

DECLARATION

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university thesis.

Prague, date 25/05/2018
.....

A handwritten signature in black ink, consisting of a stylized, cursive script with a prominent loop and a horizontal stroke extending to the right.

Signature

DEEP REINFORCEMENT
LEARNING FOR
AUTONOMOUS OFF-ROAD
DRIVING IN SIMULATION

Jacques Valentin

Specialization: Cybernetics and Robotics

Thesis Supervisor: Karel Zimmermann, doc. Ing., Ph.D.

Thesis Co-supervisor: Teymur Azayev

Czech Technical University

A thesis submitted for the degree of

Master of Science

June 2018

ABSTRACT

This thesis presents different ways to make a car autonomous. We will use the power of machine learning and neural network to “teach” a car how to drive autonomously in an off-road environment by using only a minimum set of sensors, in our case which is just a single RGB camera. We will first focus on a technique called imitation learning, it is a supervised learning algorithm which takes a lot of example pairs (image; driving command) to extract a policy that the car will use to drive in unseen situations. Then we will use the so-called reinforcement learning technique. It is an unsupervised learning algorithm which manages, by a lot of trial and error experiments, to create a policy used by the car to drive safely. We managed with these two techniques to make our car drive itself in a simulator.

ACKNOWLEDGMENTS

I first want to thank my advisor M. Karel Zimmermann who helped me during this project. I also want to thank Teymur Azayev who joined this project and helped me a lot to solve some problems I had. I finally want to thank my parents and my friends who've been there in both happy and difficult moments of this project.

TABLE OF CONTENTS

Abstract.....	i
Acknowledgments.....	ii
Table of Contents.....	iii
Table of tables.....	vi
Table of figures.....	vi
Introduction.....	1
Chapter 1 Neural network.....	3
Layers.....	4
Fully Connected layer.....	4
Convolutional layer.....	5
Dropout [7].....	7
Architecture.....	7
Loss function.....	9
Mean square error.....	9
Optimizer.....	10
ADAM.....	10
Software.....	12
Tensorflow [10].....	12
TFlearn [11].....	13
Chapter 2 Simulators.....	14

DEEPDRIVE	15
CARLA [12]	15
AIRSIM [13].....	16
Sensors	18
API.....	20
Final Choice	20
Chapter 3 Imitation learning	22
Data	23
Gathering the data	23
Splitting the training and testing data	26
Experiments	27
Fine tuning of hyperparameters	28
Results	32
Chapter 4 ReinforcEment learning.....	34
Principle	35
Reward	36
Algorithm.....	38
Technique 1: simple DQN [15].....	39
Technique 2: double DQN [15].....	40
Technique 3: Double Dueling DQN [16]	40
Comparison of thEse techniques.....	41
Results	43
Imitation learning vs Reinforcement learning	44

Future work.....	47
Using semantic segmentation for imitation learning.....	48
Adding some memory.....	48
Train also the throttle of the vehicle	49
Conclusion.....	50
Bibliography	52
Annex 1: Learning rate analysis	54
Annex 2: keep_prob analysis.....	57

TABLE OF TABLES

Table 1: neural network architecture	8
Table 2: simulators comparison.....	21
Table 3: results for imitation learning.....	33
Table 4: results for reinforcement learning	43
Table 5: imitation learning vs reinforcement learning.....	45

TABLE OF FIGURES

Figure 1: fully connected layer	4
Figure 2: convolutional layer.....	5
Figure 3: ADAM algorithm from [9].....	11
Figure 4: CARLA sensors: RGB image (left), Depth image (center), semantic segmentation (right).....	16
Figure 5: AIRSIM environments.....	17
Figure 6: position of cameras	18
Figure 7 : camera images: RGB (top left), Depth (top right), semantic segmentation (bottom left), surface normal (bottom right).....	19
Figure 8: scheme of the multi-threading algorithm used for logging the data.....	24
Figure 9: data file	25
Figure 10: Technique 2.....	25

