choices based on collected experimental data and design intuition.

These above steps were developed as a fusion between the steps suggested in Francois Chollet's "Deep Learning with Python" and Ian Goodfellow's recommendations. The addition of using functional unit diagrams is meant to appeal to a better understanding between the ML and Systems Engineering worlds in this project.

3.3.2 Definition of Design Language

A unique design language that incorporates all the above steps well will now be defined for use in this project. This design language is a combination of Functional Flow Diagrams, Functional Unit Diagrams, Unified Modelling Language, and additional features that will be defined uniquely to this document.

The first set of concepts to be defined are the 'building blocks' of a solution - or what constitutes a complete Artificial Neural Network Architecture or ML model. An ANN can be broken down into the following singular components:

1. Layers
2. Activation Functions
3. Modifiers (such as dropout)
4. Hyper-parameters
5. Optimizers
6. Cost Functions

These components are structural and procedural parts that have an impact on the ANN's use, performance, and type. For example, if a feature detection network that places images into categories is desired, the Network's architecture will be made up of Convolutional Layers, Dense Layers and ReLU Activation functions. It will also have a Categorical Cross-Entropy Cost Function and most likely a Stochastic Gradient Descent (SGD) Optimizer. Modifiers such as dropout may be optionally included depending on the problem, and a different Optimizer may be used, depending on experimental performance.

The first step in designing a network would be to put together a 'Network Block' - which illustrates the most basic information one would know about the problem before any experi-

mentation takes place.

Name	Name (FU x.0)
Type	(Architecture)
Mapping	e.g Categorical
Input	Datatype, (shape)
Output	Datatype, (shape)
Cost Func.	e.g MSE
Optimizer	e.g SGD

Figure 3.1: Network Block

A 'Network Block', or a high-level description of a Network Architecture, will appear as in Figure 3.1. The Network Block declares the type of Network Architecture being designed, the type of output mapping (Regression or Categorical Network), specifies the input (Type and Shape), the output (type and shape) and declares the loss function and optimizer initially chosen. The Network Block also contains a name and a functional unit number - which will allow it to be used as part of a Functional Unit Diagram.

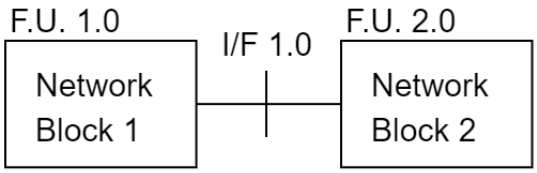


Figure 3.2: Functional Unit Diagram

Figure 3.2 shows an example of a traditional Functional Unit Diagram used to describe the immediate system that the ML Model will fit into when deployed. Each functional unit has its own number, and each connecting unit has a numbered interface. In the case of connecting two Network Blocks to each other, the interface will be defined by the output data type and shape of the first Network Block and by the input data type and shape of the second Network Block, which should correspond. Each Network Block, when defined on their own, should contain their functional unit numbers in their names - even if they are a stand-alone Network architecture that will not necessarily be immediately integrated into a larger system.

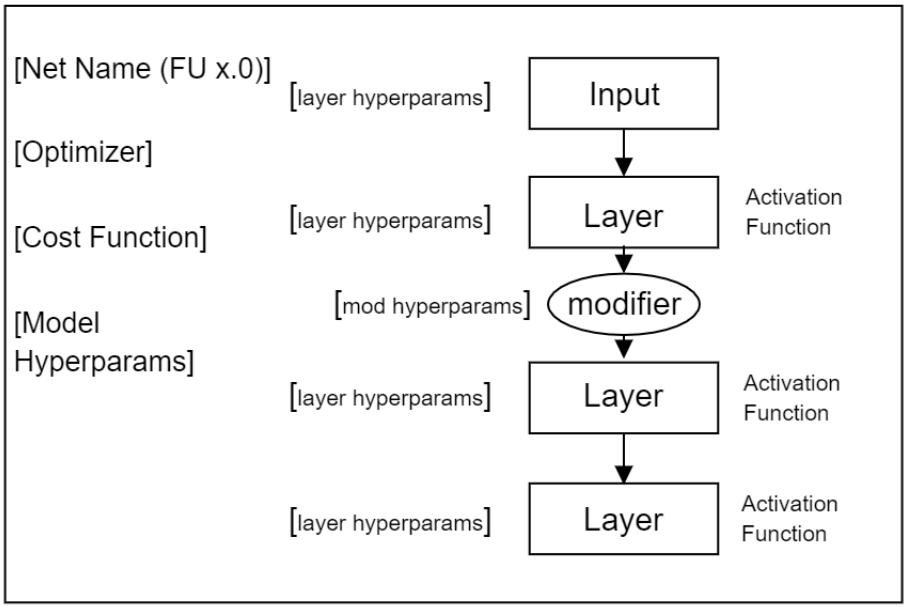


Figure 3.3: Architecture Diagram

The Architecture Diagram, pictured in Figure 3.3, is one detail level lower than the Network Block. The Architecture Diagram depicts the layers of the Network as well as other components. Modifiers, such as dropout and normalization layers, are also included in the diagram. Each layer is identified by its number and layer type. On the right hand side of each layer, the activation function used for that layer must be specified. Similarly, on the left hand side the size of the layer the number of neurons/nodes, kernel size of a convolutional layer, or other hyperparameters must be specified. The Architecture Diagram once again contains the name and Functional Unit number used for the network block. On the far left of the Architecture Diagram, the Optimizer, Loss Function and other hyperparameters used are listed.

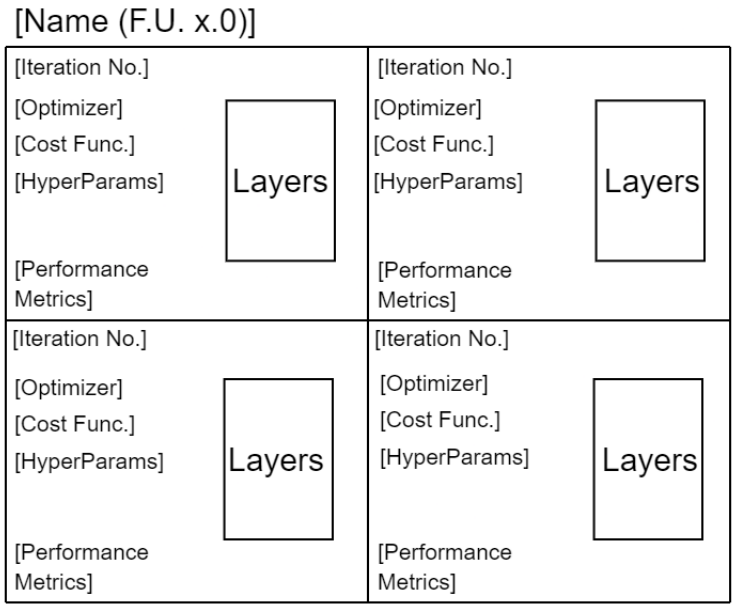


Figure 3.4: Evolutional Grid

Since iterative implementation, experimentation and re-implementation is integral to the design process of ANNs, an Evolutional Grid as depicted in Figure 3.4 is used to document the design process. An evolutional grid can have any number of one or more cells, depending on the number of iterations it took to arrive at a satisfactory design. An Evolutional Grid will appear with two columns and one or more rows. This design tool is optional, depending on the amount of experiments and iterations it took to arrive at a satisfactory solution.

In each cell of the Evolutional Grid, an Architectural Diagram for that iteration of the Network will appear. In the top left-hand corner of the cell, the iteration number, hyperparameters, optimizer and loss function used will be listed. In the bottom left-hand corner, the metrics used, such as validation accuracy, average loss, etc. will be listed. As is with the Architectural Diagram, layer sizes, activation functions and layer-specific hyperparameters will be listed.

Supplementary to or instead of the Evolutional Grid, dashboards and graphs from experiments can be included and appropriately referenced to aid in the documentation of the design process.

3.3.3 Some Detail of Design, Experimentation, and Implementation Tools

The API of choice with which to design and implement the Pilot is the Keras API. Keras provides the user access to a variety of pre-built layer types, optimization functions, activation functions, and training loops. This allows for rapid prototyping and implementation - and provides for a deeper and/or wider solution due to the abstraction of what would otherwise be a struggle with nit-pickingly small errors and troubleshooting.

However, beyond these levels of abstraction, Keras allows user finer usage and control over the Tensorflow backend via its Functional API [38].

This usage and control includes the ability to create and use custom layers, custom activation functions, custom optimization functions, and much more. It also enables a non-sequential approach to Architecture design - providing the capabilities to create multi-input multi-output (MIMO) models, and even call previously trained models as layers in a new model.

Custom layers may also be implemented with non-trainable weights to accomplish any number of functions. This has many useful applications, such as re-using a pre-trained feature extractor, or having a layer dedicated to performing a traditional canny edge detection algorithm. More complex configurations for a built-in lane detector/feature extractor for a model may also be implemented [38]. This opens a wide range of interesting topics for further work.

The second tool of importance is Google Colab - a remotely-hosted Jupyter notebook system. These notebooks will house each of the architectures developed throughout the design process - as well as descriptions and commentary of the actions taken. This also speeds up the experimentation process - since each of these notebooks are hosted on a GPU-enabled virtual machine with plenty of RAM, which drastically reduces model training times.

The final tool to be discussed is Neptune.ai - which hosts and stores the results and logs of Machine Learning experiments. Links to these online experiments are found in tables and throughout the report for reference.

Chapter 4

Experimental Design

4.1 Data and Environment

4.1.1 Data Collection

Data was collected via driving a simulated version of the Donkey Car in the Unity-Based Donkey Car Simulator, otherwise known as the Donkey Gym. This allowed the Donkey Car system to take photos of a Unity-Simulated 3D world. The data collected consisted of RGB images of the simulated tracks and their environments. The images collected were 160x120 pixels - and were the same size and quality as if they were collected via the physical Raspberry Pi camera equipped to the Donkey Car. Data was not able to be collected from physical environments due to the Covid-19 outbreak and national lockdowns. Each image collected had an associated json record of driving data linked to it.

The json records associated with each image contain the time in milliseconds since the start of data collection that the image taken, the relative angle associated with the image, and the relative throttle value associated with the image. Images and json records are associated with each other by their names - an image and json file will contain the same number in their filename.

