UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Martin Liivak

Sample-efficient Online Learning in a Physical Environment

Master's Thesis (30 ECTS)

Supervisor: Tambet Matiisen, MSc
Supervisor: Rainer Paat, MSc

Tartu 2020

Sample-efficient Online Learning in a Physical Environment

Abstract: Autonomous driving has been seen as the next breakthrough in transportation. Autonomous vehicles employ a variety of sensors to understand their surroundings, for example multiple cameras, ultrasound sensors, and LiDARs. In this work, a much smaller scale radio-controlled cars, that only carry a central camera, are used. Their effectiveness as a test-bed for validating autonomous driving methods is evaluated. Multiple neural network architectures were proposed, among which a convolutional neural network was selected as the best candidate. The network was then trained using both supervised learning and online learning, the results of which were then compared. Experiments show that online learning in a physical environment, while costly, is a significant improvement over pure supervised learning. Additionally the radio-controlled cars proved to be a good comparative test-bed for evaluating model performance in an interactive physical environment.

Keywords:
autonomous driving, deep learning, online learning

CERCS: P170 Computer science, numerical analysis, systems, control

Andmetõhus interaktiivne õpe füüsilises keskkonas

Lühikokkuvõte: Isejuhtivaid autosid peetakse järgmiseks transportatsiooni valdkonna läbimurdeks. Isejuhtivad autod kasutavad oma ümbruse evimiseks paljusid kalleid sensoreid nagu näiteks kaamerad, ultrahelisensorid ja LiDAR-id. Selles töös kasutatakse väikseid raadio teel juhitavaid autosid, mis kasutavad seevastu väheseid ja odavaid sensoreid. Neid autosid katsetatakse võimalusena isesõitvuse meetodite hindamiseks. Lisaks katsetati mitut erinevat arvutuslikku närvivõrku, mille hulgast valiti parimana välja konvolutsiooniline närvivõrk. Seda närvivõrku treeniti nii juhendatud õppe kui ka interaktiivse õppega ning nende tulemusi võrreldi. Võrdluste tulemusel selgus, et kuigi interaktiivne õpe on füüsilises keskkonnas kulukas, pakub see märgatavalt paremat tulemust, kui juhendatud õpe. Lisaks selgus, et raadio teel juhitavate autodega saab hästi hinnata erinevate mudelite efektiivsust füüsilises keskkonnas.

Võtmesõnad:
isesõitev auto, sügavad närvivõrgud, interaktiivne õpe

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll

Contents

Glossary

Agent An autonomous system that uses observations from an environment to perform its tasks.

Learner The program that trains a machine learning model.

State A set of sensor observations taken from a single point in time.

Action Some step or change that can be performed to change the state.

Policy A set of rules that decide what action to take in a specific state.

1 Introduction

Autonomous driving has widespread potential benefits such as reducing the societal losses caused by erroneous human behaviors, providing better mobility to the portion of population that otherwise would have difficulties driving, and shifting from personal vehicle-ownership towards consuming Mobility as a Service [YLCT20].

The goal of this work is to create an automated driving system (ADS) based on end-to-end neural networks capable of controlling radio-controlled (RC) model cars. These RC cars can be driven using a control interface consisting of pedals, a steering wheel, and a screen displaying the video feed that the car sees. The main method used for training these networks will be imitation learning, where the agent attempts to imitate the behavior of a human driver.

Applying autonomous driving to RC cars could be an effective test-bed for development of end-to-end driving algorithms meant for full-sized self-driving cars. This could reduce the costs when validating such agents. Additionally the collected training data and the developed software could be used for some autonomous driving courses in University of Tartu.

State-of-the-art ADS-s employ a wide selection of on board sensors, such as monocular cameras, omnidirectional cameras, and event cameras for visual information, and radar, LIDAR, and ultrasonic sensors to gather depth information. Collecting and correlating such data is generally complex and expensive. In addition such works generally do not use end-to-end networks, but instead use a collection of models separately performing the acts of sensor analysis and correlation, planning, and actions [YLCT20].

In contrast the only external sensor the RC cars used in this work have on board is a single central camera. Therefore the end-to-end networks used in this thesis only have access to the same sensor modalities as humans have in this setting. This is an open problem, because imitation learning suffers from dataset bias, overfitting, and generalization problems [CSLG19].

The main difficulties when it comes to implementing self-driving using imitation learning are the following:

- the driving agent has to be able to successfully navigate in states and configurations that were not present in training datasets,

- the driving agent should be able to avoid the accumulation of errors that may happen from suboptimal handling of these states [RGB10].

As the training data has to be collected in a physical environment by having a human driver control the car, it is unfeasible to simply collect data for all possible configurations. Additionally the potential biases that the collected datasets might have must also be taken into account, such as the drivers mostly driving in the middle of the road. In order for the learner to be able to handle potentially unseen states, handle other cars in the track in unknown configurations, and the potential changes of racetracks, it must avoid overfitting, as all of these issues require a good degree of generalization.

In order to tackle this autonomous driving problem, the machine learning pipeline capable of connecting to the existing RC car platform had to be built from scratch. During implementation a plethora of technical issues had to be solved ranging from passing data correctly to different components, to computational efficiency and memory limitations.

The main approaches used for the training the driving agents are supervised learning, and online learning using the DAgger algorithm. Multiple model architectures were trained using supervised learning and tested on the RC cars. The best architecture was then used for searching the best parameters of the DAgger algorithm. In the end the best architecture was trained using supervised learning on a well curated training dataset, after which the resulting model was fine tuned using DAgger. This process resulted in a model that was able to continuously drive on average for five laps on the race track without human intervention.

The code for this work is publicly available, with different components in separate repositories [1] [2] [3].

[1]https://github.com/martinliivak/RCSnail-AI

[2]https://github.com/martinliivak/RCSnail-Connector

[3]https://github.com/martinliivak/RCSnail-Commons

2 Background

2.1 Automated driving systems

Hardware-software systems that can execute dynamic driving tasks on a sustainable basis are called automated driving systems (ADS-s). The adoption of deep learning in computer vision, new sensor modalities, such as lidars, and an increase in public interest and market potential has sped up the research and industrial implementation of ADS-s with varying degrees of automation [YLCT20].

In general there are two design philosophies of ADS-s: it can be designed either as a standalone, ego-only system, or it can be a connected system. In ego-only approach, a single self-sufficient vehicle carries everything necessary for automated driving operations at all times. In contrast, connected systems distribute the basic operations of automated driving amongst multiple infrastructure elements [YLCT20].

These two philosophies are in turn implemented using either modular or end-to-end designs. Modular systems are structured as a pipeline of separate components linking sensory inputs to actuator outputs. The pipelines are typically structured as follows: the raw sensor inputs are fed into localization and object detection modules, the outputs of which are in turn used in scene prediction, and finally these results are used for decision making. The motor commands are generated at the end of the stream by the control module. End-to-end driving systems on the other hand generate signals required for motion directly from the sensory inputs [YLCT20].

State-of-the-art ADSs employ a wide selection of onboard sensors. High sensor redundancy is needed in most of the tasks for robustness and reliability. Hardware units can be categorized into five: exteroceptive sensors for perception, proprioceptive sensors for internal vehicle state monitoring tasks, communication arrays, actuators, and computational units. Exteroceptive sensors are mainly used for perceiving the environment, which includes e.g. driveable areas, obstacles, and other vehicles. Camera, lidar, radar and ultrasonic sensors are the most commonly used sensors for this task. Proprioceptive sensing, i.e. measuring the vehicle's own parameters, such as speed and acceleration, is used to operate the platform safely. In addition to sensors, an ADS needs actuators to manipulate the vehicle and computational units for processing and storing sensor data [YLCT20].

The sensors that are used in this work will be described in detail in a later sections 2.4.1 and 2.4.2.

2.2 Imitation Learning

The ADS must be able to predict driving actions given some sequence of observations which are revealed to it over time. In a complex physical system such as driving a car, especially with the presence of other cars in the system, standard control methods cannot be used effectively to predict those actions. Instead a controller must be learned that is able to drive as well as a human [RGB10].
Imitation learning was selected as the approach for learning the controller. The purpose of imitation learning is to efficiently learn some desired behavior by imitating that of some expert's. Any system that requires autonomous behavior similar to an expert can benefit from imitation learning. In this case the expert will be a human driver. Furthermore imitation learning can be well applied to physical systems, which makes it a good solution to our problem [OPN$^+$18].

2.2.1 Supervised Learning

Supervised learning is an approach to imitation learning, where the goal is to learn some classifier to predict an expert's behavior. In order to accomplish this, the classifier is given a set of training trajectories visited by an expert where a single trajectory consists of some sequence of observations as input, and a sequence of actions executed by the expert during the visited states as the desired output. The idea is to use some statistical learning approach for the classifier to learn a policy that tries to best mimic the expert's actions given similar or identical observations [RGB10].
However, most statistical learning approaches used by this method make a crucial assumption that the observations are independent and identically distributed (i.i.d). In this case, the learner's prediction affects both the future observations and states during the execution of its learned policy, which violates the i.i.d assumption. Ignoring this issue leads to poor performance both in theory and practice [RGB10].
Regardless of this issue, pure supervised learning is still a potentially viable method and its ease of implementation motivates its usage as the first method in this work.

Additionally this approach will serve as a good baseline to compare against the improved supervised learning with interaction approach that tries to rectify these issues.

2.2.2 Online learning

The reason for the poor performance of pure supervised learning is that the learned policy will inevitably deviate from the distribution of states and observations encountered by the expert. For example, there can be a dataset, where the expert that has only driven in the center of the road. When an agent that learned from this dataset drifts away from the center, then this deviated distribution of states can cause a cascade of errors resulting in as many as quadratically dependent number of mistakes over time in expectation. This is because as soon as it deviates from the learned trajectory, it will be unlikely to be able to get back to the states provided by the expert [RGB10].
The sequential decision making problem has some important properties that distinguish it from the pure supervised learning setting. Firstly the solution may have important structural properties and constraints that lead to some multi-step plan, secondly there is some interaction between the learner's decisions and its own input distribution, and thirdly the cost of training data can be very high, which means that the solution cannot rely on simply more data [OPN$^+$18].
While learning a policy from some training dataset, sequential decision making is closely related to pure supervised learning, but distinctions arise in practice due to the structural properties of policies that are being imitated. Because of the physical system, the difficulty of "resetting" state and restarting predictions is very costly, as it is infeasible to do programmatically and requires human intervention.
In pure supervised learning, the source domain corresponds to expert demonstrations and the target domain to learner reproductions. In sequential decision making, the demonstration dataset does not cover all possible situations since collecting such all-encompassing expert demonstrations is almost always too expensive and time-consuming. As a result, the learner must be able to adapt to the unknown states it will very likely encounter [OPN$^+$18].
Online learning using supervised learning is an effective method that solves the aforementioned issues. It is an iterative method in which some additional data becomes available in a sequential order at every step i. This data is then used to produce a new

policy π_{i+1} at each step, as opposed to pure supervised learning where some singular training dataset is used to produce the final policy. Online learning algorithms observe the loss at every step i incurred by policy π_i and provide a new policy π_{i+1}. The aim of an online learning algorithm is to minimize the expected loss over the resulting sequence of policies $\pi_1, \pi_2, \ldots, \pi_N$. The loss functions may also vary over iterations [KSJK13]. This method is illustrated in the following figure 1.