Figure 11 : split train/test	26
Figure 12: effect of learning rate onto learning process.....	28
Figure 13: learning rate comparison	29
Figure 14: effect of <i>keep_prob</i> on the learning process.....	30
Figure 15: <i>keep_prob</i> comparison	31
Figure 16: final learning curve.....	32
Figure 17: Pseudo code for DQN	39
Figure 18: DQN learning process	41
Figure 19: double DQN leaning process.....	41
Figure 20: double dueling DQN learning process	42

INTRODUCTION

The first experiments on self-driving cars were conducted in the 1920s¹ in some labs. Nowadays we see companies like Tesla and Valeo which are making cars autonomous for everyone. Those improvements were made thanks to the evolution of artificial intelligence techniques. Moreover, thanks to the evolution of graphical units simulators became really realistic so people don't need real car to design artificial intelligence algorithms which need a lot of experiments and data. Simulators also made the development of such algorithm cheaper and accessible to almost anyone.

In [1] they use Google Street View pictures, which cover almost all the cities in the world, to learn a policy- with reinforcement learning algorithm- to navigate vehicles through a city. That shows that nowadays with artificial intelligence techniques and the amazingly big amount of data we can gather, we don't need straight rules to navigate a vehicle. They also show that unlike most mobile robots which need a map to navigate to a particular destination, here we just need pictures of the environment without creating a map of it. To do that they trained a reinforcement learning algorithm and increased gradually the complexity of the tasks. A lot of different reinforcement learning algorithms are described in [2]. They analyzed the different improvement of deep Q learning algorithm and show how we can combine them to get better results. Reinforcement learning algorithm became "famous" when a computer defeat a human for the first time in the game of go, in [3] the authors explained how they used reinforcement learning for tree searching in order to play the game of Go. The algorithm explained in [3] is an evolution of the first algorithm described in [4] which was also using imitation learning to learn a policy to play go. Imitation learning is another technique used to navigate a vehicle through a city, in [5] they described how the used imitation learning from expert driving data to do that. They trained their policy in a simulator and they were also able to put this policy into a 1/5 scale truck to test it into the real world. That

¹ https://en.wikipedia.org/w/index.php?title=Autonomous_car&oldid=838072806

shows that simulators help to gather realistic data which can be used to train neural networks that can perform well in the real world as explained in [6].

For this project, we decided that we wanted to use a simulator and these techniques to make an autonomous off-road driving vehicle. We looked at different simulators and decided that instead of navigating through a city we will navigate through a less structured environment like a countryside track. After finding the right simulator we made a car autonomous using the two different techniques explained previously. We first trained a neural network using imitation learning. We tried different techniques to gather the data and also tried different architectures for the network. After finding the right architecture and getting some results with imitation learning, we implemented a reinforcement learning algorithm to compare the results of both techniques.

CHAPTER 1

NEURAL NETWORK

One of the goals of this project is to study how we can use artificial neural networks to make a car autonomous. We will consider that the reader knows what a neural network is and its principle. We will start with a focus on the different layers that we used in the network. Then we will focus on the activation function, the loss function and the optimizer we used.

Layers

FULLY CONNECTED LAYER

Fully connected layers are the most common layers of neural networks. Let's see how it works.

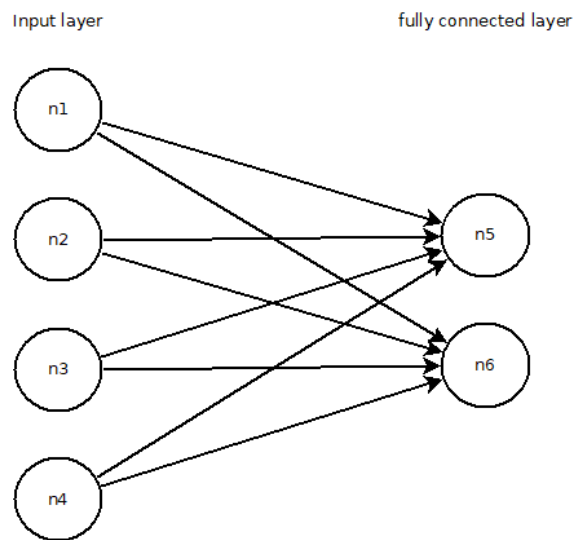


Figure 1: fully connected layer

On this figure, we see that the fully connected layer is composed of the neurons n5 and n6, and each neuron is connected to all the neurons of the previous layers. We have the following result

