FACULTY OF ENGINEERING AND ARCHITECTURE

Applying end-to-end imitation learning for real time perception and control of autonomous vehicles: from simulation to real world environments

Wouter Vandendriessche
Student number: 01600456

Supervisors: Prof. dr. ir. Aleksandra Pizurica, Prof. Frank Lindseth (Norwegian University of Science and Technology (NTNU))
Counsellor: Nina Žižaki

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2019-2020

GHENT UNIVERSITY

Abstract

Autonomous vehicles are no longer a thing of the future. The technology is here and getting better every day. The current systems are however typically bound to specific controlled geographical areas, limiting their usability. There is still a lot of work and research to be done to make a truly independent autonomous vehicle. Advancements in deep learning have increased the validity of using end-to-end systems as a promising alternative approach to current systems.

This thesis explores some of the possibilities of these end-to-end systems and compares the performance of different architectures and techniques i.a. the importance of using temporal data, the importance of the quality of the dataset, classification vs. regression and the effect of increasing the complexity of the system.

This work also explores the implementation of these architectures on the JetBot robotic test platform for the task of both lane following and following navigational directions in a simplified urban environment.

The architecture proposed by *Aasbø and Haavaldsen, "Autonomous Vehicle Control: End-to-end Learning in Simulated Environments."*[1] is used as a basis. The idea is explored further by applying the findings on the jetbot platform, performing further tests and validating results.

The findings in this thesis show that even a simple deep neural net can achieve full autonomy, given a sufficiently large dataset of high quality. The results with more complex models on the JetBot platform were not promising, with the vehicle regularly ignoring commands or swerving out of lane. Further experiments hinted that this is probably because those models were overqualified for the simple environment as well as the use of a (too) limited dataset.

Acknowledgements

I would like to thank my supervisors Frank Lindseth and Aleksandra Pizurica for giving me the opportunity to write a thesis on this promising technology. Without the guidance and the resources provided by the NTNU Autonomous Perception Lab (NAP-Lab), this work would not have been possible.

Contents

List of Figures

List of Tables

Acronyms

ALVINN Autonomous Land Vehicle In a Neural Network. 21

ANN Artificial Neural Network. 4, 16, 26

API Application Programming Interface. 26, 27

CAD Computer Aided Design. 26

CAN Controller Area Network. 22

CNN Convolutional Neural Network. 16, 21, 23, 24, 36, 43, 45–48

CUDA Compute Unified Device Architecture. 25–27, 34

DARPA Defense Advanced Research Projects Agency. 21

DAVE DARPA Autonomous Land Vehicle. 21, 35

ELU Exponential Linear Unit. 8, 35

GPU Graphical Processing Unit. 26, 27, 34

Grad-CAM Gradient-weighted Class Activation Mapping. 38

LAGR Learning Applied to Ground Robots. 21

LiDAR Light Detection And Ranging. 19

LSR Least Squares Regression. 38

LSTM Long Short-Term Memory network. 18, 23, 36, 38, 40, 42, 43, 45–49

MDP Markov Decision Process. 20

MSE Mean Square Error. 9, 22, 38

NTNU Norwegian University of Science and Technology. 3

ReLu Rectified Linear Unit. 7

RGB Red Green Blue. 25

RNN Recurrent Neural Network. 17, 32, 49

ROS The Robot Operating System. 26, 27, 48–50

SGD Stochastic Gradient Descent. 10, 11

SOM System on Module. 25

V2V Vehicle To Vehicle. 50

Glossary

Gaussian blur A Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. 14, 34

PID controller A proportional–integral–derivative controller or PID controller is a control loop mechanism employing feedback that is used in applications requiring continuously modulated control. 30

YUV colorspace YUV is not an acronym. YUV is a color encoding system that defines a color space in terms of one luma component (Y) and two chrominance components, called U (blue projection) and V (red projection). 21

1

Introduction

1.1 Background and Motivation

We are at a crossroads in the history of human transportation. The time of manual driving and its inevitable human errors will become a thing of the past. According to the European commission of Mobility and Transport, there are still more than 40 000 deaths on EU roads each year with more than 90% caused by human error, of which 10 to 30% caused by distraction[2]. The need for smarter and safer cars is high on the agenda.

More and more startups in this field are emerging and some big players have started investing, such as Waymo (Alphabet Inc.) who have been developing their technology since 2009, have already started a commercial taxi service in selective regions (known as level 4 autonomy or high automation [3]). Other, well established car companies, such as GM, BMW, Nissan, Ford and many others have also invested billions trying to get ahead of the competition.

Today's most used approach is an explicit decomposition of the problem where the different sub problems such as sensor fusion, lane marking detection, path planning, and control get solved by separate modules, each using different technology stacks and techniques.

Another approach is to develop one coherent end-to-end system (such as a deep neural network) which takes as input all sensor data and directly generates the output commands for the vehicle.

Such a system both reduces complexity and required knowledge of different domains. A deep neural net based end-to-end approach can e.g. be trained using Imitation Learning. This way, the system learns to drive by studying the behaviour of an expert driver. This thesis will explore the field of imitation learning for end-to-end systems using only camera input. The findings will be applied on the JetBot robotic test platform in a miniature real-world environment.

1.2 Objectives and Research Questions

1.2.1 Problem definition

The overall goal of this thesis is twofold.

Initially, this work is a continuation of the work of *Aasbø and Haavaldsen, 2019*[1]. The authors proposed a deep learning architecture using different aspects from recent research to create an end-to-end system for autonomous driving. In their work, the importance of dataset size and optimal parameters were measured and analysed. This thesis investigates the importance of the used architecture aspects as well as validates the achieved results.

In addition,this study explores the possibility of applying these systems into a simple real-world scenario using a small robotic test platform called JetBot. This includes examining how well these models perform in real life compared to their performance in simulated urban environments.

1.2.2 Research questions

- What is the importance of dataset balancing?
- What impact does the recurrent module of the end-to-end architectures have on a model's performance?
- Does the model architecture benefit from a more complex feature extractor?
- Do these methods translate to a simple real-world environment?
- Do the findings of performance differences correlate to the findings in the real-world environment?

1.3 Contributions

This is mostly an exploratory work. It covers deep learning as well as exploring the capabilities of the JetBot robotic test platform. The thesis engages into the topic of artificial neural networks for the use of end-to-end learning for autonomous vehicles.

Several models were trained, comparing their performance in simulation as well as on the JetBot platform. The real-life tests using the JetBot show that following commands while keeping in lane is possible, but that a larger dataset is needed to increase reliability. The tests also show that a simpler architecture performs better in this case.

This being the first operational trial using the JetBot platform at NTNU, the acquired experience exploring the possibilities and limitations of the system, sets the stage for the use in further projects.

1.4 Thesis outline

Chapter 1: Introduction consists of a general introduction to the problem and poses research questions, objectives and describes this thesis's contributions.

Chapter 2: Background and related work contains the general background information needed on artificial neural networks and on the relevant technology for this thesis, backed up by related work.

Chapter 3: Methodology describes the methods used to collect, process and train on data as well as the proposed architectures and the methodology for testing.

Chapter 4: Experiments and results covers the conducted experiments with their results and a discussion of the findings of each experiment as well as a more general discussion. The discussion reflects on both the experiment outcomes and on the choices that were made.

Chapter 5: Conclusion and Future work draws a conclusion and explores potential future work to be done.

2

Background and related work

2.1 Artificial neural networks

Artificial Neural Networks are a means of doing machine learning in which a computer learns to perform a task by observing and analysing a set of examples. An ANN is built up out of thousands or even millions of simple processing nodes that are densely connected together to fulfil complex functions that have proven hard or even impossible to solve with classical computation.

Usually, an analogy with the human brain is made, where the computational nodes in the ANN are compared with biological neurons. It is important to note however that this is a loose comparison as (at least in the current state of artificial intelligence) the neurons in the brain are vastly more complex then artificial neurons.

Currently, most neural nets are organized into densely interconnected "Feed Forward" layers. Feed Forward implies that data flows through the network in one direction: each node receives data from several nodes in the preceding layer, performs a computation on that data and sends it along to connected nodes in the following layer.

All in all, a neural network is basically a group of simple linear functions stacked together in a hierarchical way to form a very complex nonlinear function.

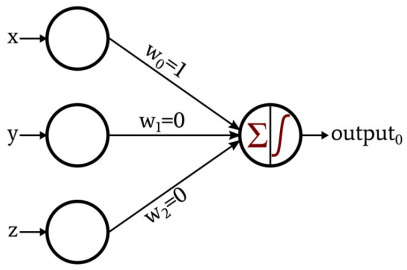


Figure 2.1: The modern perceptron

2.1.1 The *perceptron*

The perceptron is a type of artificial neuron first developed in the 1950's and 1960's by Frank Rosenblatt[4], inspired by earlier work of Warren McCulloch and Walter Pitts. It is an algorithm for learning a binary classifier called a threshold function: a function that maps its input, a vector with real values, to a single binary output.

$$f(x) = \begin{cases} 0, & \sum_j w_j x_j + b > 0 \\ 1, & \sum_j w_j x_j + b \leq 0 \end{cases} \tag{2.1}$$

The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j + b$ is less than or greater than some threshold value, called the bias b (see Equation 2.1). The bias is an indication of how easily the node is activated while the weights signify how important each of the inputs is.

The issue with this type of artificial neuron is that a small change in the input does not result in a small change in output. A minuscule adjustment in the weights or the bias of a single perceptron may flip its result from 0 to 1, causing an extensive behavioural change of the entire network. This makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour.

Therefore, the artificial neuron used today is somewhat different. The output of the neuron is now the weighted sum of its inputs, transformed by an activation function that limits the possible range of the output as well as introduces non linearity.

2.1.2 Activation functions

Activation functions have the crucial task of adding non-linearity to the nodes of a network. Without this the network would collapse into an overly-complicated linear function, unable

to fit any complex data. There are many different activation functions, each with their own respective strengths and weaknesses.

Sigmoid

The sigmoid is a good activation function for classifiers. It tends to bring the activations to either end of the curve. Making clear distinctions in prediction.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

Another advantage of this activation function is that, unlike a linear function, the output, also known as the activation, is always going to be in the range (0,1). This is desirable since unbound activations will almost certainly lead to an explosion in activation size.

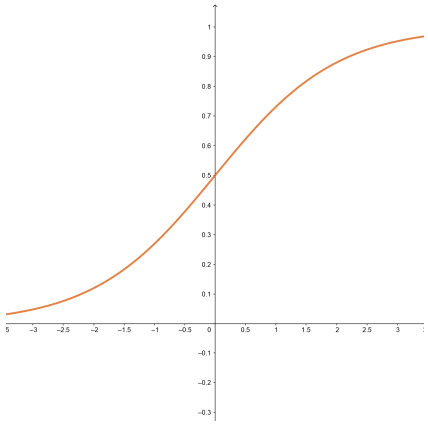


Figure 2.2: The Sigmoid activation function

The main disadvantage is that the gradient towards either end of the sigmoid will be small. This is the main cause of the problem of "vanishing gradients" where the network stops learning or learns drastically slow. There are ways around this issue and this is still the most popular activation function to date.

Tanh

Tanh is very similar to the sigmoid function. It is in fact a scaled version of the sigmoid with the main difference being that the gradient of the Tanh is steeper and that the function bounds the output to the range (-1, 1) instead of (0, 1).

$$\text{Tanh}(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.3}$$

Deciding between the sigmoid or Tanh depends mostly on the requirement of the gradient strength. Tanh also suffers from the vanishing gradient problem.

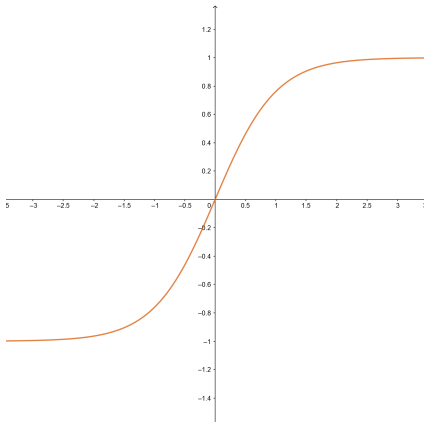


Figure 2.3: The tanh activation function

ReLU

ReLu or Rectified Linear Unit has become very popular due to its simplicity. When the input, x, is positive, its output is x, otherwise it's 0. At first this looks as if it has the same problem as having just a linear output, but ReLu is in fact non-linear. It has the advantage of being very simple to compute as well as not suffering from the vanishing gradient problem.

$$\text{ReLU}\,(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{2.4}$$

The main issue here is that the function has no upper bound on its output. This allows the activations to blow up to very large values. Another downside, called "the dying ReLu problem", comes from being zero for all negative values. Once a neuron gets negative it is unlikely to recover.

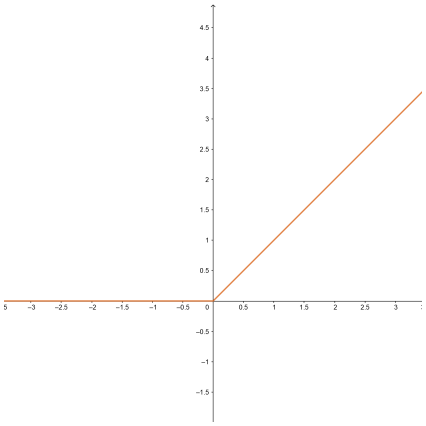


Figure 2.4: The ReLU activation function

ELU

ELU is a fairly new activation function. It aims to fix the dying ReLU problem by allowing negative values. ELU also tries to make the mean of the activations closer to zero which speeds up training.