4.1.2 Data Details and Handling

The total dataset used throughout this project consisted of three sub-datasets: collected from the Generated Road Environment, Generated Track Environment, and Warehouse scene in the Donkey Car Simulator respectively. Each of these scenes vary in lighting, complexity, and in some cases, lane line type. Figure 4.1 below shows the three simulated environments side-by-side:

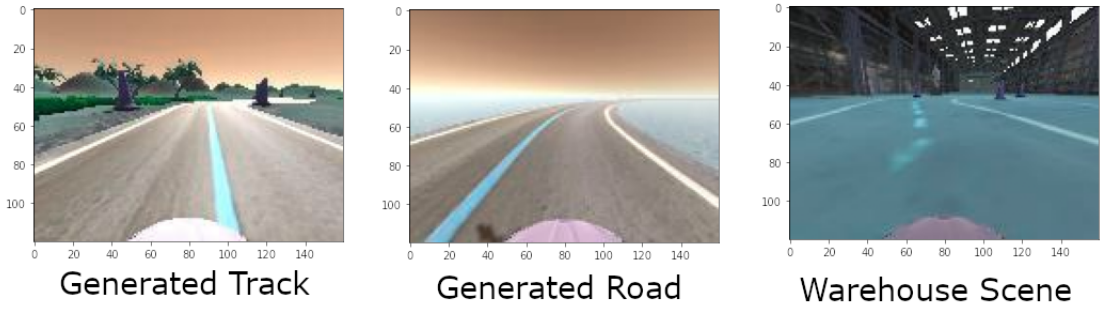


Figure 4.1: Dataset Images Examples

The full dataset consists of 41 155 images. For each sub-dataset (now referred to as "datasets"), Table 4.1 shows the total number of images:

Table 4.1: Number of Images per Dataset

Dataset	**Images Count**
Generated Track	15 995
Generated Road	17 100
Warehouse Scene	13 060

In order to effectively train a model, a good training dataset with a representative validation set must be generated. This is accomplished by reading the images for the desired dataset into 3D arrays, with the angle and throttle values read into respective 1-D arrays. The dataset arrays are sorted according to the numbering of the filenames, to ensure that the correct record values (angle and throttle values) correspond with the correct images. Once the image and record arrays are ordered, images and their corresponding record values are removed at random from the main dataset array and populate validation set arrays according to the proportion of validation data required. Figure 4.2 shows the flowchart of this process:

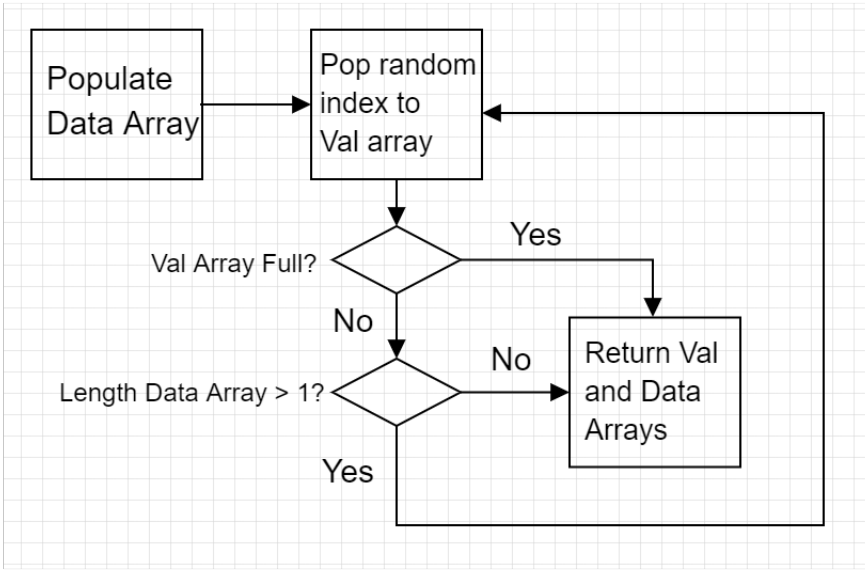


Figure 4.2: Validation and Training Data Split Process

4.1.3 Data Analysis and Exploration

Beyond the datasets consisting of images with corresponding floating point labels, additional characteristics may be defined. Shannon Entropy was developed by Claude Shannon in 1948 as a means to measure the "amount of information" present in a sequence [39, 40]. The higher the entropy, the higher the information content. In image processing, Shannon's entropy is sometimes used to assist compression algorithms, in this case it will be used to determine the relative "complexity" of the grayscale versions of images used in the datasets.

Shannon's Entropy is calculated via the following [41]:

$$S = \sum p(x) * \log_2 p(x), \tag{4.1}$$

where $p(x)$ refers to the probability of a value x occurring in a probability distribution p. In the case of images, and of the Python module used (skimage), $p(x)$ refers to the probability of pixel value x occurring in the input greyscale image.

The greyscale images used in each dataset have pixel values that range from 0 to 255, meaning that they are 8 bit images. In this case, a Shannon entropy of 0 for an image will indicate a low pixel variation, while a Shannon entropy of 8 would indicate a very high pixel variation.

Table 4.2 shows the average Shannon entropy for the greyscale images of each dataset.

Table 4.2: Shannon Entropy Values per Dataset

Dataset	Mean	Mode	Max	Standard Dev
Generated Track	7.033191	7.443463	7.6027517	0.25775644
Generated Road	6.9513736	6.924553	7.1446533	0.09255418
Warehouse Scene	6.423074	6.5263515	7.0427766	0.16749667

Along with the images in each dataset, throttle and angle data is provided. The throttle data collected for each dataset remained constant at 5 % of top speed throughout - so an analysis will not take place on the throttle data. However, the angle data for each dataset will be analysed for its mean, mode, and standard deviation. Histograms of the angle data for each dataset will also be drawn up. This angle data may be biased to contain left turns, right turns, or be straight ahead, also depending on track layout. This was not addressed in the generalisation process of this project. However, taking angle biases into consideration would be useful in future attempts at generalisation.

Table 4.3 shows the relevant statistics for the angle data of each dataset - where -1.0 indicates a complete left turn and 1.0 a full right turn.

Table 4.3: Statistics of Angle Data per Dataset

Dataset	Mean	Mode	Max	Min	Standard Dev
Generated Track	0.10230716	1.5308085e-16	1.0	-0.963338	0.20399229
Generated Road	0.0009642312	1.5308085e-16	0.3568837	-0.49260035	0.13310972
Warehouse Scene	0.047873676	1.5308085e-16	0.7978186	-0.8569053	0.1599931

Figure 4.3 below shows the histograms of the angle data per each dataset:

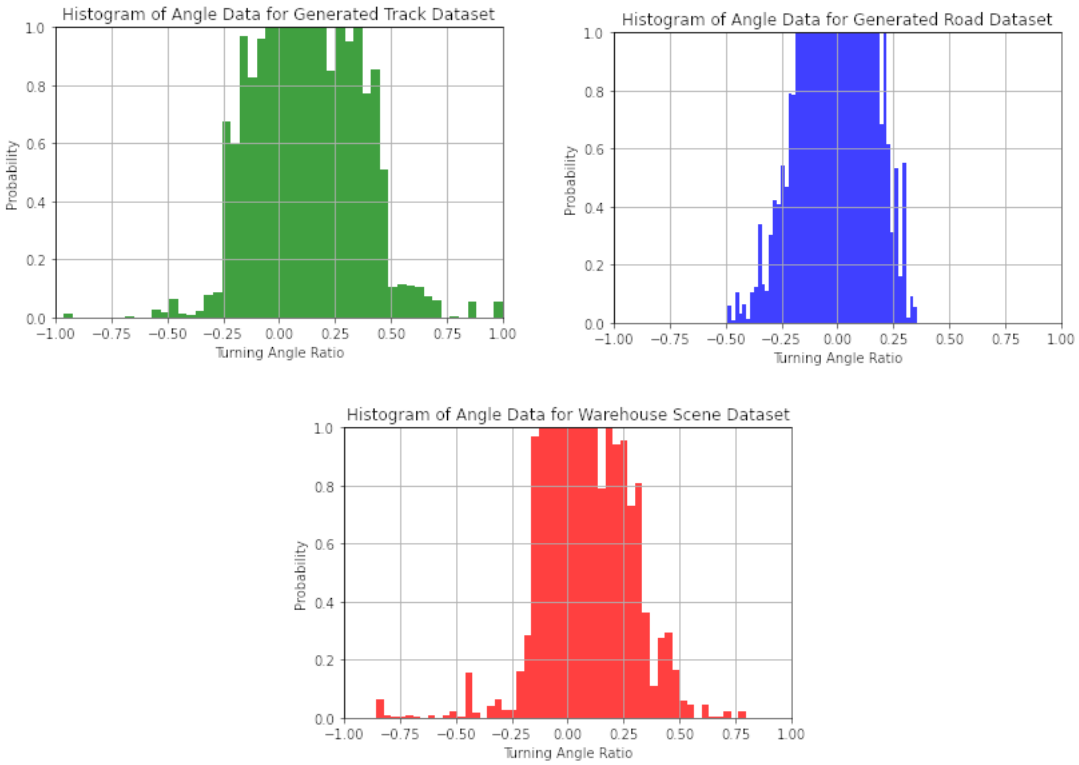


Figure 4.3: Angle Value Histograms Per Dataset

4.2 Performance Goals

The metrics that will be used are the model's R-squared or R2 score, the Mean Absolute Error (MAE) and the Root Mean Squared Error (RMSE) of the predictions. Additionally, the Keras' internal metrics, such as Validation Loss, will be used to compare model evolutions with one another. Additionally, the average deviation of the model will be used, which is calculated as follows:

$$\%dev = \frac{y_{true} - y_{pred}}{y_{true}}, \tag{4.2}$$

$$\%dev_{avg} = \frac{\Sigma \%dev}{N}, \tag{4.3}$$

where y_{true} depicts the expected output, y_{pred} the model's prediction, and N the total number of predictions.

The R2 score, or coefficient of determination, is used to determine what amount of variance in the dependent variable (predictions) is caused by variation in a particular input variable (the image data). An R2 score close to 0 indicates that the predictions are randomly caused, and an R2 score closer to 1 (or 100%) indicates a complete dependence on the input variable - i.e., a model with a higher R2 is the "better" model. The R2 score is calculated as follows [42]:

$$R2 = \frac{\Sigma(y_{true} - y_{pred})}{\Sigma(y_{true} - y_{mean})}, \tag{4.4}$$

where y_{mean} is the mean of the true labels.

The angle data that ranges in floating point values from -1 to 1 are essentially values that indicate 100 % left or 100 % right. This means that metrics such as the MAE and RMSE may be expressed as percentages themselves. Internal metrics - both validation and training loss - shall also be considered and expressed as percentages. These losses depend on user configuration at implementation/design.

Table below shows the minimum metric requirements, or performance goals, for each metric:

Table 4.4: Metric Requirements/Performance Goals

Metric	**Value**
MAE	0.05
RMSE	0.05
% Deviation (%)	5
Prediction Accuracy (%)	5
R2	0.95
Validation Loss	0.05

Each of the minimum metric requirements are set to 5 %, since that is a standard manufacturing tolerance used. Additionally - this vehicle is a scale vehicle that will also mostly be implemented virtually - so safety issues will not be caused by this vehicle veering too far from the intended steering. Additionally - small deviations in steering at least up to 5 % of the intended angle are likely to be corrected by following actions of the model, as the model is continually making predictions and sending steering signals to the vehicle. In short - the metrics of this model's base performance are quite subjective. The goal of the project is to get consistent performance over several environments once a suitable model is developed.

These minimum metric requirements are only a guideline for what is base-line acceptable: throughout the design and development of the models it will be a priority to make these metrics as good as possible (sometimes even lower than 1% if possible). The best model results will be picked from those models that perform better than the minimum requirements, but models performing poorer than minimum requirements in one or more requirements will be rejected. Internal metrics such as Validation Loss are computed both during training after every completed epoch. The value computed at the last saved epoch is taken for consideration. Average % Deviation, Prediction Accuracy, RMSE, MAE and R2 are computed with a Python script after an evaluation has been run on the best saved epoch of the model.

4.3 Base Network Evolution

Links to the relevant experiments are included throughout the text and in Table 4.5 below:

Table 4.5: Links to Neptune Experiments Conducted

Experiment Code	**Experiment Name**	**Link**
LIT-136	Base CNN Build	https://ui.neptune.ai/charag/Littlefoot/e/LIT-136
LIT-138	Base CNN Train	https://ui.neptune.ai/charag/Littlefoot/e/LIT-138
LIT-173	ROI Evaluation	https://ui.neptune.ai/charag/Littlefoot/e/LIT-173
LIT-175	Threshold Evaluation	https://ui.neptune.ai/charag/Littlefoot/e/LIT-175
LIT-181	CED Evaluation	https://ui.neptune.ai/charag/Littlefoot/e/LIT-181
LIT-185	Diverse Datasets	https://ui.neptune.ai/charag/Littlefoot/e/LIT-185
LIT-186	Diverse Datasets	https://ui.neptune.ai/charag/Littlefoot/e/LIT-186

During this design process, over 200 experiments were run. These experiments were run after changing the amount of layers in the architecture, the types of the layers, and the layer sizes. Other changes included redoing the method of splitting data into training and validation sets, changing optimizers, batch sizes, and epochs. After running each experiment, losses and accuracies throughout training were logged, and an evaluation was run on the best-performing version of that model. The metrics according to the performance goals are recorded for comparison. Highlighted experiments will be shared in this document.