$$n_j = \sum_{i=1}^4 w_{i,j} \cdot n_i + b_j \text{ for } j = 5,6$$

Where $w_{i,j}$ and b_j are the weight and the bias that will be trained to fit our database. We can then see that for each neuron of the fully connected layer there are $l+1$ parameters where l is the number of neurons in the previous layer. Then we need to train $(n+1) \cdot m$ parameters where m is the size of the fully connected layer. The problem with this kind of layer is that it takes a lot of parameters (if we want to feed a 120×120 images to a fully connected layer with a size of 10 it takes 144 000 parameters), and does not preserve the spatial arrangement of an image, because we have to flatten it before feeding to the fully connected layer.

CONVOLUTIONAL LAYER

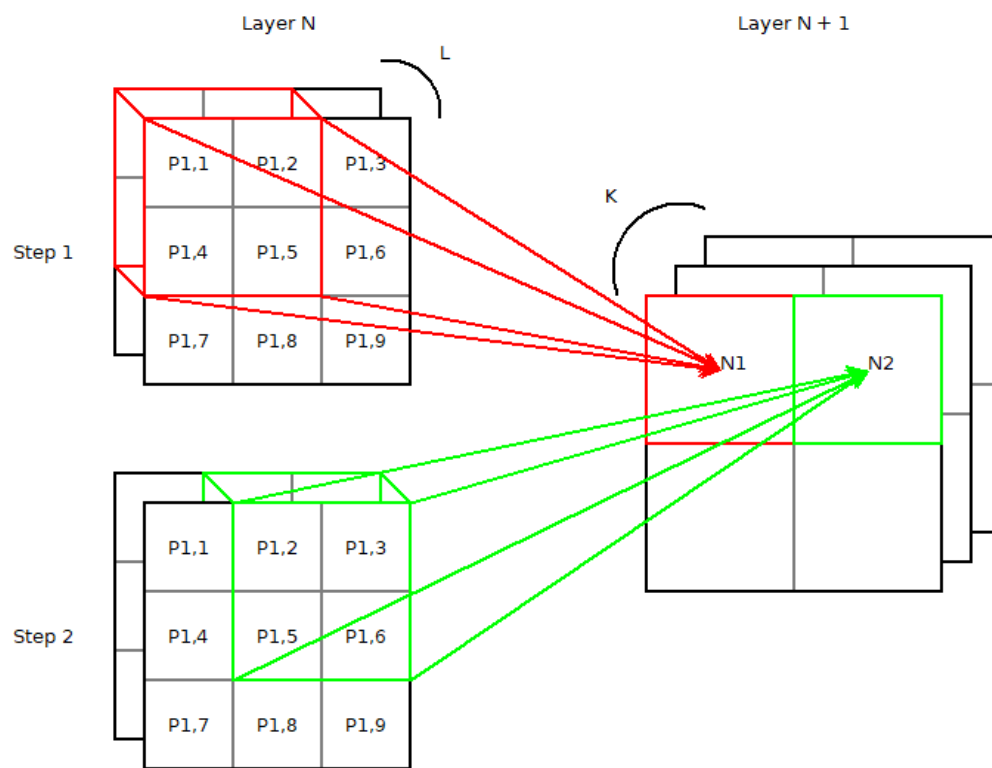


Figure 2: convolutional layer

Unlike the fully connected layer, the convolutional layer preserves the spatial structure of an image. In figure 2, we can see a convolutional layer with K filters. The principle is easy to understand, we take a window of size $n \times m$, in our example the size of the window is 2×2 , which will slide through the image. And then we cluster the pixels inside the window to feed it to a neuron in the next layer. So we have:

