Tampere University

Eetu Manninen

AUTONOMOUS RACING ROBOT

Hardware and Software Implementation

Faculty of Information Technology and Communication Sciences
Bachelor’s Thesis
June 2020

ABSTRACT

Eetu Manninen: Autonomous Racing Robot: Hardware and Software Implementation
Bachelor's Thesis
Tampere University
Bachelor's Degree Programme in Information Technology
June 2020

Artificial intelligence and machine learning are used to solve more and more different and complex problems. One popular machine learning subfield is imitation learning, where robot learns to solve a specified task using demonstrations from a human expert. Imitation learning is popular because it is easy to show an example solution to a task and let the machine figure out how to replicate it. One of the most used methods of imitation learning is behavioural cloning, where robot learns from expert demonstrations to map observations to actions.

This thesis proposes a simple hardware and software implementation for a small-scale autonomous robot. Behavioural cloning is used to teach a robot to autonomously drive around a small test track. Training data contains about 50 minutes of video, which corresponds to about 58000 images as well as control commands for those images. Training image set consists of 3 different classes, which correspond to the three different movement commands. The tests show that the proposed implementation works well on the chosen test track.

Keywords: neural network, imitation learning, behavioural cloning, autonomous driving

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

CONTENTS

1. INTRODUCTION 1
2. RELATED WORK 3
3. ROBOTIC SYSTEM ARCHITECTURE 5
3.1 Hardware 5
3.2 Software 7
3.2.1 Control Commands 8
3.2.2 Image Capturing and Motor Control Program 8
3.2.3 Controller Programs 9
4. NEURAL NETWORK CONTROLLER 10
4.1 2D Convolutional Layer 10
4.2 2D Max Pooling Layer 11
4.3 Flatten Layer and Fully Connected Layer 11
4.4 Rectified Linear Unit and Softmax Activation Function 12
4.5 Adam Optimizer and Categorical Cross-Entropy Loss 12
5. EXPERIMENTS 15
5.1 Training Data Collection 15
5.2 CNN Training 16
5.3 TensorRT Optimization 17
5.4 Self-driving Tests 18
6. CONCLUSIONS 20
REFERENCES 21

LIST OF SYMBOLS AND ABBEREVIATIONS

AI	Artificial Intelligence
CNN	Convolutional Neural Network
FPS	Frames Per Second
GPU	Graphics Processing Unit
IL	Imitation Learning
NN	Neural Network
ReLU	Rectified Linear Unit
RPi	Raspberry Pi

1. INTRODUCTION

World Health Organization lists road injuries as one of the leading causes of death in 2016 [1]. There are many factors contributing to road traffic injuries such as inadequate traffic laws, unsafe vehicles, unsafe road infrastructure and human interaction [2]. By far the biggest contributor of these is the human interaction. Despite strict system of rules even the best human drivers make mistakes every now and then. One of the proposed solutions to this problem is to take humans out of the equation and let machines handle the driving. Artificial intelligence (AI) systems aren't perfect drivers either, therefore more research is needed if they are to replace human drivers. [3]

The continued increases in computing power has enabled the integration of AI in many applications including vehicles. This is one of the reasons why self-driving technology is now seeing a surge in popularity. Many car manufacturers and tech companies are taking on the challenge of developing their own self driving car technology [4][5][6][7][8][9]. Some of them having promising results [10].

The purpose of thesis is to implement simple small-scale hardware and software solution for autonomous driving which can easily be expanded upon. This thesis is done on a real robot on a short racing test track, seen in Figure 1. The goal is to make the robot autonomously and reliably drive around the track without any human intervention.

Figure 1. *Racing track*

The proposed Neural Network (NN) implementation takes learning from demonstration approach to imitate the behaviour of a human driver. Expert demonstrations used for

training are created on the same track with a human driving the robot. The expert drives with the same information and control possibilities the robot has when it is driving autonomously.

For sensor data, the proposed implementation uses one front-facing camera to capture images of the racing track. Camera is a cheap and small sensor therefore ideal to use in situations where camera can get reliable and clear data. One camera is enough on this closed track when there are no obstacles to avoid. But this sensor arrangement might not be enough if there are many moving obstacles in the way.

Related work is presented in Chapter 2. Chapter 3 explains what kind of robotic architecture is used in this thesis. Neural network controller is discussed in Chapter 4. Chapter 5 shows the NN training methods and the self-driving test performed and their results. Finally, Chapter 6 presents the conclusions of this thesis and discusses possible related future work and how this implementation can be expanded and improved upon.

2. RELATED WORK

The huge computational capacity current computers have, and the huge volumes of data modern sensors are able collect, are couple of the factors that contributed to the surge of AI applications. However, current trends are to reduce the amount of generated data and to reduce the training time and to make AI applications more power efficient. Making AI application more power efficient enables them to be used on mobile devices. [11]

Imitation learning (IL) is growing area in research due the recent surge in AI applications. Another reason why IL is growing area is because of the ease of showing examples of a task and letting NN to learn the behaviour. Letting NN to learn the behaviour of an expert is far easier than to program precise rules. [11]

One of the first success stories of IL was the Autonomous Land Vehicle In a Neural Network or ALVINN from the year 1989. It managed to autonomously drive a test vehicle at a speed of ½ meters per second along a path thought a wooded area of the Carnegie Mellon University campus with a fraction of the computing power modern computers have. [12] Many of the more recent papers about IL and autonomous robots are based on small unmanned air vehicles rather than big land-based vehicles. Giusti et al. in their publication A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots used IL to teach a small robot to follow forest trails [13]. Other quite similar usage of IL can be found in the publication Learning monocular reactive UAV control in cluttered natural environments by Ross et at. In their publication, Ross et al. used IL to teach the robot to avoid trees in the forest. [14]