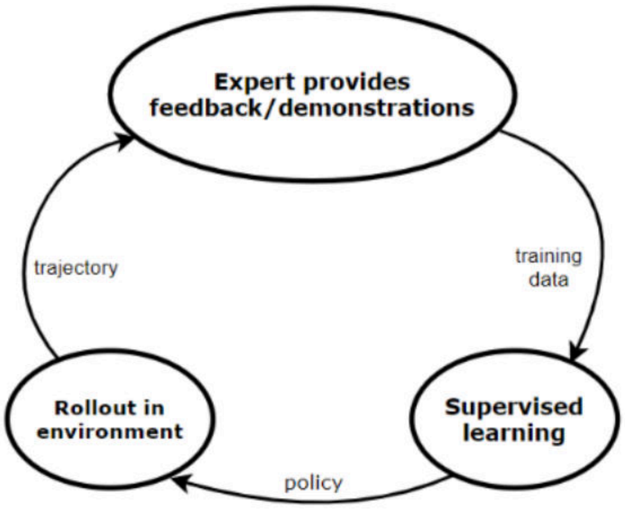


Figure 1. General flow of online learning using supervised learning [Zol19].

Online learning with supervised learning is also closely related to reinforcement learning (RL), which tries to obtain a policy that maximizes an expected reward signal. In RL, a reward function is employed that encourages a desired behavior. However, if there are optimal (or reasonably close) expert demonstrations that provide prior knowledge, it allows imitation learning to be much more efficient than methods where this data is not available, such as basic reinforcement learning. Recent work demonstrates a potentially exponential decrease in sample complexity in learning a task by imitation rather than by trial-and-error induced by RL, and empirical results have long shown such benefits as well [OPN$^+$18].

2.2.3 No regret algorithm

Related to online learning algorithms are no-regret algorithms. A no-regret algorithm is an iterative algorithm where the difference between the average loss ℓ_i incurred by π_i and the minimum average loss incurred by some policy π in the sequence approaches 0 as N approaches infinity. This difference γ_N is called the average regret.

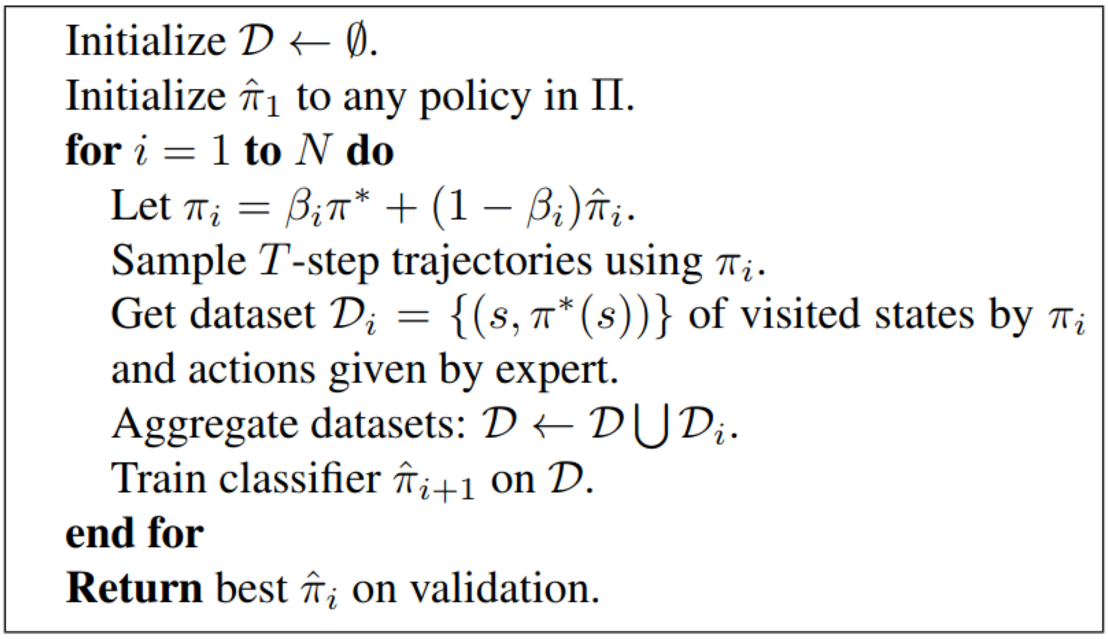
$$\frac{1}{N}\sum_{i=1}^{N}\ell_i(\pi_i) - \min_{\pi\in\Pi}\frac{1}{N}\sum_{i=1}^{N}\ell_i(\pi) \leq \gamma_N$$

Figure 2. Regret inequation [RGB10].

It has been shown that no-regret algorithms can be used to find a policy which has good performance guarantees under its own distribution of states with both infinite and finite sample losses [RGB10].

2.2.4 Dataset Aggregation (DAgger) algorithm

The main algorithm that is used for this work is called DAgger and it is categorized under supervised learning with interaction algorithms. The algorithm is closely related to no-regret online learning algorithms. This means that it results in a deterministic policy that is able to achieve good performance guarantees under the distribution of states that are induced by itself [RGB10].
The algorithm proceeds as follows. At the first iteration, it uses the expert's policy to gather a dataset of states D and train a policy $\hat{\pi}_1$ that best mimics the expert on those states. At every iteration the expert is queried for its next actions given the encountered state. In order to better leverage the expert in the learning setting, the algorithm uses a modified policy π_i, which consists of both the expert policy π^* and the policy $\hat{\pi}_i$. During the collection of new states, based on the output of a probability function $_i$, the policy π_i either returns the expert's actions, or the actions predicted by $\hat{\pi}_i$. New states and expert's actions are all added to the dataset D. The $n-\mathrm{th}$ iteration results in a policy $\hat{\pi}_{n+1}$ which is the policy that best mimics the expert on the whole dataset D [RGB10]. The pseudocode for this algorithm is shown on the following figure 3.



```
Initialize $\mathcal{D} \leftarrow \emptyset$.
Initialize $\hat{\pi}_1$ to any policy in $\Pi$.
for $i = 1$ to $N$ do
    Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.
    Sample $T$-step trajectories using $\pi_i$.
    Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by $\pi_i$
    and actions given by expert.
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.
    Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.
end for
Return best $\hat{\pi}_i$ on validation.
```

Figure 3. Dagger algorithm [RGB10].

The reason for allowing the expert to control the actions for a fraction of a time is that as the first iterations usually result in policies that make more mistakes and visit irrelevant states as the policy improves. β_1 is typically selected to be equal to 1, so that some initial untrained policy does not need to be used. The only requirement to $\{\beta_0, \ldots, \beta_N\}$ is that it has to be a sequence, the average of which is zero as N approaches infinity [RGB10]. In other words, DAgger increases the dataset at each iteration by executing a mixture of the latest best found policy and expert's policy, while querying the expert for every state, and trains the next policy under the aggregate of all collected datasets. The intuition behind this algorithm is that over the iterations, a set of states is being built up that the learned policy is likely to encounter during its execution based on previous training iterations.

The general loop of this algorithm in practice is illustrated in the previous figure 4. The expert will provide its driving inputs for some N DAgger iterations. The iterations will elapse for m number of encountered states, after which a new policy is trained and evaluated. Additionally some probability distribution function capable of producing the sequence of $\beta-$s must be selected. This means that the best combination of N, m, and β

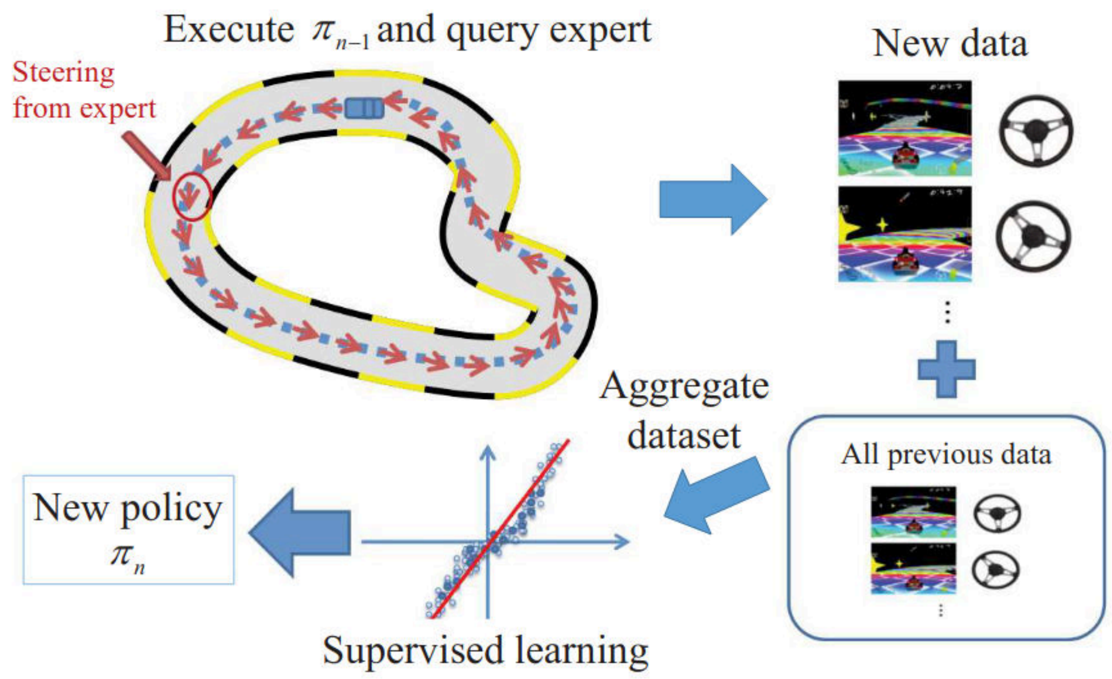


Figure 4. Usage of DAgger algorithm in a practical driving scenario [OPN+18].

must be found experimentally.