$$\begin{cases} N_1 = \left(\sum_{j=1}^L W_{j,1} \cdot P_{j,1} + W_{j,2} \cdot P_{j,2} + W_{j,3} \cdot P_{j,4} + W_{j,4} \cdot P_{j,5} \right) + B \\ N_2 = \left(\sum_{j=1}^L W_{j,1} \cdot P_{j,2} + W_{j,2} \cdot P_{j,3} + W_{j,3} \cdot P_{j,5} + W_{j,4} \cdot P_{j,6} \right) + B \end{cases}$$

We see that for the first channel of the layer $N+1$ we will need $(m * n * L + 1)$ parameters, so in total we need $(m * n * L + 1) \cdot K$ parameters. The different settings we have are the size of the kernel, the number of filters (K in our example) and the stride. The stride is the number of pixels the kernel is moving to the right and down. These parameters have an influence on the size of the next layer, as we see on the figure 2 the layer $N+1$ has a size of 2×2 and the size of the layer N is 3×3 . So it is a way to decrease the complexity of the problem. In our network we will use that instead of the pooling layer.

DROPOUT [7]

A big problem in Machine learning is overfitting. It means that the network is perfectly trained based on the data we used to train it but it will behave very badly on data it never saw. To prevent this in a neural network we use a layer called dropout. This layer will drop neurons with some probability. Thus it reduces the correlation between the layers and prevents overfitting.

Architecture

To find the correct architecture we get some inspiration from the network described in [8]. In this paper, they also implement a neural network for a self-driving vehicle using a single camera which led us into thinking that it was a good architecture. But it was a really heavy network used to detect a lot of features like road signs and there was more than 4 billion parameters to train. Since we don't need this we simplified it by removing some layers or just changing their sizes. After a lot of experimentation, we finally found an architecture which seemed to work. We can see the final architecture in table 1.

The input layer is a grayscale picture from a camera which we normalize to get a mean of 0 and a variance of 1. This technique is commonly used to make neural networks converge faster.

The output layer is a vector of size NUM_OUTPUT. This number depends on the data we have and how we want to control the car, we will talk about this in the next chapters. It can be 1 2 or 3 that depends on what we want to control.

- 1- Continuous control for the steering only
- 2- Continuous control for steering and throttle
- 3- Discrete control for the steering only

Because we used different approach the target of the network changed. If we use a continuous control, the target is just the control command of the vehicle for the image which is fed into the network. When we used a discrete control, the problem

becomes a classification problem so the target is a one hot vector. A one-hot vector is a vector full of zero except one element, the one which represents the class we desire for a given state, which is one.

<u>Layer</u>	<u>Type</u>	<u>Size</u>	<u>Number of parameters to train</u>
1	Input layer	120x120x1	0
2	Convolutional Kernel : 5x5; Stride : 2x2	58x58x24	624
3	Convolutional Kernel : 5x5; Stride : 2x2	27x27x36	21636
4	Convolutional Kernel : 3x3; Stride : 1x1 Flatten ²	25x25x64 40000	20800 0
5	Fully-connected	10	400010
6	Dropout	10	0
7	Output layer	NUM_OUTPUT	11xNUM_OUTPUT

Table 1: neural network architecture

² Layer to flatten the previous layer into the fully-connected layer

Loss function

The loss function is the mathematical tool which helps to determine how far the model is from the ground truth. During the different experiments we did we used two different loss functions because we approached this problem from two different angles. But at the end, we used only the mean square error loss function.