IL methods aim to mimic human experts' behaviour. The goal of IL is that a machine agent learns how to map observations to actions. Agent learns those mapping from the demonstrations of the given task. The use of IL enables to make the learning process faster by knowing the correct actions at training time. It also enables to make training process safer by limiting the number of different actions the agent can perform. Compared to reinforcement learning where agent tests different possibly dangerous actions, IL is safer because it is limited to predefined set of actions that the expert has shown. [15] However, showing the correct actions is not the only way to teach an agent like Gandhi et al wrote in their publication Learning to fly by crashing. The authors used examples of collisions to teach the agent what not to do. Their conclusion was that showing negative examples was a viable method to teach an agent. [16]

The main IL methods are behavioural cloning and inverse reinforcement learning. The main difference between these two is that the behavioural cloning aims to learn the trajectories of the expert while inverse reinforcement learning aims to learn the reward function or the intent of the expert. One of the problems with behavioural cloning is that if the agent is put into a state that the expert observations have not covered then the agent doesn't have any data on how to recover from that situation. [16] In that case the behaviour of the agent is undefined, and it can lead to a failure. One of the solutions is to get more observations of different situations. Other solution could be that there would be other reinforcement learning systems or sensors to detect and correct untrained states.

3. ROBOTIC SYSTEM ARCHITECTURE

In this chapter the used hardware and software components are presented. The robot used in the experiments is shown is Figure 2 and the racing track used is shown in Figure 1. Hardware configuration consists four parts Nvidia Jetson Nano, Raspberry Pi (RPi), MonsterBorg and a laptop. Software implementation is divided into three parts RPi, laptop/Jetson and Convolutional Neural Network (CNN) training. Hardware configuration is discussed in Chapter 3.1 and software implementation is presented in Chapter 3.2. CNN training is discussed in later chapters, in Chapters 5.1 and 5.2.

Figure 2. *MonsterBorg robot*

3.1 Hardware

Figure 3 show how the different hardware components are connected to each other. Jetson Nano and Raspberry Pi are onboard the MonsterBorg and connected to each other via an ethernet cable. The laptop is connected wirelessly to the RPi using Wi-Fi. Laptop is only used for training data collection and starting the python programs on the RPi and Jetson Nano.

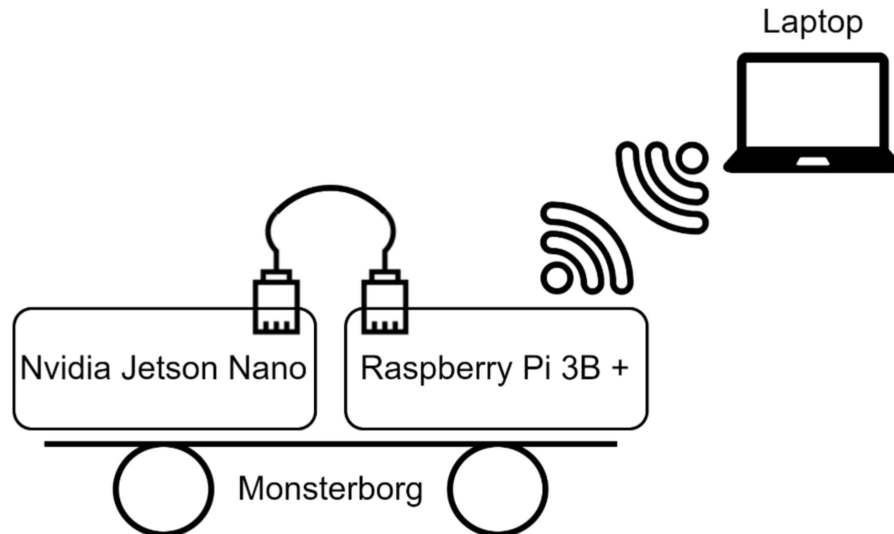

Figure 3. *Hardware connections*

MonsterBorg was chosen as the platform which the robot was built on. MonsterBorg includes four direct current motors with wheels, aluminium chassis, mounting holes for RPi and its camera. MonsterBorg also includes a ThunderBorg motor controller. [17] ThunderBorg motor controller runs on voltages between 7 and 35 volts and can supply 5 volts to RPi to which it attaches directly. ThunderBorg can supply up to 5 amperes to both motors and it can control the rotation speed of the motors via Pulse Width Modulation. [18] Three cells series and two cells parallel 14,4-volt lithium-ion battery was used to power the MonsterBorg. Separate 5-volt battery was used to power the Jetson Nano.

For controlling the motors on the MonsterBorg a RPi was used. RPi 3B+ is ARM based single-board computer designed to be used in wide range of electronics, IOT and computing applications. RPi 3B+ has wireless network adapter, wired ethernet port, connector for a camera and 40-pin GPIO header. [19] In this thesis the wireless connection is used to connect the RPi to the controlling laptop, and the wired ethernet connection is used to connect Jetson Nano to the RPi. Camera is connected to the camera connector and it is used to get images of the track. ThunderBorg motor controller is connected to the 40-pin GPIO header of the RPi.

For autonomous driving Nvidia Jetson Nano was used to drive the Neural Network (NN). Jetson Nano is ARM based computer designed to be used as a low-power platform for AI and IOT applications. Nano has 128-core Nvidia Graphics Processing Unit (GPU) onboard for accelerating NNs and other AI algorithms. Jetson Nano also has wired ethernet port, connector for a RPi camera and 40 GPIO pins which can be used to control peripherals and sensors. [20] In this thesis Jetson Nano is used to run a CNN and send the results over the wired ethernet connection to the RPi.

Figure 4 shows all the previously discussed hardware components except MonsterBorg chassis or the used Lenovo Yoga 900 laptop. The make and the model of the used laptop is irrelevant in this case as long as it has Wi-Fi capabilities.