After completion of all experimental rounds, the following model was produced:

Name	Base CNN
Type	2D - CNN
Mapping	Regression
Input	Float, (120,160,3)
Output	Float, (1)
Cost Func.	MSE
Optimizer	ADAM

Figure 4.4: Base CNN Network Block

This model, whose architecture is shown in Figure 4.5 was trained over 300 epochs, with Keras Earlystopping enabled. This meant that if the validation loss of the model did not improve over a certain number of epochs (in this case 20 epochs), the training would be cancelled. The training loop was also set up to only save the best version of the model - or the model at each new best validation loss (since validation loss was set as the training metric to measure).

This model began with two convolutional layers and a single dense layer - this architecture was gradually built up in order to decrease the validation loss as much as possible. Once the validation loss began to show significant progress - pruning methods were introduced. The pruning method used in this case was Dropout - a layer which will randomly cull a certain percentage of neurons on each training loop. Many different Dropout rates, arrangements and quantities of Dropout layers were tested until the best configuration was found.

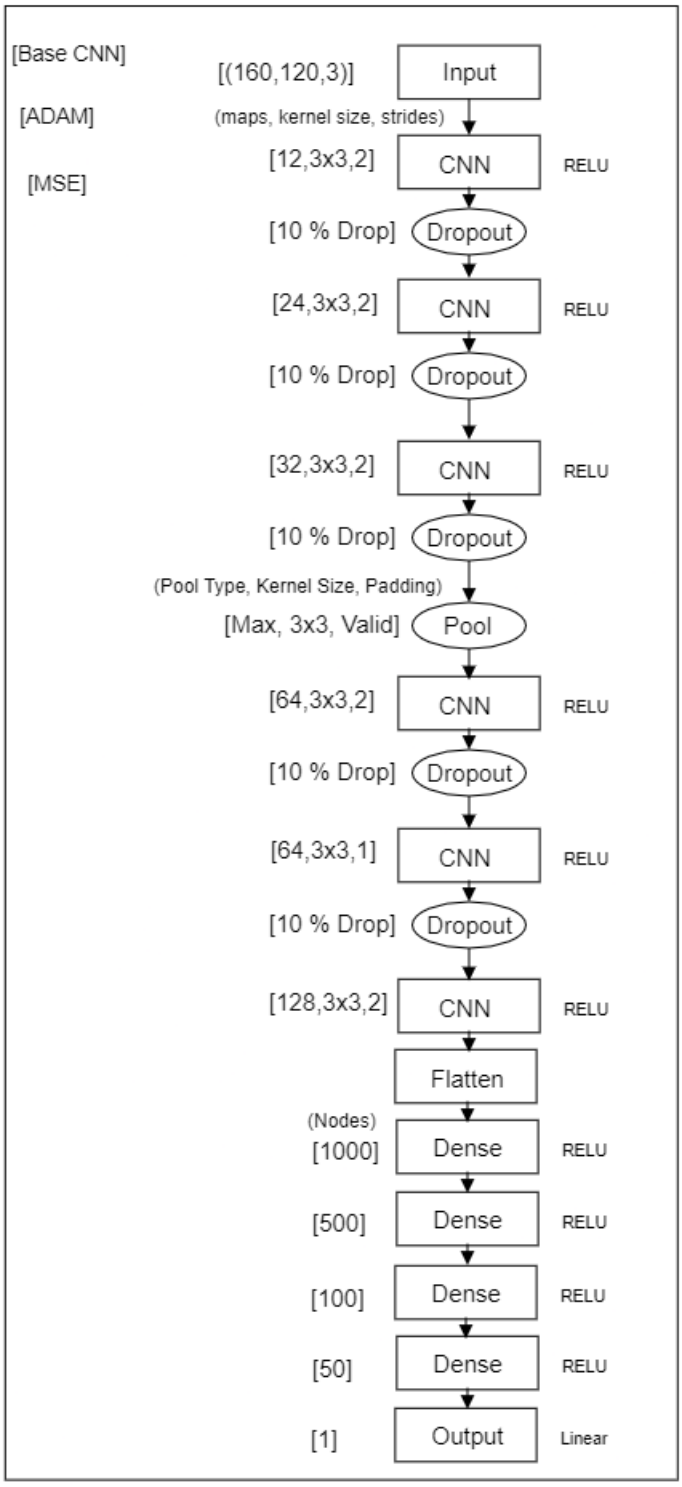


Figure 4.5: Base CNN Architecture Diagram

The final statistics of the model may be found in Experiment LIT-272 on Neptune. (https://ui.neptune.ai/charag/Littlefoot/e/LIT-272)

Table 4.6 provides the relevant parameters and values of performance metrics for the final Base CNN architecture (trained and evaluated using data from the Generated Track):

Table 4.6: Base CNN Characteristics and Metrics

Characteristic/Metric	Value
Convolutional Layers:	6
Dense Layers:	4
Activation Used (All Layers):	RELU
Output Activation:	Linear
Optimizer:	Adam
Loss Function:	Mean Squared Error
Pruning:	10% Dropout after every Conv Layer
Validation Loss:	0.16%
Prediction Accuracy:	98.16%
Average Percentage Deviation:	1.83%
R2 Score:	0.9613
RMSE:	4.06 %
MAE:	1.83 %

The final model in the Base CNN experimentation set has an average deviation of 1.1 % when dealing with validation data extracted from the dataset used to train it. None of these validation images formed part of the training data - however they were from the same set. This indicates good performance in similar environments - i.e. environments with similar lighting, background, and track shape.

Table 4.7 shows some results of the build-up experiments. Only highlights are included in this table for the sake of brevity.

Table 4.7: Chronological Evolution Table of Base CNN - Highlights

Exp. ID	Conv. Layers	Dense Layers	Dropout	Val. Loss	Pred. Accuracy	Avg Deviation
LIT-59	2	1	-	49.15 %	- %	-
LIT-73	3	2	-	11.27 %	75.7 %	24.3 %
LIT-75	3	3	-	3.69 %	- %	-
LIT-77	4	3	-	2.18 %	92.75 %	7.25 %
LIT-100	5	3	-	2.65 %	- %	-
LIT-136	6	4	10 %	1.29 %	95.58 %	4.42 %

These experiments were initially logged in notes recorded in Roam Research (links in Table 4.8 below).

Table 4.8: Links to Original Experiment Notes (Roam)

Note Title	Link
Daily Experiments	https://roamresearch.com/#/app/CharaScribbles/page/_Ete_KPKs
Build Regression CNN Experiment	https://roamresearch.com/#/app/CharaScribbles/page/ulyDVTGlk
Build Regression CNN Experiment II	https://roamresearch.com/#/app/CharaScribbles/page/jbZMau7Ss

However, these initial experiments were conducted before the performance metrics were bolstered with the addition of RMSE, MAE, and R2 metrics. Additionally - the data split used to train each of the intermediate models in the experiments was not as optimized as the split described in Figure 4.2, since some labels referred to the wrong images, hobbling the results. The initial Roam Notes are insightful of the process, but from experiment 81 onwards these notes were not recorded in roam with the same measure of detail, instead leaving the details to Neptune - meaning that a majority of the architectures will have to be rebuilt as closely as possible based on clues from Neptune to re-run the experiments with the new metrics.

Experiments 59, 73, 77, 100 and 136 are rebuilt, rerun, and recorded with the new metrics in Table 4.9 below. These experiments are run with a more optimized data split.

Table 4.9: Chronological Evolution Table of Base CNN - Highlights With New Data Split

Exp. ID	Conv. Layers	Dense Layers	Dropout	Val. Loss	Avg. Deviation	RMSE	MAE	R2
LIT-256	2	1	-	0.39 %	3.44 %	6.27 %	3.44 %	0.9061
LIT-260	2	3	-	0.25 %	2.22 %	5.03 %	2.22 %	0.9394
LIT-261	4	3	-	0.17 %	1.8 %	4.13 %	1.8 %	0.9594
LIT-262	5	3	-	0.19 %	2.07 %	4.37 %	2.07 %	0.9533
LIT-265	6	4	-	0.21 %	2.09 %	4.59 %	2.09 %	0.9496
LIT-245	6	4	10 %	0.16 %	2.12 %	3.95 %	2.12 %	0.9615

4.4 Establishing Baselines

In order to determine the performance required from the model that may be declared as 'generalised', a baseline, or 'ignorant', performance must be determined to compare with. This is done by running the Base CNN model, which had been trained on only the Generated

Track dataset, on the Generated Road and Warehouse Scene datasets, which it had not had contact with before that point.

The model's performance shall be measured according to the metrics set up in Section 4.2, and their values recorded.

The model was trained and saved as 'BaseCNN_LIT-138.h5' - this h5 file was stored as an artifact in Neptune Experiment LIT-138 (https://ui.neptune.ai/charag/Littlefoot/e/LIT-138). An evaluation was then run on this model with validation data from the Generated Track Scene, the Generated Road Scene, and the Warehouse scene.

The results of the baseline evaluation are shown in Table 4.10 below.

Table 4.10: Baseline Evaluation Results (LIT-138)

Dataset	Validation Loss	Prediction Accuracy	Average Deviation
Generated Track	0.43 %	97.53 %	2.47 %
Generated Road	31.23 %	61.44 %	38.56 %
Warehouse Scene	36.70 %	60.89 %	39.11 %

A re-run of the baseline test using the model trained in experiment LIT-272 (using the new data split) as well as new metrics yields the following Baseline results in Table 4.11:

Table 4.11: Baseline Evaluation Results (LIT-272)

Dataset	**Val. Loss**	**Pred. Accuracy**	**Avg. Deviation**	**RMSE**	**MAE**	**R2**
Generated Track	0.18 %	98.16 %	1.83 %	4.06 %	1.83 %	0.9613
Generated Road	1.98 %	68.99 %	31.01 %	14.07 %	14.49 %	-0.1186
Warehouse Scene	5.76 %	75.07 %	24.93 %	24 %	19.88 %	- 1.25

Even on arguably similar data - such as the Generated Track with similar lighting and road markings - the model performance is greatly reduced. This shows that the current model, when trained on a given set of data, does not effectively detect lane lines that may be universal (as is attested by its performance on similar lines with similar lighting). It also seems to rely on other queues in the image data besides lane lines - such as the cones next to the road, trees in the background, or other prominent features in the warehouse.

In order to improve this model's performance, Feature Engineering and Data Diversification are explored as generalisation methods.

4.5 Experimenting with Generalisation Methods

4.5.1 The Effects of Feature Engineering on Performance

To determine the performance of a network when using Feature Engineering (FE) in data - the model is first trained using the generated track dataset. Before training the model on this dataset, this data is first passed through a set of FE operations. The data on which this model will be tested will be passed through the same set of operations - as it is assumed that these FE operations will be executed in real-time as the vehicle is driving.

The FE operations used in this trial include a region of interest crop, canny edge detection, and image thresholding. Each one of these feature engineering operations shall be tested on their own and then compared in Table.

These models are tested on the generated road dataset, generated track dataset, and warehouse scene dataset as in the baseline establishment test.

