Technion
Israel Institute
of Technology

CGGC *ISL*

Formula Technion 2018
Self-Driving Vehicle Algorithm

Project Report

Dean Zadok, Tom Hirshberg, Amir Biran

Supervised by

Dr. Kira Radinsky (eBay, Technion), Dr. Ashish Kapoor (Microsoft Research)
Mr. Boaz Sternfeld (Technion)

This project was carried out under the supervision of Dr. Kira Radinsky (eBay, Technion), Dr. Ashish Kapoor (Microsoft Research), and Mr. Boaz Sternfeld (Technion), in the Faculty of Computer Science.
We would like to thank ISL, managed by Aram Movsisian, and CGGC labs for supporting us with the working space and hardware with which the project was realised.

Contents

1 Abstract

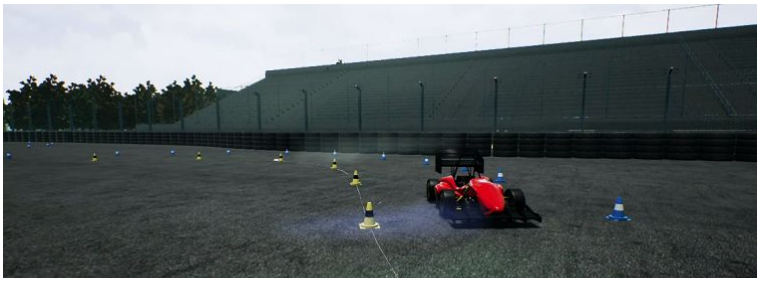
"Formula Technion" is a cross-faculty endeavour that has been practiced and run (mainly by BSc. students) since 2012, with the goal being the construction of a fully functioning race car from scratch within only one academic year, every year.
The "Formula student association" consists of over 600 universities from around the world. Every year the association organizes competitions in various countries, in which teams from universities from around the world compete. Through recent years, the Technion project has seen successes in competitions in Italy, Czech Republic, and Germany.
The rise of self-driving cars (led by the deep-learning revolution in computer vision and inference for decision problems) has had its effect on the competitions: 2017 was the first year in which FSG - the German competition - had opened its doors to student-made self driving cars. In the summer of 2017 at the Technion, some of the students who have participated in the previous years' competition have decided to open a parallel team to the one dealing with the human-driven car. The new team was set to tackle the ambitious venture of **creating the first self-driving car that's designed, built and programmed solely in Israel**, and specifically - by students alone, including the tasks of acquiring proper funding and sponsors, student recruitment, and finding mentors. The goal was set - to send a fully functioning self-driving car to Germany by July 2018, knowing that a successful run of the project could potentially incur the participation of students in projects relating to the rising field of autonomous vehicles in years to come.
This Project report deals with the work of the algorithm team, consisting of computer science students, tasked with **the design and implementation of the inference algorithm for the self-driving vehicle**. The algorithm will function as a predictive model for steering in real-time, in a reliable and smooth manner.

2 Introduction

The objective of the project was to create an inference algorithm for steering a formula student car in the setting of the 2018 international self-driving formula student competition in Germany. The competition takes place on a professional Formula racetrack and two lines of traffic cones (blue and yellow cones) define the track. Note that the track is unknown before the race, so our algorithm's goal is to be accurate on an unfamiliar track. The race scene also includes ad signs, tires, grandstands, fences etc.

Formula student driverless racetrack

In order to train our algorithm, we had to have a training environment. Due to safety constraints, lack of accessible closed road track and fuel saving, we used a simulation for building our training environment, recording data and training our algorithm with it. We simulated the dynamic model of our car, different and varied racetrack scenes and weather conditions. Since we couldn't train on real tracks at all, it was important to create the scene to be as similar to the real environment as possible. So, the transition from the simulation to the real world will be more smooth.



Formula student driverless racetrack simulation

The main goal of our algorithm is to predict the next state of the car in real time, using recorded images and the last state of the car. A state is defined by: steering angle, speed, throttle and

brakes. We decided to focus only on predicting the next steering angle, while other parameters of the state are derived from it and calculated due to the car's mechanical abilities.
Our algorithm is based on deep learning methods. Through the project, we tried different approaches such as reinforcement learning and imitation learning. For each method, we tested several algorithms, changes within those algorithms and tuned hyperparameters, as will be discussed later in detail.

One of our experiments in the Technion

The last part of our road to have a driverless formula car is to transfer the algorithm to a specified hardware and test our algorithm with the whole systems of the real car.
After that we are going to present our formula car to our contributors by demonstrating our abilities in a well prepared event. The next mission is to be the first Israeli team to participate in formula student driverless competitions in Germany next year.

Previous Work

The industry of autonomous driving had changed a lot over the past few years. The improvements in computer vision applications using deep learning tools led us to move on from classic algorithms into faster and more efficient implementations, for models which are more accurate in their feature extraction capabilities.

DARPA challenge[1]

One of the first important steps in autonomous driving history was DARPA challenge. Started in 2004, the goal was to develop an autonomous vehicle that is capable of navigating desert trails and roads at high speeds.
A certain challenge in DARPA was the urban round, which included driving 97 km through an urban environment with the goal of properly interacting with other vehicles on the road. The

main focus was on a team with a vehicle named “Boss”. This team invented an algorithm based on the mechanism control of the car that helped them win the challenge.

“Boss”, the vehicle that won the urban challenge

Their algorithm was based on trajectory planning. By solving past challenges in computer vision, perception and distance measurements, they were able to transform the dynamic model of their car into an algorithm that instructs how much to accelerate, turn or slow-down and for how long.

Previous year

In August 2017, the first driverless formula student competition took place in Hockenheim, Germany. Only 15 teams qualified for the competition and only 4 of them passed the regulations. There are aspects in which a racing autonomous challenge can be harder than the urban challenge. Controlling the behaviour of the car, while maintaining a steady perception of the space, is harder than it looks. Since it’s the first competition, most of the teams took the conservative approach by conducting a safe algorithm to get through the race track carefully and steadily.

In this context, using deep learning in order to perform decision-making is quite risky, the car should be reliable enough to hold on to the track while taking sharp turns. Some of the teams tried that approach and none of them succeeded in getting a fair amount of reliability.

The technology that won this competition is called SLAM, simultaneous localization and mapping. In this method, the vehicle drives very slowly for the first lap. This is done in order to map the track accurately, so in the next time the car drives through the track it will be able to drive faster by maneuvering between obstacles without using visual sensors.

The winner of the first competition, AMZ Zurich, with their SLAM technology[2]

And yet, none of these methods have the potential to perform the decision-making in a fast and accurate manner like deep learning, which is exactly what we attempted to do this year.

3 Training Environment

Introduction

Training a deep learning algorithm to perform autonomous driving requires hours of recorded driving data. Since it was not possible to record such amounts of data with the real car, we needed a simulation that could demonstrate a realistic formula student environment.
We started by taking an open source simulation that was created by Microsoft, called Airsim[3]. The simulation is based on Unreal Engine 4 (UE4), which is commonly used for creating computer games.
At first, the simulation's main objective was to train drones to perform autonomous flying. After that, they started to program various cars into the simulation in order to perform autonomous driving. So, we implemented our own vehicle model and environment into the simulation.

Graphic Car Model

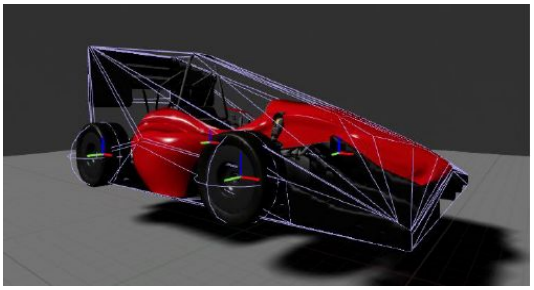
In order to simulate the real physics of the car, we used a graphic model that designed by the real car's components. We converted the model from SolidWorks into Rhino, than into 3DS-Max and at last, imported it into UE4. The procedure of importing models into UE4 removes materials and textures. Because of that, we also adjusted the mass of the components, materials and textures.
Using this graphic model enabled us to place a camera in the same position as the real car's camera, and also to show the right point of view of the camera, considering the car's measures. Moreover, the material of the car simulated the reflection of the sunlight on the camera and illustrate real life conditions.

The actual car

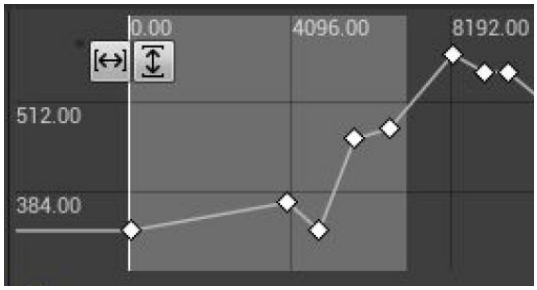
Simulated car model in UE4

Dynamic Car Model

The default dynamic model in the simulation was based on a vehicle which is heavier and taller than our vehicle. Also, the components themselves are different than our components.



Car physics model



Torque graph

Therefore, in order to complete our graphic model, we needed to change parameters related to the behaviour of the vehicle. Along with the students who built the real car, we investigated the dynamic model in the simulation. Parameters such as engine and gear properties, body weight and dimensions, converted to resemble the real car's behaviour on the road.

The results of changing these parameters resulted not only in a more realistic drive, but also in better grip, steady acceleration and faster response.