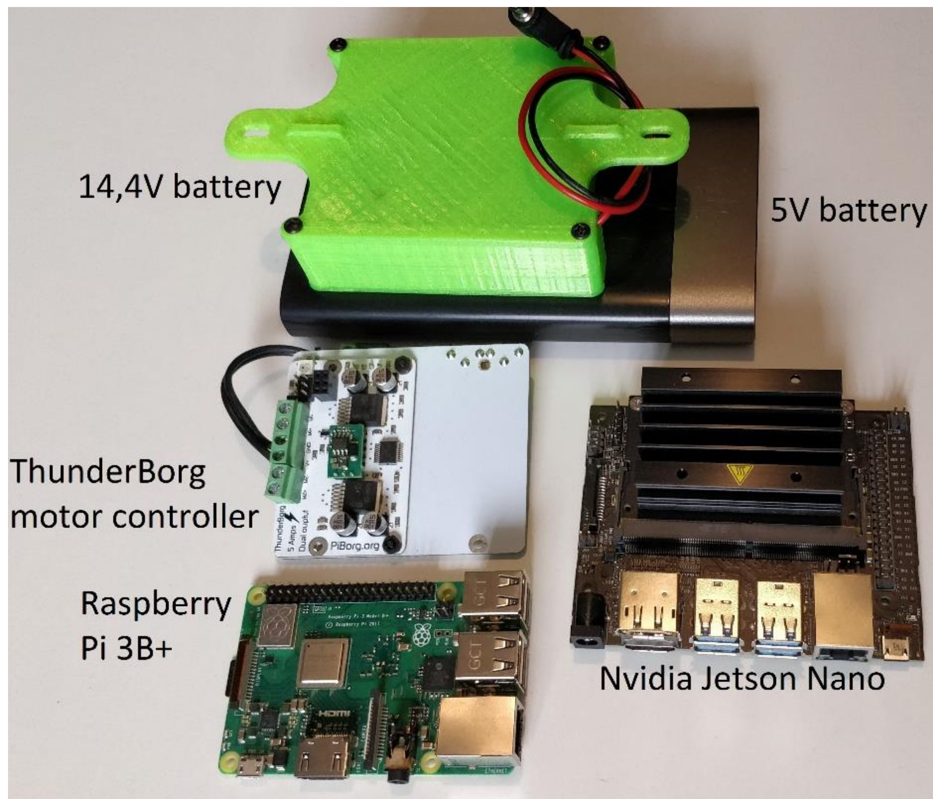

Figure 4. *Hardware components*

3.2 Software

Figure 5 shows and overview on how the different devices and programs interact with each other. Figure 5 shows the order in which the steps occur and example data for every step. The top part shows an example where the laptop is the controller device and the expert gives the control command via the laptop's keyboard. Bottom part shows an example where Jetson Nano is the controller.

Chapter 3.2.1 presents the control commands. The image capturing and motor controller software running on RPi is discussed in Chapter 3.2.2. Controller programs that are running on the laptop and the Jetson Nano is discussed in chapter 3.2.3.

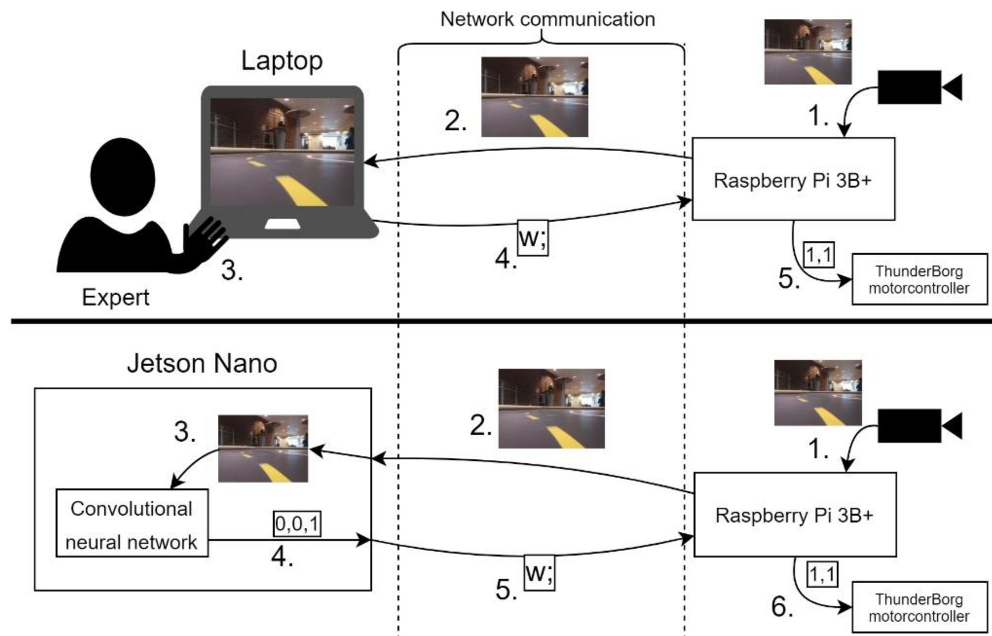

Figure 5. *Software interactions*

3.2.1 Control Commands

There are four basic movement commands to control the robot. Those commands are "w" straight, "a" turn left, "s" backwards and "d" turn right. Turn left and turn right commands are implemented by slowing down the wheels on the inside curve and speeding up the wheels on the outside curve. That is why they need either "w" or "s" command at the same time to change direction of the robot.

An example command could be "w,a;" telling the robot to move forward and turn left. Colon separates the different parts of the command and semicolon indicates the end of command. Control commands are sent between the devices using User Datagram Protocol to reduce unnecessary latency.

3.2.2 Image Capturing and Motor Control Program

Program running on RPi is written in python 2.7 and is responsible for capturing and sending images of the track and receiving control commands from one of the controllers. RPi is also responsible for sending motor control commands to the ThunderBorg motor controller. The ThunderBorg library is based on python 2.7 and there was no reason to port it over to newer versions of python.