Effects of ROI

An ROI filter, blacking out the top third of the image, was applied to all images in both the training and evaluation datasets. Figures 4.6, 4.7, and 4.8 below show samples from each datasets before and after the ROI filter was applied:

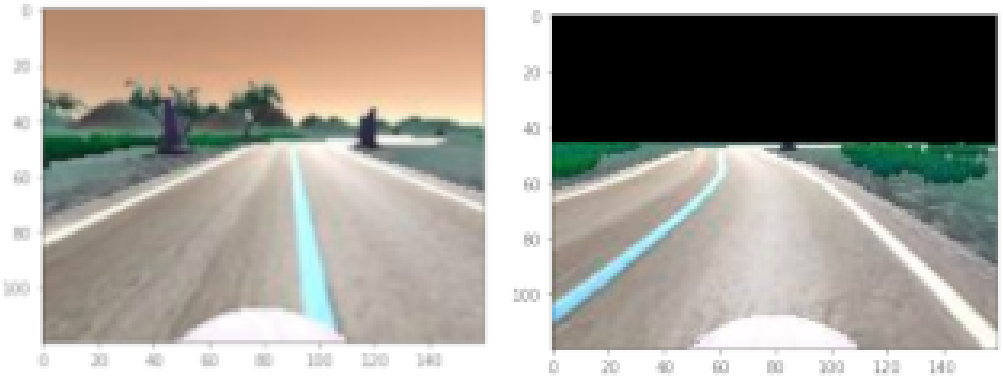


Figure 4.6: Generated Track Dataset Before and After ROI Filter is Applied

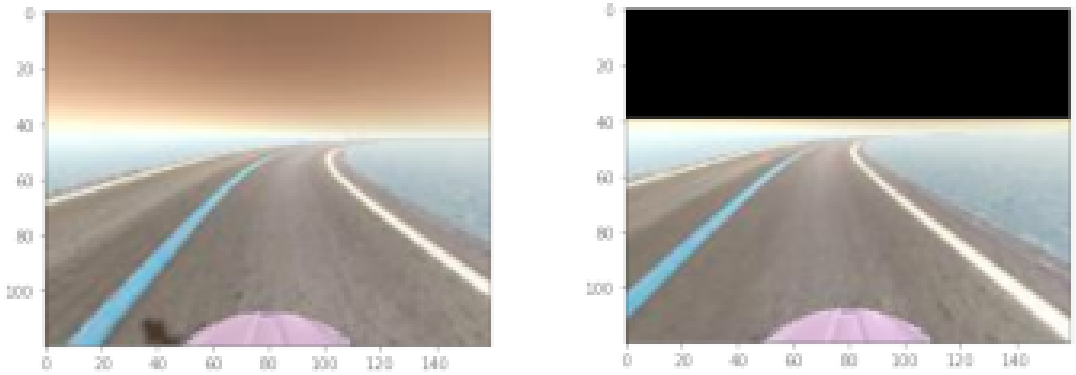


Figure 4.7: Generated Road Dataset Before and After ROI Filter is Applied

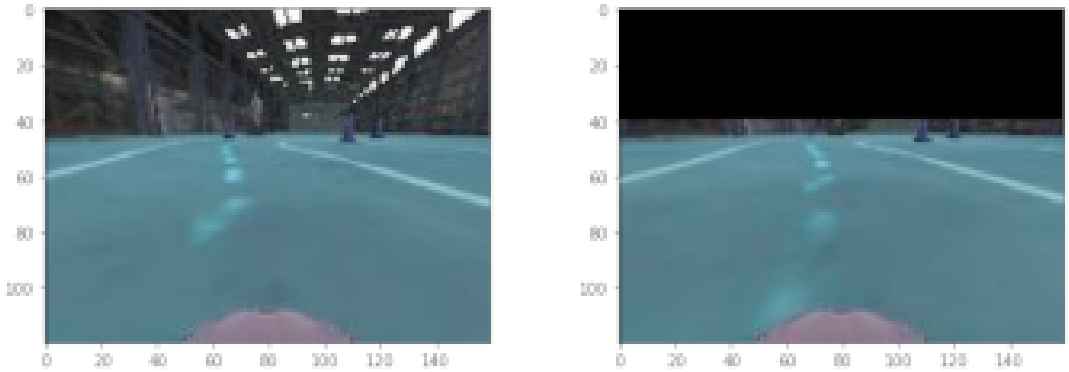


Figure 4.8: Warehouse Scene Dataset Before and After ROI Filter is Applied

Table 4.12 below shows the Shannon Entropy values per dataset after ROI Filtering is applied to the grayscale versions of these images. (Grayscale images used to simplify analysis)

Table 4.12: Shannon Entropy Values per Dataset with ROI-filtering

Dataset	**Mean**	**Mode**	**Max**	**Standard Dev**
Generated Track	4.978903	5.2763643	5.3838797	0.17689756
Generated Road	4.932608	4.8795	5.138629	0.09973487
Warehouse Scene	3.853292	3.8812633	4.4822664	0.15233901

The statistics when training the model with ROI-filtered generated track data is found in Neptune Experiment LIT-277 (https://ui.neptune.ai/charag/Littlefoot/e/LIT-277) and in Table 4.13 below.

Table 4.13: Model Training Overview

Statistic	Value
Loss	0.11 %
Val. Loss	0.22 %
Epochs	140

Table 4.14: ROI Evaluation Results

Dataset	Val. Loss	Pred. Accuracy	Avg. Deviation	RMSE	MAE	R2
Generated Track	0.22 %	97.62 %	2.37 %	4.71 %	2.37 %	0.9463
Generated Road	4.51 %	51.43 %	48.53 %	21.27 %	17.31 %	-1.546
Warehouse Scene	4.44%	80.90 %	19.09 %	20.89 %	15.23 %	- 0.7063

Effects of Thresholding

A threshold filter passes over the image and blacks out all pixels under a certain colour threshold. This enables some form of rudimentary lane detection. In each channel of the RGB image, the thresholding function will black out each pixel under a value of 150 - effectively highlighting the white and yellow lane lines in the image and blacking out everything else.

Figures 4.9, 4.10, 4.11 and below show the training and evaluation data before and after applying the threshold filter.

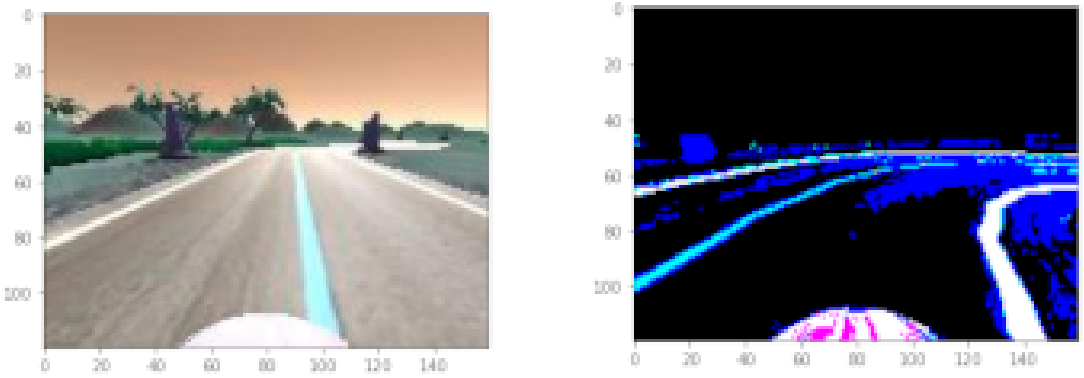


Figure 4.9: Generated Track Dataset Before and After Threshold is Applied

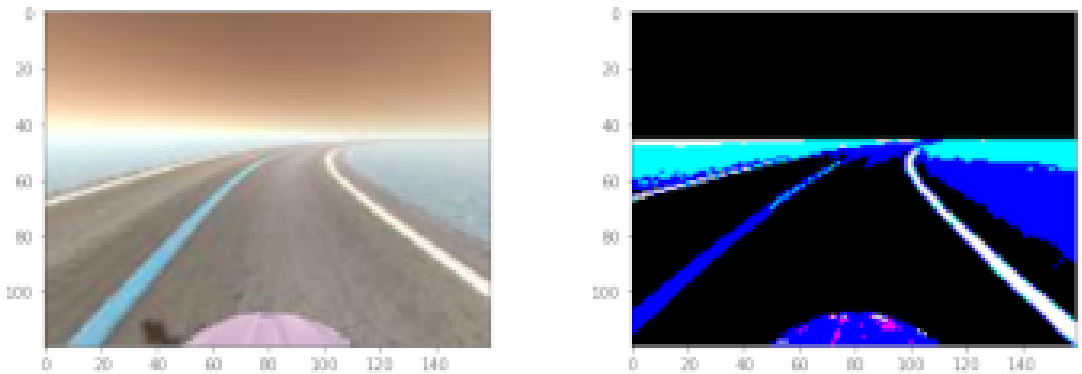


Figure 4.10: Generated Road Dataset Before and After Threshold is Applied

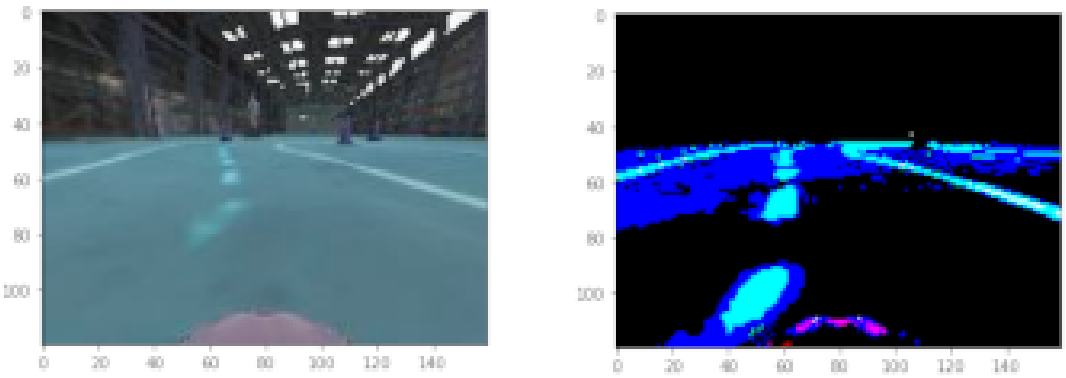


Figure 4.11: Warehouse Scene Dataset Before and After Threshold is Applied

Table 4.15 below shows the Shannon Entropy values per dataset after ROI Filtering and Thresholding is applied.

Table 4.15: Shannon Entropy Values per Dataset with ROI-filtering and Threshold

Dataset	**Mean**	**Mode**	**Max**	**Standard Dev**
Generated Track	0.94560945	0.9500301	0.9650344	0.010531155
Generated Road	0.9569369	0.9562948	0.97098106	0.008593745
Warehouse Scene	0.9996187	0.99999994	1.0	0.008764012

The training statistics for the model using the thresholded generated track data is available in Neptune experiment LIT-175 (https://ui.neptune.ai/charag/Littlefoot/e/LIT-175) and in Table 4.16 below.

Table 4.16: Threshold Model Training Overview

Statistic	Value
Loss	0.071 %
Val. Loss	0.145 %
Epochs	202

The results of the evaluation are found in Table 4.17 below:

Table 4.17: Threshold Evaluation Results

Dataset	Val. Loss	Pred. Accuracy	Avg. Deviation	RMSE	MAE	R2
Generated Track	0.31 %	97.58 %	2.41 %	5.34 %	2.41 %	0.9332
Generated Road	1.37 %	75.09 %	24.90 %	11.71 %	8.88 %	0.2257
Warehouse Scene	2.21%	87.03 %	12.96 %	14.65 %	10.34 %	0.1613

Effects of Canny Edge Detection

Canny Edge Detection (CED) is accomplished via a convolutional filter that seeks out the starkest difference between adjacent pixels and labels these pixels as edges. The effect on the image is that the edges are outlined as white, and the rest of the image is blacked out. Figures 4.12, 4.13, and 4.14 show both training and evaluation data before and after CED is applied. CED is applied to an image which had been thresholded to increase the likelihood that only lane lines are outlined.