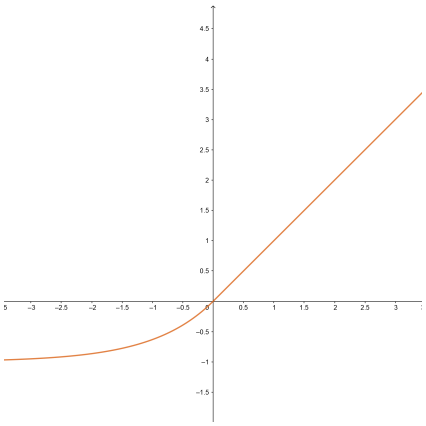


Figure 2.5: The ELU activation function

$$\text{ELU}\,(x) = \begin{cases} x, & x > 0 \\ \alpha(e^x - 1), & x \leq 0 \end{cases} \tag{2.5}$$

2.1.3 Training

Training a neural network is the process of finding the optimal value for each parameter in the network (for all the weights w_{ij} and biases b_{ij}). To achieve this, the weights and biases get initialised randomly and a cost function (often called a Loss function) is defined which assesses the quality of the network. It does this by describing how close the network's output is to its desired target for a certain input.

Cost functions

There are various different cost functions for numerous different applications. Here we use the most common function for regression problems: Mean Square Error (MSE). If a vector of n predictions is generated from a sample of n data points, and Y is the vector of observed values being predicted, with $\hat{Y}$ being the predicted values, then the MSE of the predictor is computed as in equation 2.6.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{2.6}$$

Back propagation

To adjust the parameters in the network the gradient descent optimisation algorithm is used. Here, the gradient for each parameter is calculated (the gradient of a parameter being the partial derivative of the loss in regards to that parameter). The value of each parameter is then updated by taking a step of size α (the learning rate) in the downward direction of its gradient. This process is repeated until a local minimum for the cost is reached.

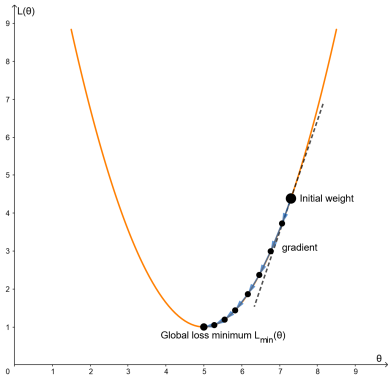


Figure 2.6: Gradient descent

Because neural networks are so complex, sometimes having hundreds of thousands of parameters, using classic calculus to derive each parameter would be extremely impractical. Thus, an algorithm called back propagation is used. It consists of two phases: the forward pass calculates both the result of the network at each layer as well as the local gradient of each layer with respect to its following layer. The backward pass then simply applies the chain rule to compute the gradients for each parameter with respect to the final cost. It *propagates* the cost backwards, hence the name back propagation.

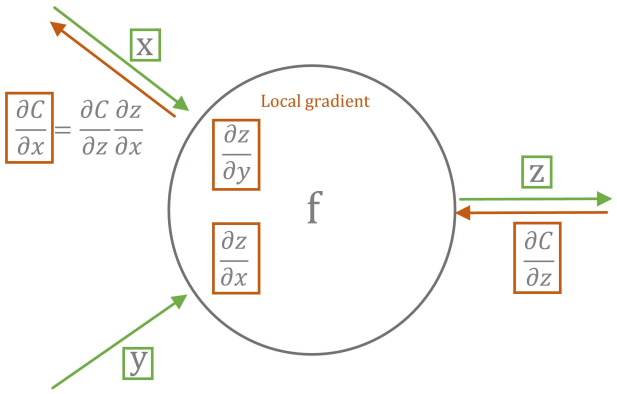


Figure 2.7: How back propagation would work through a single node.

Optimizers

The way parameters of a neural network are updated is determined by the used optimizer. Optimizers are mathematical functions that modify the network's parameters in order to minimise the cost. The gradients of the loss function act as a guide, telling the optimizer in what direction to move to reach a local minimum.

Stochastic Gradient Descent or SGD is a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). The weight update for SGD is given in equation 2.7, where α is the learning rate that dictates how much the weights get adjusted in each iteration.

$$w_{i,j}^{t+1} := w_{i,j}^{t} - \alpha \frac{\partial x}{\partial w_{i,j}^{t+1}} \tag{2.7}$$

Adam [5] is an adaptive learning rate optimization algorithm that aims to improve on SGD. It calculates an adaptive learning rate for each individual parameter using the mean (the first moment) and the variance (the second moment) of the gradient. The weight update can be seen in equation 2.8.

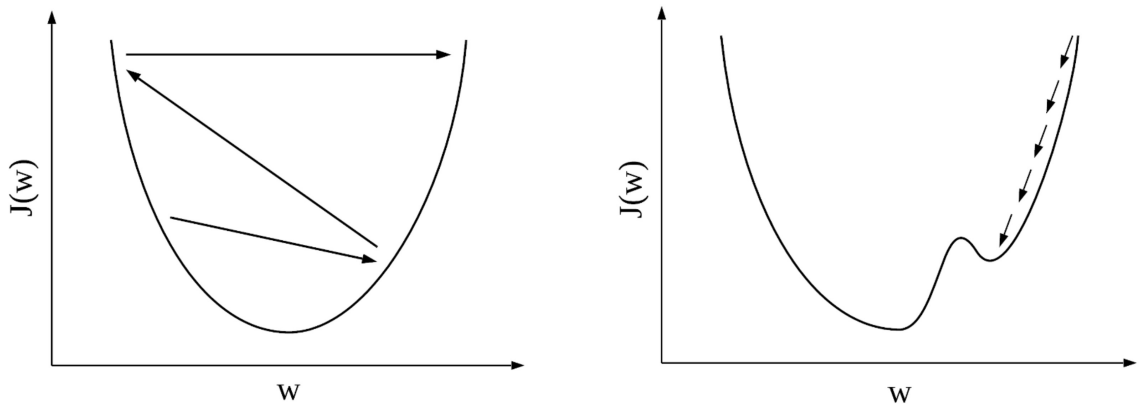
$$w^{t+1} := w^t - \alpha \frac{\hat{m}^t}{\sqrt{\hat{v}^t} + \epsilon} \tag{2.8}$$

Here, $\hat{m}^t$ and $\hat{v}^t$ are the first and second moment of the gradient, while ϵ is a constant with a typical value of 10^{-8}.

First published in 2014, Adam showed huge performance gains in terms of training speed. Unfortunately, it has been shown that Adam often finds a worse solution than stochastic gradient descent. A lot of research has since been done to address these issues.

Learning rate

The learning rate regulates how much the weights get adjusted in each training step. Choosing an optimal learning rate is important. When it is set too low the network will learn very slowly and it might fail to escape a local minimum, making it unable to reach a global minimum and thus an optimal solution. When set too high, the network might overshoot the global minimum causing the performance of the model (such as its loss on the training dataset) to oscillate over training (as illustrated in Figure 2.8).



(a) Large learning rate (b) Small learning rate

Figure 2.8: The effect of the learning rates. Figure (a) illustrates a large learning rate that is too high. Figure (b) depicts a small learning rate that gets stuck in a local minimum.

2.1.4 Momentum

By updating with only a subset of the data samples, the path taken by stochastic gradient descent will "oscillate" towards convergence. Momentum is a technique which considers the past gradients to smooth out the update. It computes an exponentially weighted average of the gradients and uses that to update the weights instead, making it converge faster than the standard gradient descent algorithm. How the weights are updated is shown in equation 2.9, where γ governs how much the weight updates are affected by previous weight changes.

$$w_{i,j}^{t+1} = w_{i,j}^{t} - \alpha \frac{\partial L}{\partial w_{i,j}^{t}} + \gamma \Delta w_{i,j}^{t+1} \tag{2.9}$$

2.1.5 Generalization

Generalization techniques are used to reduce the errors introduced into the network as a result of the choice of dataset[6]. All standard neural network architectures such as the fully connected multi-layer perceptron are prone to overfitting[7]. When a model over-fits, the error on the training dataset will keep diminishing to almost 0 whilst performance on unseen data will get worse. In this case, the network has started to memorise features which are unique to the training data instead of finding meaningful, general features. When a model performs poorly on both training and testing data however, it might be under-fitted. This might be because the network was not trained for long enough or is too simple for the task at hand.

Early stopping is a common generalization technique in which the evolution over time is tracked on a validation set. The validation set is a small part of the training data-set on which the model is not trained. Figure 2.9 shows an example of a training loss curve and a validation loss curve. This approach uses the validation set to anticipate the behaviour in real use (or on a test set), assuming that the error on both will be similar. The validation error is used as an estimate of the generalization error [8]. Validation error curves are rarely as smooth as in figure 2.9 and thus various criteria exist which determine when early stopping should actually take place.

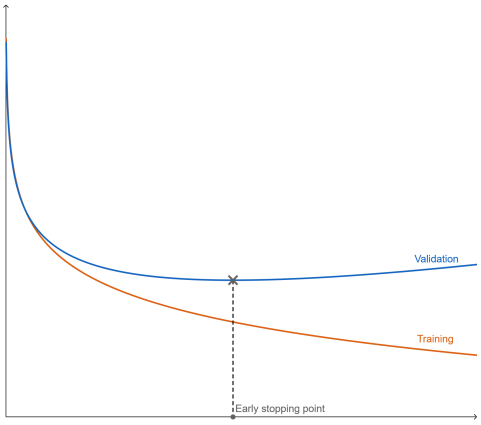
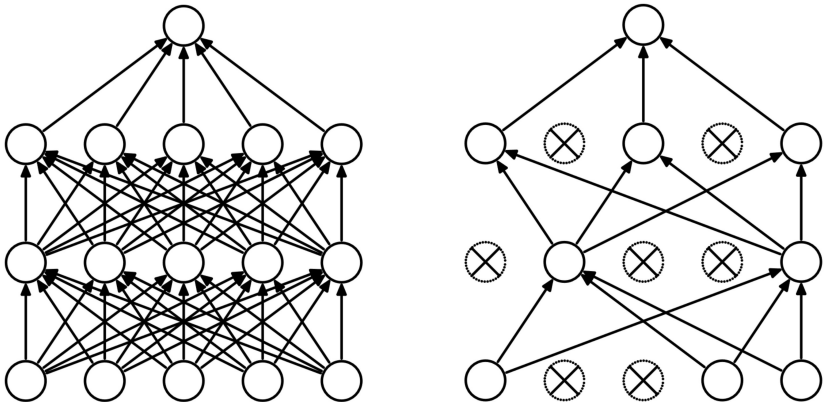


Figure 2.9: Early stopping

Dropout is another popular generalization technique. First introduced in 2014 [9], dropout aims to prevent overfitting by providing a way of combining many different neural network architectures into one. The term "dropout" refers to randomly dropping out units in a neural network by temporarily removing them, along with all their incoming and outgoing connections, as shown in Figure 2.10.



(a) Large learning rate (b) Small learning rate

Figure 2.10: Dropout Neural Net Model. Figure (a) illustrates a standard neural net with 2 hidden layers. Figure (b) is an example of a thinned net produced by applying dropout to the network on the left.

Data Augmentation is an approach in which the size of the dataset is artificially expanded by applying transformations on the existing data. If a model has a lot of parameters, it has to have a proportional amount of examples to learn from to get good performance. Transformations used for augmentation should keep all important features but expand the dataset. In the case of a dataset of images, transformations can be mirroring, rotating, shifting the hue, perspective transformation, adding noise (e.g. in the shape of a Gaussian blur), etc. The simple case of flipping each image around one axis already increases the dataset size by a factor of 2. Data augmentation is especially helpful when working with images, videos, and text sets.

Weight regularisation. While training neural networks, there is an opportunity for some very large weight values to crop up. This happens because these weights are focusing on certain features very specific to the training data which causes them to continuously increase in value throughout the training process. Huge weights are very susceptible to small changes resulting in many incorrect predictions arising on the test data, decreasing the generalisation of the neural network.

$$L1 : \lambda * \sum |W| \qquad (2.10) \qquad\qquad L2 : \lambda * \sum W^2 \qquad (2.11)$$

Weight regularisation includes part of the weights in the loss function, so that weights are also minimised whilst training. There are two methods of doing this, called L1 and L2 regularisation (equation 2.10 and 2.11, where λ signifies the extent of weight change). These expressions are simply added to the overall loss function of the neural network.

2.1.6 Batch normalisation

Batch normalisation is a method that normalises activations in a network across each mini-batch. For each feature, it subtracts the mini-batch mean and divides the feature by its mini-batch standard deviation (Equations 2.12, 2.13, 2.14). This forces the features to have a 0 mean and a unit standard deviation. To avoid problems where a large activation might have actually benefited the network's performance, batch normalisation adds two additional learnable parameters: the mean and magnitude of the activations (it scales the normalised activations and adds a constant, see Equation 2.15). This allows the magnitude and mean of the activations to be controlled independent of all other layers, essentially "smoothing out" the loss surface, making it easier to navigate.

$$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad (2.12) \qquad \sigma_\beta^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta) \quad (2.13) \qquad \hat{x_i} \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}} \qquad (2.14)$$

$$y_i \leftarrow \gamma x_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{2.15}$$

2.1.7 Transfer Learning

Transfer learning is a technique in which a model trained for a task is reused as the starting point for a model designed for a different task. It is an optimisation which allows for rapid progress or improved performance when modelling the second task. The source model is often pre-trained, but sometimes, just the untrained model is reused and trained from scratch. This may involve using all or only parts of the model, depending on the modelling technique used. Optionally, the model may need to be adapted or refined on the input-output pair data available for the task of interest.

This technique is especially useful for the early layers in convolutional networks, as features are more generic in early layers and more original-dataset-specific in later layers[10]. *Olivas Et al.*[11] describes three possible benefits from transfer learning: a higher initial skill (before refining the model), faster convergence (the rate of improvement of skill is higher during training) and a higher convergence (better final network performance).