RPi captures images sized 480x360 pixel at either 20 or 30 Frames Per Second (FPS) depending on which device is the controller. When laptop is acting as the controller the frame rate is limited to 20 FPS, because the wireless connection causes too much latency on higher frame rates. There is no such problem on the wired connection and the

frame rate is limited by the speed of which the receiver can process the images. In this implementation the Jetson can process images at 30 FPS and therefore RPi camera frame rate is limited to 30 FPS. Images are sent to the receiver using Transmission Control Protocol where RPi acts as the server and receivers act as clients.

Target speeds for the motors are set according to the control commands received from the controller. Motor speeds are controlled in a loop where in every cycle current motor speeds are changed a little towards the target speeds. This slightly reduces the responsiveness of the robot but makes it more resilient to small steering errors coming from the controller. It also makes the movements smoother by removing sudden movements happening when target speeds changes drastically.

3.2.3 Controller Programs

Both controller programs are written in python 3.7. They are responsible for receiving images from the RPi and sending commands back to it. The two programs quite similar to each other because the Jetson version is modified version of the laptop program. The only thing they differ from each other is the way they acquire the control commands.

Controller program running on the laptop shows the human user the image stream and receives keyboard presses from the user and sends them over the network to the RPi. While the program running on Jetson doesn’t have the component where images are shown to the user. Also, the component where keyboard presses are received from the user is replaced with a component that computes commands with the NN.

Laptop program also handles the training data capturing. It saves the received images into a video using computer vision library OpenCV and saves the sent commands to a text file in the form of one line per command. Commands are synchronized to the image so that every time the laptop receives a new image it sends a command back. This way every frame of the saved video is automatically annotated and can be easily used for training.

4. NEURAL NETWORK CONTROLLER

The used CNN is implemented in TensorFlow 2.1 using the Keras frontend. The structure of the implemented model is shown in Figure 6 on page 14. Model has five convolutional layers and five max pooling layers. After the convolution and pooling layers there is a flatten layer and four fully connected dense layers. The following subchapters present more in-depth information about the layers of the model.

Many different configurations were tested. Different number of layers and parameter for those layers were tested and, in the end, decided to try to reduce the number of parameters the model had. The reasoning behind the decision to reduce the number of parameters was to reduce overfitting and to ensure that the Jetson Nano could run the model.

In addition to the model configuration presented in Figure 4, during training dropout layers were added after the first two dense layers to reduce overfitting. Dropout layers set on random some percentage of outputs to zero. This makes harder for model to learn specific information from the training data and helps the model to generalize the data better. [21][22] Every dropout layer had 30% drop rate.

4.1 2D Convolutional Layer

Convolutional layer is one of the basic building blocks of a CNN. Convolution layer extracts features from the input image using a set of filters. These filters are learnable and are learnt during the training process. Convolutional layer performs convolution between the input image and the learnt filters, to produce a feature map of the input. [23][24] Convolutional Layer has three main configurable parameters that are depth, stride and zero-padding and activation function with which the user can tune the layer fit the intended purpose. [22][24]

Depth corresponds to the number of filters the layer has. With a large number filters the layer can extract more features, but with increased computing requirements [22]. In his thesis the number of filters is increased as the model get deeper. The first two layers have 32 filters, the next two have 64 filters and the last layer has 128 filters.

Stride is the number of pixels by which the filter matrix is moved. Having a stride of 1 means that the filter is moved by one pixel at a time. A larger stride leads to fewer extracted features but reduces overlapping [22]. Every convolutional layer in the implemented model uses a stride of 1 which is the default value in Keras.

Zero-padding means that zero values are added to the edges of the input image. If no zero-padding is used the size of the output feature map is smaller than the input image. The reason is that the filters need adjacent pixels to perform the convolution and, on the edges, there is no adjacent pixels without zero-padding. [22] In the implemented models' convolutional layers zero-padding is used to avoid the reduction in size.

Activation function is used to define how the layers' output behaves when given an input and to add non-linearity to the layer [23]. All convolutional layers in the implemented model uses Rectified Linear Unit as their activation function. Rectified Linear Unit is discussed in Chapter 4.4.

4.2 2D Max Pooling Layer

Pooling is used to reduce the computational cost by reducing the number of parameters to learn. The reduction of parameters also helps to reduce over-fitting. Pooling also make the network more invariant to small changes or distortions in the input image. There is couple of different types of pooling layer, for example average pooling, sum pooling and max pooling. In this thesis max pooling us used as the pooling layer. Max pooling outputs the largest value in the current window. Max pooling layer forms a new smaller feature map by filtering the whole map with a maximum filter. [22]

The first configurable parameter of pooling layers is window size. Every pooling layer in this model uses windows size of 2x2 pixels. Because the input images are not very high resolution, 2x2 pixel window performs well to reduce the number of parameters. For larger images larger window sizes might be more suitable to reduce the parameter count even faster.

Other configurable parameter for pooling layers is stride which is discussed in the previous chapter. In this case the used stride value is the Keras default which is the same as the window size. There is also parameter for padding which is also discussed in the previous chapter. In all pooling layers the Keras default padding of no padding is used.

4.3 Flatten Layer and Fully Connected Layer

Flatten layer takes features that are in a multidimensional matrix and rearranges them to a one-dimensional vector. This allows connecting multidimensional feature maps from for example 2D convolutional layer or 2D max pooling layer to a fully connected layer. [25]

Fully connected layers' purpose is to form connections between high-level features and that way classify an image. Fully connected layer as its name suggests is fully connected

to previous layers' neurons. Action that one neuron in fully connected layer performs is a weighted sum with every connection to previous neurons getting its own weight or multiplier. Training changes these weights to improve the layers ability predict the right outcome. [25]

Number of neurons is one of the main configurable parameters of a fully connected layer. In the implemented model the number of neurons decrease as the model gets deeper. First fully connected layer has 150 neurons, second has 75, third 25 and the last one has only 3. The other parameter to configure is the activation function. The model uses Rectified Linear Unit function for the first three fully connected layers and Softmax for the last one. Softmax function is discussed in the next chapter.