Landscape and Features

As mentioned before, the Formula Student competitions take place at professional Formula race tracks. In particular, the driverless competition's track is built on a professional track zone and is marked by two lines of cones.

Scene

Simulating a Formula race track, we used an already-prepared scene. We added special asphalt to the scene, which we used to place driverless tracks on. Besides cones and asphalt road, the scene also consists of grandstands, trees, signs and fences.

A part of the scene

Traffic cones

The traffic cones which are being used in Formula student competitions are specific cones, according to the race's regulations.

To perform a realistic hit result at cones, we imitate the cone's realistic physics. We modeled the specific cones according to the regulations. We also ordered some of the actual cones to build precise models, by measuring their dimensions, weights and adjusting colors accordingly.

Formula student traffic cones

Track creation tool (spline)

When using learning algorithms that are based on visual recognition, it is important to use various tracks consisting the data distribution, in order to reduce overfitting. We built a track creation tool in order to build tracks in a more simple, easy and fast way.

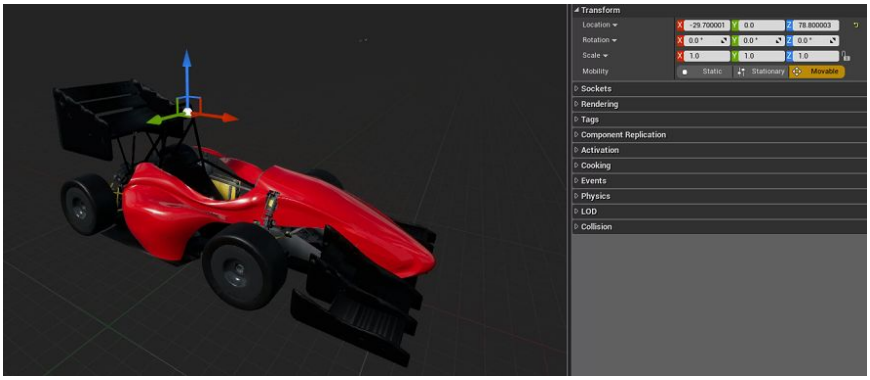
Track creation tool

By only clicking on a point on the road, we can edit the track as follows: The points set the right line of the track (yellow cones), The tool then automatically builds a parallel left line (blue cones) at a constant distance. Moreover, the tool automatically places each cone at each line in a constant distance from the previous one**.**

Camera

Since the goal was to transfer the algorithm from the simulation to the real car, we needed to simulate the exact image the vehicle's camera will capture.

In order to simulate this, we fixed the camera's place and properties: we measured the exact location of the camera on the real car and placed it in the simulation accordingly. Also, we changed the camera's field of view and lens light sensitivity similarly to the real camera.



Changing the location of the camera

Even though we fixed most of the camera's parameters in the simulation, some of them were not available for adjusting, thus they did not exactly resemble the real camera's parameters. The exposure measurement of camera depends on light intensity and our real camera exposure can be fixed only mechanically. So, in the simulation we didn't change that parameter, but in real environment we adjusted it according to specific lighting.

4 Training the Model

Introduction

At the beginning of our project, we were unsure about the proper flow of training we should use, and so we attempted two different methods - imitation learning using recorded data, and reinforcement learning using DQN[4] approach. After the unsuccessful attempts at straight-up reinforcement learning and imitation learning, we decided on changing our strategy to a two step procedure:

1. Training a model using imitation learning
2. Performing transfer learning to a suitable network for reinforcement learning (fine-tuning step)

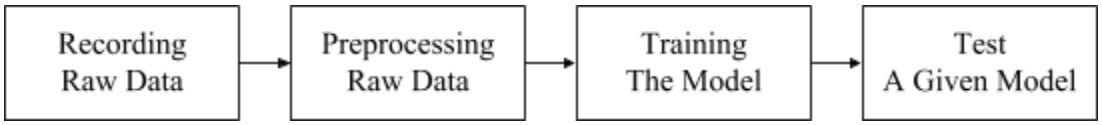
This change in approach had a positive impact on results. Although it is noteworthy that our final model - which performed best - was trained with imitation learning only.

End-To-End Steering Inference

End-to-end inference using DNNs is the usage of a trained neural network model to perform inference over given inputs, with no intermediate steps (i.e. feature-engineering) from the input. Promising results of using this approach in recent years had encouraged the industry to use end-to-end approach for many applications. For example, Nvidia has made efforts to create a comprehensive system for autonomous driving - this system is mainly based on a set of neural networks specially designed for the purpose. One of the neural networks designed, labeled "PilotNet"[5], was created and fine-tuned for the inference of steering angles given sensor data. In our project, after a period of time trying different options for architectures to become our feature extractors (conventional CNNs, ResNet50 etc..), we settled on using "PilotNet". This Architecture has been used and altered extensively during the project, undergoing a variety of modifications: the addition of Batch Normalization and Dropout, non-trivial activations, several inputs along different layers of the net, choosing a proper ROI for the input images, data-distribution engineering, consecutive-frames as input etc.
The different trials and errors leading up to our well-performing end model will be covered in the next sub-chapters of the report.

Phase 1: Imitation Learning

The imitation learning procedure is divided into 4 main steps:



1. **Recording raw data** using AirSim and specially designed maps. Each recording session creates a folder which will store the relevant information of the training samples: The car-state data (including the steering angle) with it's timestamp is stored alongside the corresponding images.

2. **Preprocessing raw data** - the user chooses raw data folders (recording sessions) which will be used to pre-process. Using a designated script, the images and car-state data are saved into a ".h5" file. For perspective, on our final training, the pre-processed files generated took about 20 Gigabytes of disk space, and represented 4 hours of recordings.

3. **Training the model -** Based on shuffled pre-processed data, batches are produced and trained with the chosen model architecture. In the training procedure, we use a parameter called "patience", to base a stopping mechanism. This hyperparameter points on when to stop the training procedure - when the validation loss had not improved within a certain number of epochs. Whenever an epoch ends with the validation loss being smaller than the previously seen, the current model is saved for future testing.

4. **Test a given model** on a desired map in AirSim - a saved model is used to infer the steering in real-time while maneuvering through a given track.
   The speed of the car in the simulation, while testing a model, is managed using the formula: $car.throttle = x - (y * |z|)$ where $x, y \in (0, 1)$ are hyperparameters which are fine tuned for each model (each model handles driving in different speeds), and $z \in [-1, 1]$ is the scaled neural net prediction. This determines the throttle based on the steering angle. A sharper angle would lead to a softer throttle push (and in turn, to slowing down), and vice versa.
   It's important to note that we classified different models by time on track, not by their loss on test set, as will be discussed later in detail.

Architecture

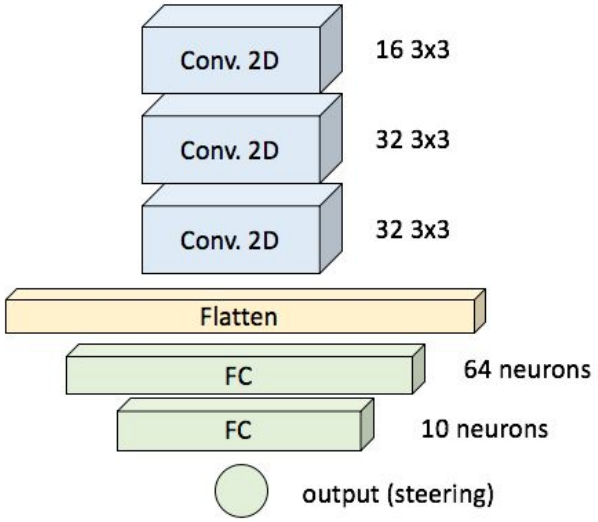
Initial Attempts

At the beginning of our project, we tried to investigate convolutional neural networks and how changes we do can affect the results.

We started from a simple CNN, composed of 3 convolutional layers and 2 fully connected layers. We got pretty good results, but noticed that taking a well known model that already proved itself is a better option.

Like in most of deep learning related use-cases, when using a high number of parameters, we received a massive overfitting on the train set.

Therefore, we realised that using deeper networks like ResNet would be an overkill to our problem. So, after many researches, we found PilotNet, a network architecture suited for our cause.

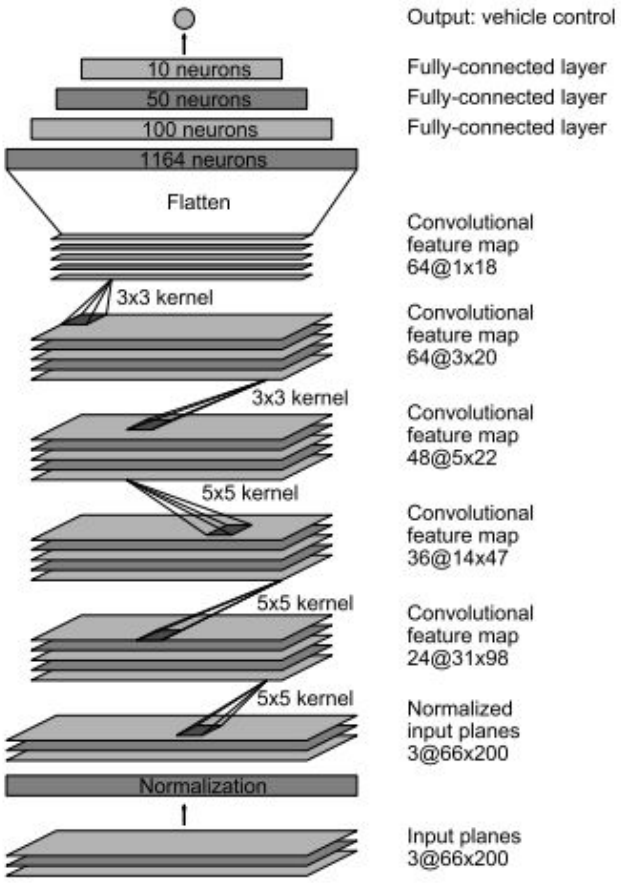


PilotNet

PilotNet, a neural net designed by Nvidia for steering-angle inference had come to our attention after reading the accompanying paper[5]. We then decided that working with this architecture as a basis for improvement would prove itself down the road. Our initial results for this architecture were somewhat unsatisfying, but later we managed to change parts in this architecture and received better results.