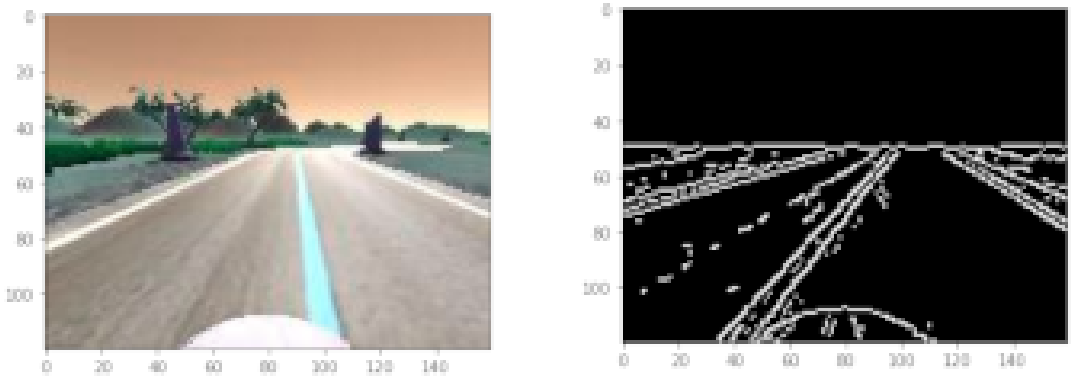


Figure 4.12: Generated Track Dataset Before and After CED is Applied

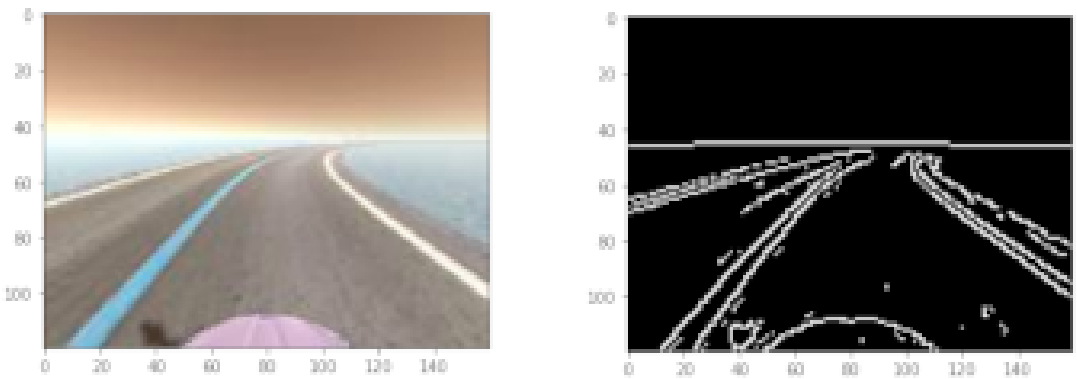


Figure 4.13: Generated Road Dataset Before and After CED is Applied

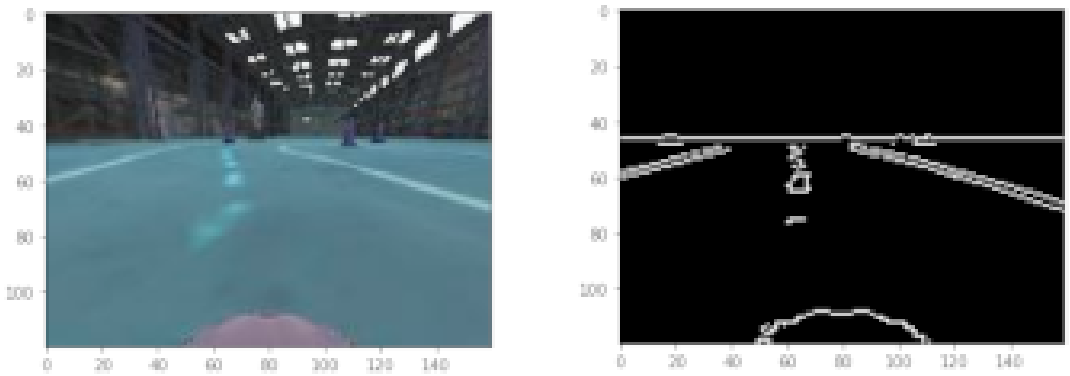


Figure 4.14: Warehouse Scene Dataset Before and After CED is Applied

Table 4.12 below shows the Shannon Entropy values per dataset after CED is applied.

Table 4.18: Shannon Entropy Values per Dataset with CED

Dataset	**Mean**	**Mode**	**Max**	**Standard Dev**
Generated Track	0.66634953	0.6671482	0.7101069	0.0110008335
Generated Road	0.6478013	0.6497805	0.69632244	0.0110134175
Warehouse Scene	0.4065076	0.40107712	0.5236449	0.03093578

The training statistics of the model with CED-filtered data is found in Neptune experiment LIT-181 (https://ui.neptune.ai/charag/Littlefoot/e/LIT-181) and in Table 4.19 below.

Table 4.19: CED Model Training Overview

Statistic	Value
Loss	0.11 %
Val. Loss	0.24 %
Epochs	246

Table 4.20: CED Evaluation Results

Dataset	**Val. Loss**	**Pred. Accuracy**	**Avg. Deviation**	**RMSE**	**MAE**	**R2**
Generated Track	0.25 %	97.60 %	2.39 %	4.87 %	2.39 %	0.9415
Generated Road	1.88 %	70.05 %	29.94 %	13.69 %	10.68 %	-0.0583
Warehouse Scene	2.23 %	86.55 %	13.44 %	14.71 %	14.71 %	0.1542

4.5.2 The Effects of Data Diversification on Model Performance

In this set of training - a model trained using data from all scenes will be used, excepting data from the Warehouse scene. This model is tested in all three environments as in the tests before. However, all data from the Warehouse environment will be 'blind' test data, meaning that the models trained with data diversification will not have anything to do with warehouse data before testing the fully trained models.

The total records used to train the diversified model amount to records when data from the generated road and track datasets are merged.

Table below shows the Shannon Entropy for a combined and diversified dataset:

Table 4.21: Shannon Entropy Values For Diversified Dataset

Mean	**Mode**	**Max**	**Standard Dev**
6.961362	-	7.6027517	0.41891217

Experiment LIT-283 (Available:https://ui.neptune.ai/charag/Littlefoot/e/LIT-283) on Neptune logs the model's training performance. The training stats of the diversified model is shown in Table 4.22

Table 4.22: Model Training Overview

Statistic	Value
Loss	0.12 %
Val. Loss	0.525 %
Epochs	247

The evaluation results of the model in all three environments is found in Table below:

Table 4.23: Diversified Model Evaluation Results

Dataset	**Val. Loss**	**Pred. Accuracy**	**Avg. Deviation**	**RMSE**	**MAE**	**R2**
Generated Track	0.29 %	97.58 %	2.41 %	5.00 %	2.41 %	0.9381
Generated Road	0.03 %	97.07 %	2.92 %	1.9 %	1.04 %	0.98
Warehouse Scene	3.41%	82.75 %	17.24 %	18.28 %	13.75 %	- 0.3057

This experiment set had begun already using the new split algorithm shown in Figure 4.2.

4.5.3 The Effects of Combining Feature Engineering and Data Diversification

It has been shown that both Feature Engineering and Data Diversification positively contribute to the performance of the model on unfamiliar data. Now, the effect of combining various Feature Engineering and Data Diversification techniques will be explored and tested. For these experiment sets, the datasets from the generated road and generated track will be merged as with the Diversification experiments, with the tested Feature Engineering techniques applied to the merged dataset.

Diversification and ROI

Experiment LIT-290 (Available:https://ui.neptune.ai/charag/Littlefoot/e/LIT-290) on Neptune logs the model's training performance with diversified ROI-modified data. The training stats of the diversified model is shown in Table 4.24

Table 4.24: Model Training Overview

Statistic	Value
Loss	0.12 %
Val. Loss	0.22 %
Epochs	299

The evaluation results of the model in all three environments is found in Table 4.25 below:

Table 4.25: Diversified Model with ROI Evaluation Results

Dataset	**Val. Loss**	**Pred. Accuracy**	**Avg. Deviation**	**RMSE**	**MAE**	**R2**
Generated Track	0.27 %	97.18 %	2.81 %	5.2 %	2.81 %	0.9355
Generated Road	0.09 %	95.27 %	4.72 %	3.13 %	1.68 %	0.9440
Warehouse Scene	8.33 %	69.42 %	30.57 %	28.74 %	24.39 %	-2.22

Diversification and Thresholding

Experiment LIT-300 (Available:https://ui.neptune.ai/charag/Littlefoot/e/LIT-300) on Neptune logs the model's training performance with diversified Threshold-modified data. The training stats of the diversified model is shown in Table 4.24

Table 4.26: Model Training Overview

Statistic	Value
Loss	0.08 %
Val. Loss	0.28 %
Epochs	180

The evaluation results of the model in all three environments is found in Table below:

Table 4.27: Diversified Model Evaluation Results

Dataset	**Val. Loss**	**Pred. Accuracy**	**Avg. Deviation**	**RMSE**	**MAE**	**R2**
Generated Track	0.41 %	96.96 %	3.03 %	6.32 %	3.03 %	0.9035
Generated Road	0.22 %	92.98 %	7.01 %	4.66 %	2.50 %	0.8777
Warehouse Scene	2.95%	84.39 %	15.60 %	16.97 %	12.44 %	-0.1253

Diversification and CED

Experiment LIT-289 (Available:https://ui.neptune.ai/charag/Littlefoot/e/LIT-289) on Neptune logs the model's training performance with diversified Threshold-modified data. The training stats of the diversified model is shown in Table 4.24

Table 4.28: Model Training Overview

Statistic	Value
Loss	0.12 %
Val. Loss	0.525 %
Epochs	247

The evaluation results of the model in all three environments is found in Table below:

Table 4.29: Diversified Model Evaluation Results

Dataset	**Val. Loss**	**Pred. Accuracy**	**Avg. Deviation**	**RMSE**	**MAE**	**R2**
Generated Track	0.44 %	96.49 %	3.5 %	6.65 %	3.5 %	0.8950
Generated Road	0.28 %	91.65 %	8.34 %	5.26 %	2.97 %	0.8445
Warehouse Scene	2.29%	86.51 %	13.48 %	14.90 %	10.76 %	0.1324

4.6 Conclusion and Final Solution

The impact of FE and feature extraction on the performance of a model on new data is undeniably positive. Methods of automating the process, or making this feature detection more adaptive, could include training a Segmentation Network. Additionally, since the input to the model is a three-channel image (with each channel being convolved by each filter), these channels may be exploited in the feature extraction process by having a CED modification, Threshold modification, and a Grayscale version of the image each occupying one of the three image channels.

The Keras Functional API, as described in the document, was eventually utilised in restructuring the designed model to be fit for simulator integration. This API may also be utilized to build certain FE operations into the model itsef - eliminating the need for additional programming in the Donkey Car system.

The iterative process of the model design allowed for rapid testing and discovery of issues. This also allowed for the refining of requirements as new knowledge arose - without the need to completely scrap a design and start again afresh. This is believed by the author to be one of the strongest points of the ML design methodology. This design methodology also appears to fit in seamlessly with the paradigms of traditional Systems Engineering Design - showcasing its validity and usefulness.

Chapter 5

Implementation and Evaluation

This chapter deals with the implementation of the final design in the Donkey Car Simulator, and evaluates the performance of the solution against the client's requirements and specifications.

5.1 Description of Implementation Environment

Orginally, the Donkey Car would have been initially tested in the simulator and then implemented continuously in the physical Donkey Car system. However, due to space, time and scope constraints developing over the course of the year, the simulator was selected as the final testing and implementation environment for this iteration of the project.

This section deals with a thorough description of the implementation environment as it appears in the Donkeycar 3.1.12, donkey-gym 1.0.14, and Donkey Sim 5.16.20 releases from the DIYRobocars open source community.