4.4 Rectified Linear Unit and Softmax Activation Function

Rectified Linear Unit (ReLU) has become common for many types of NNs. That is because in many situations a model that uses ReLU is easier to train and achieves better performance than when other more complicated activation functions are used. [25][26] ReLU is a piecewise linear function which allows it to be more simple than other activation function like Sigmoid or Tanh. ReLU consists of two linear function which are

$$f(x) = \begin{cases} 0, & if\ x < 0 \\ x, & if\ x \geq 0 \end{cases}, \tag{4.1}$$

where x is the input to the ReLU. ReLU outputs zero when its input is negative and when the input is positive it outputs the same value it received as input. [26][27]

Softmax is often used as the activation function for the last layer of a NN. That is because Softmax outputs probability distributions based on the input and when used on the last layer it outputs the probabilities of the labels for the input image. In practise Softmax takes a vector of numbers like those values from previous neurons multiplied by weights as its input. Then Softmax takes exponents of each element of the input vector. Next the Softmax normalizes these exponential values by diving by the sum of all those exponents. After normalization sum of all the output vector probabilities add up to one. [24][25]

4.5 Adam Optimizer and Categorical Cross-Entropy Loss

The implemented model uses Adam as its optimizer algorithm. In this mode the prediction accuracy is the metric that Adam tries to optimize. For the loss function Cross entropy loss is used.

Adam is an adaptive learning rate optimization algorithm to update the weights in the network. Adam or Adaptive Moment Estimation is relatively new optimizer first published

in 2014. Adam uses adaptive learning rate to minimize the problem of overfitting the model without sacrificing speed during training. Adam combines the advantages of AdaGrad and RMSProp optimizers to provide robust and well-suited optimizer for wide range of different problems in the field of machine learning. [28]

Cross-entropy loss is a popular choice when dealing with classification problems. One of the reasons is that it has been proven to work quite well to solve them. It works very well with Softmax activation function on the last layer of the model. It is used to compare the one-hot encoded label vector to the probability distribution output vector from the Softmax layer. With this comparison cross-entropy loss determines how far of the predictions were. [25] Cross-entropy is defined as

$$J = -\frac{1}{N}\left(\sum_{i=1}^{N} y_i * \log(\hat{y}_i)\right), \tag{4.2}$$

where J is the loss, N is the number of classes, y_i is the ground truth probability of a class and $\hat{y}_i$ is the predicted probability of a class. [25][27] Cross-entropy loss is dependent only on the probabilities of the correct classes. Unlike in Mean square, loss distribution of wrong predictions does not matter, only the probabilities of the right answers.

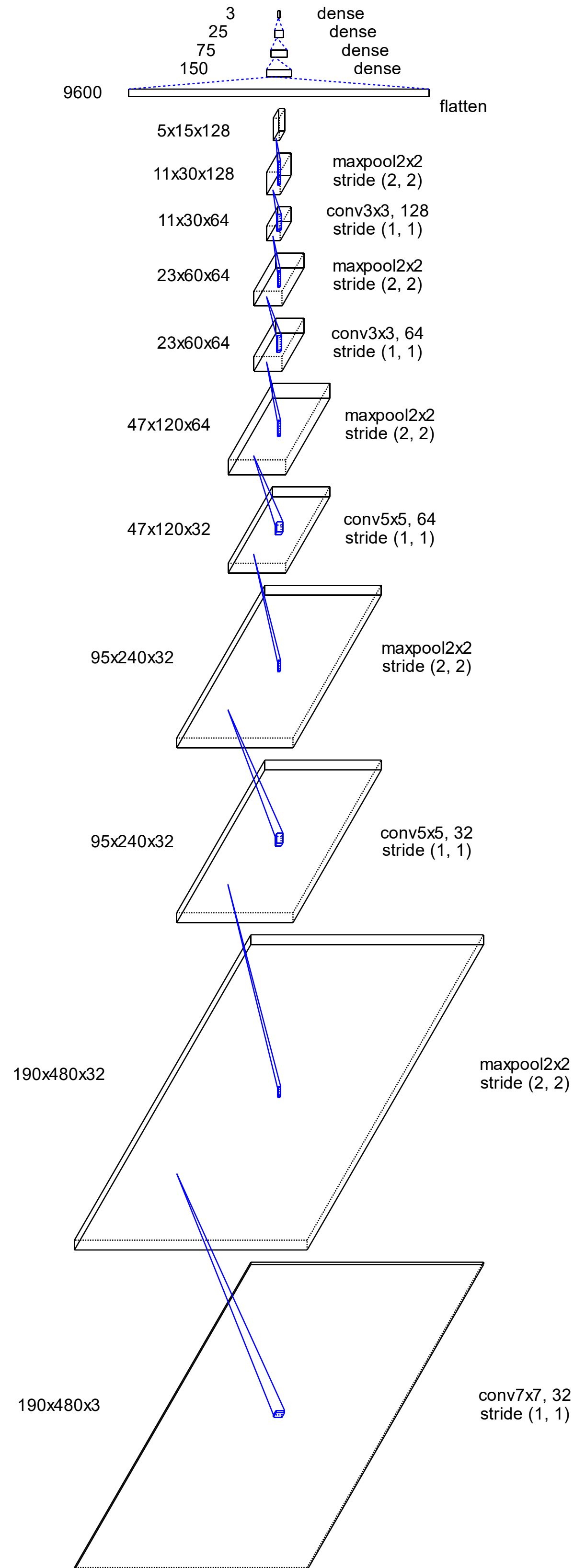

Figure 6. *Structure of the neural network*

5. EXPERIMENTS

In this chapter the training process and the three self-driving tests are discussed. Chapter 5.1 presents the training data collection process and the needed pre-processing of the collected data. Chapter 5.2 presents the training process and Chapter 5.3 presents the TensorRT optimization. Chapter 5.4 presents the three self-driving tests.