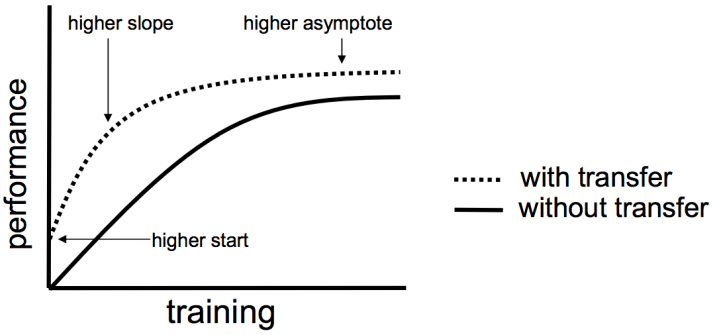


Figure 2.11: Ways in which transfer learning might improve training.

2.2 ANN Types

2.2.1 Convolutional Neural Networks

Regular neural networks don't scale well to full images. For images which are only of size 32x32x3 (32 wide, 32 high with 3 colour channels), a single fully-connected neuron in a first hidden layer of a regular neural network would have 32*32*3 = 3072 weights. Clearly this structure does not scale to larger images, as an image of e.g. 200x200x3, would already lead to

neurons that have 200*200*3 = 120,000 weights. A convolutional neural network (also known as CNN, or ConvNet) is a type of deep neural networks which takes advantage of the fact that the input is an image. The CNN can assign importance to various features in the input through the learnable weights and biases, independent of their location in the image. While in primitive methods, filters to find features in the image are hand-engineered, ConvNets have the ability to attain these filters through training. A CNN also keeps the original image structure, preserving the spatial information of the input where a typical ANN would flatten the input to a one-dimensional vector.

Every layer of a ConvNet uses a differentiable function to transform one volume of activations into another. There are three main types of layers to build a CNN architecture: Convolutional Layers, Pooling Layers, and Fully-Connected Layers.

Convolution Layer

Convolutional layer's parameters consist of a set of learnable filters. Every filter is spatially small, but extends through the full depth of the input volume. During the forward pass, each filter moves by a given amount across the width and height of the input (it convolves over the image) and computes dot products between the entries of the filter and the input at every position. This results in a 2-dimensional activation map which gives the responses of that filter at every spatial position. The amount of filters determines the depth of the output map. Padding may be added to the input to prevent a change in spatial dimension.

Intuitively, stacking multiple convolutional layers will result in the later layers activating on features with higher abstraction (E.g. an eye or an entire face) while early layers will learn to recognise very simple features such as horizontal or vertical lines.

Pooling Layer

Pooling layers are often periodically added in-between successive convolutions as a means to progressively reduce the spatial size of the data. This reduces the amount of parameters and computation in the network, also reducing overfitting. In max pooling, this is achieved through dividing each depth slice in the input volume into subsections of a small spatial size, say 2x2, and then simply taking the maximum of each section to form the output. Other techniques, such as average pooling or sum pooling, work in much the same way, only replacing the max operation.

Fully-Connected Layer

At the end of a series of convolutional and pooling layers (known as the feature extractor layers), there are usually one or more fully connected layers. These process the high-level features of the image, produced by the feature extractor, to form the final prediction.

2.2.2 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of neural network which allows the output from the preceding time step to be added to the current input. This combined input forms the internal (hidden) state of the network, which allows the RNN to exhibit dynamic temporal behaviour. RNNs can use their internal memory to process arbitrary sequences of inputs. This allows for four different configurations of input/output relations making it useful for wide variety of applications.

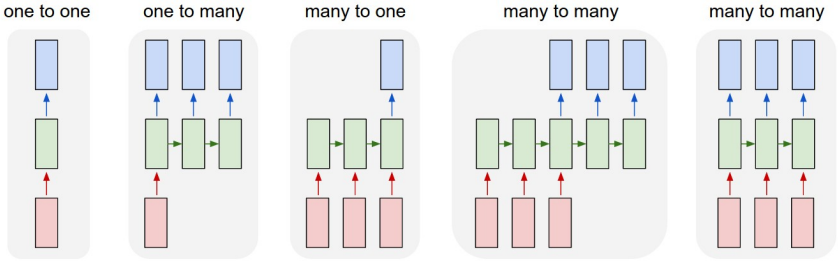


Figure 2.12: Types of RNN operations, from left to right: (1) one-to-one, (2) one-to-many, (3) many-to-one, (4) many-to-many

Figure 2.12 shows the possible configurations. Each rectangle is a vector: input vectors are red, output vectors are blue and green vectors hold the RNN's state. One-to-one is processing without RNN, from fixed-sized input to fixed-sized output. One-to-many can be used to produce an output sequence (e.g. image captioning takes an image and outputs a sentence) while many-to-one expects a sequence input and gives a fixed length output (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). Finally, Many-to-many processes one sequence and gives another sequence back (e.g. Translation: an RNN reads a sentence in English and then outputs a sentence in Dutch).

Long Short-Term Memory networks

The main appeal of RNNs is the idea that they might be able to connect previous information to the present task. This works well for short-term memory tasks and, in theory, RNNs are also absolutely capable of handling long-term dependencies. In practice however, this is not the case

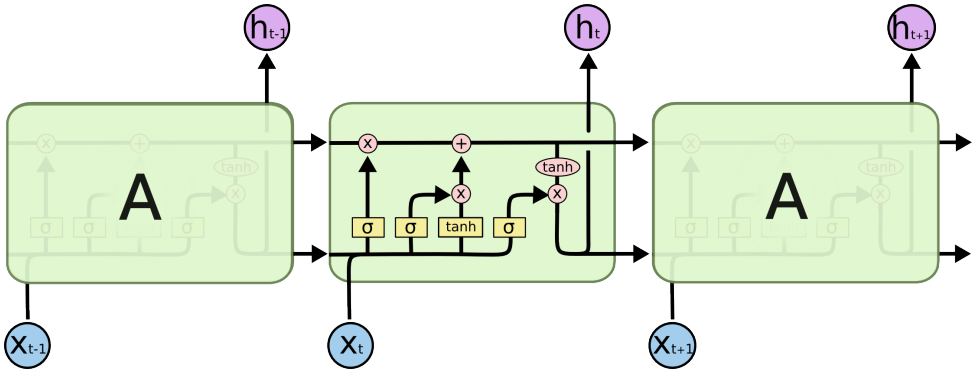


Figure 2.13: A repeating LSTM cell: yellow squares represent Neural Network Layers, red circles represent point-wise operations and arrows represent the flow of data

due to a trade-off between efficient learning with gradient descent and latching on to information for longer periods [12].

A Long Short-Term Memory network (LSTM)[13] does not have this problem. An LSTM cell has three inputs: a hidden state, a data input, and a cell state with the cell state holding long-term dependencies. At every time step, three gates regulate what information is kept (remember gate), thrown away (the forget gate) and added to the cell state (the input gate). The full architecture can be seen in Figure 2.13

2.2.3 Residual Neural Networks

The accuracy of Neural networks should increase with an increasing number of layers. This is only true to a certain point, after which the accuracy gets saturated and then degrades rapidly. The problem of *vanishing gradient* creeps up: in some cases, a weight's gradient becomes undesirably small, effectively preventing the weight from changing its value. If a network becomes sufficiently deep, it may not be able to learn even simple functions.

Residual layers address this issue by introducing skip connections: the input of a residual block gets added to its output before it is fed into the next block. Essentially, this allows the propagation of larger gradients to the initial layers (without passing through non-linear activation functions, which cause the gradients to explode or vanish) so that they can learn as fast as the final layers, giving the ability to train deeper networks.

$$M(x) = y \qquad (2.16) \qquad F(x) = M(x) - x \qquad (2.17) \qquad M(x) = F(x) + x \qquad (2.18)$$

Usually, a deep learning model learns the mapping, M, from an input x to an output y, as in

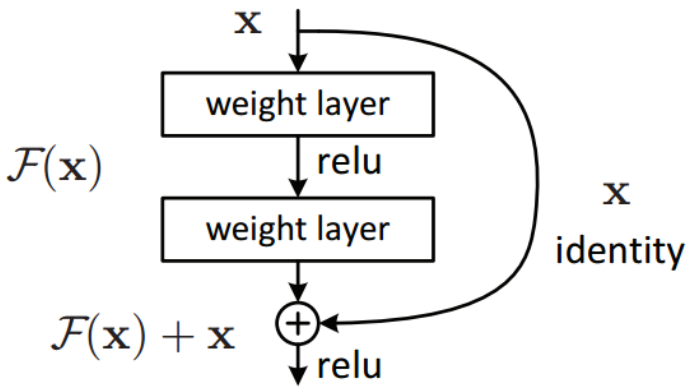


Figure 2.14: A single residual block

equation 2.16. Instead of learning a direct mapping, the residual function uses the difference between a mapping applied to x and the original input, x, as seen in equation 2.17. A skip layer connection thus uses equation 2.18.

This way, a residual neural net lets the layers directly fit a residual mapping, as it is easier to optimise this residual function $F(x)$ compared to the original mapping $M(x)$[14].

2.3 Teaching Autonomous Vehicles to drive

2.3.1 Mediated Perception

The mediated perception approach to autonomy connects multiple separate components which are each responsible for a different relevant subset of driving[15]. Some components may, for example, be responsible for processing sensory input (using computer vision techniques to detect lane-lines, traffic signs, traffic, ...) while other components can process the resulting information to perform decision making.

Considered as the state of the art technique, this approach is the most developed and the most widely adopted in the industry. This method however does not always work well in complex traffic situations that cannot easily be characterized by analytical models[16]. One reason for this is the utilisation of careful feature engineering, which increases the likelihood of missing important details. Another disadvantage of this technology is the high cost associated with hardware (e.g. LiDAR sensors and radar). Furthermore, most mediated perception approaches rely on creating high definition maps of roads, which renders the system unable to drive in unknown locations.

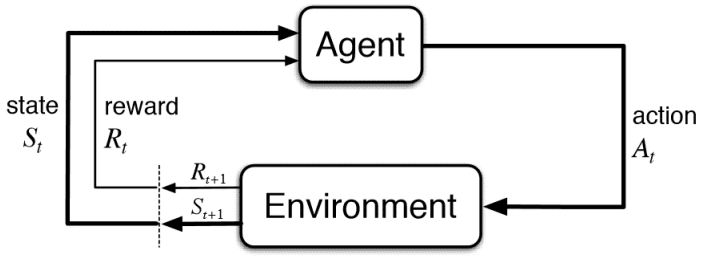


Figure 2.15: Reinforcement feedback loop. Starting at time step t, the agent observes the state s_t and reward r_t. When the agent takes action a_t, the environment returns a new state s_{t+1} and reward r_{t+1} for time-step $t + 1$.

2.3.2 End-to-end Learning

End-to-end Learning promises to address the steering, throttle, and braking predictions with a single neural network, greatly simplifying the process (the different components do not have to be designed and optimised separately with human intervention, minimising the required setup for a running solution), making it a hot topic in the research field.

Reinforcement learning approaches the problem by defining the vehicle as an agent following a policy. This agent has the ability to make actions in a dynamic environment, letting it explore. During exploration, a feedback is given to the agent in the form of reward. A possible reward metric for lane following could for example be the distance from the vehicle to the center of the lane[17] but designing reward functions can be challenging, as the center of the lane may not always be clearly defined. Such an agent, environment and reward system is traditionally known as a Markov Decision Process (MDP). The agent's policy is optimised using a deep neural network to maximise the expected future reward. An example of the reinforcement feedback loop can be seen in Figure 2.15. The exploration factor of reinforcement learning raises some serious safety concerns as agents need to be able to make mistakes in order to learn. This is probably a reason for development, outside of simulation, being slow. Recent work[18] has shown however, that the application of deep reinforcement learning to a full sized autonomous vehicle is possible and shows much promise.

Imitation Learning (also known as behavioural cloning) denotes a supervised learning technique in which a single network is trained to mimic an expert's actions. An imitation learning model is trained using a dataset of observations labelled with the corresponding correct decisions to be made. Generally, imitation learning is useful when it is easier for an expert to demonstrate the desired behaviour rather than to specify a reward function which would generate the same behaviour or to directly learn the policy.

In the case of autonomous driving, a dataset can consist of a recording of human driving. This

recording usually contains images, steering wheel angles, throttle and brake controls. The loss of the network can then be calculated by comparing the recorded controls with the prediction the network made based on the images. This way, the policy π_θ is adjusted so that the taken action for a given state s_i is closer the expert's action a_i for all recorded states and actions (see equation 2.19).

$$minimize \sum_i L(a_i, \pi_\theta(s_i)) \tag{2.19}$$

Relevant past experiments

ALVINN (Autonomous Land Vehicle In a Neural Network) is the first known project that attempts to use an end-to-end neural network for autonomous driving[19]. The used model is a 3-layer fully connected neural net. The input consists of a combination of a laser range finder and a gray scale forward-facing image for a total input resolution of 1217 units. The output layer consists of 45 nodes that represent points along the curvature the vehicle should follow in order to navigate. One node in the output layer is used to loop back to the input, meant to serve as a feedback for the road intensity.

Published in 1989, ALVINN was very limited by the computational power available at that time. Despite this limitation, the network was able to accurately complete a 400 meter path in a wooded area. Even back then, simulated road images were used for training. ALVINN has inspired many more recent end-to-end approaches.

DAVE (DARPA Autonomous Land Vehicle) was a project funded by DARPA, that explored the idea further[20][21]. The most noticeable improvements were the use of a CNN and a stereo camera using the YUV colorspace, which has shown to increase accuracy[16]. DAVE demonstrated the potential of end-to-end learning (it was used to justify starting the DARPA Learning Applied to Ground Robots (LAGR) program[22]), but performance was not yet sufficiently reliable to provide a full alternative to modern modular approaches (the mean distance between crashes is about 20 meters in complex environments[20]).

DAVE-2 by Nvidia[23], also known as PilotNet, aimed to prove the feasibility of end-to-end systems, building on the work of *LeCun et. al*[20]. The idea is that end-to-end learning leads to better performance and smaller systems because the internal components self-optimise to maximise overall system performance, instead of optimising human-selected intermediate criteria.

