Examensarbete
15 högskolepoäng, grundnivå

Usage of End-to-End Machine Learning for Self-Driving Vehicle

Användning av end-to-end maskininlärning för självkörande fordon

Pratchaya Khansomboon

Examen: Högskoleingenjör 180 hp
Huvudområde: Datavetenskap
Program: Datateknik och Mobil IT
Datum för slutseminarium: 2023

Handledare: Paul Davidsson
Examinator: Johan Holmgren
Abstract

This thesis is about the usage of a machine learning model for self-driving vehicles running on a small mobile computing unit, in this case, a smartphone. We use NVIDIA’s PilotNet model which is a simple feed-forward machine learning model for steering a vehicle. Their testing is conducted on a real-world vehicle and the model is used for lane keeping. Instead, we’ve adopted to be able to drive around an oval track that has no lane marking. The primary goal was to be able to run the model on a smartphone and this was a simple task as we’ve seen that the inference time is small enough that it can run at 30 FPS. As of now the model only generate a good steering output in the testing phase with prerecorded data and was only able to complete a corner on one side of the track.
Sammanfattning

## Contents

1 Introduction .......................... 1
  1.1 Problem Domain ......................... 1
  1.2 Research Question ....................... 1
  1.3 Scope and Limitations ................... 2

2 Theory .............................. 3
  2.1 End-to-End Machine Learning ............... 3
  2.2 Tensor ................................ 3
  2.3 Activation Function ....................... 3
  2.4 Loss Function .......................... 4
  2.5 Neural Network .......................... 4
  2.6 Hyperparameters ......................... 7
  2.7 Fully connected layers .................... 7
  2.8 Convolution and Cross-Correlation ........... 7
  2.9 Convolutional layers ...................... 8
  2.10 Convolutional Neural Network .............. 8

3 Related Work ......................... 9

4 Research Methodology ................. 10
  4.1 Methodology of choice ................... 10
  4.2 Literature Study ........................ 10
  4.3 Experimental Setup ...................... 11
  4.4 Concept ................................ 11
  4.5 Procedure ................................ 11
  4.6 Methodology Discussion .................. 11

5 Results ............................... 13
  5.1 Development Environment .................. 13
  5.2 Artefact ................................ 13
  5.3 Dataset ................................ 13
  5.4 Training ................................. 15
  5.5 Evaluation ............................... 16
  5.6 Model deployment and Real World testing ... 17

6 Discussion and Analysis ............... 18
  6.1 Dataset .................................. 18
  6.2 Training and Testing ...................... 18
  6.3 Pipeline ................................. 18
  6.4 Experimental ............................. 18

7 Conclusion and Future Works .......... 19

Glossary .................................. 22

A Environment ............................. 23
  A.1 Python packages ........................ 23
  A.2 Dockerfile .............................. 23

B Build Steps ............................. 24
  B.1 Anaconda commands ...................... 24
B.2 Docker commands
1 Introduction

Machine Learning for self-driving vehicles is being used widely as a tool to further automate our modern transportation. In recent years technology for machine learning computation is getting more accessible which makes it easy to run complex models on low-power devices[1]. The modern smartphone includes a powerful processor for both general tasks and task-specific, in our use case with a commonly named neural processing unit (NPU)[1]. These specialised hardware makes running machine learning models much faster than traditional hardware. This gives us access to embed tasks such as driving a vehicle onto a small device. The specialised hardware includes a faster way to compute for example matrix multiplication that is used a low in a neural network computation. With end-to-end machine learning technique that make the model learns all the steps between the initial input to the final output results. This makes it so that there are less engineering work in developing specific model for different tasks.

The data collection and experiments in this work are carried out on a micromobility vehicle developed and provided by LEVTEK SWEDEN AB. The vehicle is a new type of electric micromobility vehicle, designed to be lightweight, stable and practical for everyday tasks. LEVTEK’s vehicle is electronically controllable regarding both speed, braking, and steering, making it suitable for advanced automated functionalities for safety and other features such as remote and autonomous driving. The vehicle system can be extended with additional sense and compute units for vision, running neural networks, etc. With this technology, LEVTEK SWEDEN AB is striving to deliver a cutting-edge solution that enhances the riding experience while also ensuring safety and reliability.

1.1 Problem Domain

The use of a smartphone for deployment enables the model to be embedded in a real-world scenario with low latency and high efficiency. As modern smartphone processors have neural processing units for handling AI-related workloads, they are well-suited for running the trained model in real time.