2.3 Models

In order for an ADS to navigate some driving scene, it must at least have some rudimentary understanding of it. Deep neural networks as general function estimators have proven to be a promising tool that can be used for path planning required for navigation in driving scenarios. Given their continued successes, they will be used as the models for our system [GTCM20].
Additionally, given that the system used in this thesis provides the most important feedback of the driving scene in the form of a video feed, some kind of image recognition is required. Convolutional neural networks (CNNs) have proven to be very effective in the required image recognition. Additionally the CNNs will be coupled with some multi-layered perceptrons (MLPs) in different network configurations.

2.4 RC car overview

The RC cars that were used in this thesis have the measurements of 190 * 80 * 60mm and weigh 280 grams. One of the cars used in this thesis can be seen in the following figure 5.

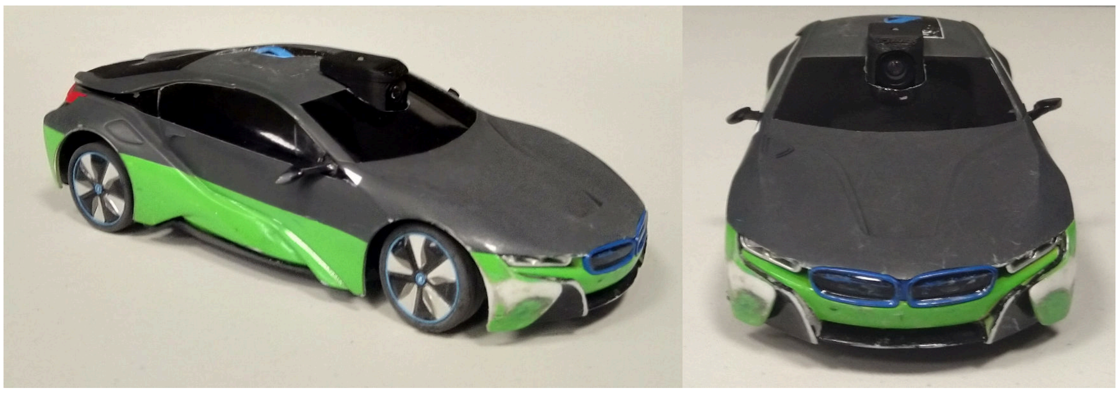


Figure 5. Picture of the RC car used in this work.

2.4.1 Exteroceptive sensors

The only external sensor on board the RC cars at the time of writing this thesis is a single central camera. The camera provides HD images in 1280x720 resolution with a frequency of 30 frames per second. The latency between the camera capturing the frame and the pipeline receiving it is approximately 100-150 milliseconds.

2.4.2 Proprioceptive sensors

RC car steering is controlled by a servo that is capable of emitting its own position meaning it can be used for monitoring the true state of steering. The electrical motors used in the RC cars are brushed DC motors, which unfortunately do not have sensors for measuring their rotational speed, therefore they are not able to emit their own position. To somewhat bypass the latter issue, the last throttle and gear input that reached the car can be emitted as a virtual reading of the motors. If the controlling mechanism takes this design into account, then the motors can also be used as virtual proprioceptive sensors, although their readings may drift and become suspect over time.

2.4.3 RCSnail architecture overview

Currently there is one race track with multiple RC cars that can race against each other. These cars are controlled from control panels that have screens, steering wheels, and pedals. One of the control panels can be seen on the following figure 6.

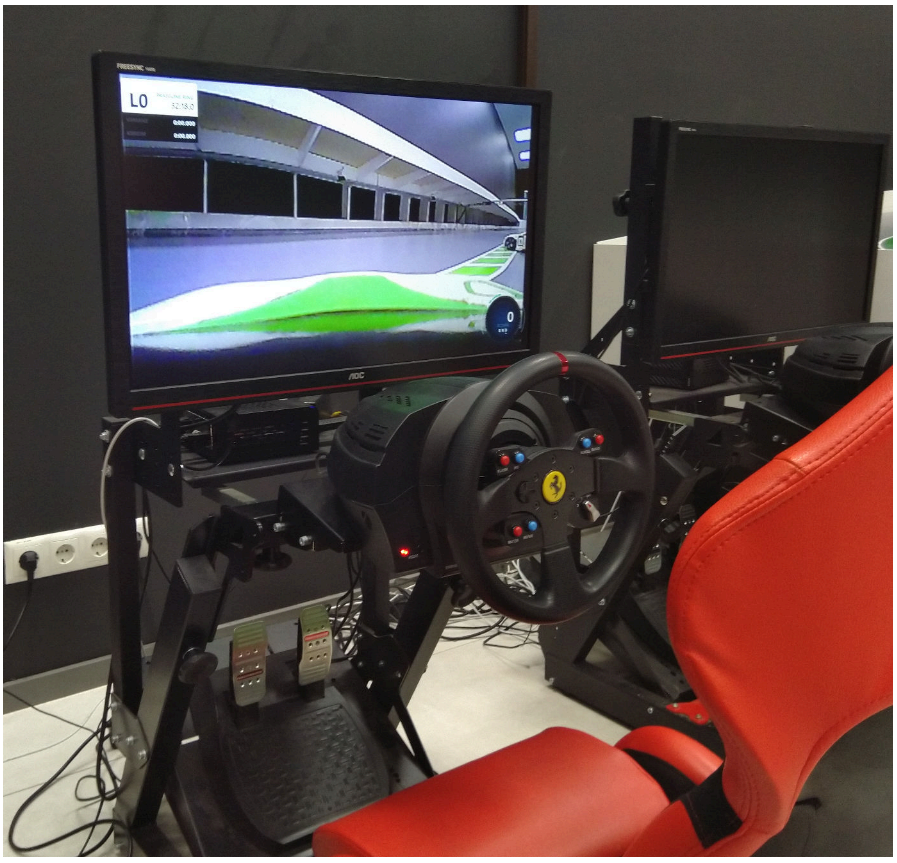


Figure 6. Picture of one of the driving interfaces.

The cars are connected to the controllers using WebRTC, which creates a peer-to-peer connection for lower latencies. The car sends the panels its current video image, which is displayed on the screen, and the control panels use their controller states to send commands to the car.

3 Methods

3.1 Pipeline

3.1.1 Overview

The resulting ADS will be a connected system due to the car being too small to be able to carry the compute required for calculating predictions on board, and installing dedicated additional hardware was found to be overly complicated and expensive. The connected system is composed of the car and a connected separate computer responsible for the machine learning pipeline. This pipeline governs everything starting from receiving the data consisting of the input video and car state data, and ending with sending the next predicted commands back to the car. The following sections will outline the exact components of this pipeline.
For the algorithmic implementation of the learner, the end-to-end approach will help keep the complexity of the system down.

3.1.2 Data

The pipeline starts with receiving a live video feed. The video frames are received paired alongside the state of the car in that frame. The video frames are originally received as 640x360 pixels large with three, RGB (red-green-blue), colour channels with values in [0, 255]. These frames are then downscaled to 180x120 size to reduce the required compute and memory footprint. The frames are also trimmed, by cutting off the top portion of the frame, as it does not contain any useful information to the model. An example of the downscaled frame and the trimming as can be seen on the following figure 7. This helps us further reduce the data footprint. After the cut, the frames are also normalized, after which the pixel values are in [0, 1.0].

The car state consists of a set of values with the current steering, throttle, gear, and braking. Throttle and braking have real number values in [0, 1.0], where 1.0 corresponds to maximal value of throttle or braking. Steering has real number values in [-1.0, 1.0], where negative values correspond to turning left and positive values correspond to turning right. Gear has integers in -1, 0, 1, where -1 is the reverse gear, 0 the neutral gear, and 1 the forward gear.

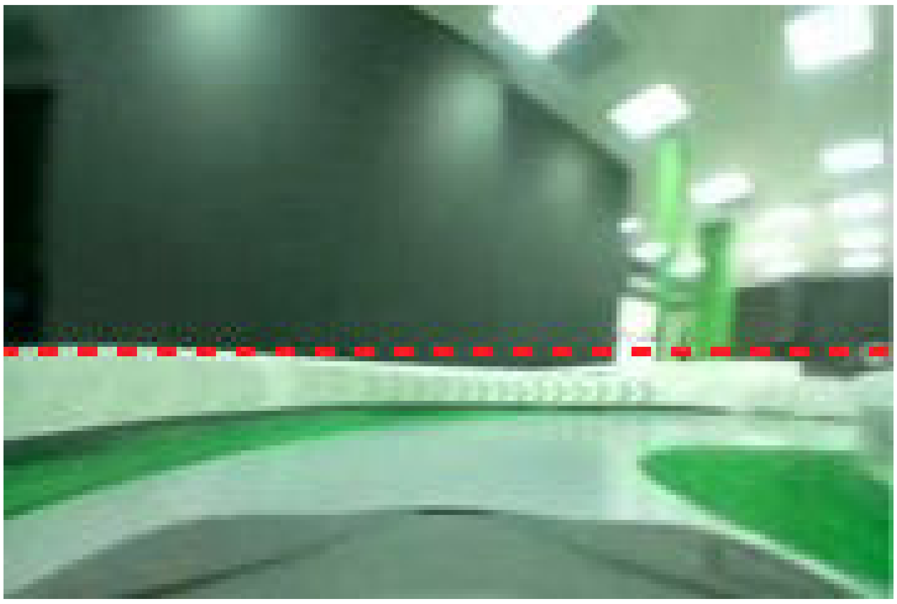
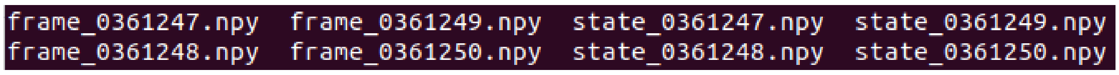


Figure 7. Downscaled frame with the trim line in red.

