AI RC Car Racing

Martin Liivak
martin.liivak@gmail.com

Meri Liis Treimann
mltreima@ut.ee

Markus Loide
markusloide@gmail.com

Sebastian Värv
sebastian.varv@gmail.com

Abstract Autonomous or self-driving racing is an evolving sport which is gaining popularity among various race disciplines. A number of events and series have been launched, including Formula-e spin-off Roborace and Formula student driver-less. Many car manufacturers are putting much effort into developing sophisticated driver aids and even self-driving cars. For example in July 2015 Audi did beat the racing driver in world famous Sonoma raceway using self driving model of RS7, named Robby. Robby was a second generation self driving racing pilot, a successor to Bobby which was widely known for its record of beating 240kph on a main straight on Circuit Of The Americas (COTA) race track. In addition a start-up based in China called NIO broke lately a self-driving car lap record on COTA. The main difference in our case is that we have miniature cars and much less sensors. This also might be in our favor, because we do not need to take account several external factors such as weather conditions and other factors which influence traction and power output.

Keywords Convolutional neural networks, autonomous cars, self-driving cars

1 Introduction

The main goal of this project was to create an artificial neural network that is capable of controlling a remote control car based on live visual from a camera attached to it. We used remote control car racing track operated by RCSnail OÜ and located at Tartu SPARK Democenter. There, the live feed from the camera attached to the car is shown on monitor and the car can be controlled with driving wheel and throttle and break pedal. The setup of RCSnail track is good for this project because it provides an opportunity to test end develop the model in a safe and controlled environment.

The biggest difficulties to solving the self-driving car problem revolve around data. Even in the case where the network is trained to control a car in one quite specific environment, there can be some unpredictable conditions like different lighting (either artificial or natural), small tire slips, or other cars racing the track that do not even have to appear in camera view to affect the driving commands. In order to call the car truly autonomous, the training data should include many different situations rather than a few perfectly driven laps. Gathering the data itself is not as big of a problem, but as it is important that the data covers as many different conditions and situations as possible, it might take some time

and planning to get a good set of training data. Secondly, as the only input to the network is the feed from the camera, there arises a need to estimate the speed of the car, because a human driver bases the commands given to the car also on current speed.

In this work we used OpenCV to work with video data and Keras API to build the network. We used a convolutional neural network as we deal with visual recognition. We trained the network using behavioral cloning.

2 Background

Although the dream of autonomous cars began already in the beginning of 20th century, the idea of using artificial neural networks with visual feed as input was not realized before late 1980s, when researcher Dean Pomerleau created ALVINN, an Autonomous Land Vehicle In a Neural Network. Even though ALVINN could successfully use the images from camera and laser range finder to follow the road and avoid obstacles simultaneously, it was still only partly autonomous, because the speed and brake had to be controlled by a human driver.

A lot has changed in the field since ALVINN. Computing power has increased enormously. Nowadays, self-driving cars are equipped with a number of different sensors from detecting objects in every direction and their distance to measuring humidity and other environmental factors. This all together provides more than enough information to successfully outperform a human driver.

3 Methods

This section gives an overview of the data that were used to train and test the network, and the network itself.

3.1 Data

The data for training and testing the network consisted of two parts: recorded video streams from the camera attached to the car and corresponding control commands (labels). The data was collected multiple times and included laps driven by our team as well as during different events that took place at SPARK.

Videos

The visual inputs were recorded as 60 frames per second 720×400 video streams with 3 color channels (RGB). We gathered about 2 hours and 10 minutes worth of video data.

We used OpenCV (Open Source Computer Vision Library) to read in and preprocess the videos. Because with 60 fps the information difference between two frames is marginal, we decided to decrease the frame rate. We chose 5 fps which seemed to more or less match with the number of commands a human driver could give to the car within one second, mostly because of reaction time. We obtained this by reading in every 12th frame, starting from the first. We then normalized the RGB values of the chosen frames and, more importantly, resized images to 64×64 pixels.

In addition, as the image of the road from the car's current position is the only input for the network, it is difficult to estimate some commands, e.g. break value, because they depend on speed information which we do not have. This problem can be approached by using last n frames to estimate the speed at each moment. We decided to use the last 4 frames for each prediction.

The frames from each video were stored as a $(n, 64, 64, 12)$ numpy array, where the first dimension n corresponded to the number of groups and the other three represented a group of four consecutive images (in RGB). An example "row" of prepared visual data used for training the network can be seen in Figure 1.