MEAN SQUARE ERROR

$$(1.1) \quad MSE = \frac{1}{N} \cdot \frac{1}{f} \sum_{i=1}^f \sum_{j=1}^N (y_{i,j} - \hat{y}_{i,j})^2$$

The equation (1.1) describes the mean square error where:

- N is the number of data in the batch
- f is the number of features we want to predict, in our algorithm
 $f = NUM_OUTPUT$
- $y_{i,j}$ is the ground truth value for the feature i in the batch j
- $\hat{y}_{i,j}$ is the predicted value for the feature i in the batch j

The mean square error is usually used for regression problems. In this project, we used it when we wanted to control the car with a continuous input. But we have also used it in our DQN reinforcement learning algorithm which is a classification problem, where we want to learn the Q policy which will be used for the classification.

Optimizer

To train a neural network we need to minimize the loss function. The way to do it is to feed the network with an input where we know the ground truth prediction, and then we need to backpropagate the loss function through the network to update each weight and bias of the network. The optimizer is the tool used to do this.

ADAM

In this project, we used the ADAM optimizer. But first, let's take a look at the gradient descent optimizer. This optimizer updates the weight using the formula (1.2)

$$(1.2) \quad w_i \leftarrow w_i - \eta \frac{\partial J}{\partial w_i}$$

Where:

- J is the loss function
- η is the learning rate

We see in the formula that η is a constant which describes how much we have to follow the gradient. But if we are too close to a local minimum we can “skip” it if η is too big. ADAM optimizer solves this problem by updating the learning rate

through the learning process for each weight. The algorithm is described in [9] as you can see in figure 3.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0 : Initial parameter vector
 $m_0 \leftarrow 0$ (Initialize 1st moment vector)
 $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
 $t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
 $t \leftarrow t + 1$
 $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
 $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
 $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
 $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
 $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
 $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 3: ADAM algorithm from [9]

Software

To implement our artificial neural network we used two libraries in python, Tensorflow, and Tflern

TENSORFLOW [10]

Tensorflow is an open source library developed by Google. It is a really useful library because it can compute either on a CPU or on a GPU, and it is the same code for both of them which is very useful when one wants to work on a cluster. For example, one can start to design one's network on one's computer without a GPU and then finalize the training on a cluster with multiple GPUs. It is also very easy to use thanks to all the tutorials that one can find on the internet.

The main principle of Tensorflow is to work with graphs, the graphs are composed by:

- Tensors, which are the variables represented by matrices. They are represented in graphs by edges.
- Operations that operate on the tensors. They are represented in graphs as nodes.

TFLEARN [11]

Tflearn is an abstracting layer on Tensorflow which simplify the implementation. It provides a lot of different layers like convolutional and fully connected. It also provides a simple way to fit our model to the data and then load and save our model. One of its nice features is that it is totally compatible with Tensorflow so every operation made with Tensorflow are compatible with Tflearn. This can be useful to implement some features which are not provided by Tflearn.

CHAPTER 2

SIMULATORS

For this kind of problem, self-driving car, it is much easier to use a simulator to simulate the car in different type of environment. As we can see in [6] simulators are a good way to generate annotated data, it doesn't need a large amount of human effort to annotate all the images we can get, for example for semantic classification. It also reduces the time to get a large variety of images. It is easy, for example, to get images in a raining environment but in the real life, we have to wait until it rains. So we can understand through this example the potential of a simulator for deep learning algorithms. For this project, we found different simulators, which can simulate a car in different environments, and which were released at the beginning of our project.

DEEPRIVE

We won't talk a lot about this simulator, but like in [6] they used the open world game GTAV to train self-driving car algorithms. But Take-Two Interactive© forced them to shut down the project so we couldn't move forward in this direction. But finally they moved to unreal engine and released the second version of Deepdrive, but it was too late for us.

CARLA [12]

CARLA is an open-source simulator created to simplify autonomous driving research. It simulates an urban environment (because at the beginning of the project we didn't know that we wanted to focus on off-road driving), where we can find different static objects like buildings or vegetation but also dynamic objects like pedestrians and cars.

As you can see in figure 3 the simulator provides different sensors, RGB image (Left), depth image (Center) and semantic segmentation (Right).