3.1.3 Data storage

Both the video frames and the car state are stored in memory by default during the runtime of the program. Storing unpacked video, even when downscaled, is very memory extensive. With the hardware used in this work, approximately 120 minutes of unpacked video will cause the system to run out of memory. This means that larger datasets consisting of multiple shorter videos with a combined duration exceeding the limitation couldn't be used. To solve this issue, these smaller datasets were read into memory one by one, and every frame and car state within was stored into separate files on the disk. This way the much more voluminous disk space can be used for storing the unpacked video. The downside of this approach is that the high speed of memory is lost, as disk IO operation limitations will become a bottleneck. Given that the training datasets must be significantly larger than two hours, and that an SSD disk has a fairly high IO speed, it was found to be an acceptable tradeoff. This file system allows for easier sampling to be done later on, as the indexes of the files can be tied directly to the sampling. This means that data sampling can be reduced to simple index multiplier manipulation. Depending on the starting configuration the session video can be stored in memory, and can be saved

afterwards as a regular AVI formatted video file. Alternatively the session video can be stored as per frame NumPy array (.npy) files in case training was required during the sessions.
The car state data is collected in a comma-separated value (.csv) file if the in-memory configuration is used, and otherwise in separate per-state npy files that have paired names to the video frame files.



```
frame_0361247.npy  frame_0361249.npy  state_0361247.npy  state_0361249.npy
frame_0361248.npy  frame_0361250.npy  state_0361248.npy  state_0361250.npy
```

Figure 8. Some examples of the stored .npy files and their naming convention.

In case of storing per-instance files, there will be some N frame and state files. These files are indexed by the number suffix in the filenames, in order to guarantee the expected order of files. Some examples of such files can be seen on the above figure 8.

3.1.4 Data collection

The data collection itself is fairly straight-forward. The data is collected by driving the car for extended periods of time. The data collection is done with the fully in-memory configuration, as these training data collection sessions can be done in shorter, usually about 10-30 minutes long, sessions. This configuration is used, as the formatted video files are several orders of magnitude smaller in size than the partial files, and will therefore be significantly easier to back up over the internet. In the end the numerous training drives will result in a number of AVI video files and .csv files.
During the data collection effort was put into driving the laps in a specific manner, such as making sure that both sides of the track were traversed equally frequently, so that the resulting dataset contains a reasonable amount of data in addition to having data from driving in the middle.

3.1.5 Sampling

After a sufficient dataset has been collected, it generally makes sense to apply some sort of data-level techniques to avoid having imbalances in the data. In literature data-level techniques have been used for data sampling and given that they have proven themselves, this work will be following in their footsteps [JK19]. Additionally there could be some unwanted data in the training sets, such as the driving instances which lead to a crash, which should be removed before training.

3.1.6 Upsampling

Statistical learning methods tend to become very skewed towards the most frequent instances in the dataset, which causes bad behaviors when the learner encounters some of the more uncommon instances. This is in our case unwanted behavior, as the driving system has to be able to drive straight with maximal throttle in many cases and be fully turning with very little throttle in some other rarer cases.
One method of data sampling that helps against this issue is called upsampling. The idea of upsampling is to increase the frequency of some uncommon instances appearing in the dataset. As a result the dataset is artificially enlarged with duplicated less common data. When upsampling, some limit must be selected below which a sample range is considered to be uncommon, and which therefore has to be artificially duplicated until it reaches that limit. In this work the data was split into bins by value steps of 0.01. The sampling is based on the counts of data points in these bins, and its limit was selected to be the mean of these counts, as in practice this limit proved to be a good middle ground between insufficient and overly aggressive sampling.

3.1.7 Recovery sampling

Ideally the driving system should be able to drive out of situations where it cannot proceed by moving forwards. For instance if the car manages to drive perpendicularly into a wall, then the only way it could proceed on the track is to reverse back into the track. This procedure will be called a “recovery”. In order to teach the learner recovery behavior, a significant amount of instances where this happens must be added to the training dataset. However before having to recover, the car must first be coerced into a suboptimal situation. The coercion part however would be detrimental if given to the learner for fitting, as it would demonstrate the learner how to drive into a wall.
To avoid this issue, a sampling method was created where it detects in the dataset when a recovery is being made and removes some of the training instances beforehand that lead to this recovery.

3.1.8 Augmentation

Data augmentation is generally a method that allows us to increase the diversity of the dataset without actually collecting extra data. There are multiple video augmentation techniques on image input data that could be used e.g cropping, padding, and flipping. Some augmentation techniques will be used that make the most sense given the nature of the datasets.
In this work brightness augmentation was picked, as it allows the model to better generalize across images that have different lighting levels. In addition it was also decided to use horizontal flipping augmentation. However when flipping the image, the direction of the steering data must also be inverted. The idea behind this augmentation is that the resulting dataset allows the learner to better generalize on different configurations of tracks.

3.1.9 Learning hyperparameters

A crucial part of training the neural network is the stochastic gradient descent (SGD), which is an iterative method for optimizing the network. The specific SGD optimization method used in this work was Adam, which had been designed specifically for deep learning [KB14] and has proved itself in practice [Rud16]. The main hyperparameter of Adam is called learning rate, which was selected to be 3e-4, as it has been considered to be a good base value in practice [And19].
During the training of a neural network the training dataset is split into batches. Batch size is a hyperparameter of SGD that determines the number of training samples after which the network's internal parameters are updated.
Finally there is another SGD hyperparameter that controls the number of complete passes through the training dataset called epochs. The best value of epochs can be found by observing when the loss on validation datasets over epochs converges to a stable value. Additionally, the lower the number of epochs, the less time is spent on training, which has practical value during DAgger learning, where the human expert generally does not want to spend multiple hours in total waiting between driving sessions.

3.1.10 Loss

When fitting models, the value of some loss function is used for SGD. This means it could theoretically be used as a measurement of goodness of the model. However given that loss only measures the similarity of model prediction to the actions of the human expert, and that the states and actions in the validation datasets are technically i.i.d means that loss is not a good metric for actual driving ability.
While loss cannot be used to infer learner's effectiveness on the track, it can still be very effectively used for detecting whether overfitting is taking place and whether the epoch count should be changed.
Mean absolute error was selected as the loss function, as it has been shown to work well in literature related to automated driving [GLDS20].

3.2 Model architectures

The following are the main architectures which were used in this work and resulted in models that had sufficient promise to be tested on the track. All the models are trained on the same dataset.

3.2.1 Simple CNN model

The first model is a relatively simple yet clever architecture engineered and published by Nvidia for end-to-end driving. In the original work, three frontal camera images with different view angles were used for training the model. When trained, the model is given central camera frames as input and predicts corresponding steering commands [BTD+16].
However in this work, the model was trained with input from a single front-facing camera image, and predicts both steering and throttle commands. The architecture can be seen on the following figure 9.

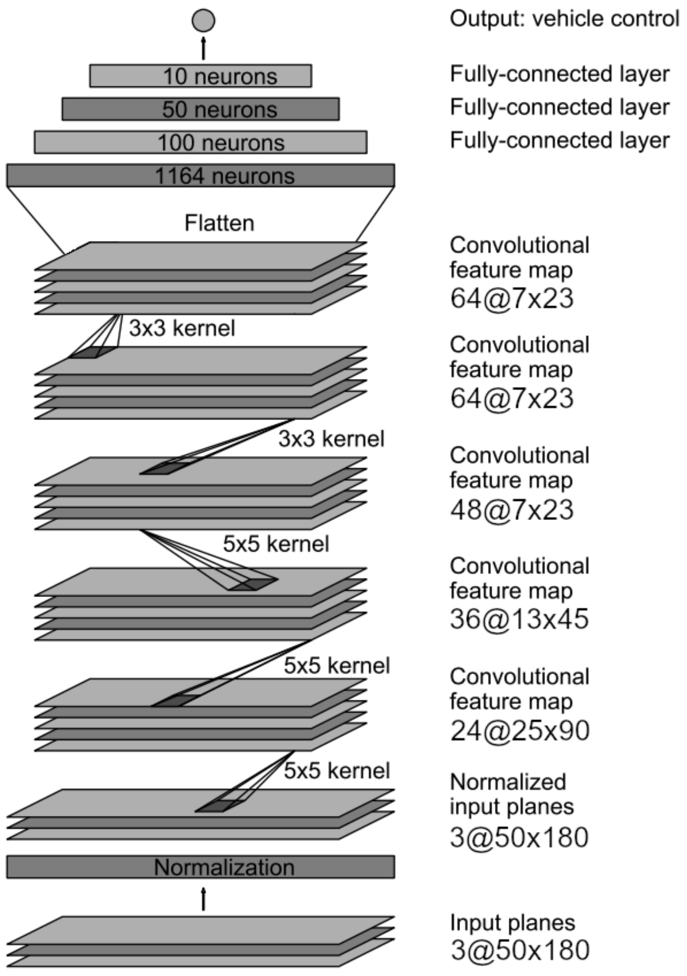


Figure 9. CNN architecture. Figure is taken in a modified form from [BTD+16]

In order to help combat overfitting, based on prior experience, L2 regularization was used on every layer in this architecture with the weight decay of 5e-4 that has been found to be a good value in literature [KSH12] [Cho16].

3.2.2 Branched MLP+CNN model

This model uses two separate inputs. The first inputs are the simple numerical inputs, which contain the steering, throttle, and gear values of the previous car state. The second input is the video frame. These inputs are passed into a model with two branches. The first branch is an MLP that consumes the numerical data, and the second branch leads to the simple CNN for handling the image data. The outputs of these two branches are passed into final fully connected layers, producing the steering and throttle commands as final outputs.

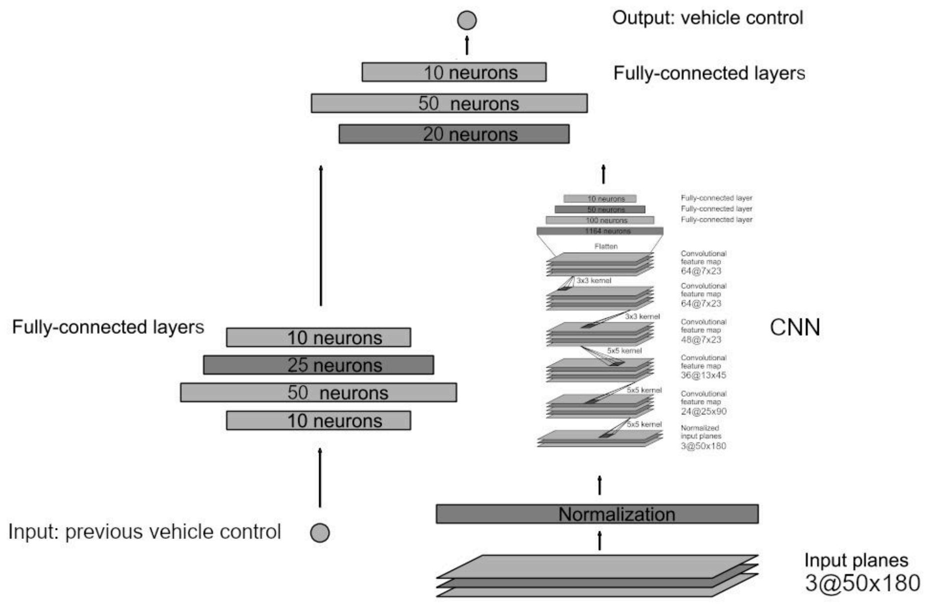


Figure 10. CNN+MLP architecture.