The inference time of the model on a smartphone is investigated and how it performs in a real-world scenario, i.e. does it performs well driving around on the track it has been trained on? The model will be trained on a dataset collected from a vehicle and will be deployed on a smartphone. The smartphone will be connected to the vehicle, and the vehicle will be driven around a track. The model will be evaluated on its inference time and accuracy.

1.2 Research Question

RQ1 What is the inference time of the end-to-end machine learning model on a smartphone?
   RQ1.1 Is the model able to run in real-time?

RQ2 What is the model steering accuracy when driving the vehicle compared to a human?
1.3 Scope and Limitations

The following limitations are imposed on the project. The camera position when recording the video data is fixed to the vehicle and no change to it will be made during data collection and testing. This ensures that the same settings on the vehicle can be as closely as possible both during data collection and running the model. An existing application made by LEVTEK SWEDEN AB is being used that can communicate with vehicle and control it.

As the communication is being wirelessly transferred using Bluetooth® we have found that the communication rate when receiving sensory data is limited at 10 Hz. This might give some challenges if the vehicle is to be driven at high speed. To accommodate this problem both testing and data collection is done at low speed.
2 Theory

In traditional approaches for self-driving vehicles, tasks such as perception, decision-making, and control are decomposed into separate subproblems and solved independently. In contrast, end-to-end machine learning considers the problem as a whole and trains a single neural network to map raw sensory inputs to control outputs directly. This usually implies that more training data is needed.

2.1 End-to-End Machine Learning

End-to-end machine learning is an approach to designing and deploying machine learning systems where a single, unified model is used to automate an entire task, from input data to output predictions, without relying on separate, handcrafted features or intermediate models.

In traditional machine learning pipelines, data is preprocessed, transformed, and filtered to create features, which are then used as input to a separate model to make predictions. End-to-end machine learning removes the need for this manual feature engineering step by learning directly from the raw data.

2.2 Tensor

Tensors in computer science are generalisations of matrices to higher dimensions and can consequently be treated as multidimensional fields.[2] In figure 1 shows 4 different ranks for tensor. The illustration is showing a scalar (single value) as a cube in figure 1a, these can be thought of as holding a number or any kind of data representation.

![Tensor Rank Illustrations](image)

Figure 1: Tensor rank illustrations. (a) Rank 0: Scalar (shape [ ], 0-axis), (b) Rank 1: Vector (shape [4], 1-axis), (c) Rank 2: 2D-Matrix (shape [4, 4], 2-axis), (d) Rank 3: 3D-Matrix (shape [4, 4, 4], 3-axis)

2.3 Activation Function

Activation function is used in neural networks to introduce non-linearity, a neural network without activation functions would not be able to learn complex functions and be limited to modelling only linear relationships between the input and output data, i.e. a linear function. A commonly used activation function for visual feature extraction is ReLU (Rectified Linear Unit). In the paper, [3] in section III.A the ReLU function is used to clamp the negative value of a sum of input signals. Equation 1 is applied element-wise on the input tensor. Figure 2 shows an illustration of the function.
ReLU\( (x) = (x)^+ = \max(0, \ x) \)  \hspace{1cm} (1)

Leaky ReLU is another variant of ReLU, but instead of clamping any value below zero to zero it "leaks" the output. This can be seen in figure 2. Equation 2 is similar to 1 with the only difference that we can determine how much of the output below zero should be leaked.

\[
\text{LeakyReLU}(x) = \max(0, \ x) + \text{negative\_slope} \cdot \min(0, \ x)
\]  \hspace{1cm} (2)

where \text{negative\_slope} controls the angle of the negative value slope. In figure 2 Leaky ReLU slope has the value of 0.1.

\begin{center}
\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{activation_functions.png}
\caption{Illustration of ReLU and Leaky ReLU activation function.}
\end{figure}
\end{center}

\subsection*{2.4 Loss Function}

In machine learning and optimisation, a loss function, also known as a cost function or objective function, is a mathematical function that measures the discrepancy between the predicted output and the actual output or target value.

The loss function provides a way to evaluate how well a model is performing on a specific task and is typically used in the training process to optimise the model’s parameters. The goal is to minimise the value of the loss function over the training set, which implies that the model is able to predict the output as close as possible to the true output.

Different types of loss functions are used depending on the task and the type of output. For example, in regression problems, the mean squared error (MSE) loss function is commonly used, while in classification problems, the cross-entropy loss function is often used.