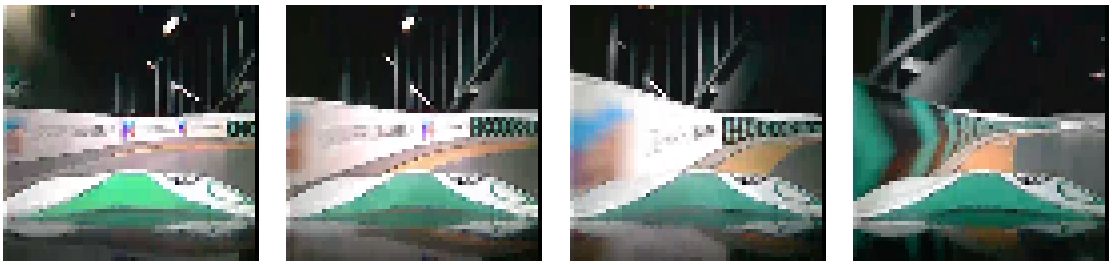


Figure 1: An example of four consecutive frames after preparing the visual data.

Due to the amount of video material being larger than what can fit into the memory of the hardware used for model training, we attempted to use generator-based fitting approaches. To this end various methods were tried, the first one just reading the provided video files in a sequence as they were needed. It quickly became apparent that the lack of data shuffling is negatively affecting performance. As the next approach, building upon Keras' Sequence class was attempted, which allows for an arbitrary "index" of training data to be called. This method, however, was not very conducive to dealing with video data, where a chosen video file had to be opened and the proper frame sought out every time the fitting function called for a new piece of data, which led to very long training times.

To improve the situation, the final generator approach was to generate data from all the provided videos at once. As the videos varied in length, one loop over the longest video would guarantee getting different parts of the track at the same time due to the shorter videos being selected multiple times. This approach performed better than the previous ones, however the speed was still nowhere near to having the data already in memory.

Commands

The control commands were stored as .csv files for each recorded video. The files had the following columns: timestamp (seconds from the start of the corresponding video), corresponding frame number, steering (from -1 for left to 1 for right), braking (from 0 to 1), throttle (from 0 to 1), and gear (0 for reverse, 1 for neutral, 2 for drive). An example of commands can be seen in Table 1.

Table 1: Example of control commands.

Timestamp_s	FrameNo	Steering	Braking	Throttle	Gear
9.554702	550	1.53e−5	0.5	0.5	1
9.569478	551	−0.0007613438065	0.4689666629	0.4689666629	1
9.586818	552	−0.007993509993	0.1799666733	0.1799666733	1
9.601655	553	−0.01363934483	0	0	1

3.2 Network Architecture

We tried several different network structures during the development, all the models were implemented using Keras library. The first model we implemented was time distributed LSTM. We decided to use this model because LSTM has loop back connections which are important because we have no data about the velocity of the car at certain time moment and dime distributing would give us information about the change of frames over short amount of time which would help the model to sense the speed. Our first hope was that LSTM can derive the speed using previous states without explicitly coding it. During the testing period the model worked seemingly well, but on the validation it produced constant output. Unfortunately we were unable to find the actual cause of this problem.

The second approach was to train the model just using LSTM, without the dime distributed part and feeding in frames one by one. In this case our hypothesis was that LSTM should be capable of extracting features and detecting environmental variables such as speed itself. Unfortunately it did not produce satisfactory results, which lead us to the next model where we added convolution part to the front to enhance the feature detection from the image. We also did try to use GRU instead of LSTM, which surprisingly produced better results by giving us explained variance score of 0.16. Explained variance turned out not to be sufficient to describe our models, since in the real world testing the models with lower explained variance score tended to perform better.

Network architecture with LSTM on top of convolution yielded much better results, but had a capacity to overfit the network. Our aim was to reduce network capacity and we decided to remove the LSTM part on top and feed in frame sets of four frames, which would simulate the time distributing, while keeping the number of network parameters fairly low to avoid overfitting. This did work quite well and we ended up using the simple convolutional network connected to dense. We adjusted number of layers, regularization strength, dropout probability and other hyper parameters according to the size of training data and trained several networks using each approach.

We used real life validation on all the models. In some cases we used combination of manual throttle and model controlled steering. This worked out quite well in some cases, because throttle sensitivity does depend a lot on battery voltage and it is very easy to lose control and spin out in case of accelerating out of slow corners.

We finally settled for a CNN shown in Figure 2. The network has $96,388$ parameters in total.