The attached diagram is the original PilotNet architecture devised by Bojarski et al.[5] as a part of Nvidia's work in the field. It consists of a normalization layer, several convolutions which function as feature extractors and three fully connected layers which function as the selection layers. This network has about 27 million connections and 250 thousand parameters. A necessary change we made in the network was the addition of a **TanH activation at the end of the network**. This activation generated the required steering output for AirSim simulation framework on the scale between (-1) and (1).

Another change was the addition of **previous car state as input**. Our assumption was that the network could utilize information regarding the car's speed, throttle and steering angle in the previous state, in order to better infer the next state's steering. That is in a similar manner to how humans consider their current momentum, when choosing an immediate action for trajectory planning. This model had a hard time dealing with curves, leaving us to consider steps of modification to the network.



Grid-Search on GCP

We have managed to attain a sufficient budget for use on google cloud platform for performing high computing power demanding tasks. We could now use powerful Tesla K80 GPUs. Thus, we began working on heavy processes of grid-search, further fine-tuning the hyperparameters for our training procedure. We tried different values for the following:

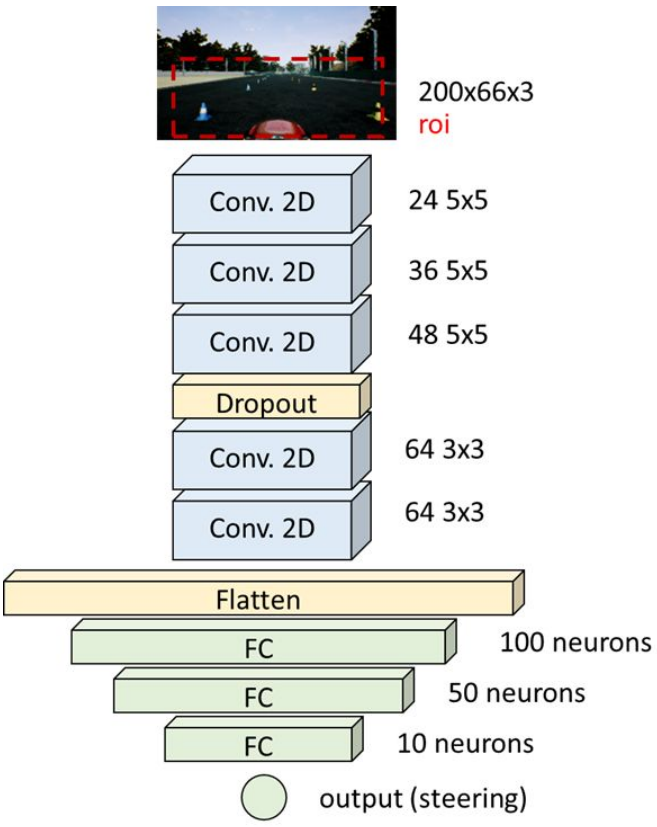
- Regularization strength
- Learning rate
- Presence of Batch-Norm at different sets of layers
- Dropout rate

- Weight initializer

Another attempt suggested by our mentors was to change the input from just one image to two or four consecutive frames. The consecutive frames did not produce successful models in our case. Grid search had eventually confirmed that indeed we were using good hyperparameters in the first place, leading us to search for other paths.

Addition of Dropout and Leaky-ReLU

At this point, the output of the network was a number between (-1) and (1) signifying the steering angle, Our analysis of the performance of the car showed a lack of ability to deal mainly with left turns. This implied that negative values are vanishing during the training procedure, most likely due to the presence of dead ReLUs. So, an attempt of training was set using Leaky-Relu as the activation. This allowed for non-positive value propagation to only slightly hinder the training rate, and to avoid an abundance of dead ReLUs in the net. This addition, alongside the addition of a Dropout layer, produced a great improvement, to the point of having the car drive on an unseen track (although being one without sharp turns) for up to 3 minutes. This model too had many drawbacks, e.g. a lack of ability to deal with shadows of any kind on the track, plus still having a difficulty in dealing with sharp turns.



Removing Car State From the Input

Our initial assumption was that increasing the amount of inputs to the model, will improve the accuracy of it. After experiments with and without the car state as input, we discovered a regularization in the behaviour of the driving style. For example, after adding the drunk-style driving (will be described in the next section), the model showed better driving without the car state (smoother).

Adaptation To Other Frameworks

There are some layers that are cannot be used in different frameworks. These layers are not basic ones, such as LeakyReLU. Our goal is to implement our algorithm on embedded architectures (will be described widely later). After “simplifying” our net, we should be able to transfer the model into other frameworks, such as Caffe and TensorRT, correctly without changes.

By changing the range of the steering angle to be between 0 and 1, instead of between (-1) and (1), we avoided using LeakyReLU (which is an unsupported layer in Caffe) and used regular ReLU instead.
We also switched our output activation from TanH to Sigmoid, thus complying with the new scale. The truth labels were normalized to fit this scale.

Data

Not only the model's architecture, but also the training-data influences the quality and prediction accuracy of a deep learning model. As the project progressed, we noticed just how much the nature of the data had a major influence on the algorithm's accuracy.
As mentioned at section 2 of this report, we trained our model on a virtual environment, from which we collected our data.
We examined the quality and effect of different datasets in two aspects: the way we collect data, and the data distribution.

Collecting the Data

The input of our imitation learning algorithm is composed of images from the point of view of the car's camera. Therefore, we placed the camera in the simulation at the exact place as the real car's camera, as mentioned before. In addition, we adjusted the virtual camera parameters to resemble the real camera's parameters. By doing that, we narrowed the gap between the virtual and real environment.
Our first attempts at collecting the data were performed by steering the car in the simulation with keyboard keys (right and left arrows), which meant that the steering values were binary {-1,1}. This resulted in a sharp, coarse and disrupted steering.
To improve the results, we steered the car in the simulation with a playstation 4 joystick (DualShock). With the joystick, we were able to keep on continuous changes in the steering angle. However, it was hard to record a precise and smooth driving style. The difficulty was derived from our inability to maintain a stable grip of the joystick at a certain stance.
Our last attempt (for now) to improve the results, was steering the car with a steering wheel (Logitech g920). We adjusted the steering wheel's parameters to the real car's steering wheel dynamics. The results improved significantly. First, our recording ability of the data was much more precise, and second, the algorithm's output (steering value) was much smoother and behaved in a more continuous manner.
Due to availability constraints, we recorded the driving of only several drivers. None of them was a professional driver. Nevertheless, the recorded sessions were precise and eventually proved to be sufficient for our cause.

Playstation 4 joystick *Logitech g920 steering wheel*

Data Distribution

We used several methods to vary the data. A major part of our experiments was to find the right balance between the percentages of each of the methods' prevalence in the data distribution. We also needed to discover what is the sufficient amount of recorded driving time, to achieve a high accuracy of the driving algorithm, while not exceeding recording-time and memory constraints. The methods we used:

Track type

We built several virtual different tracks. Almost half of the recorded data was taken from drivings on tracks which were nearly straight. For the other recordings, we used tracks in which the number of left turns and the number of right turns were balanced. As we have observed, in previous Formula Student competitions most of the tracks were built as being mostly straight lines with some turns, and therefore we chose the tracks as mentioned.

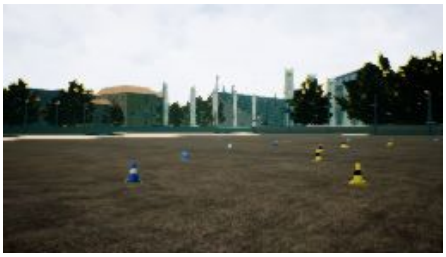
Drunk-style driving

Driving constantly sharply from one edge of the road to the other. This kind of driving style enabled us to add an unusual type of driving. Without this method, our recordings were "safe and steady", keeping close to the center of the road, not able to learn extreme situations.
This method served the purpose of adding data samples of the car drifting to the edge of the road and returning back to the center.