The implementation setup used in this project runs on Donkeycar 3.1.12 (available: https://github.com/autorope/donkeycar), donkey-gym 1.0.14 (available: https://github.com/ tawnkramer/gym-donkeycar), and Donkey Sim 05.16.20 (Pandemic Edition) (available: https://github.com/tawnkramer/gym-donkeycar/releases). All of this software is installed on a single host laptop.

The Donkey Sim and Donkey Gym are developed using Unity Tools: the Donkey Sim itself is created in Unity, and the Donkey Gym piggybacks off Unity's OpenAI functionality. Donkey Car itself expects to communicate its pilots outputs to an existing physical servo motor and receive images from a physically existing camera. To work around this, Donkey Gym stands in for physical hardware by providing the feedback from the OpenAI gym (a 120x160x3 image, among other data) as the input from the "camera", as well as translating the outputs from the pilots as movement signals that the vehicle controlled by the OpenAI gym will understand.

Figure 5.1 below illustrates this relationship:

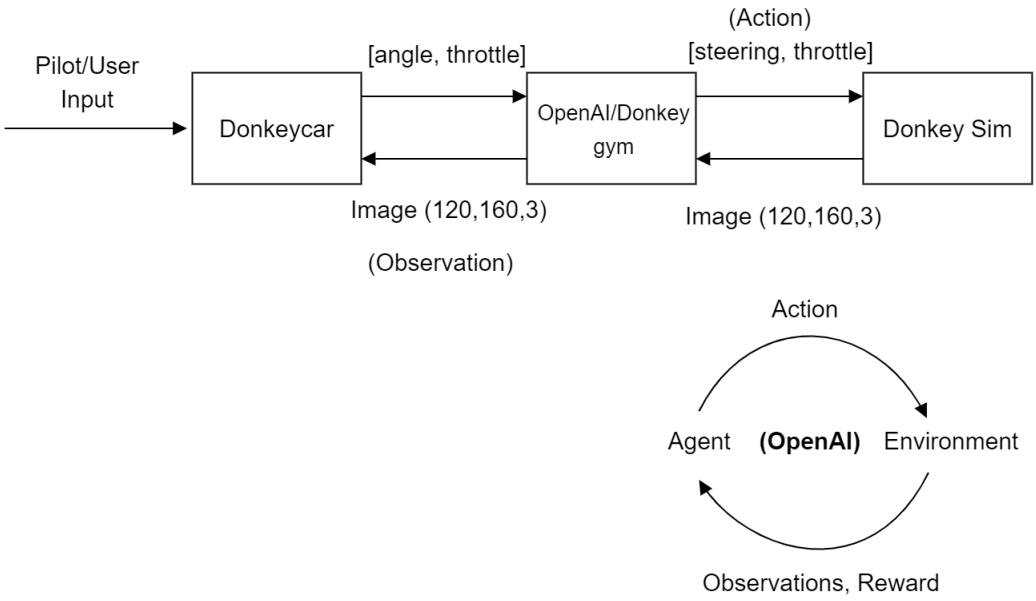


Figure 5.1: Donkeycar, donkey-gym, and Donkey Sim Relationships

The OpenAI gym works on a basis of providing actions to the environment, and retrieving observations and rewards back from that environment to the agent.

The user may drive the virtual vehicle around the environment to collect data for training, or deploy a pre-trained Pilot in the selected environment. The environment in which the simulation takes place is selected in the donkey car configuration file - where Donkey Gym selects an environment from the list of available environments.

The Donkey Sim and Donkey Car software "hosts" a control website at localhost, known as the Donkey Monitor, from which the user can either drive the car in the selected environment using the mouse-based virtual joystick, or allow a selected model to Pilot the car. Figure 5.2 shows this interaction and usage flow:

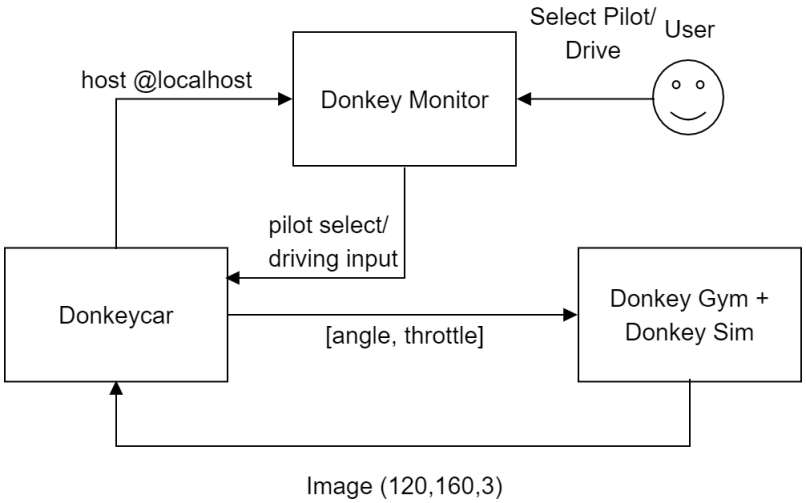


Figure 5.2: Caption

Description of Donkeycar

Donkeycar has several support structures in place besides the Pilot itself to make self-driving possible. A typical use of the Donkeycar software can be illustrated as follows: a single control script houses an instantiation of the main Vehicle class. This Vehicle class contains the ability to allow data flow between different "Parts" continuously and in the correct order - such as data flowing from a camera module to a Neural Network for inference, and from the Neural Network to the Part in control of outputting signals to a Servo Motor.

A Part is a Class that contains the methods, variables and methods required for a specific 'working function' of the car. A Part may be responsible for retrieving images from a hardware camera, processing and storing an image in a directory, providing a connection to the Donkey Car System via the Donkey Monitor, or housing the models used for training and driving a Pilot. Each of these parts has an 'incoming' data flow and an 'outgoing' data flow. For example, the Keras.py Part that may either refer to an existing model structure in the Donkey Car system, or load a .h5 model file for inference, takes as incoming data images and json records, and outputs angle and throttle values to the part communicating with donkey gym in this case.

Any part may be created and added to the Donkeycar drive loop, as long as it follows the convention of having at least an init method and an update method.

Figure 5.3 shows the "Drive Loop", which iterates through each Part attributed to the vehicle class, calls the update method on each Part in sequence. The update method of each part applies any processing to the incoming data to that Part, and then pushes that data on to the next part in the Drive Loop sequence:

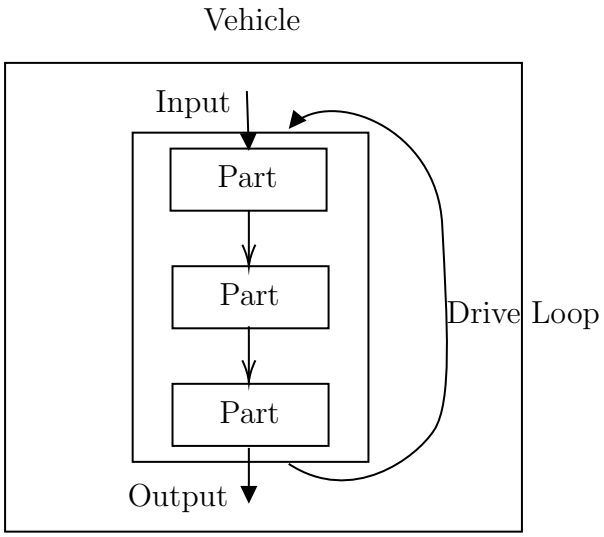


Figure 5.3: Vehicle Class and its Part Classes

Due to Keras' ability to load .h5 files that contain both the trained weights of the model and the model architecture - models that are built and trained outside of Donkeycar may be easily and quickly loaded for use without any integration with the rest of the system required, as long as there are no Tensorflow and Keras version conflicts. When driving, an .h5 file may be loaded to function as a self-driving Pilot. This Pilot takes over the functions of the user, inferring angle and throttle values based on the incoming images from the rest of the system.

5.2 Implementation Methodology

The actual implementation of the final Pilot was rather simple after all the installations took place - with the bulk of the work taking place during the Experimental Design phase and with the already existing support structure of Donkeycar itself. Some difficulty was met with Tensorflow version conflicts - which was resolved by undergoing a custom installation process and creating a suitable Anaconda environment. The Feature Engineering algorithm was added to the cv.py part of the Donkeycar, and a new management script version was created that included this part in the Drive Loop. The Pilot was tested via running the "drive" command and pointing the Donkeycar system to the model saved on disk.

5.2.1 The Installation Process

Donkeycar is a python library. The operating system of the laptop used is Windows 10, so Anaconda was installed and added to the system path so that Donkeycar could be operated from the Windows Powershell cli. Git was installed natively using the PowerShellGet package manager so that the rest of the installation process could take place.

The official Donkeycar installation process involves creating an Anaconda environment from a pre-prepared .yml file so that the dependencies, which inlcude Keras and Tensorflow,

are automatically installed. However, the only usable Tensorflow versions on Colab with GPU acceleration are Tensorflow 1.15.2 and Tensorflow 2.x, since Colab custom-compiles their Tensorflow distrobutions for their GPU accelerators. The default Tensorflow version of Donkeycar is 1.13.1. This Tensorflow version caused conflicts when trying to run models trained in Tensorflow 1.15.2, since Tensorflow 1.15.2 supports "ragged tensors" which Tensorflow 1.13.1 does not support - and regardless whether or not ragged tensors were used the metadata of the models' .h5 files referenced it - so Tensorflow 1.13.1 was unable to run the models trained on Colab.

To work around this, a custom conda environment was set up with specifically Python 3.7 selected and Tensorflow 1.15.2 pre-emptively installed via pip before all the other dependencies were installed. Python 3.7 was selected due to the fact that Python 3.8 does not support any Tensorflow version lower than 2.0, and Python 3.6 is incompatible with some of Tensorflow 1.15.2's dependencies.

Once the Tensorflow conflicts were resolved, the Donkeycar installation process could proceed via just cloning the repository from git and running the installation script provided.

The donkey-gym installation process was similar, with cloning the repository into the desired folder and running the installation script. The Donkey Sim for Windows was downloaded as a zip file and extracted - all the assets needed and an .exe file for the sim neatly in place.

The simple installation and implementation process besides the Tensorflow versioning conflicts allowed for more extensive time spent on the Design and Experimentation process of the project.

5.2.2 Feature Engineering Port and Running the Pilot

The final feature engineering algorithm used converts the incoming image to grayscale, finds the average pixel value, and based of the average pixel value performs Adaptive Gaussian Thresholding [43, 44]. The resulting image (where theoretically only the lane lines should remain) is then run through Canny Edge Detection.

All the image operations were implemented with OpenCV 2 methods and functions - since OpenCV 2 is a library already included in the Donkeycar suite and a high-level implementation of the operation allows for fewer points of failure.

The feature engineering algorithm was implemented as a Part class in the Donkeycar cv.py library file. The main control script, manage.py, was modified to include the ImgCanny Part (the custom image preprocessing part) between the camera module and the rest of the vehicle operation Parts, and renamed as managecv.py to allow parallel testing of models with and without Feature Engineering involved.