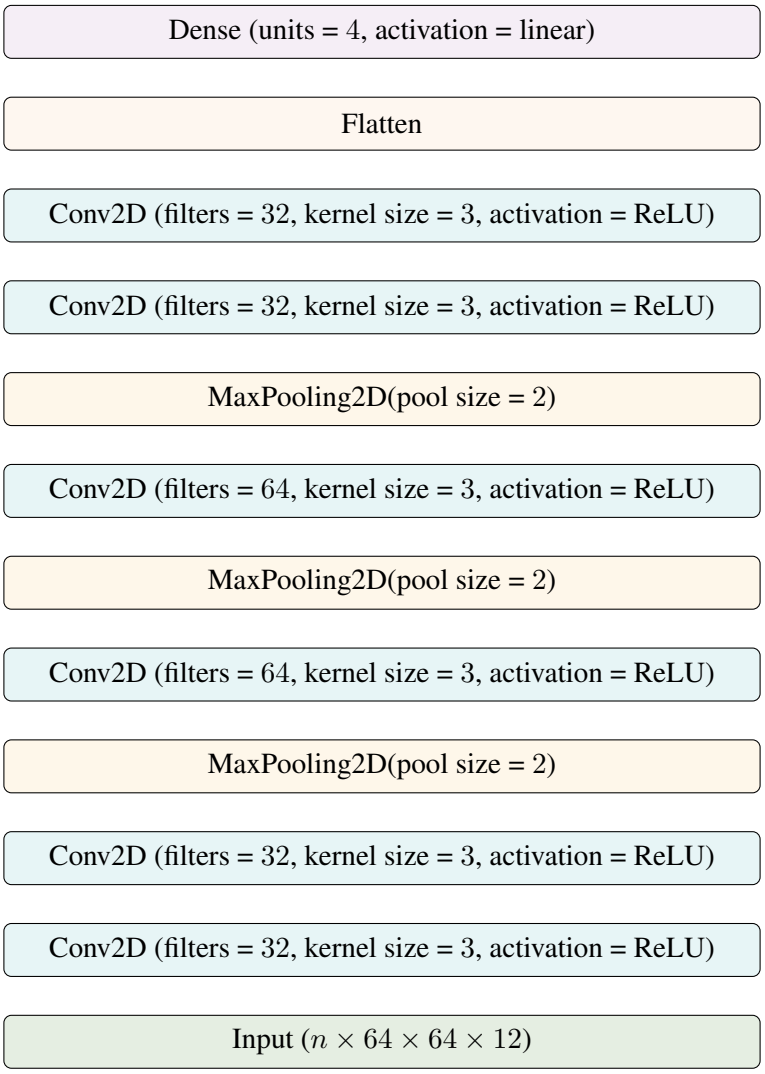


Figure 2: Architecture of the network.

Evaluation

We used explained variance as the evaluation function for our network and performed 10-fold cross-validation using scikit-learn. Explained variance did not give us satisfactory results and in the end we used real-life validation for each model to measure its performance. In some cases we combined manual (human-controlled) throttle management and model controlled steering in order to evaluate steering output and throttle output separately. This turned out to be a good method, because in some cases throttle management was completely off but controlling the steering worked seemingly well.