5.1 Training Data Collection

Training data was collected while a human expert was driving around the track. Data was save to the laptop which the expert used to control the robot during data collection. The expert used the laptops keyboard to input commands. Image data was recorded to video files and action data was collected to text files. Figure 7 shows and example of a training image and corresponding expert action.

Figure 7. *Example training data.*

To reduce complexity only three different commands were chosen that the robot could perform. The chosen commands were drive straight, turn left and turn right. Because of the decision that the robot always had to move forward, the action data must be filtered to remove every command and the corresponding image from the dataset that did not have “w” in the command. The final dataset had a little bit of a bias towards straight commands because about half of the commands were straight as seen in Figure 8.

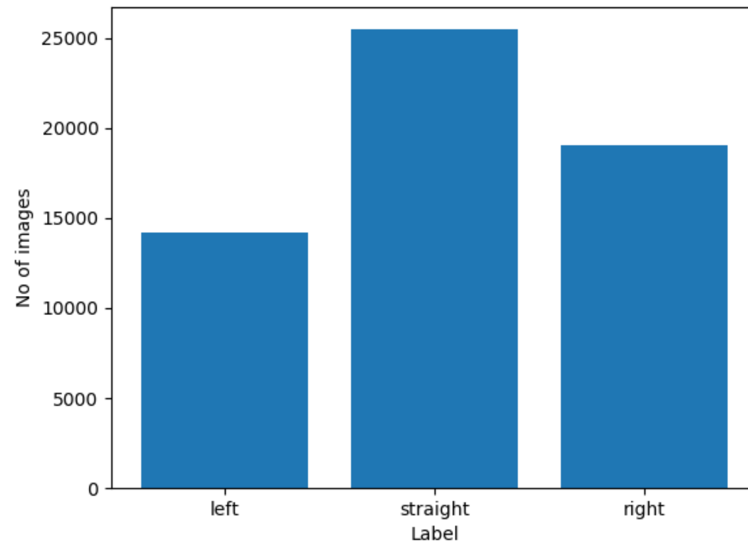

Figure 8. *Image label distribution.*

Actions shown in the Figure 5 had to be converted to one-hot encoded vectors for the model. Actions were converted so that “1,0,0” represents turning left, “0,1,0” represents turning right and “0,0,1” represent driving straight. Before using images from the camera, pre-processing has to be done. The following steps describe the implemented pre-processing steps.

1. Cropping images: The images are cropped from the top by 170 pixels in order to remove unnecessary data and to focus on the region of interest which is the track. Other benefit is that smaller image is computationally faster meaning faster training. Cropped images have the size of 480x190x3.
2. Normalization: Each colour channel is scaled to values between zero and one by dividing the pixel value by the maximum 8-bit integer value which is 255.

After the pre-processing steps, the total of number of valid action image pairs in the dataset was 58745. 49933 images were used to train the CNN and 8812 images were used to validate the network.

5.2 CNN Training

Training was done on a computer with Intel I7-8700k processor, Nvidia GTX 1080TI GPU and 16 GB of DDR4 memory running on Windows 10. First, the model was trained for 10 epochs to get information when overfitting was occurring. Figure 9 shows the validation and the training accuracy as well the loss for both the training and the validation for every epoch. Training for 10 epochs took about 25 minutes with the presented model and hardware configuration. Epoch 5 was chosen as the number of epochs to train the final model. This seemed as a good compromise between the accuracy and the loss.

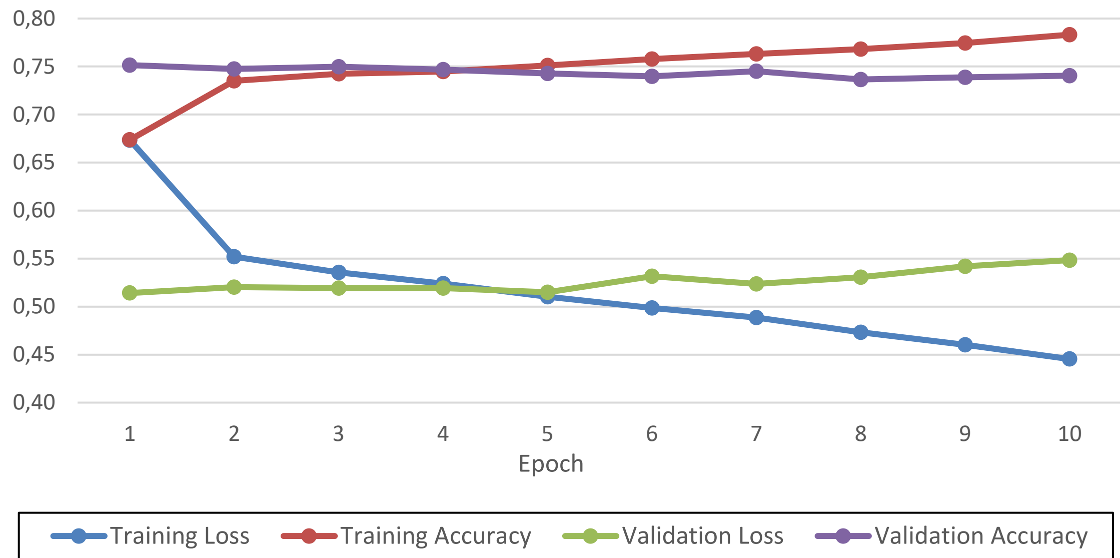

Figure 9. *Training information*

As seen in Figure 9 the validation accuracy is not very high. One of the reasons might be that there are many similar images with different labels because human expert has the ability to predict the direction from the image and for example start turning before the center line turns in the image. Other reason might be that a human can remember the track from the previous laps and can start turning in corners earlier than on the previous lap and this creates two different actions for same kind of input image. I this thesis a large dataset is used to avoid this problem.