\subsection*{2.5 Neural Network}

A neural network is a computational model inspired by the structure and function of the human brain. It consists of many interconnected processing nodes or "neurons" that work together to learn patterns and relationships from input data. It is built by using mathematical functions to model neuron connections which gives it the power of approximating any input data to a desired output. Neural networks are also known as general-purpose function approximators. These properties can be used for various tasks, including image, speech recognition, and natural language processing.[4]
Figure 3 illustrates a single feed-forward neuron. It has one input $x$ and one output $y$. The input is multiplied by a weight $w$ and added to a bias $b$ seen in equation 3. The result is then passed to an activation function $f$ to produce the output $y$. The activation function is usually a non-linear function. The bias is a constant value that is added to the result of the multiplication. The bias is used to shift the activation function to the left or right. The weight is a constant value that is multiplied by the input.

$$y = f(xw + b)$$  \hspace{1cm} (3)

To handle multiple inputs we multiply each input by its own weight then sum them together and add the bias. See figure 4 and equation 4.

$$y = f(x_1w_1 + x_2w_2 + b)$$  \hspace{1cm} (4)

This equation can be rewritten with summation notation as seen in equation 5 to accommodate for any number of inputs.

$$y = f(\sum_{i=1}^{n} x_iw_i + b)$$  \hspace{1cm} (5)

where $n$ is the number of inputs, $x_i$ is the $i$th input, $w_i$ is the $i$th weight and $b$ is the bias.

A lot of functions can not be approximated by a single neuron. To approximate a more complex function we can use multiple neurons. See figure 5 and equation 6.

$$y = f(f(x_1w_1 + x_2w_2 + b_1)w_3 + b_2)$$  \hspace{1cm} (6)
F{\textit{igure 6: Three layer neural network with two inputs} x_1 \text{ and } x_2 \text{ and using } f \text{ activation function. Note that weights are not shown.}}

\[ y_{11} = f(x_1 w_{11} + x_2 w_{21} + b_{11}) \]
\[ y_{12} = f(x_1 w_{12} + x_2 w_{22} + b_{12}) \]
\[ y_{21} = f(y_{11} w_{21} + y_{12} w_{22} + b_{21}) \]
\[ y_{22} = f(y_{11} w_{22} + y_{12} w_{22} + b_{22}) \]
\[ y = f(y_{21} w_{31} + y_{22} w_{32} + b_3) \]

During training we compute the network’s output called forward propagation/pass and calculate the loss value to the target value. Then later calculate the derivative of the loss value. Because neural networks are just complex mathematical functions, calculus can be used to find the slope of the function at any point of the step. To evaluate the derivatives/slope of these complex functions, we can compute the derivative numerically by using the definition of the derivative of a function \( f \) at point \( a \)

\[
\lim_{{h \to 0}} \frac{f(a + h) - f(a)}{h} \tag{7}
\]

where \( h \) is a small number. This is the same as the slope of the tangent line to the function \( f \) at point \( a \).[5, p.206]

To evaluate complex functions we can use the chain rule, which states that

\[
(f \circ g)'(x) = f'(g(x)) \cdot g'(x) \tag{8}
\]

where \( f \) and \( g \) are functions and \( f' \) and \( g' \) are their derivatives.[5, p.215]

Once the slope/gradient is calculated of all the weights and biases, we can update them simply by adding the learning rate \( \eta \) (step size) to nudge the output value closer to the target value.

\textit{Figure 7 shows a simple function approximation using a neural network. The network has one input and three layers with the last layer being the output to one value with LeakyReLU as an activation function. To get a good approximation another layer is added after each layer’s output, this layer is a dropout layer. Dropout layer is used to deactivate a neuron randomly during training, this gives us a better performance to minimise our loss value. A scheduler is also used to step down the learning rate, this prevents the loss value from jumping around and instead slowly converging to a minimum. The value that determines when we should step down our learning rate is the step size. The network is trained 30 000 times or epochs on the dataset provided.}
Figure 7: Neural network with one input \( x \), three neuron layers and using \( f \) activation function to approximate \( y = x^3 + 4x^2 + x + 1 \). Dropout = 0.001, learning rate = 0.01, epochs = 30 000 and step size = 15 000

### 2.6 Hyperparameters

A hyperparameter is a setting that is not learned from data during the training of the machine learning model, rather it is set before training begins. Hyperparameters define how a machine learning algorithm learns and influences the model’s overall behaviour, and can greatly affect the performance of the model. Examples of hyperparameters include the learning rate in gradient descent, the number of hidden layers in a neural network, the regularisation parameter in a linear regression model and batch size the number of data to process at the same time. The process of selecting the optimal hyperparameters for a model is called hyperparameter tuning or hyperparameter optimisation.