The proposed architecture consists of a CNN with 5 convolutional layers: the first three with a stride of 2 and a kernel-size of 5x5 and two non-strided with a kernel of 3x3 for the final two convolutions. The five convolutional layers are followed by three fully connected layers, leading

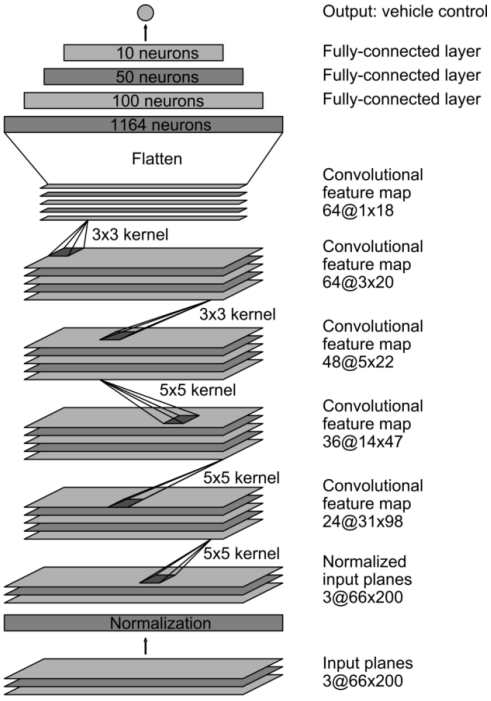


Figure 2.16: The PilotNet CNN architecture. The network has about 27 million connections and 250 thousand parameters.

the vehicle control as output. The full network architecture can be seen in Figure 2.16.

Data was collected using three front-facing cameras with a set offset and steering angles were recorded through the CAN bus of the vehicle. The training data was augmented using shifts and rotations so that the model learns to recover from mistakes.

For training, the network was fed images in order to predict a steering angle. These predictions were then compared to the ground truth by calculating the Mean Square Error. Based on this, the parameters in the network were adjusted through back propagation. After training, the model could accurately predict steering angles given a single front-facing image.

The PilotNet model was able to navigate various roads and terrains with very little error, driving autonomously 98 to 100% of the time. This promising result has generated a wave of interest and research in end-to-end approaches for autonomous driving.

Limitations

Even though the end-to-end approach has showed promising results, the technology has several limitations and challenges. One of the most problematic issues comes from the the fact that end-to-end approaches are entirely based on deep neural networks. They therefore inherit much

of the same problems. Known as *the black box problem*, deep neural networks are often criticized to be non-transparent: it is difficult to know what information a network uses as a basis for its prediction. This causes uncertainty on how the network may react to outliers in the data. Recently, a lot of research has been conducted to create explanators or explainers which try to point out the connection between input and output to represent, in a simplified way, the inner structure of machine learning black boxes[24].

Furthermore, the black box issue raises concern on deliberate attacks based on the properties of neural networks. Neural nets can be tricked into making wrong decisions based on deliberately crafted patterns in the input data[25]. In the case of autonomous driving, drawing a line on the road perpendicular to the driving direction can cause the vehicle to make a sharp turn which leads to a certain crash[26].

Approaches

The architecture proposed by *Bojarski et al.*[23], a feed-forward CNN, takes a single input image and gives the appropriate steering angle as output. This simple system is very streamlined and proved to work remarkably well. Since then, some approaches have improved this system further.

Spatiotemporal features

One such improvement introduces *spatiotemporal features* to the CNN. This follows the notion that humans don't make driving decisions based on single snapshots in time but also consider all events (points in space and time) leading up to that point. One way to add the notion of time to a convnet is through 3D convolutions[27]. Another proposed technique uses Long Short-Term Memory network (LSTM) cells which are connected to the output of the underlying CNN[28][29].

Both 3D convolutional layers and recurrent layers make use of temporal information, so these techniques combined greatly improve performance over the basic CNN[30]. On top of that, using residual networks (such as ResNet[14]) with transfer learning allows for deeper networks that converge faster and potentially to a better solution.

Conditional Imitation Learning

The CNN architectures discussed so far are really good at cloning the expert's driving behaviour but have no sense of their intent. A vehicle trained end-to-end to imitate an expert cannot be guided to, for example, take a specific turn at an upcoming intersection. Conditional Imitation

Learning aims to integrate the intent through high level navigational commands (e.g. go left, change lane, ...). In real-world scenario's, these commands could be triggered by navigation software or the car's turn-signals[31]. The commands are given as input during training, allowing the model to react differently in scenarios that require decisions.

In traditional Imitation Learning, the approximator $F(o;\theta)$ must be optimized to fit the mapping of observations o_i to actions a_i as in Equation 2.20. In contrast, the command-conditional imitation learning objective is given in Equation 2.21. The learner is additionally informed off the expert's expected behaviour c_i and it can use this extra information in predicting the appropriate action.

$$\underset{\theta}{minimize} \sum_i L(a_i, F(o_i;\theta)) \qquad (2.20) \qquad\qquad \underset{\theta}{minimize} \sum_i L(a_i, F(o_i, c_i;\theta)) \qquad (2.21)$$

Codevilla et al., 2019 "End-to-end Driving via Conditional Imitation Learning"[32] propose two techniques of implementing a conditional approximator. In the command input network, the images are fed into a CNN for feature extraction. Commands and environment measurements are then concatenated to the feature extractor's output before being fed into a fully connected network. This approach runs the risk of commands being ignored by the network.

The other proposed architecture uses the command not as input but as a selector. Measurements are still concatenated as before, but the network now has different fully connected sub-modules (also called heads) corresponding with the possible discrete commands. The command thus works much like a switch and is therefore guaranteed to have an effect during run-time. The disadvantage here is that commands can no longer be continues values, which is possible using command input network.

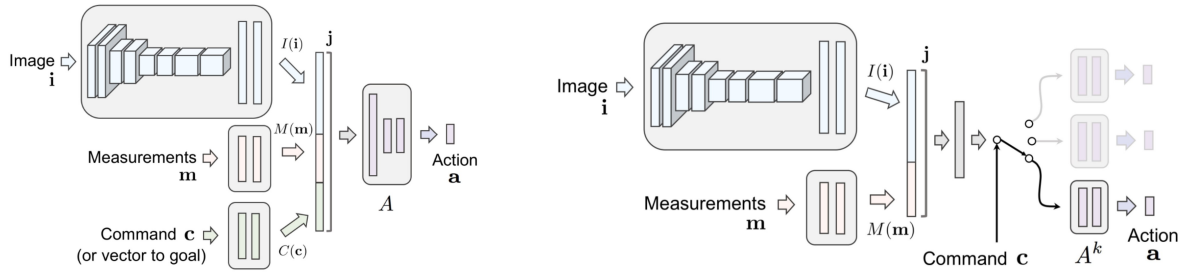


Figure 2.17: Two network architectures for command-conditional imitation learning. Figure (a) command input: the command is processed as input. Figure(b) branched: the command acts as a switch that selects between specialized sub-modules.

The authors tested both models in simulation and in a real-world suburban setting. The branched network outperformed the command input network significantly in both speed and

reliability.

2.4 Hardware

2.4.1 Nvidia JetBot

The JetBot development platform is an open-source development kit aimed at AI research. The robot (Figure 2.18) uses differential steering with two wheels in the front and a single caster-wheel in the back with a pair of drive motors which can be independently driven in both directions. The robot can drive forwards, backwards, turn and spin or pivot on the spot. A single wide-angle RGB camera is mounted to the front.

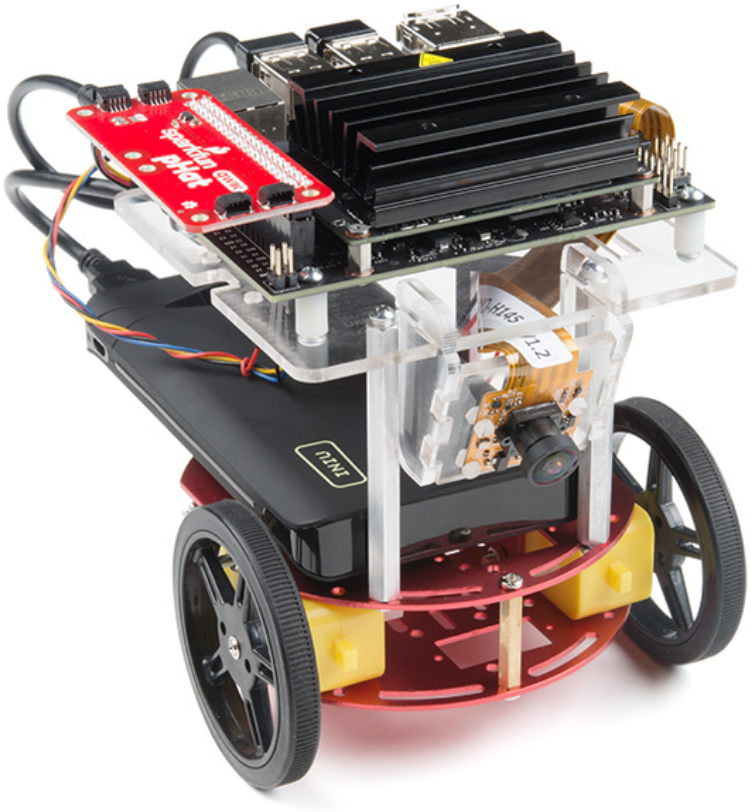


Figure 2.18: The SparkFun JetBot AI Kit, based on the open-source Nvidia JetBot

The kit is powered by an Nvidia Jetson Nano, a single-board computer with a Jetson powered System on Module (SOM) focused on running modern AI algorithms fast through Nvidia's CUDA framework (described below). The Jetson Nano delivers 472 GFLOPs of computations at a power usage of around 5 Watts.

Interfacing with the JetBot

The main way of interfacing with the JetBot out of the box is through the browser using JupyterLab. A python API allows for very simple and intuitive control of the robot's motors and camera. This is a great system to get started, but it does have limitations (mainly teleoperation is unreliable and can have major delays making it difficult to drive). A more complex but versatile way to use the robot is through ROS (The Robot Operating System), a set of software libraries and tools to build robot applications. ROS comes pre-installed on the JetBot but has a steeper learning curve than using the notebooks and is not mentioned in any of the included demos.

2.4.2 Computations with GPU's and CUDA

A Graphical Processing Unit or GPU is a specialised computation component designed to rapidly manipulate memory and perform computations on large blocks of data in parallel to accelerate the creation of images. Due to the increased popularity and demand of gaming and high complexity computer aided design (CAD), GPUs have been designed with rising internal parallelism and possibilities for vectorised programming.

The high level parallelism of modern GPUs has made them useful for more then just graphical processing. ANNs are represented in memory as mostly vectors, meaning that both training and prediction can be parallelised. The Compute Unified Device Architecture (CUDA) is a platform developed by Nvidia which allows interfacing with their GPUs for general computing.

2.5 Software

2.5.1 CARLA simulator

CARLA[33] is an open-source simulator for autonomous driving research. It provides a realistic virtual environment with assets such as urban layouts, buildings, vehicles and pedestrians. The simulation platform is highly customisable with the ability to change the behaviour of dynamic actors (e.g. pedestrians or vehicles), changing the environmental conditions (such as the weather) and customising the sensor-array with an extensive library of sensors to choose from.

CARLA is currently under active development and is therefore constantly updated. At the time of writing this thesis, there are 10 pre-built maps each having a different style and layout in order to cover many different driving scenarios.

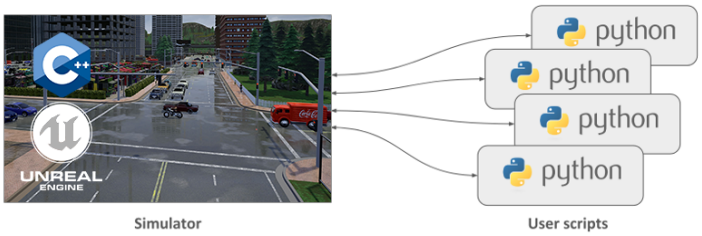


Figure 2.19: CARLA client-server structure

The simulator consists of a scalable client-server architecture where the server handles everything related to the simulation itself (rendering, sensors, physics, ...) and the client consists of a sum of modules controlling the logic of actors on scene and setting world conditions. The CARLA API is a layer that mediates between server and client and can be accessed in either Python or C++.

The server is built on the Unreal Engine 4, a general purpose 3D creation platform mostly used for game development. This allows for best in class graphics and performance with support for running the simulation on GPUs.

Other than that, CARLA has many advanced features designed to make research on autonomous driving easier. A built-in traffic control module controls vehicles besides the one used for learning to recreate urban-like environments with realistic behaviours. The recorder feature allows to record and replay complete scenarios. On top of this, CARLA can integrate with other learning environments through ROS bridge and Autoware implementation.

2.5.2 The PyTorch framework

PyTorch is an open source machine learning library, primarily developed by Facebook's AI research lab (FAIR). The library is primarily python based but is also available in C++, all be it less polished.

PyTorch provides two high-level features: Tensor based computing and automatic differentiation. Tensor based computing allows for high parallelisation using GPUs. Deep neural networks are built on an automatic differentiation system, taking advantage of the sequence of operations to differentiate gradients through the chain rule.

Tensors are homogeneous multidimensional vectors that hold numbers (integers, floating points, ...). These tensors have the added functionality of supporting CUDA operations.

Pytorch's Autograd module provides the automatic differentiation functionality but can be difficult to use. The nn module thus provides a higher-level way to create networks.

When comparing PyTorch to its main competitor, Tensorflow, the most important difference is the way these frameworks define the computational graphs. While Tensorflow creates a static graph, PyTorch uses a dynamic graph. In Tensorflow, you first define the entire computation graph of the model and then run it. In PyTorch, the graph is defined/manipulated during run time, which is particularly useful for training models with variable length inputs.

3

Methodology

3.1 Data collection and preparation

Due to the specific annotations needed for training, it is unfeasible to use a pre-existing dataset. Two different datasets were collected. The first was collected in simulation, with a realistic depiction of a real-world driving environment using a full sized car in an urban environment. The second dataset was collected using the Jetbot robot using its front-facing camera driving around on a miniature urban-like environment built up out of varying intersections.