In order to avoid overfitting, again L2 regularization with weight decay of 5e-4 was used. In addition all the fully connected layers in this architecture are followed by dropout layers, with dropout of 0.3.

3.2.3 Frozen ResNet50

This model consists mostly of a ResNet50 pre-trained on ImageNet dataset by Keras [Ker]. ResNet50 is a network architecture that works well for image recognition [HZRS15]. The weights of the ResNet50 component will be frozen, which means that its weights will not be updated during backpropagation. This component is followed by a number of fully connected layers. The model takes the camera frame as an input, and predicts steering and throttle commands.

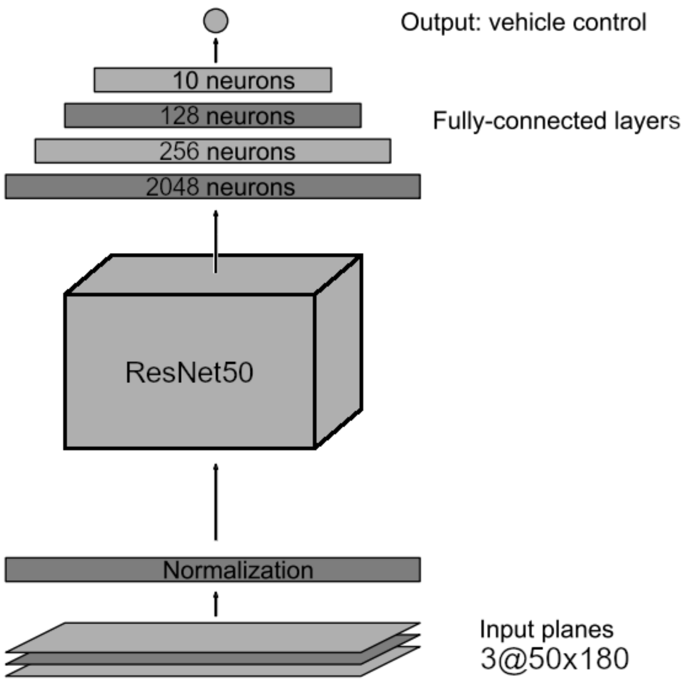


Figure 11. Frozen ResNet50 architecture.

Yet again to avoid overfitting, L2 regularization with weight decay of 5e-4 was used.

3.3 Measurements and metrics

In order to determine how well the learner is able to drive and compare it to other learners, some quantitative measurements are needed. There are some reasonable metrics that could be used for that end, which are described in detail in the following sections.

3.3.1 Error comparing to human driver

A simple metric that was used was collecting the learner's predictions made on a recorded validation lap that was not used during training and comparing the results to the human driver's actual inputs. However the problem with this metric is that the accumulating effect of learner's own errors will not be detected by this measurement, and for that reason this metric was only used for debugging purposes during the selection of models.

If a learner was incapable of producing a reasonable trajectory by this metric then it was immediately discarded.

3.3.2 Frequency of interventions

When the learner passed the human driver comparison measurement, it was introduced to the racing track. However measuring its performance on the lap has complications of its own. Even after passing the previous measurement, most learners were still incapable of traversing the entire lap without interventions. Therefore some metric is required that takes the interventions into account in order to compare their efficacy.
A good measurement that has been used in literature is the frequency of human interventions [Dep19]. This is acquired by starting the time measurement when the learner is starting to traverse the track. Upon any interventions, they are counted down, and the final sum of interventions per lap is stored. During the experiments measuring this metric, the car is reset after every lap into an identical starting position to ensure that these results are all comparable. For learners that require some interventions, but are mostly capable of autonomous driving, it is a good metric and will be used extensively. However when a learner is able to traverse the entire track length on average without interventions, a new metric is required for measuring its long term reliability.

3.3.3 Endurance drive

When the learner is able to traverse the full track with no interventions over multiple experiments, a new evaluation process is used. Unlike in the previous metric where the position of the car is reset after every lap, it will be allowed to drive continuously for an extended period of time to better evaluate its long-term effectiveness.

3.4 Memory

An experiment was carried out, wherein longer sequences of single camera frames were used as inputs to the models. Processing n multiple sequential camera frames as a simultaneous input should in theory give the model an ability to estimate the car speed, which may be beneficial for driving. These n sequential instances could be immediately following each other, or could be sampled as every last $m-\mathrm{th}$ instance.

In practice these memory frames are appended into a single image array, the result of which has dimensions identical to an $n \cdot 3$ channeled image. These are then fed as the video frame inputs to the models.
Multiple memory configurations will be compared to the no memory baseline using the model architecture that had the most promising results in order to validate whether this approach has practical value in this particular setting.

3.5 DAgger experiments

Before it is possible to use DAgger for fine-tuning models trained on static datasets, the hyperparameters of it need to be found. For this end a grid search has to be performed to find the best combinations of these parameters. Unfortunately in practice online learning takes a lot of intensive man-hours to be performed, meaning that the grid cannot be extensive and some optimization must take place when constructing the plan of experiments.
It can be expected that the model will improve, as it receives additional training. However comparing a slightly better model to an already good model is error prone. In order to better measure the effects of fine tuning, a worse baseline model is used. This way the differences are better detectable and the comparisons will be clearer. The worse baseline model is accomplished by using a significantly smaller dataset than that which was used in the previous model experiments, to train the simple CNN model. The number of interventions per lap will be used as the main measurement for comparing the results.

3.5.1 DAgger iterations count and probability distribution

As the probability distribution depends on the iteration count, and will strongly affect the effect DAgger has on the model, it is inextricably tied to the probability distribution. Therefore these two parameters have to be jointly scanned to find their best combination. In these experiments the iteration length was selected to be 800, which is approximately equivalent to driving a single lap. Firstly a probability distribution is selected. In this work it was decided to use the exponential decay function dependent on the iteration count with varied decay constants. These constants were selected from $\{-0.02, -0.05, -0.08\}$. These resulting distributions for these constants can be seen on the following graphs.

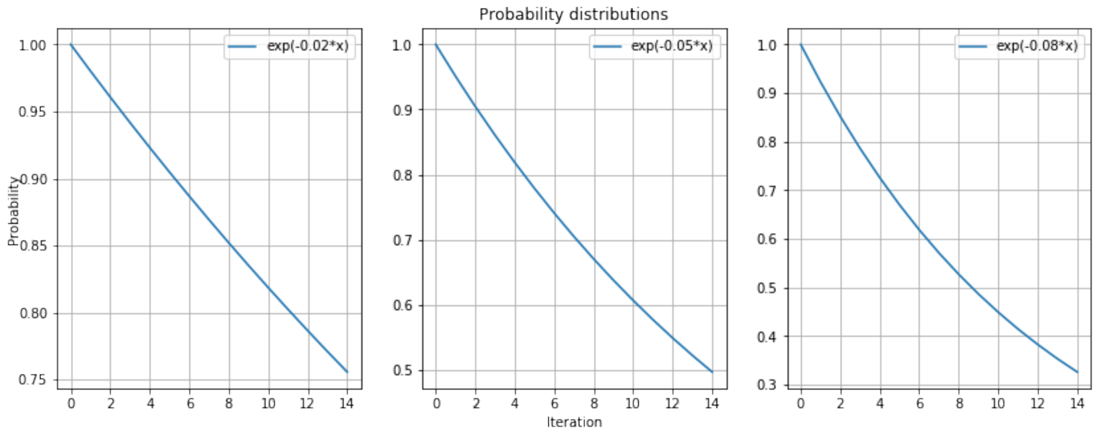


Figure 12. Probability distributions of the exponential decay function with different constants.

The first distribution value is nearly equivalent to a linear function. The second and third distributions enable sequentially faster control to the model. It is also worth noting that the final values of the probabilities decrease significantly between the constants. Alongside the decay constants, the maximum iteration count is also varied to find the best combination. The iteration counts were selected from $\{4, 8, 12\}$.

3.5.2 DAgger iteration length

After the best combination of iteration count and the probability distribution function are discovered, their values are used to find a good value for the iteration length. Given that a single lap is approximately 800 states long, the iteration lengths were selected from $\{400, 600, 800, 1200, 1600, 2000\}$.

3.6 Software architecture

The system was split into two major components - a predictor component and a connector component. The first facilitates all the steps in the machine learning pipeline:

- parsing and storing the incoming data,
- fitting the models using the stored data,

- passing the incoming data stream to models for predictions,
- sending the predictions back to the connector component.

The second component is responsible for:

- initiating and maintaining the connection with the RC car,
- providing the user with an interface for displaying the car state and live video feed,
- capturing and parsing input data from a controller,
- brokering data between the car and the predictor component.

The idea behind splitting the system into two components is that the connector component is ultimately a simple IO device that sends and receives data. When any connection issues were to happen, the connector component can be at worst easily restarted without it at all affecting the predictor component. In addition when the predictor component is integrated with the full production system, it would be connecting directly with the car system, rather than through the connector application. The split makes it natural to change the data source.
These two components communicate asynchronously using a message queue provided by ZeroMQ. This message queue implementation was chosen because of its ease of implementation and it has good performance out of the box when used in a reasonable manner.
Both components have been built in a highly configurable manner so that they can be easily adjusted or configured to support multiple algorithms and control modes such as:

- expert controlling some specific inputs, e.g the expert provides throttle and gear, while the learner controls steering,
- expert fully controlling the driving process,
- expert and learner control in a shared cooperative manner,
- learner fully controlling the driving process.

The sequence diagram of the regular flow can be seen on the following figure 13.