### 2.7 Fully connected layers

The fully connected layer connects the neurons directly to the neurons in the two adjacent layers, without being connected to any layers within them. This layer applies the linear transformation to the incoming data.

\[
y = x A^T + b \tag{9}
\]

where \( y \) is the output, \( x \) is the input, \( A^T \) is the weight and \( b \) is the bias.

### 2.8 Convolution and Cross-Correlation

Convolution is the mathematics operation on two functions \( (f \) and \( g) \) that produce a third function \( (f * g) \). Properties of convolution include linearity and shift invariant.\[6\] Shift-invariant is the discrete version of time-invariant. This property means that if we apply a delayed input signal to the system, the out will be the same as if we had applied the original, undelayed signal and then delayed the resulting output. Linearity allows us to break down complex input signals into simpler components, convolve each component separately, and then combine the results to obtain the output.

Equation 10 is a discrete convolution as it is more commonly used in computer vision.

\[
(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n - m] \tag{10}
\]
Equation 11 shows discrete cross-correlation. Cross-correlation is being used in convolutional neural networks. The only difference is that convolution is a time reversal on one of the inputs. This can be seen in the $g[n - m]$ in eq. 10, compared to $g[m + n]$.

\[
(f \star g)[n] \triangleq \sum_{m=-\infty}^{\infty} f[m]g[m + n]
\]

### 2.9 Convolutional layers

This layer determines the output of neurons connected to local regions of the input through the calculation of the scalar product between their weights and the region connected to the input volume. A ReLU activation function is applied to the output of the filter.

### 2.10 Convolutional Neural Network

Convolutional neural network (CNN) is a class of artificial neural networks (ANN) commonly applied to analyse visual imagery. CNN is based on the shared-weight architecture of the convolution kernels or filters. Using the filters that slide along the input features and generating a translation-equivalent output known as feature maps. CNN allows us to encode image-specific features into an architecture that makes it more suitable for image-focused tasks[7]. The feature maps reduce the input data into a more manageable size for later layers of the neurons, this in turn reduces the computational requirements.
3 Related Work

The usage of machine learning models for self-driving vehicles has been a big task for the past decades. The famous neural network model ALVINN[8] uses a simple 3-layer back-propagation network to handle the task of road following. Their sensory input uses images taken with a camera and a laser range finder. The input layer uses a combination of the two input sensors with the video layer having the size of 30x32 and the range finder being 8x32. Their model is a simple feed-forward network with layers being connected, thus requiring a lot of computation. A more modern way is to use CNN layers to decrease the sensory data. PilotNet[9] is a model from NVIDIA® experiments that use both CNN layers and fully connected layers as the network’s architecture. Their architecture is what we implement but with different input image colour spaces. Their evaluation is in two steps, first in the simulation and then in on-road tests. For their on-road tests, they used a car driving on the road whose steering control are connected via CAN-bus and a video feed is fed into a model.

The PilotNet model was the bases for NVIDIA PilotNet Experiments[10]. This paper expands upon the PilotNet model by utilising other machine learning to make path predictions and an environment where they can see the road surface and how the model will steer the vehicle. The model’s output has been expanded for example to predict the lane boundaries that are inferred even when there are no lines painted on the road.

To answer our research questions we used OpenBot[11] by M. Müller and V. Koltun as reference as they implement a working model and being able to run on modern smartphones. Their machine learning model is different from how PilotNet originally works. Instead of having just feed-forward model with only images that output steering angle for the vehicle, their architecture includes a command input. Because it includes the command input the model is called a driving policy network. As the name implies, the model makes the decisions not just from images but also external commands, such as drive forward, left, right or back.

For understanding the basics of how a machine learning model works and what methods to apply an introductory paper Machine Learning with neural networks by B. Mehlig[4] provides a basic understanding of what math is used and how training works.

Design science is a method that seeks to enhance technology and science knowledge bases via the creation of artefacts that solve problems and improve the environment in which they are instantiated[12]. This method is similar to what is used by the PilotNet authors and is similar to how design science works; as they’ve designed an artefact, the model. Their problem relevance is to use a machine learning model to steer a vehicle that follows the road path. Their design evolution is their PilotNet Experiments paper that makes the model more robust and makes it less of a black box by providing more information to the designer for further debugging. These methods are our starting point for the method we used.
4 Research Methodology

Selecting an appropriate research methodology is crucial for ensuring the validity and reliability of research findings, as it determines the methods and techniques used to collect and analyse data.