5.3 TensorRT Optimization

TensorRT is a C++ library that enables high-performance inference on Nvidia GPUs. It is designed to work in conjunction with NN training frameworks. Some framework such TensorFlow have built-in TensorRT components so TensorRT can be used within the framework. [29]

TensorRT optimizes the model by performing platform specific optimizations or by optimizing the model layout directly. The optimization process can remove unused layers and operations that are equivalent to no operation. It can also fuse convolutions and ReLU operations together. Optimization also modifies weights if different precision is used than on the original model. [29] Lowering the precision makes the model faster by removing unnecessary calculation when lower precision is sufficient.

Inferencing with the TensorFlow model on the Jetson Nano took too much time for it to be real-time as it needed to be. Inferencing took about 200ms when it needed to be a

fraction of that. The 200ms was measured when Jetson was on power saving "5 watt" mode. On a TensorRT model optimized and with the power saving mode the inferencing took only about 30ms thus enabling real-time operation with a 30 FPS video stream.

5.4 Self-driving Tests

In total three self-driving tests were done. In the first test the NN was on the laptop and in the second test NN was moved from the laptop to the Jetson Nano. First two tests used a NN model that used grayscale images as input. This was done to save memory on the training machine, the laptop and the Jetson Nano. In the final test, the NN was changed to a model that used colour images because of the limitations of grayscale data.

First self-driving test used the laptop as the controller where the NN was running. Because the laptop didn't have a discrete GPU the throughput of the NN was about 10 interferences per second. This was too slow to make the robot reliably react to changes in direction of the track. The wireless connection latency was also a hampering effect on the reliably. The robot often drove off the track and could not complete even one lap because of the previously mentioned limitations.

To increase the throughput, the NN was moved to the onboard Jetson Nano. With the Jetson Nano and the optimization discussed in the previous chapter, the throughput was increased to about 30 interferences per second. With this increased throughput the robot was able to react much faster and was able to follow the track quite accurately. On tighter curves the robot was not able to make the turn and drove out of the track. With a little bit tuning to the motor turning speed multipliers, the robot was able to drive the track on its own. However, with the tuned multipliers it was very hard for a human to drive the track, because the robot made too sharp turns for a human to drive accurately.

Next tests showed that the robot sometimes started to falsely follow side lines of the track. The robot didn't distinguish the side line from the center line. This problem was solved by using colour images for the NNs input. With colour images the NN was able to more easily distinguish the yellow center line from the white side line.

The last test used the model presented in Chapter 4. This model performed quite reliably, and the robot was able to drive multiple laps autonomously. The only problem remaining was that sometimes the robot turns sharply on the starting straight. The reason is that the robot confuses the starting lines to a sharp turn. **Error! Reference source not found.** shows two example images from the training dataset. Left image shows a situation where robot can confuse starting lines to a sharp left turn. Right image shows an example of a sharp left turn to ease comparison. The almost horizontal white lines might be the

reason for this confusion, but the confusion only happens when the robot arrives to the starting lines at the right angle. However, this confusion happened quite rarely so it wasn't a big problem.

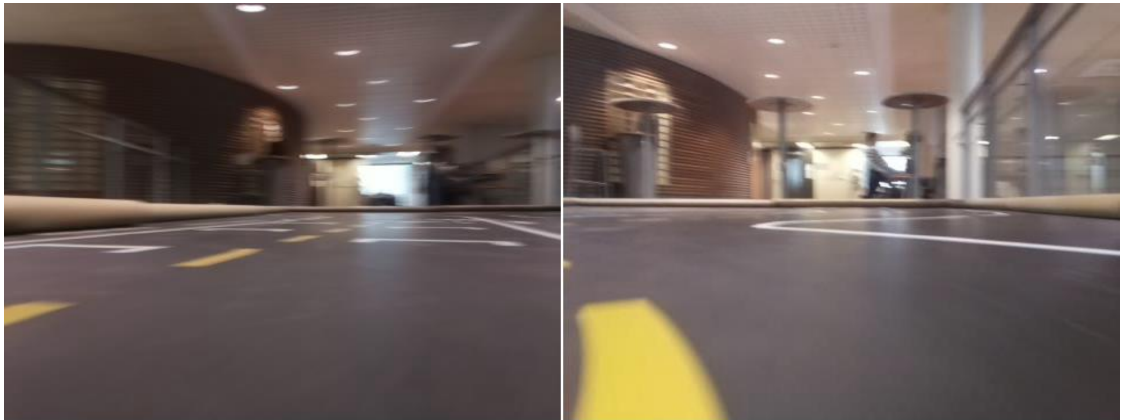

Figure 10. *Starting straight wrong turn examples*

6. CONCLUSIONS

The goal of this thesis was to implement robot that is able to autonomously drive around the chosen test track. The proposed implementation used NN to classify images to three different actions that the robot should take. Hardware implementation used Jetson Nano to compute the NN and RPi to capture images and control the robot's motors. The NN model and code used in this thesis is available at Github[1].

Experiments show that the proposed implementation works reasonably well on the chosen test track. Moving the NN from the laptop to the Jetson Nano and that way speeding up the computation made noticeable improvement to the robot's performance. NN controller's low validation accuracy did not seem to affect to the robot's performance on the track. However, this implementation could be further improved by further optimizing the NN parameters and configuration.