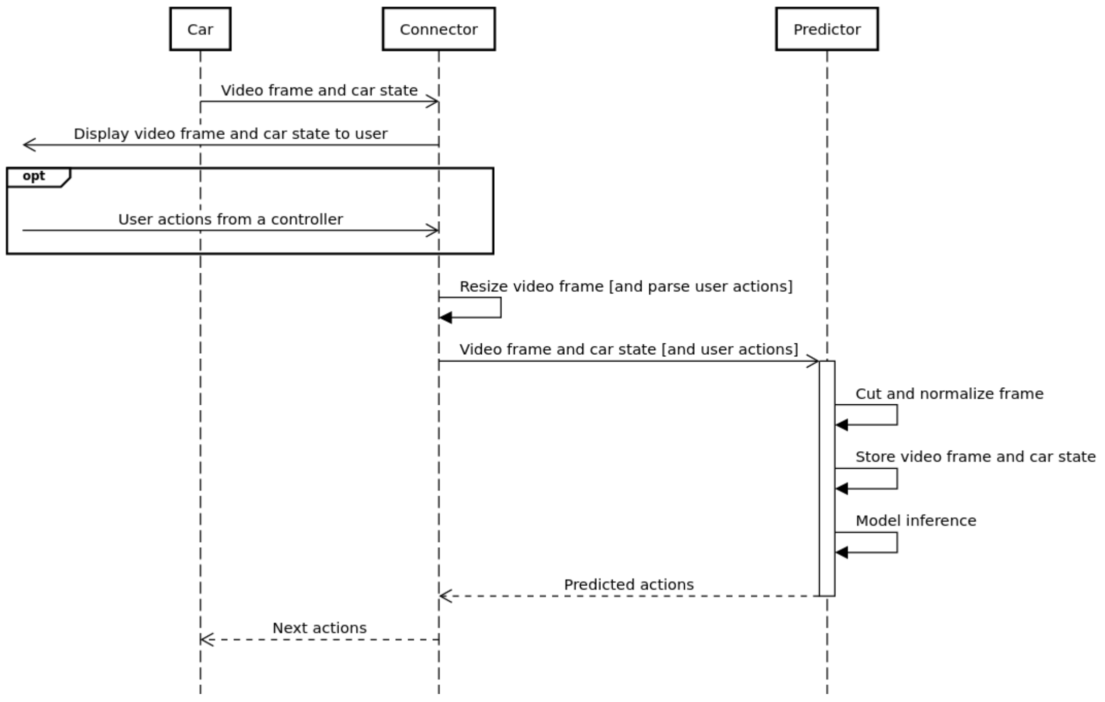

Car
Connector
Predictor
Video frame and car state
Display video frame and car state to user
opt
User actions from a controller
Resize video frame [and parse user actions]
Video frame and car state [and user actions]
Cut and normalize frame
Store video frame and car state
Model inference
Predicted actions
Next actions

Figure 13. Sequence diagram depicting the regular flow of the data pipeline.

These components were all written using Python programming language. The language was chosen because of its good deep learning ecosystem, such as TensorFlow and the libraries that were used for implementing the data pipeline such as NumPy and Pandas. Pygame library is used for translating the user's controller inputs and displaying the view from the car's camera. Python library Asyncio is used to facilitate concurrency, which is required for handling the asynchronous nature of the connections. Finally AioRTC is used for providing WebRTC bindings in Python.

3.6.1 Connector

This component deals with capturing, parsing and passing through the controller inputs, displaying the car camera's live output and transmitting the current state to the model cars using a WebRTC connection. Initially a keyboard controlling mechanism was used, but as keypresses are not continuous in nature, they do not reflect the way cars are really

being driven on the track. Additionally this approach makes collecting training data more difficult, as it is nearly impossible to hold a specific steering angle for an extended period of time, which is trivial with a steering wheel or a joystick controller. A better approach was therefore employed where an Xbox 360 controller was used. Its trigger button and joystick both enable convenient continuous inputs, which are used for throttle and steering respectively. Additionally some of the regular buttons are used for enabling manual gear changes.
In addition to controlling the car directly, this component can also send the current state and video frame via ZeroMQ to the second component, and in return receive the predicted commands and send those to the model car.
Various used libraries did not work well together with the asynchronous nature of the solution. Adding asynchronous usage to these libraries as open source work was out of the scope of this work, but the workarounds were also nontrivial.

3.6.2 Predictor

The second component establishes a connection with the connector component using ZeroMQ and waits for incoming car states and their corresponding video frames. It is capable of storing them down in either partial files for online learning, or in full files for offline learning, depending on the configuration. After receiving the data, it is passed into the configured model that is either loaded from a file or freshly created. The model's predictions are then passed back into the connector component. This component is also capable of selectively controlling the car via predicted data, e.g it could be only predicting steering values, while other commands can be provided by the expert. In addition the component can be configured to use different imitation learning algorithms, depending on the experiment.

4 Results

4.1 Pipeline

4.1.1 Data

During data collection, many hours were spent on collecting training data. When training the final models, approximately 20 hours of driving data was used. The resulting distribution of steering values can be seen on the following figure 14.

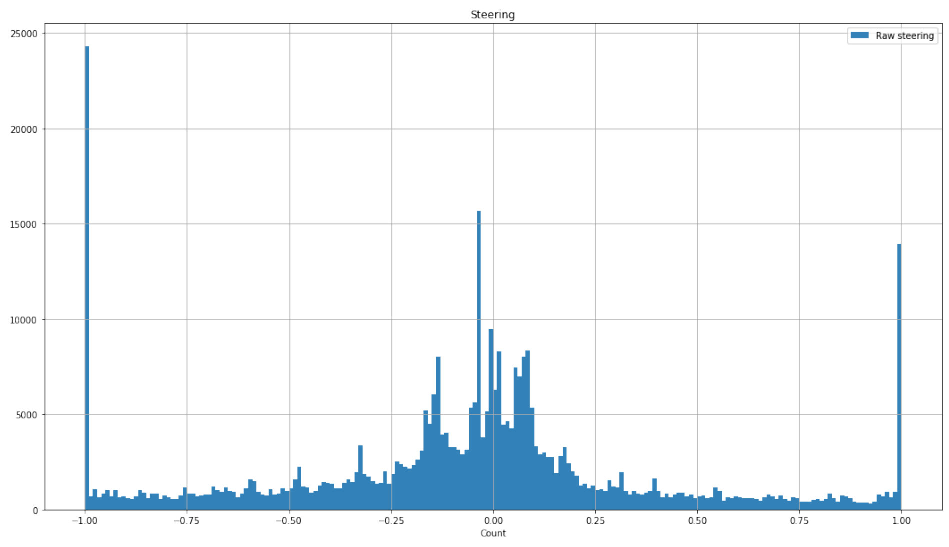


Figure 14. Distribution of steering values in the collected full dataset.

As can be seen, the distribution is the densest closest to steering values that correspond to driving straight ahead. However both extremal values are also very highly represented, which can be described by some aggressive turns in the lap, which warrant a sharper turn. Additionally the change of steering was nearly instantaneous with the controller that was used in this work, which explains the lack of intermediate values near the extremal steering values. The distribution also has a noticeable skew to the left, which is explained by the fact that the racetrack was oriented counter-clockwise.

The resulting distribution of throttle values can be seen on the following figure 15.

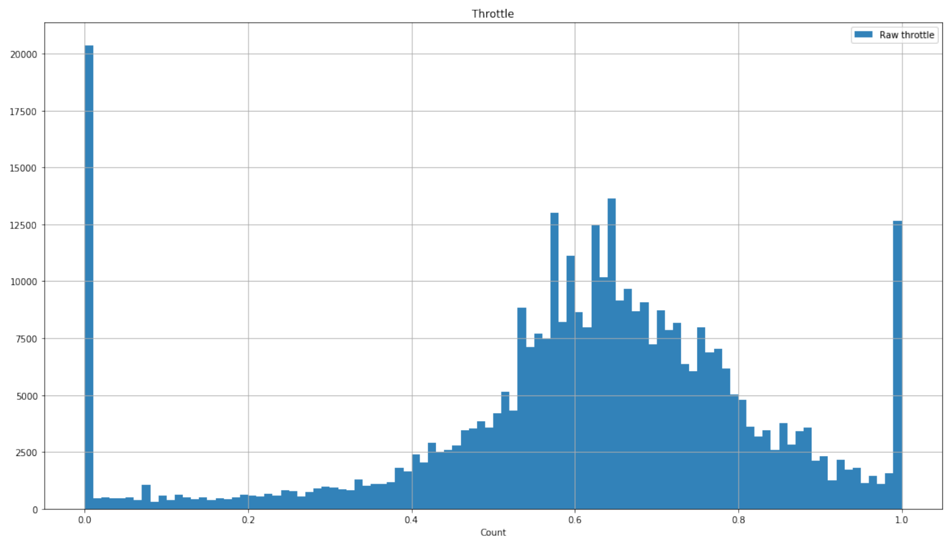


Figure 15. Distribution of throttle values in the collected full dataset.

Here a similar trend can be seen, where there is a dense distribution around throttle value of 0.65, which is approximately the speed with which it is possible to traverse most of the turns in the existing track. Additionally it can again be seen that the maximal values are well represented.
The reason for the high number of zero-throttle instances is that the car enacts engine braking, when no throttle is applied, and this method of braking was found to be sufficient meaning that the actual braking was not used in the dataset collection and experiments. In addition the very low number of samples below 0.4 is that with that throttle the car effectively enacts a weaker form of engine braking. However in practice the occurrence of these values is mostly a side-effect of passing through these values when the driver releases the throttle controller.
The maximal throttle can be explained by the fact that during the long straight parts of the lap, it makes sense to apply maximal throttle, as there is not much need to hold back. Additionally increasing or decreasing the throttle to its extremal states is almost instantaneous, which explains the lack of the throttle states close to the maximal values.

4.1.2 Data sampling

The models used in this work output both steering and throttling. Therefore the training dataset must have values for both of these controls. This creates an issue where the steering and throttling samplings would cause collisions e.g. when specific steering values are upsampled, unwanted throttle values may also get upsampled.
As mentioned earlier, throttling the values below approximately 0.4 have a similar effect of engine braking as value 0. Given that there are plenty of samples of the latter value, it was decided that throttling does not require a uniform upsampling. With steering, no such simplifications can be made, as all the values have equal importance when driving. Therefore a decision was made that steering will be prioritized when upsampling the datasets. The only downsampling that was performed on these datasets was the recovery sampling, where erroneous driving was cut out of the datasets.
The resulting sampling distributions of steering and throttling can be seen on the following figures, along with comparisons to the original data distribution.