3.1.1 Simulation

The simulator of choice was CARLA (discussed in 2.5.1). Out of all the simulators considered, CARLA was chosen due to the high level of customisation available as well as the high level of photo-realism.

The used setup for collecting data consisted of an agent vehicle, controlled through CARLA's Python API. Varying pre-defined paths were laid out through town 1. This map is situated in an urban environment and consists of two-lane roads and intersections. The map also features multiple buildings and areas with vegetation.

data	description
Center image	Image from the vehicle's center camera
Left image	Image from the vehicle's left camera
Right image	Image from the vehicle's right camera
High Level Command	The current navigational input represented as a number
Vehicle control signal	The signal the vehicle receives during recording (steering, throttle, brake)
PID control signal	The signal the PID controller generates (steering, throttle, brake).
Speed	The current speed of the vehicle
Speed Limit	The current speed limit

Table 3.1: Recorded data in CARLA

A PID controller controls the agent-vehicle and acts as the master driver. The route is recorded through 3 cameras mounted on the front of the vehicle. On top of this, the high level commands needed to navigate the route, as well as environmental information are stored alongside the camera images as can be seen in table 3.1.

This setup of using three cameras all facing forward positioned on the center, left, and right side of the vehicle was inspired by *Bojarski et. al*[23]. Each camera produces a 350x160 RGB-image. The central camera represents the car's actual viewpoint while the cameras on the side are meant to expand the dataset through the perspective of a vehicle that has drifted out of lane.

In order to teach the vehicle to correct itself in non ideal situations, a random noise was injected into the steering angle during recording, forcing the vehicle to drive slightly to the left or right for a brief moment. Only the autopilot's response to this noise was then actually recorded.

This data-collection was repeated in different weather conditions consisting of cloudy sky, clear sky, hard rain, medium rain, soft rain, clear wet roads and cloudy wet roads. This makes the data more generalised and allows the model to learn to ignore puddles, changes in ambient light and shadows.

3.1.2 Jetbot

In order to run experiments with the jetbot platform, a testing environment was created by applying painters tape in an urban street-like pattern to the floor as seen in Figure 3.1. This layout contains four types of road sections: T-junctions, crossroads, straights and corner roads.

The robot was controlled using a game controller. To achieve this, the joystick x and y position needed to be translated to a differential drive model (also known as tank drive or skid steering).

Figure 3.1: The testing and training data collection environment for the jetbot.

Deriving a control algorithm for this model requires combining two concepts: drive and pivot. It is simple to determine a mapping between a joystick X-Y input and the drive output or between a joystick X input and the pivot output. Combining the two, however, is less intuitive. The used algorithm blends the two concepts based on the Y input.

The drive mapping takes priority except when close to the midpoint of the joystick Y position at which point pivot operations get prioritised. The conversion algorithm can be implemented in a few component steps:

1. Calculate the drive turn output from the joystick X input.
2. Scale the drive output using the joystick Y input.
3. Calculate the pivot output from the joystick X input.
4. Calculate the drive vs pivot scale using the joystick Y input.
5. Calculate the final mix of the calculated drive and pivot.

Navigational input (turn left, go straight, ...) is given through the controller by pressing one of the D-pad buttons. Driving consistently using this system proved to be difficult, so data was collected in short segments. A record of what intersections had been used as well as which navigational commands were given was kept during the whole data collection process. This way, the dataset could be balanced while recording, so less post-processing was necessary.

The Jetbot's camera records 24 images per second. Alongside these images, the appropriate navigational input is stored as well as the left and right wheel speeds. The relative info of each datapoint was simply stored in the filename of the corresponding image. An overview of the content of these datapoints can be found in Table 3.2.

data	description
Image	Image from the vehicle's forward facing camera
High Level Command	The current navigational input represented as a number
Left wheel speed	The speed setting for the left motor
Right wheel speed	The speed setting for the right motor

Table 3.2: Recorded data using the Jetbot

3.1.3 Balancing the dataset

Balancing the CARLA dataset

Dropping and duplicating datapoints is a simple technique that is often used to balance datasets. In the case of an RNN however, it is important not to interfere with the temporal information of the dataset. To achieve this, two techniques were tested.

The first attempt tried to keep as much temporal information as possible by duplicating data. This was achieved by splitting each episode in the dataset into segments according to their navigational commands. This created five segment pools: straights, left turns, right turns, straight lane follows and lane follows through corners.

The balanced dataset was then built up by repeatedly selecting a segment from one of the segment pools at random. Once all segments in a pool are used, selection for that type restarts at the beginning, essentially duplicating those segments. The selection of which type of segment is taken next is weighted to keep the distribution of segments equal while building the dataset. These weights are based on the average length of the segments per pool compared to the highest average segment length. This causes segment types with less information to be duplicated more. The effect of this balancing can be seen in Figure 3.3.

For the second balancing technique, no data was duplicated. Instead, the dataset was balanced in the sequencer (the part of the code responsible for picking each data point/sequence of datapoints to train on). The dataset was divided into segments of the correct sequence length used for training. Each of these segments was represented by its most dominant navigational command (e.g. if a segment has 10 datapoints where 4 are lane following but 6 are a right turn, that segment is counted as a right turn segment). Segments were then shuffled and dropped in order to reach the desired ratios. The result of this balancing technique is visible in Figure 3.4

Finally, the steering angles were balanced out. Small steering angles were dropped to make the distribution more even. The result of this can be seen in Figure 3.5.

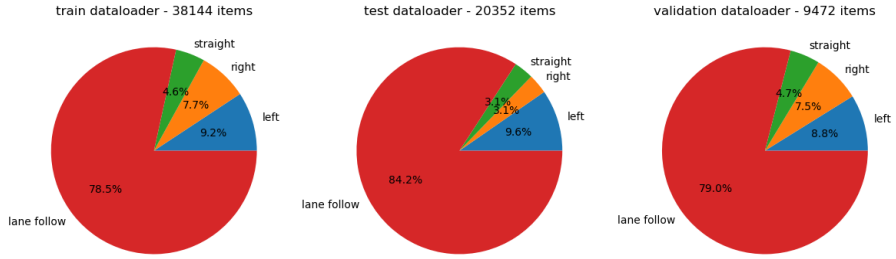


Figure 3.2: Dataset distribution before balancing

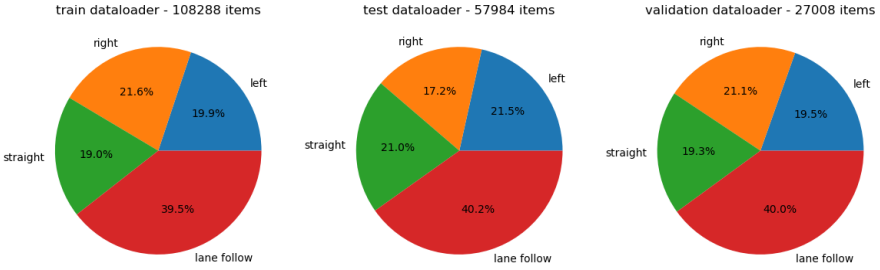


Figure 3.3: Dataset distribution after balancing through duplication.

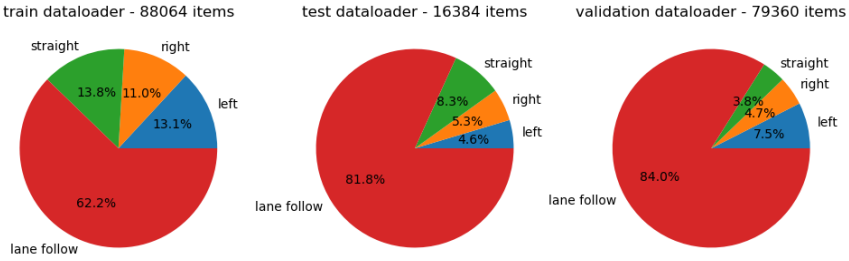
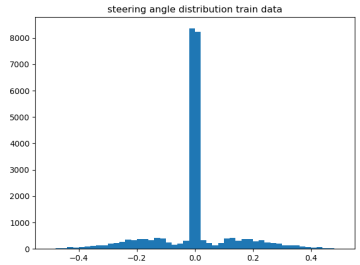


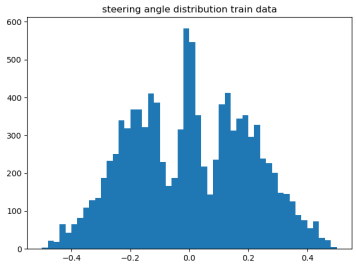
Figure 3.4: Dataset distribution after balancing by dropping data.

Balancing the jetbot dataset

Most balancing based on the navigational commands of the jetbot dataset happened during data collection by keeping track of how many segments were recorded of each type. On top of this, the entire dataset was mirrored to mitigate bias towards any one direction. This technique was not possible for the simulation's data as this would motivate the vehicle to drive into the oncoming lane without trying to recover.



(a) Steering angle histogram before balancing.



(b) Steering angle histogram after balancing.

Figure 3.5: Steering angle distribution before and after balancing.

3.1.4 Training the model

Both the models for use in simulation and for the jetbot were trained in the same way. The only difference being the output of a jetbot model was a pair of values representing the absolute speed of each motor, where the output of a simulation model was three values corresponding with the steering angle, throttle and brake.

Training procedure

All architectures were implemented using PyTorch (Section 2.5.2) and trained on an Nvidia GeForce 2080 SUPER GPU, using the CUDA computational framework (Section 2.4.2).

A custom data loader and sequencer organise the dataset into usable batched data-sequences for training. The data was split into 80% training data and 20% validation data. The models were all trained using the Adam optimizer (Equation 2.8) with a learning rate of 1e-4. The model's weights were periodically saved throughout the training process so that the weights associated with the smallest overall error could be selected for testing. In other cases, models could be compared after the same amount of training steps, regardless of how long that model actually trained for.

Data augmentation

Augmentation proved to be absolutely crucial to generalise the model and allow it to navigate previously unseen environments and situations. During training, each image was first cropped by removing the top 50 pixels, removing much of the sky, which was considered not to contain any useful information. The cropped image then had a random chance to be augmented using one or more transformations. These transformations consisted of random shifts in brightness, changes in hue, adding a Gaussian blur over the entire image and adding random dark spots to

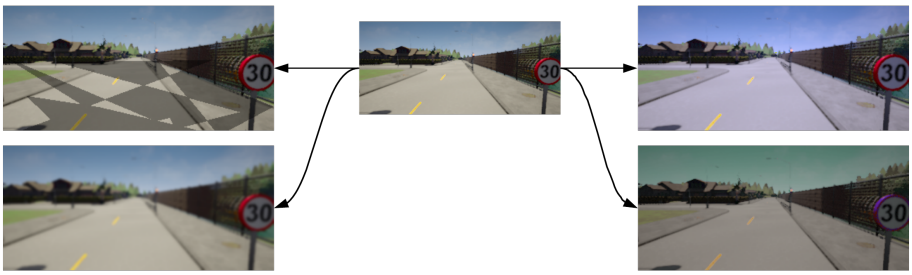


Figure 3.6: Used data augmentations: (top left): simulated shadows, (top right): random brightness shift, (bottom left): Gaussian blur, (bottom right): random shift in hue.

the image to simulate shadows. The augmentations are shown in Figure 3.6.

3.1.5 Model architectures

One of the goals of this thesis is to explore the performance of different architectures and what the impact is of different components/aspects of these architectures.

Plain CNN architecture

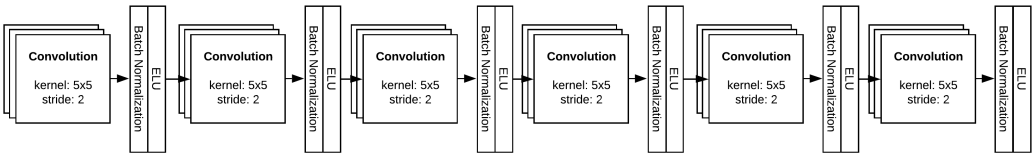


Figure 3.7: Architecture of the CNN Feature extractor.

The first model is based on the DAVE-2 architecture[23]. It is built up out of 6 convolutional layers followed by 4 fully connected linear layers. The model takes as input a sequence of RGB images of shape 3x110x350 and concatenates them depth-wise into one input tensor. After concatenation follows the convolutional layers. The first three layers use a 5x5 filter with a stride of 2, while the last three use a 3x3 filter with a stride of 1. All convolutional layers use the ELU activation function (Equation 2.5) and apply batch normalization (chapter 2.1.6). The output is then flattened into a one dimensional vector of length 1024.

Next, the output is concatenated with the navigational input (a one-hot encoded value of length 6) and external state information, producing a vector of length 1033. The concatenated vector is

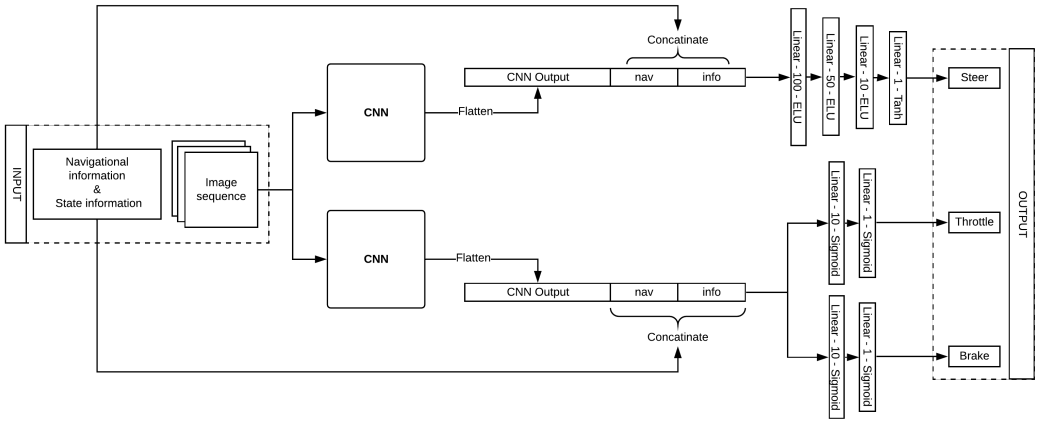


Figure 3.8: Plain CNN model architecture. See Figure 3.7 for the architecture of the CNN modules.

then fed through three different classifier blocks: one predicting the steering angle, one predicting the throttle and one predicting the brake. Dropout was applied to all non-output fully connected layers (Section 2.1.5). The steering classifier used the Tanh activation function (Equation 2.3) on the last layer, allowing the output to range between -1 and 1. All other linear layers used the Sigmoid activation function (Equation 2.2).