Moreover, the physic of non-player characters was really realistic, the pedestrian, for example, were walking on the sidewalk but could cross the road at any moment - which is a nice feature - to explore as many states as we can to train a self-driving vehicle.

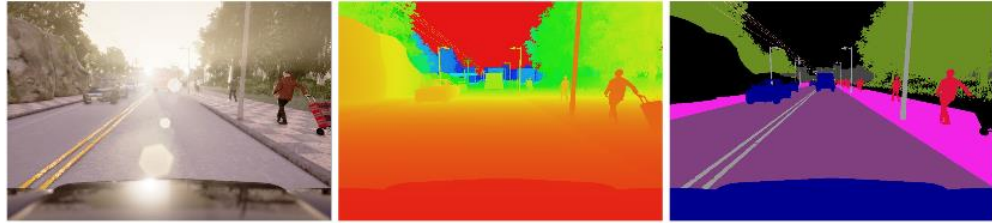


Figure 4: CARLA sensors: RGB image (left),
Depth image (center), semantic segmentation
(right)

In [12] the authors try to compare different algorithms like the one we wanted to implement and this is why we were interested in this particular simulator.

The problem we had is that at the beginning of the project, just the first version of the simulator was available and after running into some difficulties to install it, we realized that it was really slow and it was almost impossible to drive the car under such conditions.

AIRSIM [13]

Airsim is a simulator for drones and car on unreal engine environment, its purpose is to represent as closely as possible - the real world - to be able to export an algorithm trained in the simulator into the real world. It is available on both Windows and Linux and providing binaries for different kind of environments as one can see in figure 5.



Figure 5: AIRSIM environments

It also provides different interesting features

SENSORS

On each vehicle, there are 5 different cameras positioned as in figure 6

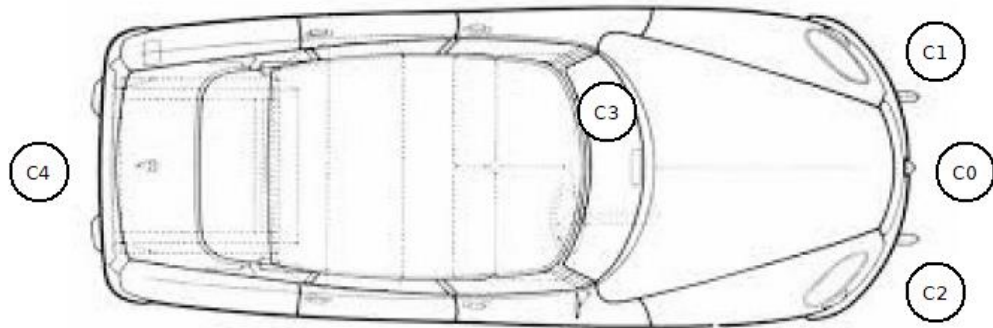


Figure 6: position of cameras

Each camera can provide different kind of images as you can see in figure 7:

- RGB images (top left)
- Depth images (top right)
- Semantic segmentation (bottom left)
- Surface normal (bottom right)