To answer the research questions in section 1.2 these steps are followed: literature study, data collection, data evaluation, implementation of the preexisting model, evaluate the model, running the model on hardware and evaluate the model on said hardware.

4.1 Methodology of choice

Design Science\cite{12} is selected as the methodology. This methodology is selected because it includes these steps that are relevant to our research.

1. Design an artefact. This is our system development that encompass the whole data collection to training to model deployment. See figure 8 to see an overview.

2. Problem relevance, these are the research questions in section 1.2.

3. Design evaluation, this stage we evaluate if the steps in the pipeline can be skip or repeated.

4. Research contributions, our goal is to give an understanding on a small model with no modification is able to function properly on a small device.

5. Design as a search process.

6. Communication of research.

The artefact we designed is a pipeline from creating training data, training, and model evaluation.

4.2 Literature Study

Our main research is focusing on the convolutional neural network, its use case in self-driving vehicles and end-to-end machine learning. What are the results of these research topics and what are the advantages and disadvantages between the traditional machine learning approach and the end-to-end method?

Literature sources mainly include arXiv\cite{13} a free distribution service and an open-access archive, IEEE and searching is conducted through Google Scholar and LibSearch database. The algorithms, activation function and computation of convolutional operation used in PyTorch are from their documentation page\cite{14}.
4.3 Experimental Setup

Development environment for both running and debugging the model is needed. A tooling system for validating both collected data and the model’s data output needs to be created. To accelerate development time a GPU (Graphics Processing Unit) needs to be used during the training of the model.

4.4 Concept

Figure 9 show a problem/issue tree for the project. It represents different modules in the project. The problem tree is split into two main sections. The system that needs to be implemented and the presentation of the findings. In the system development, we chose to have three modules, PyTorch Model modules are the actual implementation of the PilotNet model and also facilitate the training step. In the data collection and evaluation, there are three steps that need to be done, one is to collect the data and clean up for training. And finally, the CoreML is the last step that the PilotNet model needs to be converted to, to be able to run on the device.

![Figure 9: Problem/Issue tree for project tasks](image)

4.5 Procedure

1. Setup environment for development
2. Collection of video and sensor data
3. Sort and process data. As the data might not be recorded correctly.
4. Implements PilotNet machine learning model in PyTorch
5. Train the model
6. Run the model on the vehicle

When training the model, a checkpoint is saved between each epoch. This makes it easier to analyse model training progression later after training and choosing which one to use later is relatively easier.

4.6 Methodology Discussion

The methodology adopted in this thesis is design science. Design science is a research approach that aims to create and evaluate artefacts that solve practical problems. It is particularly suited for research in the field of self-driving vehicles as it involves the creation of an artefact, which is a tooling system that converts raw data, the video and sensor data into model input, model conversion to a format that is compatible with the app that needs to be run.
The design science methodology follows a cyclic process of problem identification, artefact design, implementation, and evaluation. This process involves the identification of a relevant problem in the field of self-driving vehicles, the design of an artefact that addresses this problem, the implementation of the artefact, and its evaluation to assess its effectiveness in solving the identified problem.

To evaluate the effectiveness of the artefact, various tests were conducted to assess the performance of the self-driving vehicle in different scenarios. These tests involved collecting data for different types of scenarios in the vehicle operational environment in the underpass between factories to which this will be deployed.

In summary, the design science methodology adopted in this thesis has allowed for the creation of an artefact that addresses a relevant problem in the field of self-driving cars. The self-driving vehicle and tooling system have been designed, implemented, and evaluated to assess their effectiveness in navigating through different environments safely and effectively.
5 Results

5.1 Development Environment

For the training environment both Anaconda[15] and Docker[16] were used. Anaconda was used to create a virtual environment with all the dependencies needed to run the code. Docker was used to create a container with the same dependencies as the virtual environment. A docker container is used when we are running the model on cloud GPU, this makes sure we have the same dependencies as the local environment.

To create a docker environment we created Dockerfile that is used for creating docker images that can be run on any computer that can run container environments. The Dockerfile uses Python 3.9.16 as its base image and installs the dependencies in the requirements.txt file. For more detailed commands and step see appendix B.

5.2 Artefact

To train the PilotNet model in our environment, we reimplemented the Tensorflow[17] code in PyTorch[14]. This implementation[18] implementation is used as a reference, which is the forked from Sully Chen[19] that made some modifications to the model from the paper[9].

The artefact includes model implementation, training script, evaluation script, data pre-processing script and model conversion script to CoreML[20].