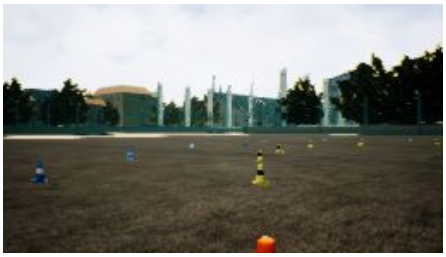
Shifted images

"Shifted" is a method of altering the camera's position to the right or the left of the car, so that it can record images in extreme conditions, simulating driving close to traffic cones at one side. In order to simulate driving back to the center from those extreme situations, we post-processed the recorded angle of the steering accordingly. Using this method improved our ability to face difficult situations that the car sometimes get into while self-driving, and get back to the center

of the track, in addition to the drunk-style driving method. It has made a great difference in the ability of our model to stay on track.
This idea is based on the article: Learning a CNN-based End-to-End Controller for a Formula race car[6].
The two images below show an example of the use of this method. The left image shows a standard position of the camera, and the right shows a tilted image to the right (150 cm).



Standard image



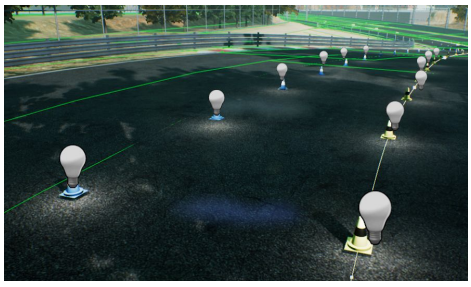
Shifted image

Distance between cones

The Formula Student competitions regulations determine the distance between cones at each side of the track to be between 300 to 500 centimeters. Simulating different distances at our training procedure enabled our algorithm to learn how to deal with various distances between the cones. We observed that in most of the competitions tracks the cones were at a distance of approximately 400 centimeters. Therefore, most of our data was recorded with this distance.

Glowing traffic cones

At first, our algorithm was unable to deal with shaded tracks. We assumed that the algorithm could not differ between cones and small light spots on the road in shady conditions. To overcome this problem we added a glowing halo around each cone located in a shady environment. The results did not improve at all.



Glowing traffic cones

Environment conditions

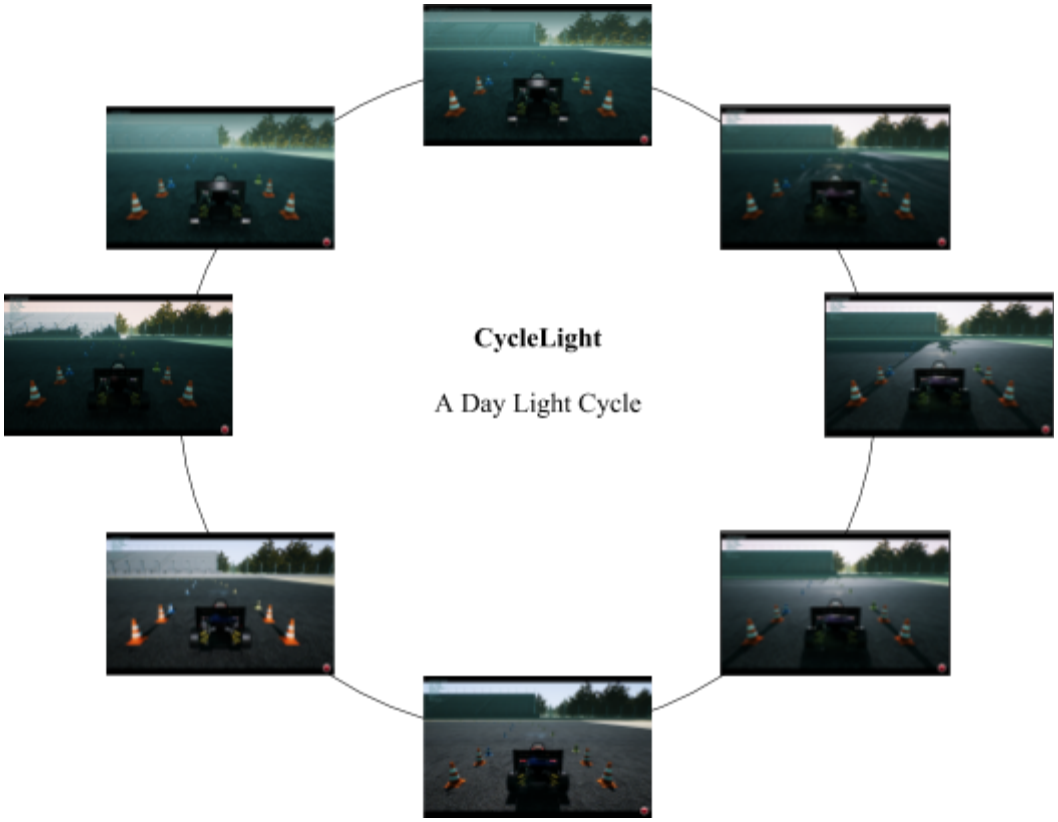
The time of day and weather in which the car will drive at is unknown. Therefore, we needed to simulate different conditions, so that the algorithm would be able to deal with as many situations as possible.

- **Temperature**: Changing the temperature in the simulation affected the RGB channels of the recorded images. We suppose that those changes helped us to overcome the difficulty in matching the real car's camera parameters.

- **Sun's intensity**: This is a changing parameter, which depends on each day's weather. Training the algorithm with different intensities enables it to deal with those changes.

- **Cloud opacity**: The more opaque the clouds are, the more they reflect the sun's light. This affects the glare on the camera's lense. In order to overcome this issue we trained our model for different cloud opacities.

- **CycleLight**: A novel method for introducing variation in recorded driving data on a simulated environment. This method provided a great means for generalization over weather and lighting conditions.
  CycleLight is an animation of a day light cycle in a changeable, potentially very short, period of time. Shortening this period of time enabled us to collect the data more effectively. Instead of recording a whole day in different conditions, we were able to record a whole day's conditions of driving in only a few minutes.
  Using this feature in our recordings, the model had become very robust to changes in shadowing conditions, daytime vs nighttime, ad signs, tire walls and various objects, and generally showed a staggering improvement in driving smoothness. It also helped in overcoming a shortage of data.

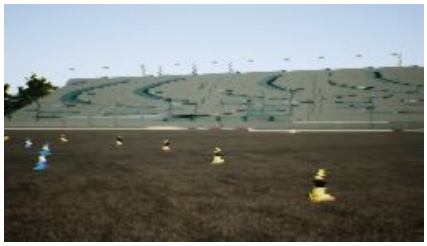


Noise

Adding noise to recorded images, with the goal of simulating camera sensor noise and other such factors. We chose to add the following disruptions:

- **Horizontal lines contribution**: This determines blend ratio of noise pixel with image pixel.

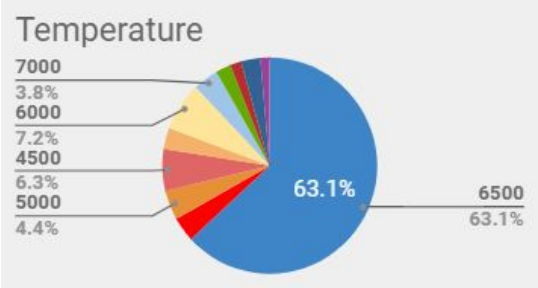
- **Horizontal distortion**: This also determines blend ratio of noise pixel with image pixel, but differently as shown in the images.

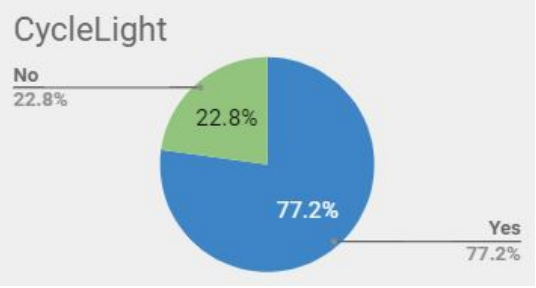


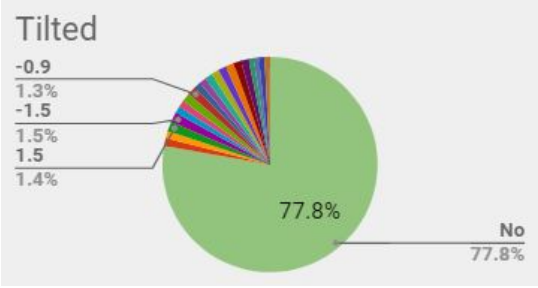
- **Light reflection on lense**: Adding light reflection on the lense in order to train the model to deal with glare.

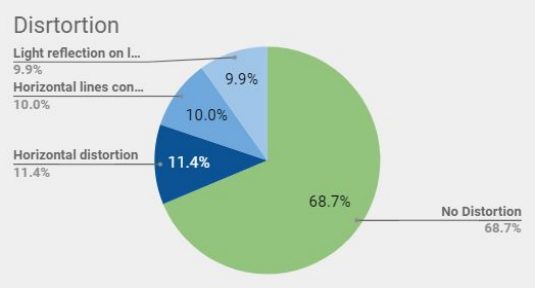
The final data distribution

The data distribution (that produced the best trained model), after many experiments, consists of approximately 4 hours of driving and 189,093 images.
The final data distribution is shown below:



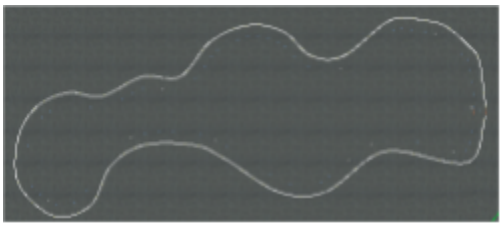




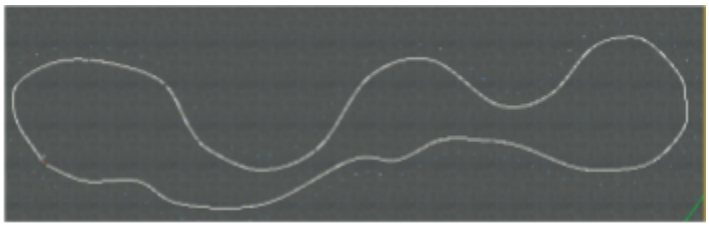


Track maps:

The tracks “spline_all_road” and “spline_all_road2” are long and are nearly straight tracks. All of the other tracks that we built for recording drivings are shown below. These tracks are much more curved and short.