Possible future hardware improvements could be changing the camera setup. One change could be to move the camera higher of the ground so that the robot could see farther and react earlier to changes in the track. Another change could be to increase the field of view of the camera to make the robot more aware of its position on the track. Other hardware improvement could be to remove RPi from the configuration and connect the camera and motor controller directly to the Jetson Nano. This improvement would decrease the latency between the image capture and the movement action.

Software improvements could be to change the NN controller to output steering angle and throttle values instead of direction commands. This change would make the robot movement more precise and possibly faster. Another improvement would be to add a system to help the robot to avoid obstacles on the track. This improvement would make the robot more reliable and in addition, it would enable multi robot racing.

[1] https://github.com/Manninee/Monsterborg

REFERENCES

[1] World Health Organization. 2018. The top 10 causes of death. https://www.who.int/news-room/fact-sheets/detail/the-top-10-causes-of-death [referred 23.3.2020]

[2] World Health Organization. 2020. Road traffic injuries. https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries [referred 23.3.2020]

[3] N. A. Greenblatt. 2016. Self-driving cars and the law. IEEE Spectrum, vol. 53, no. 2, pp. 46-51. doi: 10.1109/MSPEC.2016.7419800.

[4] Tesla, Inc. 2020. Autopilot. https://www.tesla.com/autopilot [referred 6.5.2020]

[5] Uber Technologies, Inc. 2020. Self-Driving Car Technology. https://www.uber.com/us/en/atg/technology/ [referred 6.5.2020]

[6] Waymo LLC. 2020. Home. https://waymo.com [referred 6.5.2020]

[7] Honda Motor Company, Ltd. Automated Drive. https://global.honda/innovation/automated-drive/detail.html [referred 6.5.2020]

[8] Ford Motor Company. 2019. Autonomous Vehicle. https://corporate.ford.com/company/autonomous-vehicles.html [referred 6.5.2020]

[9] Nissan Motor Co., Ltd. 2017. Self Driving Autonomous Car. https://www.nissanusa.com/experience-nissan/news-and-events/self-driving-autonomous-car.html [referred 6.5.2020]

[10] J. Fingas. Engadget. 2018. Waymo launches its first commercial self-driving car service. https://www.engadget.com/2018-12-05-waymo-one-launches.html [referred 6.5.2020]

[11] A. Hussein, M. M. Gaber, E. Elyan, C. Jayne. 2017. Imitation Learning: A Survey of Learning Methods. ACM Comput. Surv. 50, 2, Article 21 (June 2017). doi: 10.1145/3054912

[12] Pomerleau, A. 1989. ALVINN: An Autonomous Land Vehicle In a Neural Network. Url: https://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf

[13] A. Giusti et al. 2015. A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots. IEEE Robotics and Automation Letters, vol. 1, no. 2, pp. 661-667. doi: 10.1109/LRA.2015.2509024.

[14] S. Ross et al. 2013. Learning monocular reactive UAV control in cluttered natural environments. 2013 IEEE International Conference on Robotics and Automation, Karlsruhe, pp. 1765-1772. doi: 10.1109/ICRA.2013.6630809.

[15] D. Gandhi, L. Pinto, A. Gupta. 2017. Learning to fly by crashing. 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, pp. 3948-3955. doi: 10.1109/IROS.2017.8206247.

[16] T. Osa, J. Pajarinen, G. Neumann, J.A. Bagnell, P. Abbeel, J. Peters. An Algorithmic Perspective on Imitation Learning. 2018. Foundations and Trends in Robotics Vol. 7, No. 1-2. pp.1-179. Url: https://arxiv.org/ftp/arxiv/papers/1811/1811.06711.pdf

[17] PiBorg. MonsterBorg - The Ultimate Raspberry Pi Robot. https://www.piborg.org/robots-1/monsterborg [referred 19.3.2020]

[18] PiBorg. ThunderBorg - Dual 5A Motor Controller with DC/DC & RGB LED. https://www.piborg.org/motor-control-1135/thunderborg [referred 19.3.2020]

[19] Raspberry Pi Foundation. Raspberry Pi 3 Model B+. https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/ [referred 19.3.2020]

[20] Nvidia Corporation. Jetson Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/ [referred 19.3.2020]

[21] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, 2014, Dropout: a simple way to prevent neural networks from overfitting, Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929-1958.

[22] A. Géron. 2018. Neural networks and deep learning. O'Reilly Media, Inc. Url: https://learning.oreilly.com/library/view/neural-networks-and/9781492037354

[23] J.D. Kelleher. 2019. Deep Learning. Cambridge, MA: The MIT Press. Url: https://viewer.books24x7.com/Toc.aspx?bookid=147440

[24] U. Michelucci. 2018. Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks. Dübendorf: Apress. Url: https://learning.oreilly.com/library/view/applied-deep-learning/9781484237908

[25] M. Bernico. 2018. Deep Learning Quick Reference. Birmingham: Packt Publishing. Url: https://learning.oreilly.com/library/view/deep-learning-quick/9781788837996/

[26] F. Ertam, G. Aydın. 2017. Data classification with deep learning using Tensorflow. 2017 International Conference on Computer Science and Engineering (UBMK). Antalya. pp. 755-758, doi: 10.1109/UBMK.2017.8093521.

[27] I. Goodfellow, Y. Bengio, and A. Courville. 2016. Deep Learning. MIT Press. Url: https://www.deeplearningbook.org

[28] D.P. Kingma, J. Ba. 2015. Adam: A Method for Stochastic Optimization. 3rd Inter. Conf. for Learning Representations. Url: https://arxiv.org/pdf/1412.6980.pdf

[29] Nvidia Corporation. 2020. TensorRT Developer's Guide. Url: https://docs.nvidia.com/deeplearning/sdk/pdf/TensorRT-Developer-Guide.pdf [referred 5.5.2020]