5.3 Dataset

The dataset used for training and evaluation is collected by driving the vehicle around an oval-shaped track see figure 10. It is recorded by a smartphone, the video feed is from the main camera and the sensor data is from the vehicle that is transferred wirelessly over Bluetooth®. The data rate being sent from the vehicle is at 10Hz.

Data recorded has the following categories:

- Video feed from the main camera at 30 FPS
- Send timestamp (Time when the data was requested by the smartphone)
- Receive timestamp (Time when the data was received by the smartphone)
- Sensor data from the vehicle at 10 Hz
  - Accelerometer
  - Gyroscope
  - Quaternion (Vehicle Orientation)
  - Throttle
  - Steering Angle (Degree)
  - Speed (m/s)

Figure 10 shows layout of the track. We collect the data by driving the vehicle around the track and testing the model if it could complete a corner or successfully drive around the whole track. The data collected includes both clockwise and anti-clockwise driving.
To use the data for training, we need to preprocess the data. The preprocessing script is included in the artefact. FFmpeg[21] is used for generating the images from the original video. The preprocessing script do the following step:

1. Read the sensor data from csv file.
2. Use ffmpeg to generate images from the video feed.
3. Calculate the timestamp that is the difference between the sent and receive timestamp.
4. Calculate which images is in the timestamp range.
5. Remove unused images.

This gives result in a dataset that contains the following:

- Images of the video feed
- Comma separated values (CSV) file that contains the following:
  - Timestamp (ts)
  - Image name (frame)
  - Steering angle (steering_angle)
  - Throttle (throttle)
  - Speed (speed)
  - Quaternion (q1, q2, q3, q4)
  - Accelerometer (acc_x, acc_y, acc_z)
  - Gyroscope (gyro_x, gyro_y, gyro_z)

For PilotNet we only use the images and the steering angle for training and evaluation. The images are resized to $200 \times 66 \times 3$, (WIDTH $\times$ HEIGHT $\times$ COLOR CHANNEL). The colour channel is normalised to $[0, 1]$ instead of using it in the original range $[0, 255]$. See figure 11 for the final image size and the field of view of the input image.
Our dataset is split into two parts, training and validation, see table 1 for the dataset statistics. The number represents how many data points we have total, this includes sensor data and the image. Because the recorded data is over different days, the split is done by manually selecting a few of the recorded data for validation from each day and the rest is used for training data.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Train</th>
<th>Validation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>50 000</td>
<td>2 200</td>
<td>52 200</td>
</tr>
</tbody>
</table>

The generated dataset includes a total of 52 200 data points, this is how much sensory data we have collected in total. This comes up to about 1.45 hours of recorded data if translated to time. This is significantly less than what the PilotNet team had. Their data collection includes 3 hours of driving and simulation time in the virtual environment.

5.4 Training

The training is conducted mostly on a local computer with a NVIDIA® RTX 3060 Ti GPU with 8 GB of VRAM. Because the model is quite small and easy to run the training takes about 10 minutes to complete. The hyperparameter we used with the dataset that gives the best result is in table 2. The batch size is chosen deliberately to be 32 because it is small enough for the GPU’s memory we are using and big enough that it can process multiple inputs at the same time. Batch size number determines how much data it is sampled on every computation. The epoch’s length is chosen after running a few training at different size and we observed that the model was ready to be used after about 16 epochs. This can be seen in figure 12 at the epoch and later is almost leveling out completely.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizer</td>
<td>AdamW</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.001</td>
</tr>
<tr>
<td>Batch size</td>
<td>32</td>
</tr>
<tr>
<td>Epochs</td>
<td>32</td>
</tr>
<tr>
<td>Step size</td>
<td>8</td>
</tr>
<tr>
<td>Gamma</td>
<td>0.1</td>
</tr>
<tr>
<td>Loss function</td>
<td>Mean squared error</td>
</tr>
<tr>
<td>Dropout</td>
<td>0.2</td>
</tr>
</tbody>
</table>

We found that lowering the dropout probability to 0.2 gives us a good result in terms of data points prediction output is closer to the target. When using higher probability, the output is more smooth but makes the model not outputing steering angle large enough to complete a corner. After each training is done, the model’s checkpoint is saved together with its current seed from the random number generator that is used to shuffle the training data input and the current epoch iteration. We later use the saved checkpoint to evaluate the model and convert it to a CoreML model.
Figure 12: Model’s accuracy under training measured.