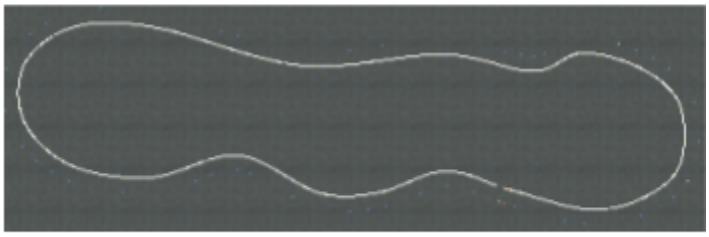


IM_MAP_1

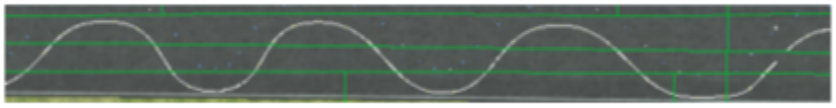


IM_MAP_2

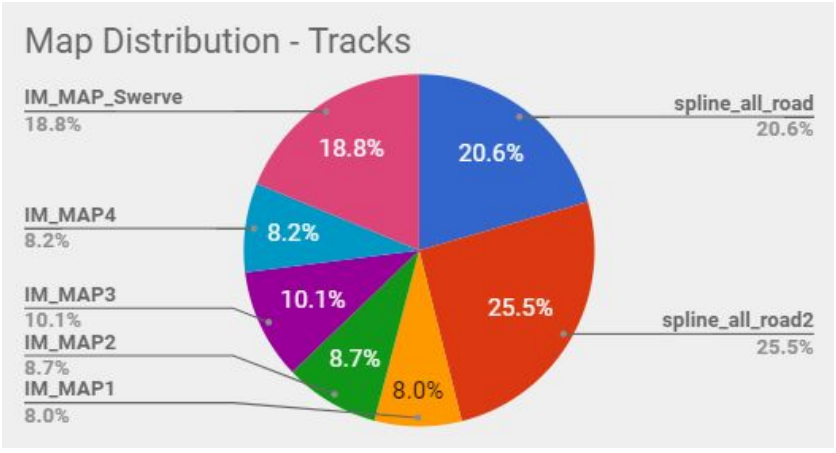
IM_MAP_3

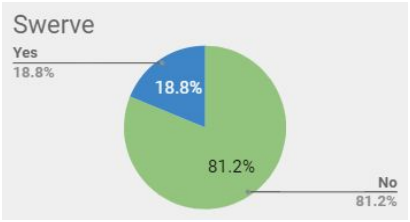


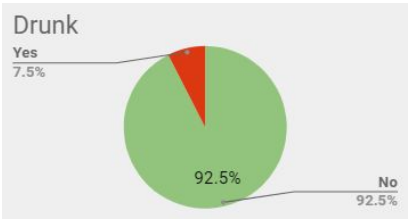
IM_MAP_4



IM_MAP_Swerve





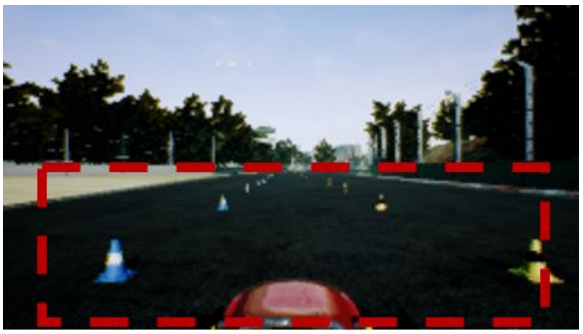


Augmentation

Region of interest (ROI)

Past papers, regarding autonomous driving using deep learning, discovered that only part of the picture is relevant to make the decision of the driving. In our picture, the part that is above the road can confuse the model with unimportant objects, such as flying birds or different shapes of clouds. So, we removed these unimportant parts and kept only the relevant region that describe our situation.

This also conserve resources by eliminating irrelevant parts of the network.



Shown in red is our region of interest as a rectangle of 200 by 66 pixels

Brightness

One of the main things to notice when transitioning the algorithm to receive images from real cameras as input, is the difference in light exposure. Changing the simulated camera to behave exactly like the real one is close to impossible at our position and we needed another way to cover light exposure control, this is why we had duplicated the picture in the pre-processing phase in variations of up to 40% of the current brightness.

Zero drop percentage

To have a variety of steering samples in a batch and to avoid overfitting on straight driving, we used zero drop percentage method. In this method, we drop a certain portion of the samples with zero value of steering.

Since we changed the model of the car from the original one in the simulation, we had to change the implementation of this method function. We change it to remove roughly half of the 0 steering angle samples while training the net. This proved to be useful, since the car then became more robust to sharp turns.

Phase 2: Reinforcement Learning

In reinforcement learning, the agent learns how to behave in an environment by performing actions (some of which being randomized) and improves by analyzing the results and optimizing the model accordingly. In our case the agent is the car, the environment is the track built of traffic cones and the actions and results are steering angles.
After examining different algorithms, we chose to use DQN[4] (deep Q-learning) for our task. Our goal, which is to drive on a track bounded by cones, has attributes that are consistent in a way to some of Atari games attributes. We chose the DQN algorithm because it is commonly used in solving Atari games. It uses replay memory and sample the next state from experience by Q-learning updates. The score an action gets is set by a reward function. Figuring out the best reward function is a difficult task.

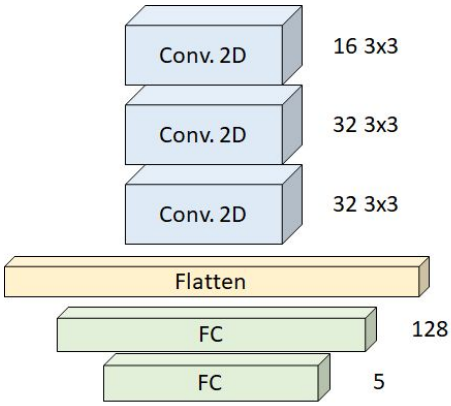
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

The full pseudocode from DeepMind's paper on DQN

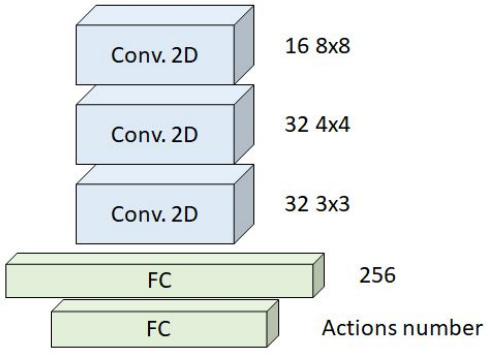
With a successful deployment of reinforcement learning, the algorithm can potentially make driving decisions better than a human driver. However, this procedure takes a lot of time because the agent has to observe many environment possibilities with a rather slow training procedure, in our case.

Our first attempts to use DQN were with CNTK framework, inspired by AirSim's code[7]. Later on, we changed to use Keras framework with Tensorflow backend, inspired by AirSim's cookbook.

The architecture we used in both frameworks was based on three convolutional layers, and two fully-connected layers, with some changes as follows:



Keras implementation

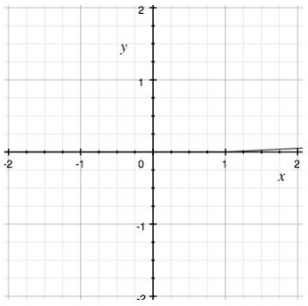


CNTK implementation

Our main work was to try and test different reward functions. For each reward function we also supplied a "collision reward", in which we gave a bad score for bumping into objects (cones for example). We tested the following reward functions:

- **Reward by speed:**
  The reward function is defined by:

$$f(x) = \max\{\tanh\left(\frac{x - speed_{\min}}{speed_{max} - speed_{\min}}\right) * const, 0\}$$

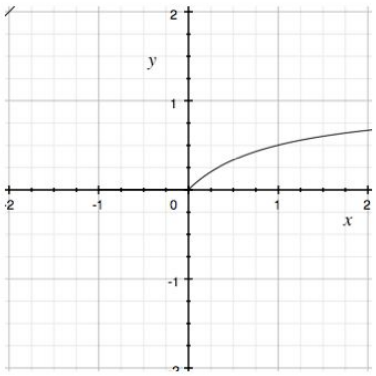


x, in meters per second, is the car's speed

  The argument *speed max* was fixed by the car's abilities, *speed min* was fixed to 1 and *const* is a hyperparameter.