Figure 5.4 illustrates the flowchart of the feature engineering algorithm:

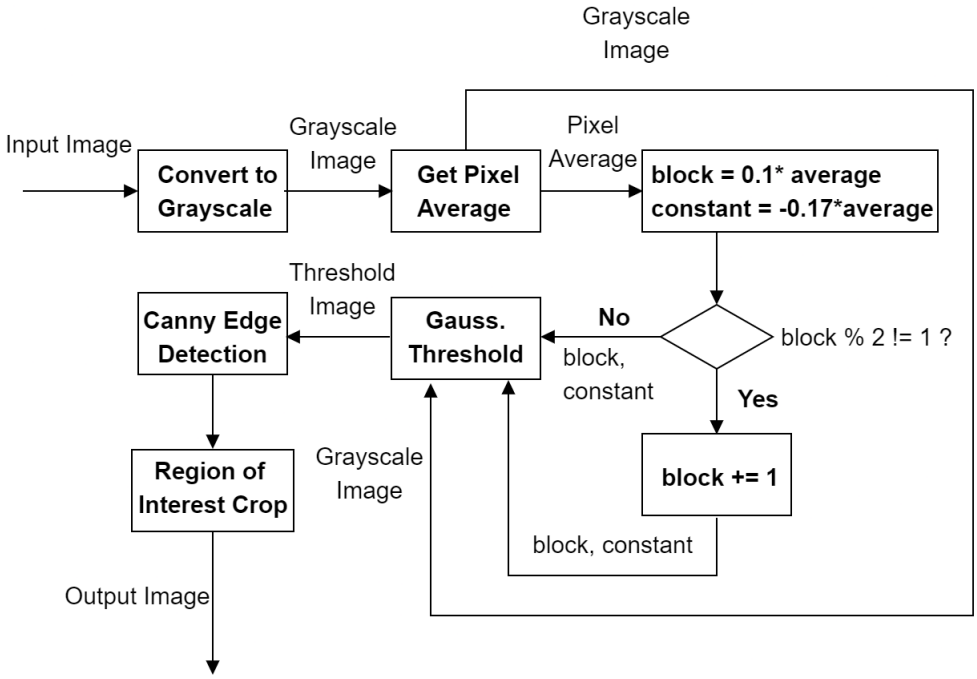


Figure 5.4: Feature Engineering Algorithm Flow Diagram

Figure 5.5 illustrates Functional Unit Diagram of the final implementation of Feature Engineering and Pilot within the Donkeycar and simulated system, where ImgCanny refers to the Feature Engineering algorithm and the Pilot refers to the CNN:

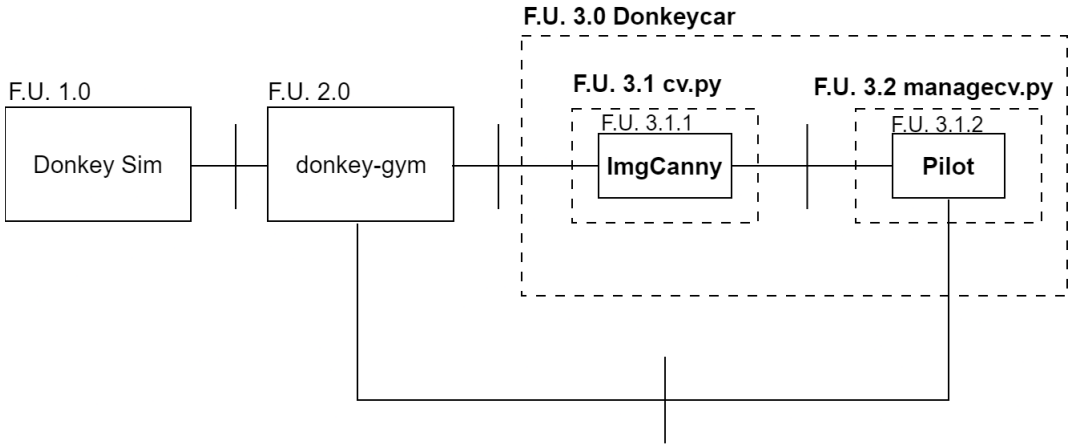


Figure 5.5: Final System Functional Unit Diagram

The manage.py, or managecv.py file, is used within a Command Line Interface (cli), in this case, Windows Powershell with the donkeycar conda environment activated. This file accepts the command "drive" with or without the option "–model". If the drive command is given without the "–model" option, the Donkeycar simply hosts the Donkey Monitor with the option for the user to drive the car through donkey-gym in the Donkey Sim via a software joystick. However, is the drive command is passed with the –model option, a relative path to

the required .h5 file of the model must be given. This allows the user to either control the car via the software joystick in Donkey Monitor, or select the "Local Pilot" control option, which hands over the reigns to the trained Neural Network Pilot.

5.3 Evaluation and Testing

5.3.1 Performance Requirements and Spec Doc Review

According to the specification document - the design problem was to develop a self-driving car pilot that **reliably** follows a track with some variations. Variations was broken down to mean one or more of the following:

1. Changes in Track Shape
2. Changes in Environment
3. Changes in Track Length
4. Changes in Lighting

The **reliability** of the vehicle was broken down to mean the following:

1. The vehicle does not have all four wheels leave the bounds of the track more than twice
2. The vehicle, in event of leaving the track with all four wheels, is able to recover at least once

The original specification document elicited requirements for tests being conducted on the Physical Donkey Car System to different compliance levels. Each compliance level stood for a different level of generalisation: such as generalising between different lighting conditions, then different environments, and then different track shapes. However, before the final baseline Neural Network could be completed, South Africa was put under COVID-19 lockdown, and several resources such as space for the test track was no longer available once access to campus was revoked. Even though access to campus was later restored, it was decided to make the implementation a Donkey Car Simulator only implementation.

5.3.2 Performance in Simulated Environment

The same three environments used in the experimental design process as datasets (test and training) are now used as test environments.

The Generated Track Environment consists of a track with the following features:

- Solid Yellow Middle Line
- Solid White Side Lines
- Single Hairpin Turn
- Bright midday lighting
- Some track reflection
- Green hills, trees and long grass in background
- Orange cones around track.

The Generated Road Environment is a randomly generated long road with the following features:

- Solid Yellow Middle Line
- Solid White Side Lines
- Randomly generated road with few to many sharp-angled turns.
- Very bright lighting
- Lots of track reflection
- Desert environment
- No cones or any items surrounding track

The Warehouse Scene depicts one of the early DIYRobocars competition warehouses, and includes features such as spectators and grafitti on the walls. The features of this environment include:

- Dashed yellow centre line
- Solid white side lines
- At least one hairpin turn
- Dull indoor lighting
- No track reflection
- Indoor environment with grafitti on the walls
- Cones, tyres, tables and spectators surrounding track

Four different models are evaluated. One model trained only with unmodified Generated Track Data, one model trained with diverse data from the Generated Track and Generated Road, a model trained on Generated Track data with Canny Edge detection applied, and a model trained on diverse data with Canny Edge Detection applied.

Each model is allowed to travel around the track for one lap, or till the end of the generated road in the case of the Generated Road Environment, while being timed. The time at which the model leaves the track for the first time, as well as the second time, is recorded. Any recoveries are also logged, and if the model completes its course the total time taken is also logged. These results are found in Tables 5.1 below:

Table 5.1: Simulation Performance Results - Generated Track - Data Used in All Training

Model Name	**Dataset**	**Feature Engineering**	**Time to First Failure**	**Number of Failures**	**Times Recovered**	**Time to Final Failure**	**Time to Completion**
Vanilla	Generated Track	None	n/a	None	n/a	n/a	00:57:58 min
Vanilla Diverse	Diverse	None	n/a	None	n/a	n/a	00:58:30 min
Track Canny	Generated Track	Canny Edge Detection	n/a	None	n/a	n/a	00:56:96 min
Diverse Canny	Diverse	Canny Edge Detection	n/a	None	n/a	n/a	00:59:10 min

Table 5.2: Simulation Performance Results - Generated Road - Data Used in Some Training

Model Name	**Dataset**	**Feature Engineering**	**Time to First Failure**	**Number of Failures**	**Times Recovered**	**Time to Final Failure**	**Time to Completion**
Vanilla	Generated Track	None	00:13:64 min	2	2	00:28:25 min	n/a
Vanilla Diverse	Diverse	None	03:31:40 min	1	1	n/a	07:14:67 min
Track Canny	Generated Track	Canny Edge Detection	01:02:00 min	1	n/a	01:02:00 min	n/a
Diverse Canny	Diverse	Canny Edge Detection	03:45:47 min	1	1	n/a	07:17:40

Table 5.3: Simulation Performance Results - Warehouse Scene

Model Name	**Dataset**	**Feature Engineering**	**Time to First Failure**	**Number of Failures**	**Times Recovered**	**Time to Final Failure**	**Time to Completion**
Vanilla	Generated Track	None	00:10:70 min	1	None	00:10:70	n/a
Vanilla Diverse	Diverse	None	00:10:71 min	1	None	00:10:71 min	n/a
Track Canny	Generated Track	Canny Edge Detection	00:28:80 min	1	1	n/a	01:45:78 min
Diverse Canny	Diverse	Canny Edge Detection	n/a	None	n/a	n/a	01:41:30 min

Tables 5.1, 5.2 and 5.3 show that the final Pilot design - which includes Canny Edge Detection and diversification of training data, performs the best in each environment and is the best performing model in the Warehouse Scene - which is a completely new environment to the model.

This same model was also integrated with and tested on the physical Donkey Car, but not to the same level of success. Lane lines available were not wide enough, and the model did not react well to corners, either going straight ahead over the corner, or turning in the opposite direction. There are several confounding factors with a physical implementation - ranging from the calibration of the turning angles, the battery level, connectivity from the control laptop to the car, and the nature of a physical environment being far more complex than a simulated environment.

The original scope of the project involved also developing and implementing a more robust lane detection method than ROI, Thresholding and Canny Edge Detection to deal with the real-world complexities. Additionally, a physical implementation would have been developed from the beginning in tandem with a simulated implementation. However, due to changes in scope and time lost with the Covid-19 lockdown, this is moved on to further study and will be discussed in Chapter 6.

Chapter 6

Conclusions and Further Work

6.1 Project Review and Conclusion

The problem to be solved by this project was in short to get a Neural Network based self-driving car to generalise well to different environments, or at least be robust to changes in those environments. This generalisation could have been accomplished by training the model with data from diverse environments and simply re-training it every time it was going to be used in a new environment - however, this method would have been time consuming and would have neglected the opportunity to explore other solutions.

The focus of the project became generalisation - and the generalisation methods used included data diversification (to some extent) and feature engineering in the form of rudimentary lane detection. This generalisation technique performed well in the Donkeycar Simulator in both known and completely unknown environments - satisfying the performance requirements of the model **reliably** following a track in spite of lighting, environment and track shape changes. Several other generalisation methods were mentioned in Chapter 1, Section 1.3 - including K-fold training, Regularization and Pruning, besides Feature Engineering. Each of these generalisation methods would have been interesting to explore. The effect of K-fold training would have been especially interesting to see, as it is often most used for the cases where few data points are available [1]. L1 and L2 Regularization would have also been interesting to implement and observe its effects.

The project topic also tackled the nature and design of Neural Network Architectures (NNAs) suitable for image processing and Piloting the Donkeycar. The NNAs that were explored as options included Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Long Short Term Memory Networks (LSTMs), Three-Dimensional (3D) CNNs or some combination of the above. The NNA that was chosen for the experimental design and implementation process of the Pilot was a CNN with dense layers added at the end for regression. If time allowed, the performance of a 3D-CNN or RNN would have been beneficial to observe. It is suspected that these time-based architectures would perform more consistently over many different environments regardless of the level of feature engineering or

generalisation measures applied - since an additional time-based element would give a certain "context" to the model's operation. This could certainly be explored in future work.

This CNN architecture was iteratively designed and implemented using a combination of Francois Chollet and Ian Goodfellow's recommendations, as well as some original interpretation of Machine Learning (ML) design methods developed for this project [1, 2]. This design methodology was described and laid out in Chapter 3, Section 3.3.1. This design methodology can be briefly summarised as getting to know the data, defining performance metrics and requirements, choosing appropriate tools and architectures based on the problem, establishing an end-to-end testing pipeline, and building up a model until a satisfying solution is found. The building up of a satisfying solution involved iteratively implementing a CNN architecture, training it, testing it, and making adjustments based on the test results.