5.5 Evaluation

Evaluation was done in two stages, the first stage was done after training. This stage was done manually by loading the model’s saved point and using it against either the validation data or newly recorded data to see if the model’s steering output is what we expected. As mentioned in the training stage the model was underperforming depending on what hyperparameter we chose in some cases, this stage of evaluation is crucial to see if the model will be useful for our purpose. The second stage is to run the model on the vehicle, this stage is hard to quantify, but we judge it by seeing how many turns it can complete.

Figure 13 shows the steering angle prediction when driving in anti-clockwise, i.e left turns only around the oval track from the model. This model uses the hyperparameters mentioned above with a dropout at 0.5. The graph shows steering input over time, with blue being the target steering angle from human input and orange being the model’s prediction from the image at the current timestamp.

Figure 13: Steering angle prediction from the model. The model is trained with the hyperparameters mentioned above with a dropout at 0.5.

Figure 14 show the same left turn but with the model’s dropout at 0.2. The graph shows that the model’s prediction is closer to the target steering angle from human input, but noisier than the previously trained model.

The inference time for the model on an Intel Core i7 2.7 GHz Quad-Core is around 2.5 ms per
frame. This including the time it takes to convert the image to a tensor (the format the model takes in as input). We have also observed that even on the smartphone it has the same or almost better time. The inference time tested on the laptop with an Intel processor is using the CPU for the computation.

![Steering angle over time (validation)](image)

Figure 14: Steering angle prediction from the model with dropout = 0.2.

5.6 Model deployment and Real World testing

To use the trained model on the device we are using Core ML Tools\[22\] to convert it into the format the smartphone application accepts. The conversion is done by loading the saved checkpoint and converting it to the CoreML model. This process creates the file required by the CoreML framework to run our trained model. When converting the model we need to specify what input type we need, in our case ImageType is used, this specifies what input size, colour channel count and colour channel value range. ImageType is a specific type iOS CoreML uses.

When the model is converted to CoreML, we can use it with LEVTEK’s iOS application that can control the vehicle. As mentioned in the previous section the application communicate wirelessly with the vehicle using Bluetooth\[17\]. This includes both controlling servo motor for steering, setting the vehicle speed and reading the sensor data. When the model is running it is using the built-in camera of the smartphone to record the current video frame and predict the steering angle and then command the vehicle. The vehicle has a servo motor mounted on the steering column. With our testing with the collected dataset, the model was able to perform a right turn at one section of the track. It is the bottom left part of the track in figure 10.
6 Discussion and Analysis

6.1 Dataset
As mentioned before that the sensor data collected is only at 10Hz, this might introduce some problems if we were to drive the vehicle at a higher speed. Receiving only 10 Hz of data rate means that every 100 ms the model is blind to what it’ll see and the vehicle will be driven by the previous prediction. This might cause the vehicle to drive off the track. This problem is mitigated by only driving the vehicle at a slower speed, in our case, it is the same speed as walking.

6.2 Training and Testing
When training the model we observed that even with little training data it was able to quickly map the raw pixel data to steering input from the prerecorded data. The only problem that impacted this project tremendously was the dropout hyperparameter was not adjusted most of training phases which gives us the result of the model’s steering output being lower than expected. A better pipeline specifically in the training phases that has support for automatically test different hyperparameter values might better way to get a better trained model.

6.3 Pipeline
The data collection, data clean up, data conversion, data splitting, training and evaluating is not automated and is done manually. This is a very time-consuming process and can be automated. The first three-part is done manually with the help of Python script for conversion, but data splitting, training set and validation set is the most time-consuming part of the project. This part needs to be more work on to automatically select which data should be used for training and validation. In our current phase the data selection is happening randomly split by about 70% to 30%. As this happens at random, there is not a system that checks for if the data is mostly driving forward och turning around corners. The OpenBot[11] addressed this by selecting having a label in their data that describe if it was a just driving straight or turning.

6.4 Experimental
In the real-world scenario, the model performance was not as expected from how the evaluation from the prerecorded data suggested, the model doesn’t steer the vehicle enough to be able to complete a corner. As we can see in figure 13 even in evaluating stage the output of the model is not turning around the corner as much as we are expecting. This was later addressed when the hyperparameter was adjusted, specifically the dropout probability of the model. This gives us a more expected result in the evaluation stage. And this also made the model able to complete one of the corners in our real-world testing track.