- **Reward by time on track and speed:**
  The reward function for speed is defined as above and the function for time is defined by:

$$f(x) = \frac{x}{x+1} * const$$

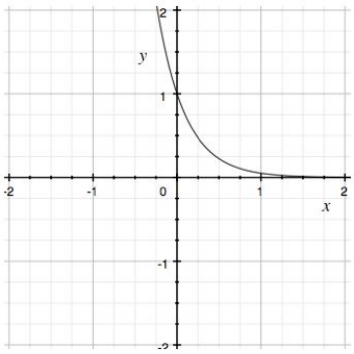


x, in minutes, is the time on the track

The argument *const* is a hyperparameter.

- **Reward by distance from the center of the track and speed:**
  The reward function for speed is defined as above. We tried two different functions for the distance reward. The first took into account the exact center of the track and the second considered an epsilon-distance from the center. The function is defined by:

$$f(x) = \frac{1}{e^{3x}}$$



x, in meters, is the minimal distance from the center of the track

The improvements of the model were limited and somewhat not satisfying. Perhaps the reason could be that we didn't have more than just a few days to try out each reward function. However, the best results were produced by using reward by epsilon-distance from the center of the track. Therefore, and also because of promising results from our imitation learning experiments, we decided to try Transfer Learning to improve our imitation learning model.

Transfer Learning

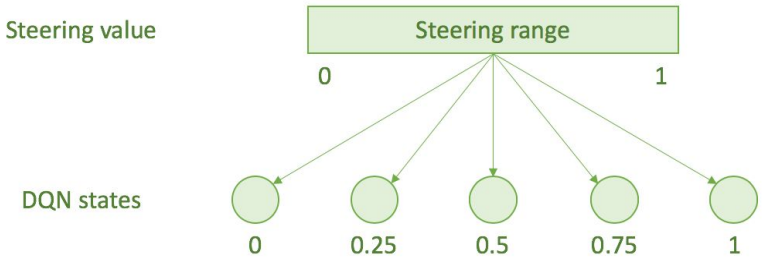
Convergence with reinforcement learning takes a lot of time, even months. But, if we start the process from a point at which the algorithm already knows how to stay on the track, this process of learning from reward can further enhance the algorithm's abilities to drive accurately, even better than human recordings.

The underlying assumption of this method, is that when we load our model from the imitation phase, we're loading a function from a state that is closer to convergence than a random state. This is why reinforcement learning phase should take much less time than before, days instead of months.

From continuous to discrete space action

Our attempts in this section were inspired by the AirSim's distributed reinforcement learning tutorial[8].

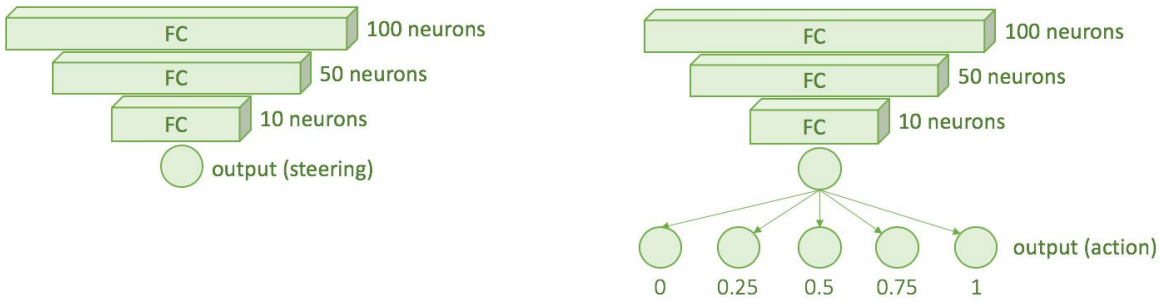
Our first problem was that our model predicted steering as a continuous value, as opposed to the DQN algorithm that predicts a state from a finite set of states. This meant that we needed to transform our model from predicting a single value to predicting probabilities over a set of states. First, we needed to convert the steering prediction to a set of states. From looking at recent results, we decided to start with 5 states.



Conversion from continuous to discrete action space

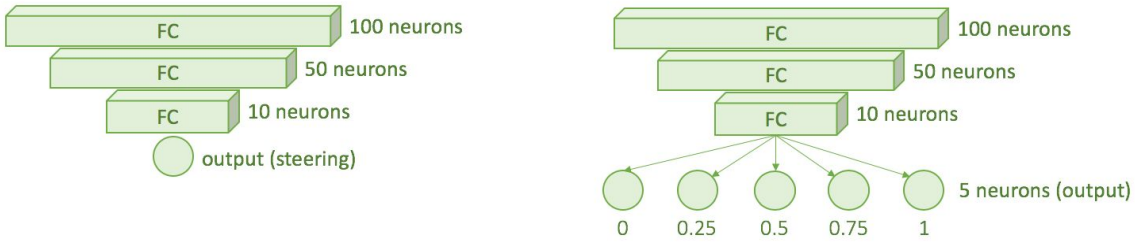
The car state is defined as steering, throttle and brake values. Since we wanted to create an algorithm that maximizes speed, but aware of the importance of staying on track, we created a function that gives high values of throttle when going straight and lower values when the prediction is to turn.

Our first attempt was to try not to change the network architecture. We added to the existing pilotnet a new hard-coded layer, which translates the given continuous value into one of the five states. Then, we continued with the DQN algorithm to compute the loss and train the network. Unsurprisingly, this shortcut didn't work well and the network didn't converge in the right way.



Left: imitation phase fully-connected layers. Right: experimental discrete action space fully-connected addition.

After that, we realised we have to change our network architecture in order to support discrete space actions. Therefore, we changed the last layer of our network to generate five values in the range [0,1]. So, when we loaded our weights from the imitation phase, instead of loading all layers weights, we initialized the output layer weights with values given from a standard normal distribution. The rest of the weights were loaded just like before.



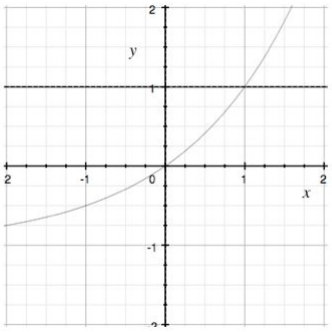
Left: imitation phase fully-connected layers. Right: final discrete action space fully-connected replacement.

Experiments

As written above, our main work in reinforcement learning was to find the right reward function that would help the network converge. According to the mentioned tutorial of the DQN, we blocked the reward functions to be in the range [0,1].

- **Reward by speed:**
  Since we realised that the algorithm already knows how to stay on the track, we thought that examining the reward from a function dependent on speed, will help the algorithm achieve better driving capabilities. We constructed a function to lead the algorithm to tend to a speed which is higher than before:

$$f(x) = \begin{cases} e^{x\ln(2)} - 1 & x < 1 \\ 1 & x \geq 1 \end{cases}$$



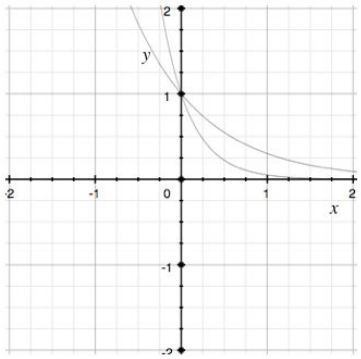
x, in meters per second, is the car's speed

As opposed to our assumption, the algorithm didn't improve.

- **Distance from the center of the track:**
  In order to achieve accurate driving in the center of the track, we came up with a method based on computing vector representations between center-points of the track (in the middle between two cones). By storing these vectors, we were able to compute the minimal distance from the center of the track at any given location of the car. We used functions similar to the one that in the previous section, with epsilon-distance from the center of the track. We tried two versions:

$$f(x) = \frac{1}{e^{1.2x}}$$

$$f(x) = \frac{1}{e^{3x}}$$



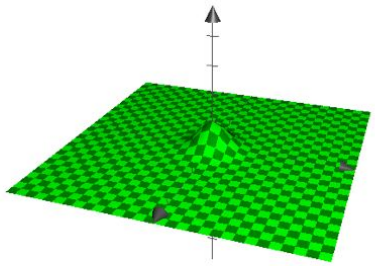
x, in meters, is the minimal distance from the center of the track

The results were better than before, but the algorithm didn't converge as expected.

- **Distance from both center of the track and required angle:**
  Experiments with computing distance from center of the track didn't prove themselves good enough. Therefore, we thought we might have missed a computation regarding the direction of where the car is heading. So, we constructed a function that computes both

distance from center of the track and the difference between the car's direction to the track line direction. The function is:

$$f(x,y) = \frac{1}{e^{2(x^2+y^2)}}$$



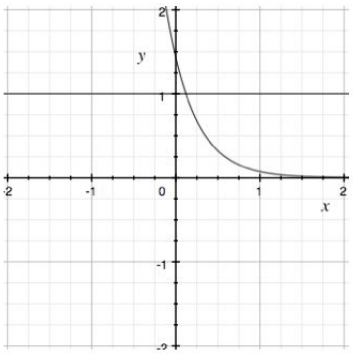
x, in meters, is the distance from the center of the track and y, in radians, is the difference between the angles

The purpose of this was that when the car is heading to the right direction and in the right steering angle, the reward should be 1. Otherwise, if one of them is not as mentioned, the reward will be lower. Surprisingly, this method turned out to be even worse than without direction reward.