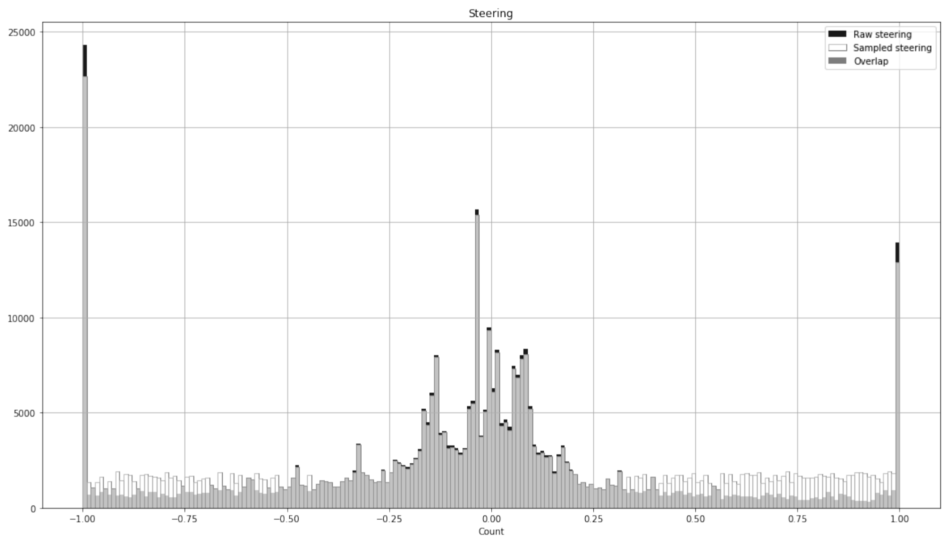


Figure 16. Distributions of the initial raw steering values and their sampled values in the training dataset.

As can be seen on the distribution, only the less common steering values closer to the extreme values had to be upsampled. It can also be observed that there are values from the raw steering distribution which were dropped as part of the recovery sampling. These values correspond mostly to very sharp turns or driving nearly straight ahead. In practice both of these occurrences were the main causes of a crash, whether intentionally or not.

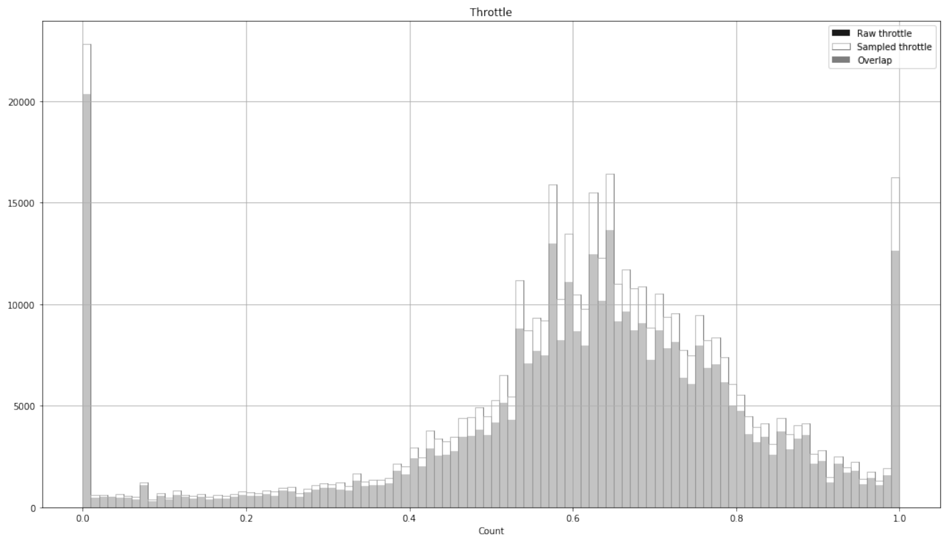


Figure 17. Distributions of the initial raw throttle values and their sampled values in the training dataset.

The throttle values were sampled as a side-effect of the steering samplings and mostly boosted the frequencies of the already known important values.

4.1.3 Learning hyperparameters

All the models were trained for 15 epochs, and after each epoch the loss on the validation dataset was recorded. The resulting plots of validation losses of the three models can be seen on the following figures below.

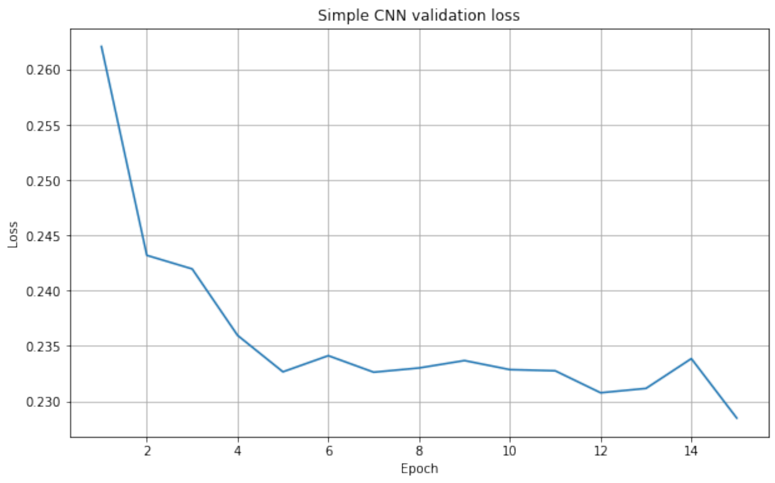


Figure 18. Simple CNN losses on validation datasets over epochs.

The number of epochs for training the simple CNN model was selected to be 8, as that seems to be the point when the validation loss has nearly converged. Although validation loss drops even lower with more epochs, this point was considered to be a good tradeoff between the resulting model accuracy and training speed.

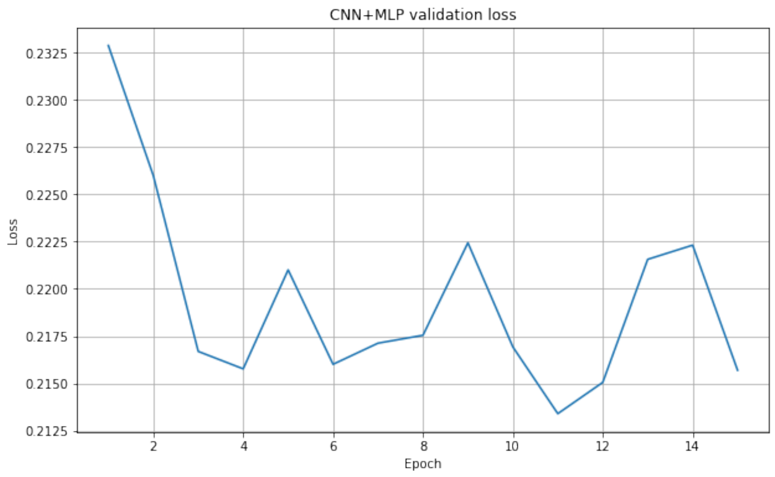


Figure 19. Branched CNN + MLP losses on validation datasets over epochs.

The number of epochs for training the CNN+MLP model was selected to be 7, as after that point the loss seems to become unstable.

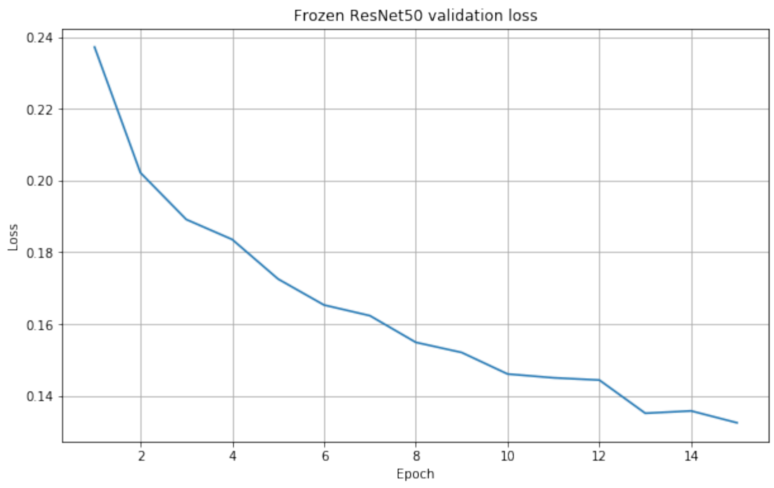


Figure 20. Frozen ResNet50 losses on validation datasets over epochs.

For the frozen ResNet50 model, the number of epochs was selected to be 11. Its validation loss continues to drop after this point, but given that this model is an order of magnitude more computationally intensive to train, it has a good tradeoff of model accuracy and training time.

4.2 Model architectures

4.2.1 Simple CNN model

Firstly the simple CNN model was trialed by streaming it the frames of a recorded validation lap, that was never seen during training. The resulting predictions were compared to the human driver's actions during that lap. The comparison results can be seen on the following figure 21.

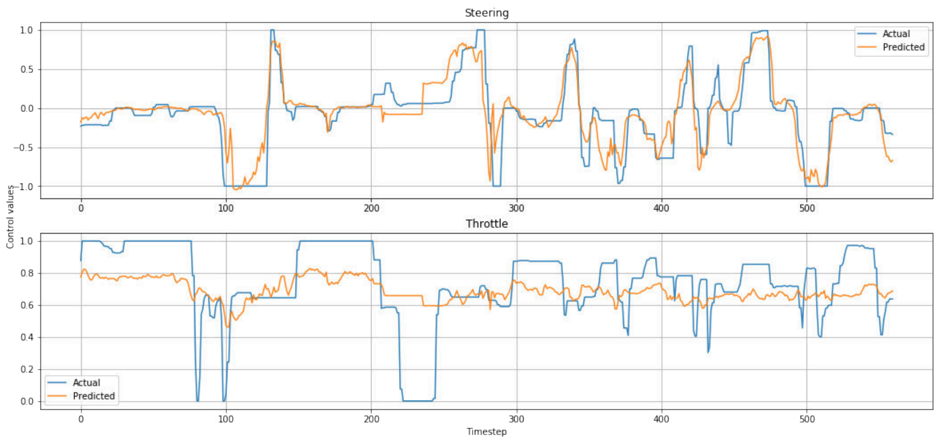


Figure 21. Steering and throttle prediction comparison to true human expert controls.

Both steering and throttle results seem to be quite good. While the throttle predictions seem to be fairly constant compared to those of the human driver's, the model was considered to be ready for track test.

4.2.2 Branched MLP+CNN model

The CNN+MLP model was tested on the same validation lap as the previous model. The comparison results can be seen on the following figure 22.