The model’s underperformance in the real world might be might related to the tracking not having enough markers. This might make it harder for the model to steer the vehicle correctly as it may not map the features it knows from training well. But when testing with the prerecorded data this problem might not be the case as it was outputing the correct steering output. It was showing similar results in the validation stage. Other problem is that there might be a delay between the smartphone sending out steering command to the vehicle. Because the incoming sensor data being sent from the vehicle is at 10 Hz, the outgoing command might also be at that speed. This might include an error that gets bigger as it drives.
7 Conclusion and Future Works

To conclude, our finding is that the modern smartphone can run the model without any problem. This is largely factored in that the usage of specialised hardware in this type of computation is fast enough for a model this size. This gives us the possibility to run the model in real-time. The model can run in real-time with a frame rate of 30 FPS as the inference time is only at 2.5 ms, this gives us a lot of room. This is the same frame rate as the camera can capture images. This means that the model can run in real-time. This answer the research question "What is the inference time of the end-to-end machine learning model on a smartphone?" and "Is the model able to run in real-time?". Running the model on the smartphone was not measured accurately but from what we have observed it is running equally fast as running it on a low power computer that doesn’t have a specific hardware instructions for machine learning and is only using the CPU as the computation unit. The smartphone is the iPhone 13, which has a dedicated hardware instructions to handle machine learning workloads.

For the question "What is the model steering accuracy when driving the vehicle compared to a human?", we only observed a good accuracy when evaluating it against prerecorded data, but when testing in the real world, its accuracy is low. As it was having a hard time completing corners. With our current testing strategy, it was able to only complete a turn on one part of the track complete without a problem.

When training the model we have observed that hyperparameter that heavily effects the model’s performance such as dropout needs to be chosen carefully. As we can see how much of a difference it makes in the results of the model’s steering output. The model trained with a dropout value of 0.5 was able to generate a more smooth steering output, but it wasn’t outputing large enough steering angle to be able to complete a corner. With a model that is trained with a lowered dropout value, it was able to turn in more, albeit more noisy as we can see in the figure 14.

Future work is to make the data collection and validation more automated by having it generate the correct data for the training and to have the ability to be able to manually cut out repetitive parts like driving in a straight line. This will massively streamline the project. And usage of simulator needs to be added in testing phase and data collection phase. Having a simulator can speed up the training and testing of the model and it can generate a better data for the model to be trained and test on. Simulator like CARLA simulation[23] can be used to generate the images and steering data and run the model in its environment.
References


Glossary

accelorometer  Accelrorometer is a sensor that measures acceleration. 13

CNN  Convolutional Neural Network. 9

dataset  A collection of data, often used as a basis for training machine learning algorithms. 13

gyroscope  Gyroscope is a device used for measuring the angular velocity of an object, the data are used for maintaining the orientation or calculate the object current rotation from a known starting point. 13

quaternion  Quaternion is a number system that extends the complex numbers. It is commonly used to describe rotation of an object in a 3D space. Quaternions are generally represented in the form $a + bi + dk$. 13
A Environment

A.1 Python packages

alive-progress==3.1.1
coremltools==6.3.0
jupyterlab==3.6.1
matplotlib==3.7.1
numpy==1.24.1
opencv-python==4.7.0.72
pandas==1.5.3
PyQt5==5.15.2
torch==2.0.0
torchaudio==2.0.1
torchvision==0.15.1

A.2 Dockerfiler

FROM python:3.9.16

WORKDIR /app

# Install tools
RUN apt update && apt install -y zsh htop ffmpeg libsm6 libxext6 python3-opencv

# Copy container config files
COPY .docker .docker
RUN sh .docker/install-neovim.sh

# Neovim config
RUN mkdir -p /root/.config/nvim
COPY .docker/init.lua /root/.config/nvim/init.lua

# Cleanup
RUN rm -rf .docker

# Python config
COPY requirements.txt ./
RUN pip install --upgrade pip
RUN pip install --no-cache-dir -r requirements.txt

# Requires to be able to use JupyterLab
EXPOSE 8888
B  Build Steps

B.1  Anaconda commands

We use these commands to create an empty anaconda environment and install the dependencies:

```
conda create --name {name} python=3.9.16 --no-default-packages
conda activate {name}
pip install -r requirements.txt
```

B.2  Docker commands

To build the docker image we use the following command:

```
docker build -t {name} .
```

This requires that we are in the same directory as the Dockerfile and requirements.txt file. To run the docker image we use the following command:

```
docker run --gpus=all -it --name {container_name} -v "$(pwd)/app" {name}
```

This command will run the docker image and mount the current directory to the /app directory in the container. This makes it possible to run the code in the container and save the results to the current directory. See appendix A for the Dockerfile and requirements.txt file.