The changes that were logged throughout the experimental process involved layer types and quantities, optimizers, validation and training losses, and datasets used. If this experimental process were to be conducted again - more controlled selection processes for the Optimizer would be conducted (i.e., comparing the losses on a standard network with the same dataset with different Optimizers), changes in the kernel sizes and layer sizes would also be logged, and activation functions would be tested for and chosen in a similiar way to Optimizers with performance comparison tests.

Additionally, a good practice suggested by both Chollet and Goodfellow is to first implement a baseline non-NNA solution for comparison with the NNA's performance as it is built up. This was not included in the design and experimentation process due to this provision being an entire project scope in its own right - however some methods have been explored in the meantime that would definitely find their way into future work for design comparisons. One of these "non-NNA" solutions is a Histogram of Gradients (HoG) descriptor that feeds into a Linear Support Vector Machine (LSVM) [45, 46]. This may even have been a candidate for a complete alternative solution due to the reported excellent performance of a HoG + LSVM combination [47].

The implementation of the final Pilot and Generalisation (Feature Engineering) solution took place in the Donkey Car Simulator - which thankfully was fully equipped to replace a Physical Donkey Car. An implementation on the Physical car was also attempted near the end of the project time frame in spite of the scope change due to Covid-19. However, several points of failure were picked up on - and a full physical implementation of the generalisation project will have to form part of further study. Preferably using real-world environments instead of simulated environments - due to the increased complexity of real-world environments, the differences in throttle and steering behaviours, and the differences in lighting to the Simulator lighting.

6.2 Further Study and Development

The ultimate goal of this project, as a DIYRobocars project, is to increase awareness of DIY-Robocars, Donkeycar, and CPDynamics (the project client) and hopefully foster collaboration with hobbyists, students and faculty with competition and the project's overall development. Donkeycar is an open source project licensed under the MIT License, and there is so much potential in every facet of the project for development and redevelopment that can be employed as Final Year Projects, other subject-bound undergraduate projects, or even Postgraduate study.

In terms of Further Study, some additional topics and/or improvements to the current project have been considered and brought to light. In terms of Generalisation-Specific further study - the effect of input image augmentation in the form of shearing, stretching, shrinking and rotation could be explored on the models' performance in new environments in conjunction with the Feature Engineering Algorithm already implemented. The models' performance in the face of blurry images may also be evaluated and solutions explored in that direction.

Continuing in the vein of Generalisation and Feature Engineering, a Segmentation Network, or U-Net as described in Chapter 2, Section 2.7.2, may be added to the model as a more robust lane detection method instead of Thresholding and Canny Edge Detection. This may be regarded as "overkill" for simply lane detection, but provides an expandability for applications like road sign detection/recognition, traffic tracking, and padestrian detection in the future.

To expand both the generalisation capacity and applicability of the model, the training and the model itself may also be split into two parts. In this scenario, a separate CNN may be trained as a "feature extractor" on hundreds of thousands of open source road marking and street photo datasets. Using transfer learning, this feature extractor's weights may then be frozen and dense layers added to it for regression. Datasets collected via Donkeycar may then be used to train the behavioural aspect of the model - with the added benefits of a robust feature extractor that adjusts to several different road markings, road widths, lightings, and environments.

Barring the applications from a generalisation standpoint for a standard Donkeycar model - Donkeycar may also be used to experiment with PID controllers in conjunction with Artificial Neural Networks. This may be done by having the model predict the angle, throttle, and actual relative speed based on both the incoming images and the rate of incoming images. This actual relative speed is then compared to the throttle, and the PID adjusts the output to the servo motor to modulate the speed against what the model actually wants to predict. This means that a model trained in a simulator, with a lower throttle value for a relatively higher "effective" speed, may run on a physical car for the same "effective" speed without modifying the training data or the model's predictions.

Bibliography

[1] F. Chollet, *Deep Learning with Python*, 2018. [Online]. Available: http://faculty.neu.edu.cn/yury/AAI/Textbook/Deep Learning with Python.pdf

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, T. Dietterich, C. Bishop, D. Heckerman, M. Jordan, and M. Kearns, Eds. Cambridge: The MIT Press, 2016, vol. 53, no. 9.

[3] H. Sankesara, "UNet," 2019. [Online]. Available: https://towardsdatascience.com/u-net-b229b32b4a71

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9351, pp. 234–241, 2015.

[5] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale Video Classification with Convolutional Neural Networks," in *Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 1725–1732. [Online]. Available: http://cs.stanford.edu/people/karpathy/deepvideo

[6] DIYRobocars, "Autonomous cars for the rest of us." [Online]. Available: https://diyrobocars.com/

[7] M. Szcys, "Self-Driving RC Cars With TensorFlow; Raspberry Pi Or MacBook Onboard — Hackaday," 2017. [Online]. Available: https://hackaday.com/2017/06/06/self-driving-rc-cars-with-tensorflow-raspberry-pi-or-macbook-onboard/

[8] J. Z. Chang, "Training Neural Networks to Pilot Autonomous Vehicles: Scaled Self-Driving Car," Tech. Rep. 402, 2018. [Online]. Available: https://digitalcommons.bard.edu/senproj_s2018/402

[9] J. Zhang, "A training method for enhancing neural network model generalisation," *Proc. Int. Jt. Conf. Neural Networks*, vol. 1, pp. 800–805, 2002.

[10] S. Mc Loone and G. Irwin, "Improving neural network training solutions using regularisation," *Neurocomputing*, vol. 37, no. 1-4, pp. 71–90, 2001.

[11] X. Jiang, M. Lu, and S. H. Wang, "An eight-layer convolutional neural network with stochastic pooling, batch normalization and dropout for fingerspelling recognition of Chinese sign language," *Multimed. Tools Appl.*, 2019.

[12] S. Wang, X. Wang, P. Zhao, W. Wen, D. Kaeli, P. Chin, and X. Lin, "Defensive dropout for hardening deep neural networks under adversarial attacks," *IEEE/ACM Int. Conf. Comput. Des. Dig. Tech. Pap. ICCAD*, 2018.

[13] L. Giancardo, O. Arevalo, A. Tenreiro, R. Riascos, and E. Bonfante, "MRI Compatibility: Automatic Brain Shunt Valve Recognition using Feature Engineering and Deep Convolutional Neural Networks," *Sci. Rep.*, vol. 8, no. 1, pp. 2–8, 2018.

[14] Y. T. C. Donkyu Lee, Jinhwa Jeong, Sung Hoon Yoon, "Improvement of Short-Term BIPV Power Predictions," 2019.

[15] D. Zimbra, M. Ghiassi, and S. Lee, "Brand-related twitter sentiment analysis using feature engineering and the dynamic architecture for artificial neural networks," *Proc. Annu. Hawaii Int. Conf. Syst. Sci.*, vol. 2016-March, pp. 1930–1938, 2016.

[16] C. L. Giles and C. W. Omlin, "Pruning Recurrent Neural Networks for Improved Generalization Performance," *IEEE Trans. NEURAL NETWORKS*, vol. 5, no. 5, pp. 848–851, 1994.

[17] F. Manessi, A. Rozza, S. Bianco, P. Napoletano, and R. Schettini, "Automated Pruning for Deep Neural Network Compression," in *Proc. - Int. Conf. Pattern Recognit.*, vol. 2018-Augus. IEEE, 2018, pp. 657–664.

[18] S. Lin, R. Ji, Y. Li, C. Deng, and X. Li, "Toward Compact ConvNets via Structure-Sparsity Regularized Filter Pruning," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 31, no. 2, pp. 574–588, 2020.

[19] D. Livne and K. Cohen, "PoPS: Policy Pruning and Shrinking for Deep Reinforcement Learning," *IEEE J. Sel. Top. Signal Process.*, vol. 4553, no. c, pp. 1–1, 2020.

[20] J. Walter, "The Mechanical Turk: How a Chess-playing Hoax Inspired Real Computers — Discover Magazine," 2019. [Online]. Available: https://www.discovermagazine.com/technology/the-mechanical-turk-how-a-chess-playing-hoax-inspired-real-computers

[21] IWM, "How Alan Turing Cracked The Enigma Code — Imperial War Museums," 2018. [Online]. Available: https://www.iwm.org.uk/history/how-alan-turing-cracked-the-enigma-code

[22] A. M. Turing, "COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, no. 49, pp. 433–460, 1950.

[23] K. D. Foote, "A Brief History of Machine Learning," 2019. [Online]. Available: https://www.dataversity.net/a-brief-history-of-machine-learning/

[24] P. Kulkarni, *Reinforcement and Systemic Machine Learning for Decision Making.* Hoboken, New Jersey: John Wiley & Sons, 2012.

[25] M. Taylor, "Neural Networks A Visual Introduction for Beginners," 2017.

[26] R. Roelofs, "Measuring Generalization and Overfitting in Machine Learning," Ph.D. dissertation, 2019.

[27] H. Zhong, Z. Chen, C. Qin, Z. Huang, V. W. Zheng, T. Xu, and E. Chen, "Adam revisited: a weighted past gradients perspective," *Front. Comput. Sci.*, vol. 14, no. 5, 2020.

[28] J. Duchi, H. Elad, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," vol. 12, pp. 2121–2159, 2011.

[29] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, pp. 1–15, 2015.

[30] M. Hazma, "The Softmax Function, Simplified," 2018. [Online]. Available: https://towardsdatascience.com/softmax-function-simplified-714068bf8156

[31] Uniqtech, "Understand the Softmax Function in Minutes - Data Science Bootcamp - Medium," 2018. [Online]. Available: https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d

[32] L. Danqing, "A Practical Guide to ReLU," 2017. [Online]. Available: https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7

[33] H. Lamba, "Understanding Semantic Segmentation with UNET," 2019. [Online]. Available: https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47

[34] Chris, "Understanding transposed convolutions." [Online]. Available: https://www.machinecurve.com/index.php/2019/09/29/understanding-transposed-convolutions/

[35] C. Nicholson, "A Beginner's Guide to LSTMs and Recurrent Neural Networks," 2019. [Online]. Available: https://pathmind.com/wiki/lstm

[36] J. L. Elman, "Finding structure in time," *Cogn. Sci.*, vol. 14, no. 2, pp. 179–211, 1990.

[37] Missing Link, "Understanding a 3D CNN and Its Uses." [Online]. Available: https://missinglink.ai/guides/convolutional-neural-networks/understanding-3d-cnn-uses/

[38] F. Chollet, "Keras API Developer Guides." [Online]. Available: https://keras.io/guides/

[39] S. Anantha, P. Eswaran, and C. Senthil, "Lossless Grayscale Image Compression using Block-wise Entropy Shannon (LBES)," *Int. J. Comput. Appl.*, vol. 146, no. 3, pp. 1–6, 2016.

[40] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.*, vol. 27, no. 4, pp. 623–656, 1948.

[41] A. Kim, "The intuition behind Shannon's Entropy," 2018. [Online]. Available: https://towardsdatascience.com/the-intuition-behind-shannons-entropy-e74820fe9800

[42] F. Boyd Enders, "Coefficient of Determination," 2013. [Online]. Available: https://www.britannica.com/science/coefficient-of-determination

[43] A. Mordvintsev and K. Abid, "Image Thresholding - OpenCV Python Tutorials," 2013. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

[44] OpenCV, "OpenCV: Image Thresholding." [Online]. Available: https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html

[45] A. Singh, "Feature Descriptor — Hog Descriptor Tutorial." [Online]. Available: https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/

[46] Mithi, "Vehicle Detection with HOG and Linear SVM." [Online]. Available: https://medium.com/@mithi/vehicles-tracking-with-hog-and-linear-svm-c9f27eaf521a

[47] H. Bristow and S. Lucey, "Why do linear SVMs trained on HOG features perform so well?" 2014. [Online]. Available: http://arxiv.org/abs/1406.2419