- **Distance from <u>recorded</u> center of the track**

  After all these attempts, we thought that our problem is that we were forcing the algorithm to drive from one center point to another in a straight line. This led us to design a new way of storing the optimal path of the track. For each track, we recorded a session of the optimal drive path, by driving slowly while being very accurate with regards to the car's location. This recorded data, consisting of thousands of center points, gave us the ability to compute the minimal distance from the real center of the track. The function is:

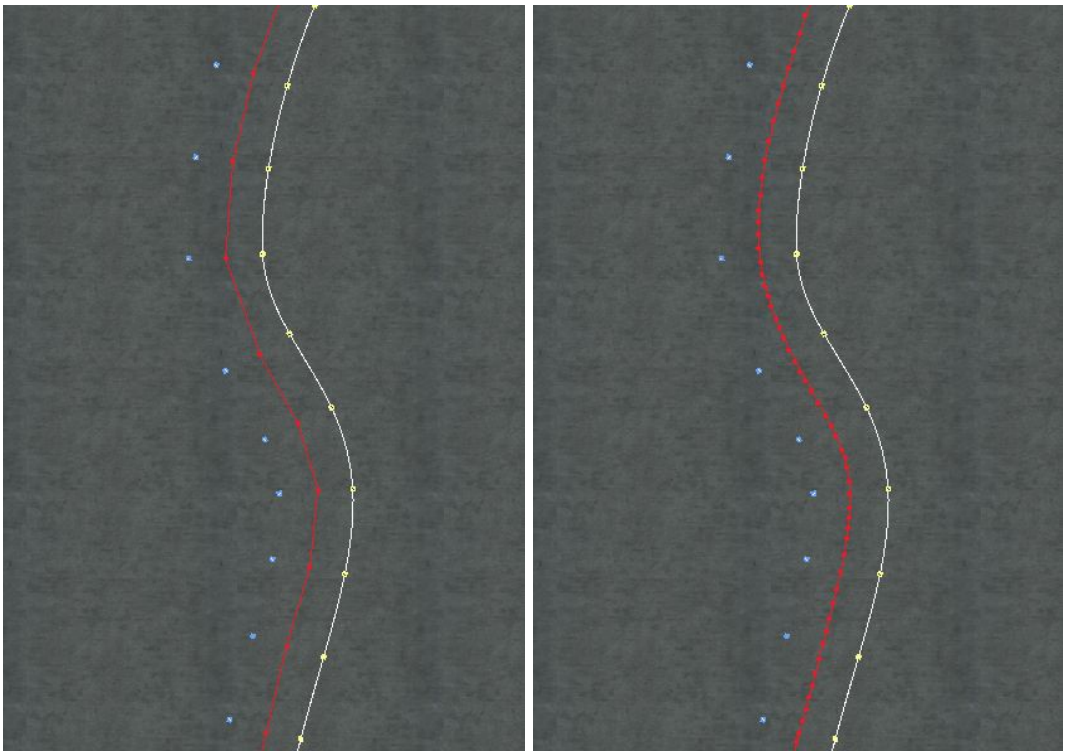
$$f(x) = \begin{cases} \frac{e^{0.36}}{e^{3x}} & x > 0.12 \\ 1 & x \leq 0.12 \end{cases}$$



x, in meters, is the minimal distance from the recorded center of the track

Since we still had a gap between each two points, we defined a circle of 12 cm radius at each point as an epsilon-neighborhood, just like in the previous section, to guarantee the

correctness of the reward function. As we thought, this method improved our algorithm and produced the best driving so far.

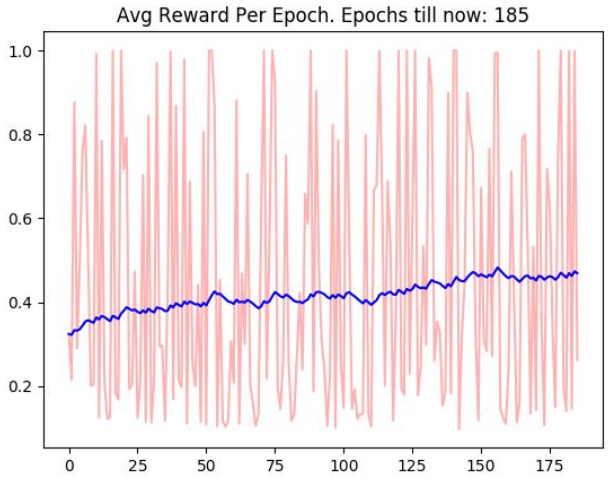


Demonstration of computing center of the track in two methods.
Left: the previous method of calculating vectors for each two center points. Right: record points while driving slowly and accurately.

In addition to these experiments, we needed to come up with a way to restore a training session to see if we could further improve a model by replacing reward function or use different parameters. We automated checkpoint saving and restoring. Then, we were able to decide if to continue the training session from the same point or to start over.
In accordance to decaying learning rate in popular implementations, improvement of the algorithm depends on the ability to try random choices to produce random states. To do so, we stored a variable of probability that decays every epoch. This means that in the beginning this variable will store a high probability, in our case 0.9, to allow big amounts of random actions. Towards the the end, this variable will be 0.2. This is to ensure that we will always look for an improvement.

Visualization

We needed to have a method to investigate the training status. In the first steps, we thought of tracking the loss values on the training samples. But since the algorithm uses high amounts of random actions in various states, we weren't able to follow some patterns in loss values. We realised that a good method to track the improvement of this procedure is the reward of the car - if the car learned to drive better, we should see an increase in the average reward, regardless to the probability of the random actions.



Example of the reward as a function of an epoch. Since the reward is noisy (in red), we presented the values with a smooth line (in blue). The function for smoothness is the exact function that is being used in Tensorboard.

This visualization helped us realise what are our problems in converging to an appropriate driving session. The first thing we noticed is that when driving in the same track for too long, it makes the algorithm overfit on specific turns, which expresses itself in a lower average of reward. This led us to create more tracks and replace them more often during training. Second, we realised that we need higher variance in our weather conditions, so we developed CycleLight (see training environment section).

Final Model

The following diagram shows the main points of change along our training procedure from the perspective of data distribution and architecture. It's important to notice, that we based our estimation of how a good a model is by letting it drive through unseen tracks. This is in contrary to the normative way of using a test set. In our case, this process was better for understanding how good the model is, since there is more than one way to drive through a track, having a test set would not necessarily generate proper feedback.

Model	Changes	Val Loss	Time On Track
Basic CNN	-	0.0570000	2 minutes
PilotNet	-	0.0000345	10 seconds
PilotNet	Leaky ReLU	0.0000279	3 minutes
PilotNet	Drunk style data added	0.0039492	5 minutes
PilotNet	Removing car state from input	0.0130848	6 minutes
PilotNet	CycleLight, added shifted driving	0.0028828	3 hours

The final model performs exceptionally well on all tested tracks. It navigates through previously unseen simulated maps, some of which with objects beside the road and all while maintaining an average speed of roughly 20 km per hour (our initial goal).

The chosen architecture is a modified version of PilotNet[5], with just a single camera image as input and a number between 0 and 1 as output of the net. The final activation is Sigmoid, and a simple normalization of the image (scale factor of 1/255) is expected to be performed by the users of the net prior to inference. The activation between layers used is ReLU and a Dropout layer with a parameter p=0.5 was also added.

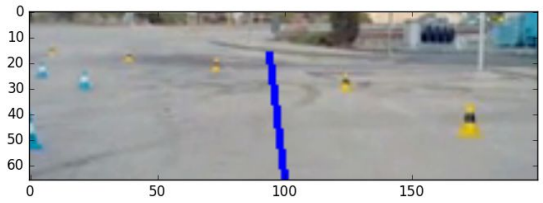
The data used consists of normative, drunk, swervey and shifted driving-style samples, alongside augmented data composed of light reflections, random horizontal lines, and horizontal distortions. It also varied in environment conditions: temperature, cone-distance, different maps, cloudiness, sunlight intensity, day-time etc. The distribution diagrams can be found in the above “Data” section.

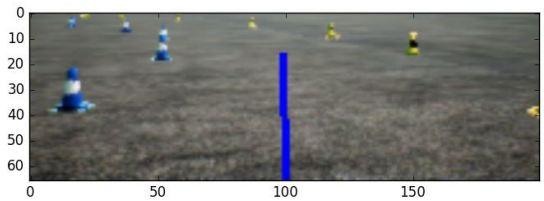
The final model can maintain its driving capabilities and ignore irrelevant objects in the ROI, as well as deal with sharp turns and changing environment conditions.

5 Theory to Practice

Introduction

The process of moving on from the simulation environment to the real world included a lot of challenges: camera adaptation, hardware and car integration, and performance assurance.
The real world can be very different from the simulation environment. Especially, the view from the camera. To overcome this, we simulated the conditions on a real track, and adapted the camera in the simulation to the car's camera, as mentioned in section 2. Moreover, we changed the car camera's parameters to fit the simulation's camera.





Real world testing *Simulation testing*

The images above were taken from tests in the real world and in the simulation environment. Due to constraints, our tests for now in the real world weren't on a suited race track, but on a regular road.

Autonomous systems work on low power computers, such as arduino or Nvidia's Jetson and Drive series. Therefore, we couldn't execute our algorithm on a standard PC and had to try different options. The transition to a low power computer consists of a complex set of actions. On such computers, the hardware specifications are less capable than PCs. Moreover, Formula cars should react quickly to steering instructions, so an optimization of the algorithm runtime must be performed. To execute our algorithm with high performance, we had to change our code and design it especially for the chosen computer's architecture.