LSTM architecture

The second model shares much of the same architecture of the plain CNN but adds an LSTM module with 10 hidden states between the CNN and the classifier heads. A sequence of outputs of the CNN gets fed into the LSTM module at a time. The hidden state produced by the input at each time step is concatenated to the next input of the sequence. At the last step, the output is directed into the (now less complex) classifier heads for steering, throttle and brake. The architecture can be seen in Figure 3.10 with a detailed view of the LSTM module in Figure 3.9.

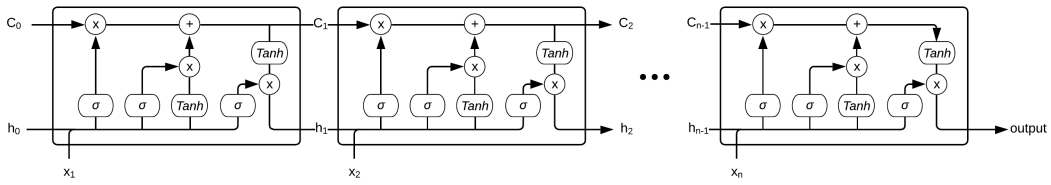


Figure 3.9: Architecture of the LSTM modules for a sequence length of n. The internal cell states C_{0-n} have 10 features each.

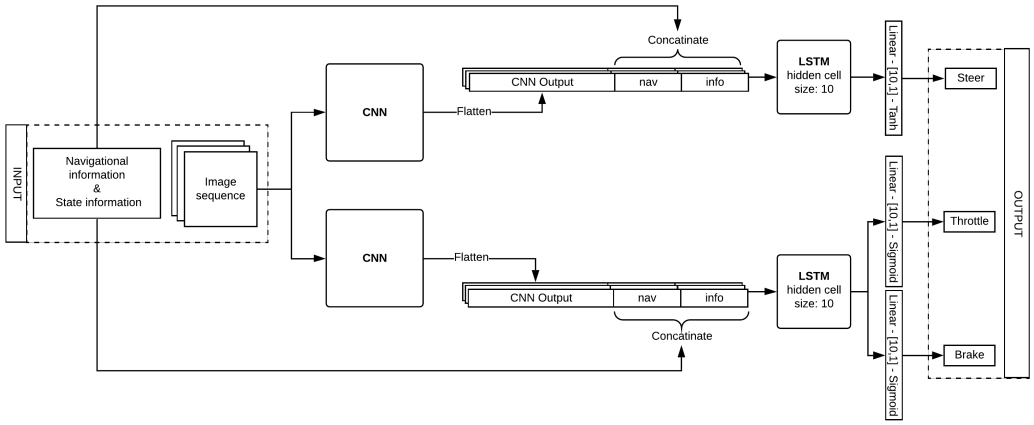


Figure 3.10: LSTM model architecture. See Figure 3.9 for the architecture of the LSTM modules and Figure 3.7 for the CNN modules.

Classification architecture

The assumption for this architecture change is that portraying a problem as a deep classification task often performs better than performing direct regression[34]. This classification can be implemented by making the task of the network to learn a sine wave that encodes the actual steering angle in its phase shift[29]. This introduces a spatial relationship between the nodes in the output layer (neurons close to each other are more similar).

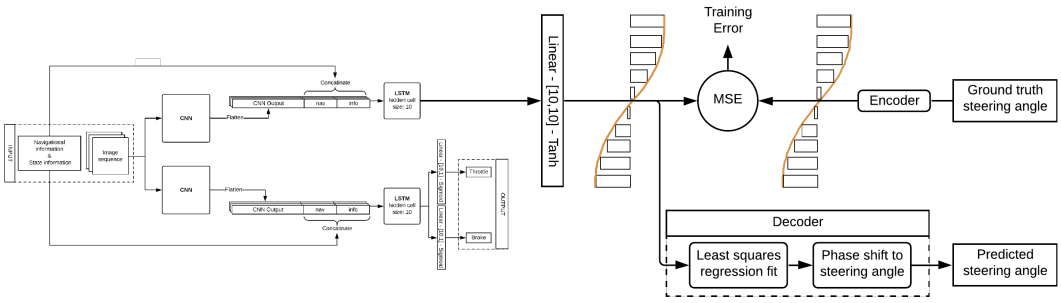


Figure 3.11: Usage of sine encoder/decoder for generating the training error and for making predictions. On the left is the LSTM architecture of figure 3.10.

The implementation of this, shown in Figure 3.11, takes any of the previous architectures and changes the number of output layers to 10. During training, the ground truth steering angle is encoded into a sine wave using Equation 3.1, where Y is a 10-dimensional vector (representing the sine wave), θ is the steering angle and θ_{max} is the maximum possible steering angle. The training error between this encoded steering angle Y and the output of the network is calculated

using MSE (Equation 2.6).

$$Y_i(\theta) = \sin \frac{2\pi(i-1)}{10-1} - \frac{\theta\pi}{2\theta_{max}} \tag{3.1}$$

For deployment, the sine wave needs to be converted back into a steering angle. The output from the network gets fitted into a clean sine wave using Least Squares Regression (LSR). The steering angle is then extracted from the sine wave using equation 3.2 where ϕ is the phase shift.

$$\theta = -\frac{2\theta_{max}\phi}{\pi} \tag{3.2}$$

3.1.6 Model analyses

Grad-CAM

In order to visualise what a trained model is looking at, a method called Gradient-weighted Class Activation Mapping (Grad-CAM) was used[35]. This technique uses the gradients of any target concept, flowing into a convolutional layer to produce a coarse localisation map highlighting important regions in the image for predicting the concept.

(a) Regions of interest in clear weather. (b) Regions of interest in rainy weather (left turn).

Figure 3.12: Example heat maps of regions of interest from the 3rd layer of the feature extractor.

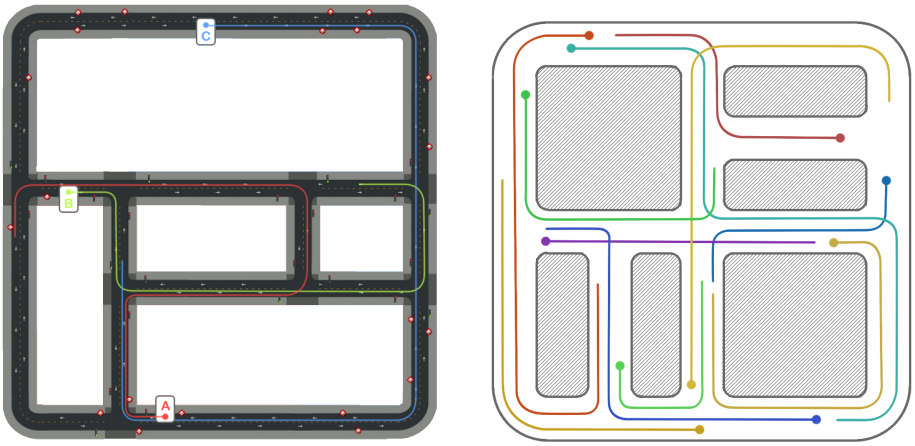
Figure 3.12a shows that the trained model (in this case the LSTM model with sine encoding) looks mostly for lines on the edge of the roads as well as the lane markings. In a rainy scenario, like in figure 3.12b, the model is less focused, but still manages to find road edges.

3.1.7 Experimental setup

CARLA test setup

The performance of each model was tested by letting it control a vehicle in previously unseen environments. The model was continuously fed images from the vehicle's forward facing central camera, while the model's predictions were applied real-time as control signals. Three different

routes were defined in which the model was expected to navigate in five distinctive weather conditions (clear noon, clear sunset, wet sunset, heavy rain at noon, and soft rain at sunset).



(a) CARLA Town 2

(b) JetBot test environment

Figure 3.13: Routes for evaluating the performance of models.

The routes were positioned in Town 2, testing the model's ability to operate in urban environments. This gives the model 15 different weather/route permutations to drive through, resulting in a combined distance of 5 km. A test is considered complete if the vehicle completes the route, gets stuck, ignores a navigational command or starts driving into the oncoming lane. The test routes can be seen in Figure 3.13a.

Jetbot test setup

The real-time performance of the models trained for the Jetbot were tested by letting the model navigate several predefined paths in the environment described in chapter 3.1.2. These 11 paths, shown in Figure 3.13b, were chosen to include as many different situations (variation in type of manoeuvres, background props, lighting conditions) as possible. Performance was then measured by what percentage of tracks were completed successfully.

4

Experiments and results

4.1 Experiment 1: The effect of dataset balancing

4.1.1 Setup

Goal The goal of the first experiment is to identify the best balancing technique out of the two techniques described in chapter 3.1.3.

Training Three models were trained on the same dataset but with different balancing techniques: one without any balancing, one using the drop technique and one using duplication. For this, the LSTM model, described in chapter 3.1.5, was used with a sequence length of 10.

Testing Each trained model was gauged by their achieved performance from a single run of the real-time test. The procedure for the real-time test is described in chapter 3.1.7. Due to the different sizes in datasets, the performance was measured at specific training steps instead of epochs.

4.1.2 Result

The results from table 4.1 show that the model trained on the dataset balanced with duplication performed substantially better. All models were tested after 11k steps of training (highlighted in gray). The duplication model yielded a performance of 37.28% at 11k training steps where the model balanced by dropping only completed 19.92%. The test results taken at 23k training steps for the duplication model performs better in sunset conditions but actually worse in noon conditions. The worst performing model was that trained without applying any balancing techniques, completing only 15.99% of the courses after 11k training steps.

balancing	training step	Clear Noon	Clear Sunset	Hard rain Noon	Soft rain Noon	Wet Sunset	Avg.
None	11818	14.70%	14.70%	21.16%	14.70%	14.70%	15.99%
None	19282	10.36%	38.00%	15.28%	17.85%	33.66%	23.03%
Drop	6048	17.85%	16.53%	26.65%	14.70%	16.53%	18.45%
Drop	11718	30.10%	10.36%	13.51%	11.96%	33.66%	19.92%
Duplication	5984	36.52%	38.00%	10.36%	14.70%	29.20%	25.76%
Duplication	11968	33.66%	44.98%	22.71%	64.17%	20.87%	37.28%
Duplication	**23936**	**13.91%**	**63.92%**	**30.10%**	**38.00%**	**63.92%**	**41.97%**

Table 4.1: Average route completion using different balancing techniques (models with the same amount of training steps are highlighted in gray)

balancing	training step	Object collision	Stuck	Lane invasion no recovery	Lane invasion with recovery	Ignore command
None	11818	6	4	5	0	6
None	19282	6	2	5	1	6
Drop	6048	13	7	7	5	1
Drop	11718	7	4	11	6	0
Duplication	5984	3	3	4	1	7
Duplication	11968	5	3	3	2	5
Duplication	23936	3	2	5	2	3

Table 4.2: Total failures using different balancing techniques

The total amount of failures of each model in the test can be found in Table 4.2. The model trained without balancing tends to ignore navigational commands, causing the test to be negative. The drop models mostly fail due to driving into the oncoming lane. Besides that, they make a lot of object collisions, which do not have the immediate effect of having the test fail but can cause the vehicle to get stuck, which does end the test. The best performing model, trained with duplication for 23k steps, had the main issue of ignoring navigational commands.

4.1.3 Discussion

The worse performance of the model trained on the dataset balanced by dropping is likely the direct effect of the smaller dataset. Another difference between these balancing techniques was the way the data was split into segments, but this should have little to no effect on performance since in both techniques, the sequence information of the data was kept intact completely. The results also show the overall importance of balancing, as the model trained without any balance has by far the worst performance.

The model trained with dropping data seems to perform best with regards to following commands. This is unfortunately not because this model is better at this task, but rather because the model usually failed before it reached difficult intersections (where other models were prone to ignore these commands).

4.2 Experiment 2: Classification vs. direct regression

4.2.1 Setup

Goal The proposed architecture in *Aasbø and Haavaldsen, 2019*[1] uses classification as output of the model (as described in section 3.1.5) instead of direct regression for the steering angle. This way, there is a correlation between the output neurons, bridging the gap between the regression and the classification problem. This experiment compares regression and sine-encoded classification by comparing their performance.

Training Two models were trained: one uses an LSTM architecture that outputs the steering angle directly, while the other model outputs 10 values representing a steering angle encoded in a sine wave. Both models used a sequence length of 10 and were trained for 23k steps (8 epoch).

Testing Each trained model was gauged by their achieved performance from a single run of the real-time test. The procedure for the real-time test is described in chapter 3.1.7.

4.2.2 Result

The results in table 4.3 show that the sine encoded architecture performs better than the direct regression architecture, attaining an average completion of 69.23% of the tracks, compared to the 41.97% of the other model. It is also notable that the sine encoded model performs more consistent across the different weather conditions.

model	training step	Clear Noon	Clear Sunset	Hard rain Noon	Soft rain Noon	Wet Sunset	Avg.
LSTM	23936	13.91%	63.92%	30.10%	38.00%	63.92%	41.97%
LSTM + sine	**23936**	**79.85%**	**85.18%**	**40.86%**	**65.03%**	**75.26%**	**69.23%**

Table 4.3: Average route completion with or without sine encoding

Table 4.4 presents the type of failures each model made during the test. The sine encoded model mostly failed because it ignored the given navigational commands while making overall less technical mistakes than the direct regression model.

model	training step	Object collision	Stuck	Lane invasion no recovery	Lane invasion with recovery	Ignore command
LSTM	23936	3	2	5	2	3
LSTM + sine	23936	3	3	1	2	6

Table 4.4: Total failures with or without sine encoding

4.2.3 Discussion

The sine encoded model performed noticeably more stable than the regression model. The turns looked smoother and performance was more consistent over the different weather conditions.