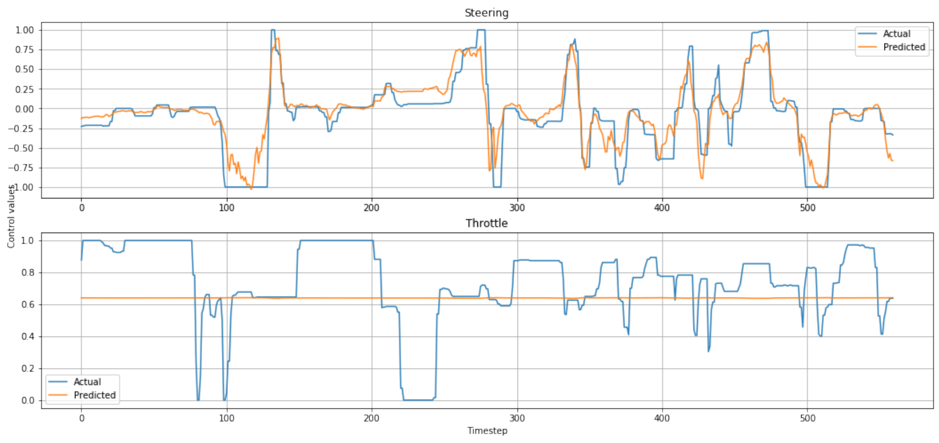


Figure 22. Steering and throttle prediction comparison to true human expert controls.

The predicted throttle is nearly constant, and even though some attempts were made to fix this issue, it was ultimately unable to be resolved with this architecture. However the steering outputs were very promising and thus this model was also considered to be ready for a track test.

4.2.3 Frozen ResNet50

The model with the frozen ResNet50 was also tested on the same validation lap as the previous models. The comparison results can be seen on the following figure 23.

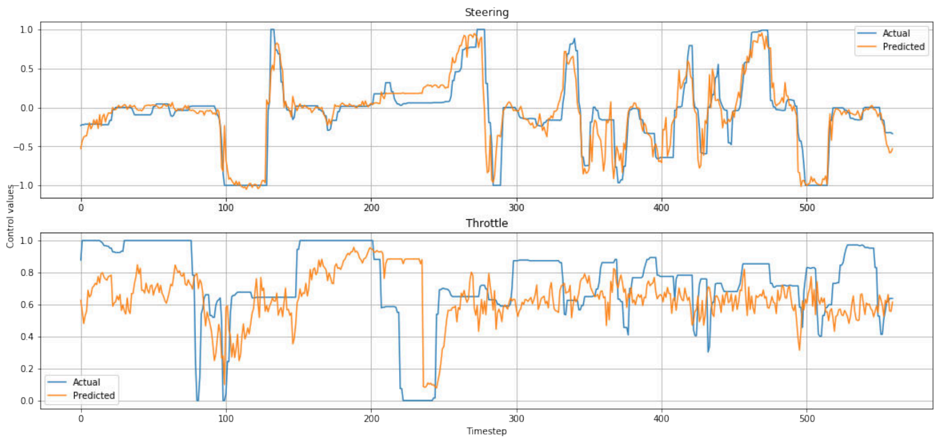


Figure 23. Steering and throttle prediction comparison to true human expert controls.

The comparison results seem to be very good on the validation dataset for both steering and throttle predictions. Given the promising predictions, this model was also considered to be ready for a track test.

4.2.4 Model performance on the racetrack

After the most promising models were identified with the human error comparison, a number of tests were conducted where these models were allowed to control the actual car on a racetrack. The resulting number of interventions and lap times were then measured as averages of five separate laps. The results can be seen on the following table 1.

Table 1. Results of the model experiments

	Number of interventions	**Lap time**
Simple CNN	0.6	48.6
Branched CNN + MLP	10.8	86.6
Frozen ResNet50	7.6	72.2

While the CNN+MLP has a good steering performance on a virtual test, where its outputs do not affect the future, it unfortunately performs terribly on an actual driving scenario. The approach has a severe issue of relying on the previous actions too strongly. This means that if some mistake was made on a previous step, the accumulation of errors is sped up, compared to a model that relies purely on the image data.
As can be seen, the ResNet50 model's approach worked significantly worse compared to the simple CNN model when trained on the same dataset. In addition the ResNet approach requires approximately an order of magnitude more time to finish its training epochs, in addition to requiring a lot more computational resources, which makes it very unwieldy for online learning purposes.
Given how much better the simple CNN model is compared to the other approaches, the following experiments will be done using this architecture.

4.3 Memory

Based on the model results the simple CNN model was used to test out different memory configurations. The following table outlines the results of different memory configurations by varying the number of remembered frames (memory length) and the interval between these frames. The same training dataset was used as in the model architecture section. The results can be seen on the following table 2.

Unfortunately it seems that the more memory is added in this manner, the worse the resulting models perform. None of these experiments managed to exceed the performance of the baseline simple CNN model.
The reason for the failure of this approach seems to be similar to the reason why the model with the MLP component performed poorly. Considering that even with no memory it is nearly impossible to have a dataset that contains all the possible states

Table 2. Resulting average intervention counts of the memory experiments

Memory length \ **Memory interval**	1	2
1	0.6	N/A
2	1.6	2.6
4	2.6	3
6	3.2	4

on the track. The longer the input trajectories are, the realm of all the possible states increases factorially. Therefore if the model has some deviations from the training set on the track, these deviations will not only affect the next action, but will persist in the input for longer, resulting in a significantly worse performance over multiple actions.

4.4 DAgger experiments

4.4.1 DAgger iterations count and probability distribution

First an initial model was trained on a static dataset using supervised learning. After this the initial model was fine-tuned using DAgger with varying iteration counts and decay constants for the probability distribution function. The iteration length was constantly 800 throughout these experiments. These fine-tuned models are then compared to a corresponding baseline model. For the baseline the initial model was incrementally trained by giving it an identical number of sequential datasets picked from the existing static training dataset. This way the baseline model was similarly incrementally trained after the initial static training and had encountered a similar amount of data.
The intervention counts were measured as an average of five laps for both the DAgger fine-tuned and the incrementally trained baseline models. The results for these experiments can be seen in the following table 3.

Table 3. Resulting average intervention counts of the DAgger iteration count and probability distribution experiments

Decay constant \ **Iteration count**	4	8	12
Baseline model	3.6	3.6	3.2
-0.02	2.4	2.4	1.4
-0.05	2.2	2.2	2
-0.08	2	2.2	2.4

As can be seen from the results, the fine tuning significantly improved the performance of the model, compared to its equivalent baseline performance. The results seem to indicate that the less aggressive drop in probability has a positive effect when the iteration count is higher. In addition there is an inverse effect with the more aggressive drop in probability, where the performance does not improve as the iteration count is increased.

4.4.2 DAgger iteration length

Using the best found iteration count and decay constant parameters from the previous section, the iteration length was varied in order to find which results in the best performing model. The baseline models were incrementally trained in the same way as outlined in the previous DAgger experiments to ensure training dataset size parity.

Table 4. Resulting average intervention counts of the DAgger iteration length experiments.

Iteration length	**Baseline**	**DAgger**
400	3.8	2
600	3.4	1.6
800	3.2	1.4
1200	3	1.2
1600	2.8	0.6
2000	2.8	1.2

Again a significant improvement can be seen over the equivalent baseline models. As can be seen from these results is that generally the longer the iteration is, the more data there is, and the better the resulting model performs. The resulting optimal length of iteration seems to be at around 1600, which corresponds to approximately two laps worth of driving. It seems that in general, the more training examples the model is fine-tuned with, the better the result is, which corresponds to general knowledge in machine learning that the more data the better. However this does not seem to hold completely, as at the length of 2000 the results are worse than at 1600. More experiments would need to be done, with finer steps, in the range 1200 to 3000 to see if multiples of track lengths have better results or if the behavior is caused by something else. Unfortunately there was not enough time to do these experiments and thus the conclusion here remains incomplete.

4.4.3 Resulting fine tuned model endurance drive

For the final experiment firstly the simple CNN model was trained on the full static training dataset. This statically trained model was then DAgger fine-tuned using the best experimentally found DAgger hyperparameters. For comparison the statically trained model was also incrementally trained in the same way as in the previous DAgger experiments. All three models were allowed to continuously control the car for 20 minutes. The resulting average intervention counts can be seen on the following table 5.

Table 5. Average intervention counts per five minutes for the 20 minute continuous endurance drives.

Model	**Interventions / 5 min**
Statically trained model	2.75
Incrementally tuned model	2.25
DAgger fine-tuned model	1

As can be seen, the DAgger fine-tuned model again shows a significant improvement in performance over both the statically trained and the incremental baseline models.

4.5 Application codebase

The applications that were created for this work are highly configurable and can be used for multiple different experiments. They also allow the usage of many different algorithms with little changes to the codebase. The codebase is publicly available and could be used as a basis for future experiments or student events with the same race cars. In addition a version of the created application will be integrated into the RCSnail production system.

5 Summary

In this work an end-to-end neural network was trained to drive a RC car. Multiple network architectures were tried out, among which a CNN architecture inspired by the Nvidia paper performed the best. DAgger online learning algorithm was then used for fine-tuning. The resulting model was able to drive five laps without human intervention. This shows that end-to-end neural networks are a viable option for autonomous driving and that a RC car system can serve as a good testbed for autonomous driving research.

Acknowledgements

I would like to thank Rainer Paat for his incredible assistance and providing me access to RCSnail facilities, and Tambet Matiisen for providing me with constant support in both practical and theoretical tasks in this thesis.

References

[And19] Andrej Karpathy. A recipe for training neural networks. http://karpathy.github.io/2019/04/25/recipe/, 2019. Accessed: 2020-8-5.

[BTD$^+$16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars, 2016.

[Cho16] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2016.

[CSLG19] Felipe Codevilla, Eder Santana, Antonio M. López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving, 2019.

[Dep19] Department of Motor Vehicles. Disengagement reports. https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/disengagement-reports/, 2019. Accessed: 2020-8-5.

[GLDS20] Zhicheng Gu, Zhihao Li, Xuan Di, and Rongye Shi. An lstm-based autonomous driving model using a waymo open dataset. *Applied Sciences*, 10(6):2046, Mar 2020.

[GTCM20] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, Apr 2020.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[JK19] Justin M. Johnson and Taghi M. Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6(27), March 2019.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[Ker] Keras. Resnet and resnetv2. `https://keras.io/api/applications/resnet/#resnet50-function`. Accessed: 2020-8-5.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[KSJK13] Purushottam Kar, Bharath K Sriperumbudur, Prateek Jain, and Harish C Karnick. On the generalization ability of online learning algorithms for pairwise loss functions, 2013.

[OPN$^+$18] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. An algorithmic perspective on imitation learning. *Foundations and Trends in Robotics*, 7(1-2):1–179, 2018.

[RGB10] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2010.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.

[YLCT20] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8:58443–58469, 2020.

[Zol19] Zoltán Lőrincz. A brief overview of imitation learning. `https://medium.com/@SmartLabAI/a-brief-overview-of-imitation-learning-8a8a75c44a9c`, 2019. Accessed: 2020-8-5.

Appendix

I. Licence