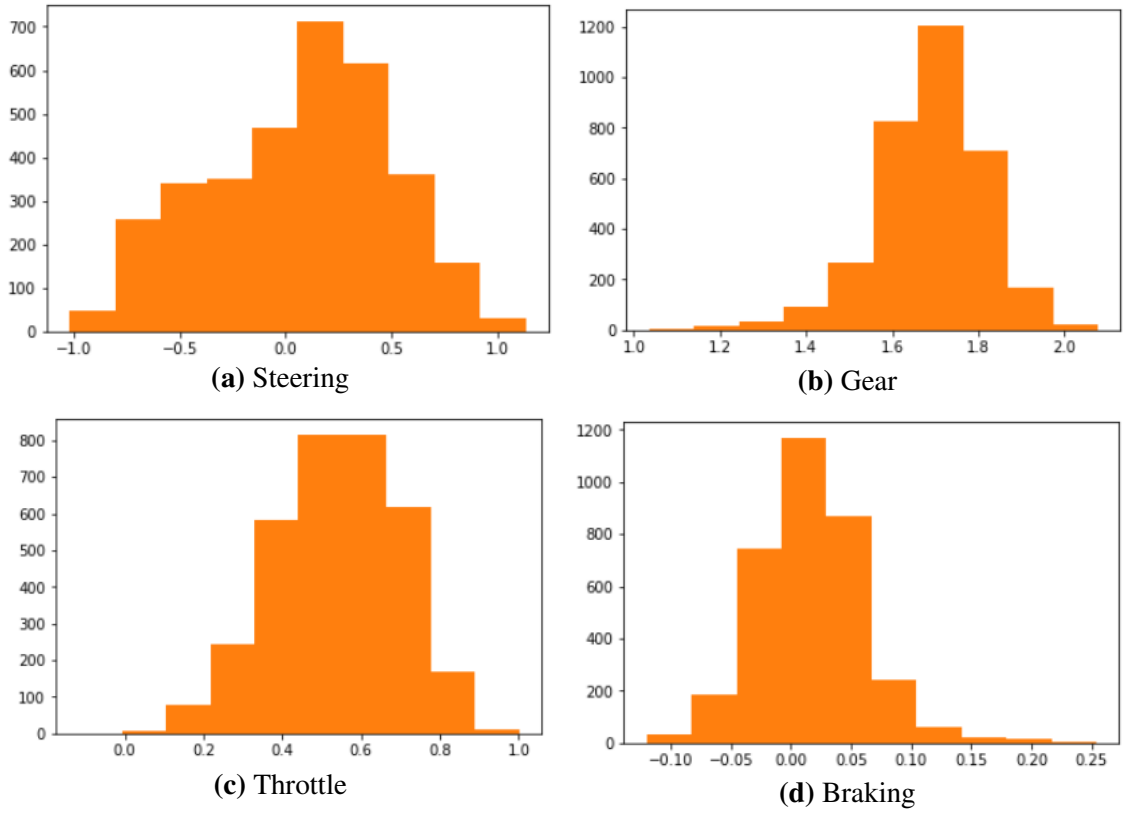
Inference

To test trained models, video stream from the applicable RC car was pointed to a TCP endpoint at 5 frames per second. Using socket programming, we read the data, performed the inference and sent the inferred commands back to the RC car on another TCP endpoint.

4 Results

We implemented various types of network architectures and used real life evaluation on all of them. The best architecture turned out to be simple convolutional model connected to fully connected network.

The distributions of different commands predicted by the network can be seen in Figure 3. We can see that the steering is more towards right than left, but this makes sense as the track is laid out clockwise.



(a) Steering **(b)** Gear

(c) Throttle **(d)** Braking

Figure 3: Distributions of training labels.

Gear values were always predicted to be over 1, which means that the network did not learn to drive in reverse. We saw this problem also when we let the model to control the car – the car got stuck at some point by trying to drive forward while facing a wall. Before forwarding the predicted gear value to the car, it is rounded so that everything above 1.4 ends up 2 (drive) and below 0.4 ends up as 0 (reverse).

Predicted throttle values also fall within a sensible range, being mostly somewhere between 0.4 and 0.8. Throttle predictions also needed some special treatment before forwarding to the car. The reason behind this was mostly the fact that there is a threshold throttle value that needs to be exceeded in order for the car to take off. We first clipped the predicted values so that they would fall within the actual range $[0, 1]$ – values outside the range were changed to the closest endpoint of the range. Next, we used Equation 1 using the clipped values to obtain the final throttle command sent to the car. This ensures that the throttle received by the car is never under 0.3.

$$\text{throttle} = 0.3 + 0.6 \cdot (1 - \exp(-2.5 \cdot \text{throttle}_{\text{clipped}})) \quad (1)$$

Braking on these cars works only for the rear axle because these RC cars do not have dedicated brakes and use engine to brake. This resulted in unexpected oversteering in case of braking and turning at the same time. Brushed electric motors have quite string engine brake which resulted very little need to use brake at all. In general case brake overrides throttle input, but in our case sometimes network predicted 0.7 throttle and 0.02 brake which resulted idling and braking due to engine brake, therefore we ignored the brake command if the strength did not reach the threshold of 0.6.

5 Summary and Further Development

In this work we created a neural network that can control a remote control car in Tartu SPARK Democenter based on live feed from a camera attached to the hood of the car. The network consisted of a few sets of convolutional layers with pooling and lastly a fully connected layer. The network was trained using behavioral cloning – the training data consisted of videos and control commands (as labels) that were given to the car by a human driver as a reaction to the visual. The network controlled the car successfully follows the track in most cases, but failed to resolve a situation where it had hit a wall face-first, implying that it did not learn to use reverse gear when the car got stuck, or that it got stuck in the first place.

RCSnail has plans to include accelerometer sensors on the car, which would allow also using reinforcement learning. Distance traveled would count as positive reward, while crashing the car against the wall would count as negative reward or penalty. Initial training could be done with behavioral cloning and fine-tuning with reinforcement learning.

Acknowledgements

We would like to thank the founder of RCSnail Rainer Paat for being most helpful by helping us with connection issues between the model and cars. We would also like to thank Tambet Matiisen and Ardi Tampuu for giving us advice on the modeling part of the work.

References

[1] Stanford University. Convolutional Neural Networks [Online course]. Retrieved from http://cs231n.github.io/convolutional-networks/ (2018, January 11).