4.3 Experiment 3: Combining the feature extractors

4.3.1 Setup

Goal The proposed architecture in *Aasbø and Haavaldsen, 2019*[1] uses two separate feature extractors for the tasks of steering and acceleration (see chapter 3.1.5). Normally when a CNN has multiple outputs or functions, only one feature extractor is used. This supposedly forces the convolutional layers to learn features that generalize better for the task at hand. The counter argument to this is that each feature extractor can learn features that are better for one specific goal. This experiment investigates if the architecture benefits from the dual CNN.

Training Two models were trained: one uses the architecture as described in section 4.3 and one uses a modified architecture where only one feature extractor is used. Both models were trained with a sequence length of 10 and a batch size of 32.

Testing Each trained model was gauged by their achieved performance from a single run of the real-time test. The procedure for the real-time test is described in chapter 3.1.7.

4.3.2 Result

The results in table 4.5 show that the model using two feature extractors performs a great deal better, with a route completion of 69.23% compared to 23.43% for the other model.

model	training step	Clear Noon	Clear Sunset	Hard rain Noon	Soft rain Noon	Wet Sunset	Avg.
One F. extr.	14960	36.52%	14.70%	10.36%	40.86%	14.70%	23.43%
Split F. extr.	**23936**	**79.85%**	**85.18%**	**40.86%**	**65.03%**	**75.26%**	**69.23%**

Table 4.5: Average route completion using one or two feature extractors

Table 4.6 highlights that the model with one feature extractor both drives more unstable as well as ignores more navigational commands.

model	training step	Object collision	Stuck	Lane invasion no recovery	Lane invasion with recovery	Ignore command
One feature extr.	14960	7	3	3	4	9
Split feature extr.	23936	3	3	1	2	6

Table 4.6: Total failures using one or two feature extractors

4.3.3 Discussion

Using two separate feature extractors for the tasks of steering and acceleration is clearly beneficial. The feature extractor responsible for steering can specialize more. Important to note is that the model with one feature extractor is vastly less complex, making it less capable of learning complicated tasks.

4.4 Experiment 4: The effects of different architecture aspects

4.4.1 Setup

Goal The goal of this experiment is to figure out the effect of temporal information by removing the LSTM module and to study if using a more complex feature extractor positively impacts

the model's competency. In layman's terms, this experiment is meant to unveil what aspects of the architecture have the most effect on overall performance.

Training Three models were trained: the CNN-LSTM combo from section 3.1.5 with a batch size of 32 and a sequence length of 10; the plain CNN from section 3.1.5 with a batch size of 32 and a sequence length of 3 and a model using ResNet as the feature extractor instead of the simple CNN with a batch size of 20 and a sequence length of 8. The batch size and sequence length of the ResNet model had to be reduced due to memory limitations, but this should have negligible influence on the final result.

Testing Each trained model was gauged by their achieved performance from a single run of the real-time test. The procedure for the real-time test is described in chapter 3.1.7. All trained models use the sine-encoder trick from section 3.1.5.

4.4.2 Result

The plain CNN model performed by far the worse, only completing 17.97% of the routes on average but actually completing only 14.70% in almost every weather condition except in "hard rain noon", where it achieved 31.08%. The ResNet model performed second to worst with an average completion of 37.50%. These results can be seen in Table 4.7.

model	training step	Clear Noon	Clear Sunset	Hard rain Noon	Soft rain Noon	Wet Sunset	Avg.
CNN	22440	14.70%	14.70%	31.08%	14.70%	14.70%	17.97%
CNN + LSTM	**23936**	**79.85%**	**85.18%**	**40.86%**	**65.03%**	**75.26%**	**69.23%**
ResNet + LSTM	21537	34.01%	19.53%	36.52%	60.91%	36.52%	37.50%

Table 4.7: Average route completion with different architectures

model	training step	Object collision	Stuck	Lane invasion no recovery	Lane invasion with recovery	Ignore command
CNN	22440	2	4	5	0	6
CNN + LSTM	23936	3	3	1	2	6
ResNet + LSTM	21537	0	1	0	0	14

Table 4.8: Total failures using different architectures

Table 4.8 reveals that most ResNet failures are due to ignoring the navigational commands. Actual driving was very stable and even better than the CNN-LSTM model, which performed best in route completeness. The plain CNN model drove very unstable, with most issues being caused by driving into the oncoming lane by taking corners too wide and not recovering.

4.4.3 Discussion

Performance of the plain CNN is about as expected. Not having as much temporal data and having less complexity renders the model incapable of learning to navigate anything more complicated than just lane following. More surprising is the poor performance of the ResNet model. It seems that the added complexity of the feature extractor overpowers the navigational command input. This is likely because the more complex model over-fits on the small dataset.

4.5 Experiment 5: testing model performance on jetbot

4.5.1 Setup

Goal The goal of this experiment was to compare the performance of various architectures in the Jetbot environment. The models were chosen to analyze the gravity of importance of the LSTM module and the feature extractor.

Training Each architecture uses direct regression to predict the absolute speed of each wheel. The models used where the LSTM model from chapter 3.1.5, the plain CNN from chapter 3.1.5 and a modified version of the LSTM model where the feature extractor (the CNN) was replaced with ResNet18.

Testing Each model's performance was assessed using the testing procedure described in section 3.1.7.

4.5.2 Result

Table 4.9 shows the average routes completed out of the predefined test routes for each model. The best performing model was the plain CNN model with a sequence length of 3. It completed 64% of the routes while the worst performing model, combining ResNet with an LSTM, only completed 18% of its assigned routes.

model	sequence length	success rate
CNN	**3**	**0.64**
CNN + LSTM	3	0.45
CNN + LSTM	8	0.36
ResNet + LSTM	8	0.18

Table 4.9: JetBot model average route completion

There were two types of failure during the test. Either the robot would drift out of lane or it would ignore the given navigational command. Table 4.10 shows the number of mistakes each model made during testing. Generally, the models with a sequence length of 8 tended to drift out of lane more often than the models with sequence length 3. With a total of 9 failures, the ResNet model made the most mistakes across the board, failing to follow the navigational commands 4 times and drifting out of lane 5 times.

model	sequence length	out of lane	ignored command	Tot.
CNN	3	1	3	4
CNN + LSTM	3	2	4	6
CNN + LSTM	8	4	2	7
ResNet + LSTM	8	5	4	9

Table 4.10: Total number of failures

4.5.3 Discussion

Surprisingly, the more advanced models performed significantly worse. This probably denotes that they are too complex for the small size of the dataset. The longer sequence length seems to have an adverse effect on performance as the LSTM model with sequence length 3 performs marginally better than the one trained with a sequence length of 8.

4.6 Discussion

The main takeaway from these experiments is definitely the importance of both dataset size and quality. This mostly comes to light when looking at the experiment with the JetBot, where the dataset size was very limited, causing more complex and capable models to perform worse. It would have been beneficial to collect a larger dataset from different environments to see how much improvement this would bring. The second aspect of dataset quality in this case is the balancing, which proved to play an important role as well.

Besides the quality of the dataset, the recurrent module, responsible for processing temporal information, has proven to be very important.

A more complex feature extractor could be beneficial to improve driving stability, but again, this added architecture complexity would have to be compensated by a larger dataset.

The findings from the (more complex) simulated environment did not translate to the simple real-world test. While the small dataset is probably the biggest culprit, the different type of

control, being differential drive, might also have had an influence. A relation between the wheel speeds (that were a direct regression output of the models) might have been beneficial. The results of experiment 4.3 indicate that using the sine encoded classification trick for the JetBot would improve stability significantly. Here, further work is needed.

The achieved result in the work of *Aasbø and Haavaldsen*[1] seems optimistic when comparing them to the findings of these experiments. The CNN-LSTM model from experiment 4.3 with sine encoder is the exact architecture described by their work, yet the results differ substantially. Their model was able to consistently complete 95.3% of the test tracks, while in this experiment, only 69.23% was completed in almost the same circumstances.

If I were to start over again, I would use ROS (The Robot Operating System) to control the JetBot. ROS allows for easier control, which would have made data-collection simpler thus more stable and less time consuming. ROS also has excellent integration with the Gazebo simulator which would have sped up development and testing significantly. A large dataset could have been collected in simulation and, through transfer learning, be applied to the real-world environment.

5

Conclusion and Future work

5.1 Conclusion

The main goal of this thesis was to explore the possibilities of the JetBot platform for the task of end-to-end autonomous driving. The platform has proven to be more than capable given a large high quality dataset. The thesis shows that a less complex model, with no RNN was better suited for this task than its more complex counterparts. Another finding is that for more complex and serious experiments, it is desirable to use The Robot Operating System (ROS) to control the JetBot.

Another goal was to explore the importance of various aspects of the proposed architecture. The most important outcome here is that the architecture should be appropriately complex for the size of the dataset and that the quality of this dataset needs to be as high as possible. Moreover, handling the issue as a deep classification problem instead of direct regression had a more significant positive impact than expected. Less of a surprise was the impact of the LSTM module to learn temporal features. Its effect was the most substantial on the overall driving quality.

5.2 Future work

While results with JetBot are promising, a larger dataset would definitely need to be collected (e.g. in a more complex, larger environment with more roads and intersection types and potentially even stop-lights and traffic signs).

Multiple JetBots could be deployed in the environment simultaneously to study V2V communication in combination with autonomous driving. For this, using/exploring ROS might be a better option as simulations (e.g. in the Gazebo simulator) could be used to speed up testing/development.

Finally, handling the control of the jetbot as a classification problem with a method similar to that of Section 3.1.5 will probably yield improved stability. As such, this is a valuable route to take for any future work.

Bibliography

[1] M. M. J. Aasbø and H. Haavaldsen, "Autonomous Vehicle Control: End-to-end Learning in Simulated Environments," Ph.D. dissertation, NTNU, Faculty of Information Technology and Electrical EngineeringDepartment of Computer Science, Jun. 2019. [Online]. Available: http://hdl.handle.net/11250/2625797

[2] "European commision of mobility and transport on the effect of Intelligent transport systems on road transport," Sep. 2016, library Catalog: ec.europa.eu. [Online]. Available: https://ec.europa.eu/transport/themes/its/road_en

[3] "J3016B: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles - SAE International," Jun. 2018, library Catalog: www.sae.org. [Online]. Available: https://www.sae.org/standards/content/j3016_201806/

[4] F. Rosenblatt, "PRINCIPLES OF NEURODYNAMICS. PERCEPTRONS AND THE THEORY OF BRAIN MECHANISMS," CORNELL AERONAUTICAL LAB INC BUFFALO NY, Tech. Rep. VG-1196-G-8, Mar. 1961. [Online]. Available: https://apps.dtic.mil/docs/citations/AD0256582

[5] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017, arXiv: 1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980

[6] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade: Second Edition*, ser. Lecture Notes in Computer Science, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer, 2012, pp. 9–48. [Online]. Available: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

[7] W. Finnoff, F. Hergert, and H. G. Zimmermann, "Improving model selection by nonconvergent methods," *Neural Networks*, vol. 6, no. 6, pp. 771–783, Jan. 1993. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0893608005801224

[8] L. Prechelt, "Early Stopping - But When?" in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science, G. B. Orr and K.-R. Müller, Eds. Berlin, Heidelberg: Springer, 1998, pp. 55–69. [Online]. Available: https://doi.org/10.1007/3-540-49430-8_3

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," p. 30.

[10] "CS231n Convolutional Neural Networks for Visual Recognition." [Online]. Available: https://cs231n.github.io/transfer-learning/

[11] E. S. Olivas, J. D. M. Guerrero, M. M. Sober, J. R. M. Benedito, and A. J. S. Lopez, *Handbook Of Research On Machine Learning Applications and Trends: Algorithms, Methods and Techniques*, 1st ed. Hershey, PA: Information Science Reference, Sep. 2009.

[12] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar. 1994, conference Name: IEEE Transactions on Neural Networks.

[13] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs]*, Dec. 2015, arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385

[15] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015, pp. 2722–2730, iSSN: 2380-7504.

[16] G. Varisteas, R. Frank, S. A. Sajadi Alamdari, H. Voos, and R. State, "Evaluation of End-to-End Learning for Autonomous Driving: The Good, the Bad and the Ugly," in *2019 2nd International Conference on Intelligent Autonomous Systems (ICoIAS)*, Feb. 2019, pp. 110–117.

[17] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv:1509.02971 [cs, stat]*, Jul. 2019, arXiv: 1509.02971. [Online]. Available: http://arxiv.org/abs/1509.02971

[18] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah, "Learning to Drive in a Day," *arXiv:1807.00412 [cs, stat]*, Sep. 2018, arXiv: 1807.00412. [Online]. Available: http://arxiv.org/abs/1807.00412

[19] D. A. Pomerleau, "ALVINN: An Autonomous Land Vehicle in a Neural Network," p. 9, 1998.

[20] Y. LeCun, "Autonomous off-road vehicle control using end-to-end learning," Net-Scale Technologies, Inc, Tech. Rep. ARPA Order No. Q458, Jul. 2004. [Online]. Available: http://net-scale.com/doc/net-scale-dave-report.pdf

[21] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. L. Cun, "Off-Road Obstacle Avoidance through End-to-End Learning," p. 8.