Real Environment Testing

The procedure of testing our algorithm in real world environment was as follows: recording videos from the real car's camera, while driving on a track made of cones and testing our algorithm offline.
Our experiments took place on a regular road, not a formula race track. We constructed tracks made of cones on a road according to the formula student competition regulations. The camera we used in the experiments had similar parameters to the simulated camera we trained our algorithm with. The weather conditions during the experiments were: summer sunny day at noon.

Due to lack of telemetry (wireless communication for receiving data from the car, like steering), we were not able to examine the output of our algorithm at real-time while the car was driving. Therefore, we had to check offline for the accuracy of our algorithm on the recorded real world driving.
An example of our algorithm's results on a recorded real world driving can be found at: https://www.youtube.com/watch?v=KTwIrgMe6OA

A picture from one of the driverless experiments, taken with a GoPro camera

The algorithm's results were sufficient, but we observed some issues. Where the track had nearby obstacles, such as big garbage cans and traffic signs, the algorithm was not accurate: it made the steering output incorrect. Nevertheless, in a formula student competition there are not supposed to be any obstacles as mentioned above.

Implementation

As was stated before, formula cars should react quickly, which means that we need the best performance we can get when we execute the algorithm on the real car. We conducted a comprehensive research on what would be the best computer for us.

Assembling on the real car

To perform inference in high frequencies, we need a powerful computer capable of managing parallel computations. The problem is that the car's chassis is not a good place for a PC. If we place the computer on our chassis, we need to absorb shocks and prevent liquids from penetrating the case. Therefore, we have two options: purchase a standard PC that is proof to damage or to use a low power computer, which is more durable than a PC.
The main advantage in obtaining a durable PC is that we don't need any adaptations in moving our algorithm, since it's the same architecture. The main disadvantage is weight. Our goals when

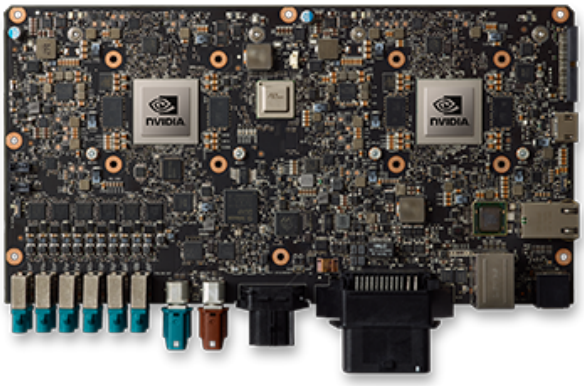
assembling a race car are to save weight as much as we can (durable PC can weigh twice than a standard one), to save power and to ease acceleration.

Durable PC case

Also, space is another issue we should take into account. Not everything can be assembled on our car and a durable PC has larger dimensions than a low power PC.

Low power computers

The big difference between PCs and low power computers is their architecture. Small computers like raspberry pi or Jetson TX are built with a CPU called ARM. This CPU is based on an architecture that is different than our regular x86 PC, which means that there will be some libraries that we need to understand how to implement, or replace them if it's impossible to do so.



Nvidia Drive PX2

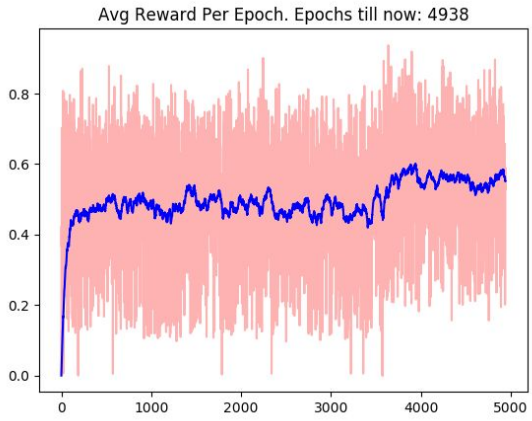
Nvidia Jetson TX2

Also, this architectures consists of smaller and weaker parts. ARM processors works in a frequency significantly lower than Intel i5 or i7 processors. These computers are usually built with MMC memory, which works on lower reading and writing speeds.
Finally, to assemble a PC on a real car, we need to supply power by connecting it to a battery. So, due to limitation in power consumption on the car, a low power computer is required.

6 Summary and Prospects

Conclusions

We started with creating a virtual environment, dedicated to simulate the real formula student racetracks, in which we implemented our own formula car model. To train an algorithm that will self-steer a car through a racetrack, we tried both imitation learning and reinforcement learning in parallel. We built network architectures to allow different training procedures. Moreover, we tried to execute these two methods one after the other using transfer learning as a mediator. With reinforcement learning, we have the potential to train an algorithm to perform better than human drivers. However, our breakthroughs were in the imitation learning procedure. We concentrated on different types of data and their distribution. For example, we augmented the data with techniques like automation of the sun's position, simulation of noise and distortions, addition of drunk driving data and shifted images to teach steadiness in the center of the track. Although we didn't succeed in converging a model with reinforcement learning, we kept on trying and investigating the possibilities of improving our algorithm until our last breakthrough. When we achieved the reliability we wanted with imitation learning, we started to focus on the implementation section.



One of our experiments in reinforcement learning.
This experiment shows an improvement after changes in the environment.

In our opinion, reinforcement learning has the potential to give us better results. The DQN algorithm leads to many possible implementations and there is still a lot to investigate. We were also limited by the amount of resources we had for this project - To perform DQN algorithm with heavy graphics simulation, we needed powerful PCs that can reduce delay in executing operations the algorithm instructs. Also, to save time, it's better to try a few attempts in parallel, if additional PCs are available.

One of our experiments in the Technion

Regarding testing in a real environment, we deduced from experiments of other formula student teams, that transforming algorithms to work on low power algorithms requires a lot of work and may lead to change in plans. Algorithms based on heavy computations of sensors usually are operated in frequencies which are not feasible for autonomous driving. We learned from that to pay attention to wasteful computations and to replace them with more efficient ones.

Open Questions

- Can a model based on learning from the recordings of a group of human drivers outperform any single one of them by his own?
  Or in other words, is there a positive correlation between the amount of human drivers performing the recordings of the data, to the end performance of the trained model, and if so, to what extent?
- Imitation-Reinforcement dialogue training: During our work, one of our supervisors, Dr. Ashish Kapoor, had suggested an intriguing training method. The process is as follows:
  a. Using recorded data, train a model with Imitation Learning.
  b. Have the model drive in the simulated environment in the setting of reinforcement learning (DQN approach). While performing reinforcement learning, record images after a predetermined time from the car's drive start, accumulating new samples and then coupling them with suitable steering labels, which are calculated based on the car's position at the moment of recording.
  c. Go back to step (a) while using the newly gathered samples.

  Under our resources constraints, we couldn't test this suggestion, but perhaps an automation of this dialogue procedure could generate a successful optimization process of the model.

Future Work

- **Improve the performance of the algorithm by using transfer learning for a long period of time**:
  As mentioned in the training section, time and lack of resources prevented us from achieving better results in this section. We believe that with more time and resources, we can prepare a plan of trials to find right ways to obtain better results than those we had so far.

- **Improve the algorithm considering nearby obstacles:**
  As opposed to what we did this year, autonomous driving algorithms usually based on separating the algorithm into blocks, in order to inspect the algorithm's analysis.
  For example, urban driving algorithms usually performs object detection and semantic segmentation in order to present the feature extraction for the given image. We can separate our model into a few models to achieve better understanding and full control on our environment. This move will require additional ways to record different datasets.

- **Improve collecting data by VR**:
  Our assumption is that the more realistic the act of collecting the data is, the more accurate a model can be in real environment. During our project, we noticed that the best results were when we collected the data with a steering wheel (not with joystick or keyboard keys). Therefore, through the usage of VR we could further enhance the quality of the data.

- **Data creation using generative models**:
  A method that is being used last years to transfer an image from one domain to another, is generative models (GANs). We can use it in order to close the gap between the simulation and the real world, by transferring recorded images from the simulation domain to the real environment domain. Than, train our algorithm on real-life images.

7 Code repository

Our code repository for the simulation and the imitation learning process can be found at:
https://github.com/FSTDriverless/AirSim

Our code repository for the implementation on a low power computer can be found at:
https://github.com/FSTDriverless/FSTImplementation

8 Bibliography

1. Autonomous Driving in Urban Environments: Boss and the Urban Challenge (DARPA challenge) https://www.ri.cmu.edu/pub_files/pub4/urmson_christopher_2008_1/urmson_christopher_2008_1.pdf\
2. Design of an Autonomous Racecar: Perception, State Estimation and System Integration https://arxiv.org/pdf/1804.03252.pdf
3. Microsoft AirSim https://github.com/Microsoft/AirSim
4. Playing Atari with Deep Reinforcement Learning (DQN) https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
5. End to End Learning for Self-Driving Cars https://arxiv.org/pdf/1604.07316.pdf
6. Learning a CNN-based End-to-End Controller for a Formula Racecar https://arxiv.org/pdf/1708.02215.pdf
7. DQN implementaion using CNTK https://github.com/Microsoft/AirSim/blob/a0e70d6e2b76ac780334229941aa79eb449bdbbd/PythonClient/DQNcar.py
8. Distributed Deep Reinforcement Learning for Autonomous Driving https://github.com/Microsoft/AutonomousDrivingCookbook/tree/master/DistributedRL