[22] U. Muller and Y. LeCun, "LAGR: Learning Applied to Ground Robotics," 2004. [Online]. Available: https://cs.nyu.edu/~yann/research/lagr/index.html

[23] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *arXiv:1604.07316 [cs]*, Apr. 2016, arXiv: 1604.07316. [Online]. Available: http://arxiv.org/abs/1604.07316

[24] V. Buhrmester, D. Münch, and M. Arens, "Analysis of Explainers of Black Box Deep Neural Networks for Computer Vision: A Survey," *arXiv:1911.12116 [cs]*, Nov. 2019, arXiv: 1911.12116. [Online]. Available: http://arxiv.org/abs/1911.12116

[25] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv:1312.6199 [cs]*, Feb. 2014, arXiv: 1312.6199. [Online]. Available: http://arxiv.org/abs/1312.6199

[26] A. Boloor, K. Garimella, X. He, C. Gill, Y. Vorobeychik, and X. Zhang, "Attacking Vision-based Perception in End-to-End Autonomous Driving Models," *arXiv:1910.01907 [cs, stat]*, Oct. 2019, arXiv: 1910.01907. [Online]. Available: http://arxiv.org/abs/1910.01907

[27] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning Spatiotemporal Features with 3D Convolutional Networks," *arXiv:1412.0767 [cs]*, Oct. 2015, arXiv: 1412.0767. [Online]. Available: http://arxiv.org/abs/1412.0767

[28] J. Y.-H. Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond Short Snippets: Deep Networks for Video Classification," *arXiv:1503.08909 [cs]*, Apr. 2015, arXiv: 1503.08909. [Online]. Available: http://arxiv.org/abs/1503.08909

[29] H. M. Eraqi, M. N. Moustafa, and J. Honer, "End-to-End Deep Learning for Steering Autonomous Vehicles Considering Temporal Dependencies," *arXiv:1710.03804 [cs]*, Nov. 2017, arXiv: 1710.03804. [Online]. Available: http://arxiv.org/abs/1710.03804

[30] S. Du, H. Guo, and A. Simpson, "Self-Driving Car Steering Angle Prediction Based on Image Recognition," *arXiv:1912.05440 [cs, stat]*, Dec. 2019, arXiv: 1912.05440. [Online]. Available: http://arxiv.org/abs/1912.05440

[31] C. Hubschneider, A. Bauer, M. Weber, and J. M. Zöllner, "Adding navigation to the equation: Turning decisions for end-to-end vehicle control," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, Oct. 2017, pp. 1–8, iSSN: 2153-0017.

[32] F. Codevilla, M. Müller, A. López, V. Koltun, and A. Dosovitskiy, "End-to-end Driving via Conditional Imitation Learning," *arXiv:1710.02410 [cs]*, Mar. 2018, arXiv: 1710.02410. [Online]. Available: http://arxiv.org/abs/1710.02410

[33] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," p. 16, 2017.

[34] R. Rothe, R. Timofte, and L. V. Gool, "DEX: Deep EXpectation of Apparent Age from a Single Image," *2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*, 2015.

[35] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization," *International Journal of Computer Vision*, vol. 128, no. 2, pp. 336–359, Feb. 2020, arXiv: 1610.02391. [Online]. Available: http://arxiv.org/abs/1610.02391

Appendices

Appendix A - Guide on JetBot setup and control

Connecting to the JetBot

Connecting to the JetBot for the first time in a new network requires a display, a keyboard and a mouse. The control board boots up automatically when power is supplied. This is done by pressing the button on the side of the power bank. Once booted, you will be greeted by a standard Gnome desktop. On the top right of the screen you can connect the robot to the required network trough WiFi. Once connection is complete, you can disconnect the display and peripherals.

Once the JetBot is connected, it will display its IP-address on the small LCD display on the back. Connect to this IP-address on port 8888 using the browser. This should open up the JupyterLab environment running on the Jetbot.

If you want to shut down the JetBot, open a terminal in JupyterLab, type "sudo shutdown now" and provide the jetson's password (By default, this is *jetbot*).

Remote control using a controller

Both manual control and the display of information happens through the Python library *ipywidgets*. This library contains a collection of interactive HTML widgets for Jupyter notebooks.

The next code snippet is an example on how to get a controller instance (given that you have a controller already connected and set up).

```
import ipywidgets.widgets as widgets

controller = widgets.Controller(index=0)  # replace with index of your controller
display(controller)
```

Next, you'll need control over the actual robot. The included JetBot library makes this very simple.

```
from jetbot import Robot
robot = Robot()
```

To link together the input of the controller with the control of the robot, another crucial library is needed: *traitlets*. Traitlets is a framework which lets Python classes have observable attributes.

This allows us to observe changes in the controller output and use that information to update the motor speeds.

```
from traitlets import observe

def update(change):
    left_wheel = controller.axes[0].value
    right_wheel = controller.axes[1].value
    robot.set_motors(left_wheel, right_wheel)

for a in controller.axes:
    a.unobserve_all()   # makes sure that no double links get made after re-executing the code
controller.axes[2].observe(update, names=['value'])
controller.axes[1].observe(update, names=['value'])
```

Controlling the JetBot by setting the speed of each wheel is, to put it lightly, fairly frustrating and impractical. To get around this, pass the joystick controls through an algorithm that converts throttle/steer to differential drive.

```
def to_differential(nJoyY, nJoyX):
    # Differential Steering Joystick Algorithm
    # ========================================
    #   by Calvin Hass
    #   https://www.impulseadventure.com/elec/
    #   translated to Python by Laurin Neff

    fPivYLimit = 32.0

    # TEMP VARIABLES
    nMotPremixL = 0 # Motor (left)  premixed output        (-128..+127)
    nMotPremixR = 0 # Motor (right) premixed output        (-128..+127)
    nPivSpeed = 0   # Pivot Speed                          (-128..+127)
    fPivScale = 0.0 # Balance scale b/w drive and pivot    (   0..1   )

    # Calculate Drive Turn output due to Joystick X input
    if(nJoyY>=0):

      # Forward
      nMotPremixL = 127.0 if nJoyX>=0 else 127.0 + nJoyX
      nMotPremixR = 127.0 - nJoyX if nJoyX>=0 else 127.0
    else:
      # Reverse
      nMotPremixL = 127.0 - nJoyX if nJoyX>=0 else 127.0
      nMotPremixR = 127.0 if nJoyX>=0 else 127.0 + nJoyX
```

```
    # Scale Drive output due to Joystick Y input (throttle)
    nMotPremixL = nMotPremixL * nJoyY/128.0
    nMotPremixR = nMotPremixR * nJoyY/128.0

    # Now calculate pivot amount
    #  - Strength of pivot (nPivSpeed) based on Joystick X input
    #  - Blending of pivot vs drive (fPivScale) based on Joystick Y input
    nPivSpeed = nJoyX
    fPivScale = 0.0 if abs(nJoyY)>fPivYLimit else 1.0-abs(nJoyY)/fPivYLimit

    # Calculate final mix of Drive and Pivot
    nMotMixL = int((1.0-fPivScale)*nMotPremixL + fPivScale*( nPivSpeed)) # Motor (left)  mixed output
    nMotMixR = int((1.0-fPivScale)*nMotPremixR + fPivScale*(-nPivSpeed)) # Motor (right) mixed output
    return (nMotMixL / 128, nMotMixR / 128)
```

And change the update code to use this function.

```
def update(change):
    steer = controller.axes[0].value
    throttle = controller.axes[1].value
    joystick_y = int(throttle * 128)
    joystick_x = int(steer * 128)
    diff_throttle = to_differential(joystick_y, joystick_x)
    robot.set_motors(diff_throttle[0], diff_throttle[1])
```

Important to know for debugging is that, as soon as a widget is used, the normal print() will not work anymore. You'll have to create an output widget and print to that.

```
output = widgets.Output()

using output:
    print("hello world!")

display(widgets.VBox([
    controller,
    output,
]))
```

Data collection

To capture images from the camera, make an instance of the Camera class included in the jetbot library and observe its changes (using the traitlets library).

```
from jetbot import bgr8_to_jpeg
from jetbot import Camera

camera = Camera.instance()

def save_snapshot(change):
    image = PIL.Image.fromarray(change['new'])
    uuid = create_filename()
    image_path = os.path.join(DATASET_DIR, uuid + '.jpg')
    image.save(image_path, "JPEG")

camera.observe(save_snapshot, names='value')
```

If the camera fails to initialise, run the command "sudo systemctl restart nvargus-daemon" to restart the JetBot drivers.

The data is stored on the JetBot's SD-card. When you want to download this dataset, I advise adding all files to an archive. This can be done using the command **"zip -r -q dataset.zip DATASET-DIR"** in a terminal. The zip file can then be downloaded by right-clicking on the file in JupyterLab's file explorer and selecting download.

Appendix B - LSTM architecture code

This appendix contains the code from my PyTorch implementation of the CNN-LSTM architecture used in this thesis. This does not include the code needed to train this architecture.

```
import torch.nn as nn
import torch


class TimeDistributed(nn.Module):
    """
    Wrapper that adapts the given module to accept time series data.
    """
    def __init__(self, module, time_steps):
        super(TimeDistributed, self).__init__()
        self.module = module

    def forward(self, x):
        batch_size, time_steps, C, H, W = x.size()
        output = torch.tensor([]).to(device='cuda')
        for i in range(time_steps):
            output_t = self.module(x[:, i, :, :])
            output_t = output_t.unsqueeze(1)
            output = torch.cat((output, output_t), 1)
        return output


class FeatureExtractor(nn.Module):
    def __init__(self, image_channels):
        """
            Is called when model is initialized.
            Args:
                image_channels. Number of color channels in image (3)
        """
        super().__init__()

        # Define the convolutional layers
        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=image_channels, out_channels=24,
                      kernel_size=5, stride=2),
            nn.BatchNorm2d(24),
            nn.ReLU(),
            nn.Conv2d(in_channels=24, out_channels=36,
                      kernel_size=5, stride=2),
            nn.BatchNorm2d(36),
            nn.ReLU(),
            nn.Conv2d(in_channels=36, out_channels=48,
```

```
                    kernel_size=5, stride=2),
            nn.BatchNorm2d(48),
            nn.ReLU(),
            nn.Conv2d(in_channels=48, out_channels=64,
                    kernel_size=3, stride=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64,
                    kernel_size=3, stride=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64,
                    kernel_size=3, stride=1),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        self._init_weights()

    def _init_weights(self):
        layers = [*self.feature_extractor]
        for layer in layers:
            for param in layer.parameters():
                if param.dim() > 1:
                    nn.init.xavier_uniform_(param)

    def forward(self, x):
        """
        Performs a forward pass through the model
        Args:
            image_input:  shape: [batch_size, 3, 110, 350]
        """
        return self.feature_extractor(x)


class LSTMDrivingModel(nn.Module):

    def __init__(self, image_channels, seq_length, sine_output=False):
        """
            Is called when model is initialized.
            Args:
                image_channels. Number of color channels in image (3)
                seq_length:  Length of one training sample sequence
                sine_output:  Wether or not the network should output a sine wave
        """
        super().__init__()
        self.sine_output = sine_output

        # Define the convolutional layers
```

```
        self.feature_extractor_steering = TimeDistributed(
                                            FeatureExtractor(image_channels),
                                            seq_length
                                          )
        self.feature_extractor_acceleration = TimeDistributed(
                                                FeatureExtractor(image_channels),
                                                seq_length
                                              )

        self.input_dim = 1033    # represents the size of the input at each time step
        self.hidden_dim = 10     # represents the size of the hidden state and cell state at each time step
        self.num_layers = 1      # the number of LSTM layers stacked on top of each other

        self.steer_intermediate_fc = nn.Sequential(
            nn.Linear(self.input_dim, 128),
            nn.ReLU()
        )

        self.accleration_intermediate_fc = nn.Sequential(
            nn.Linear(self.input_dim, 128),
            nn.ReLU()
        )

        self.steering_lstm = nn.LSTM(128, self.hidden_dim, self.num_layers, batch_first=True)
        self.acceleration_lstm = nn.LSTM(128, self.hidden_dim, self.num_layers, batch_first=True)

        if self.sine_output:
            self.steering_classifier = nn.Sequential(
                nn.Linear(10, 10),
                nn.Tanh()
            )
        else:
            self.steering_classifier = nn.Sequential(
                nn.Linear(10, 1),
                nn.Tanh()
            )

        self.throttle_classifier = nn.Sequential(
            nn.Linear(10, 1),
            nn.Sigmoid()
        )

        self.brake_classifier = nn.Sequential(
            nn.Linear(10, 1),
            nn.Sigmoid()
        )

        self.seq_length = seq_length
```

```
139
140     def forward(self, image_input, hlcs, infos):
141         """
142         Performs a forward pass through the model
143         Args:
144             image_input : shape: [batch_size, seq_length, 3, 110, 350]
145             hlcs : shape: [batch_size, seq_length, 6]
146             infos : shape: [batch_size, seq_length, 3]
147         """
148
149         batch_size, seq_length, C, H, W = image_input.shape
150
151         # steering
152         x = self.feature_extractor_steering(image_input)
153         x = x.view(batch_size, seq_length, -1)  # Flatten
154         x = torch.cat((x, hlcs, infos), dim=2)  # concatinate infos and hlc
155         x = self.steer_intermediate_fc(x)
156
157         # lstm layer
158         x, (h_n, c_n) = self.steering_lstm(x)
159         x = x[:, -1, :]  # Obtaining the last output
160
161         steer_pred = self.steering_classifier(x)
162
163         # throttle and brake
164         x = self.feature_extractor_acceleration(image_input)
165         x = x.view(batch_size, seq_length, -1)  # Flatten
166         x = torch.cat((x, hlcs, infos), dim=2)  # concatinate infos and hlc
167         x = self.accleration_intermediate_fc(x)
168
169         x, (h_n, c_n) = self.acceleration_lstm(x)
170         x = x[:, -1, :]  # Obtaining the last output
171
172         throttle_pred = self.throttle_classifier(x)
173         brake_pred = self.brake_classifier(x)
174
175         return steer_pred, throttle_pred, brake_pred
```