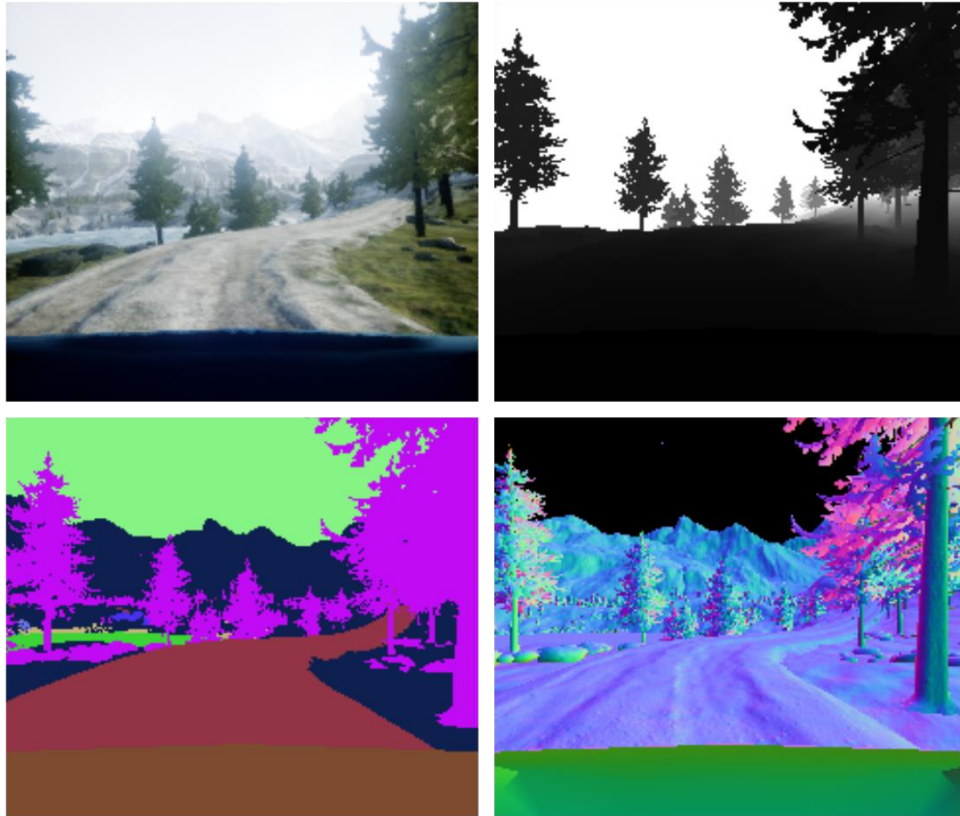


Figure 7 : camera images: RGB (top left), Depth (top right), semantic segmentation (bottom left), surface normal (bottom right)

It is also possible to get the speed of the vehicle and to see if the vehicle collided with something.

API

AIRSIM also provides an API to control the car. Using which, we can control different things:

- The throttle
- The steering
- The brake
- The gear (it can also be set to automatic)

The only problem is that when we drive the car manually we don't have access to these parameters, we will get to this point in the next chapters.

Another feature of the API is the possibility to set and get the pose of the vehicle, it helps to set the vehicle in a specified position and then train our network in different situations.

Final Choice

Now that we are aware of all the three choices we have had, we'll explain our final choice. Table 2 highlights the pros and cons of each simulator we have talked about. We finally choose the Airsim simulator because we could not do what we wanted to do with the other two. Also, because of the fact that it was difficult to put more vehicles in the simulation, we choose to use the mountain environment

that one can see on figure 5 to accomplish off-road driving which is usually sans competing vehicles in the same environment.

Simulator	Pros	Cons
Deepdrive	<ul style="list-style-type: none"> • Big open world of GTA V 	<ul style="list-style-type: none"> • Not available at the beginning of the project
CARLA	<ul style="list-style-type: none"> • Lot of different sensors • Good physics of non-player characters 	<ul style="list-style-type: none"> • Difficult to install • Only available on Linux • Not fluent and really difficult to drive manually
Airsim	<ul style="list-style-type: none"> • Lot of different sensors • Easy to install thanks to binaries • Lot of different environments • Available on Windows and Linux • Realistic physics of vehicle and environment 	<ul style="list-style-type: none"> • Need to be used to work with unreal engine to add more vehicles • Can't get the control command when driving manually

Table 2: simulators comparison

CHAPTER 3

IMITATION LEARNING

As explained in the introduction we tried two different techniques to make our vehicle autonomous. In this chapter, we will explain the first one which uses imitation learning algorithm. It is a deep learning algorithm where an expert drive manually the car while gathering the state of the vehicle, in our case it's an image taken in front of the vehicle, and the command of the vehicle like the steering and the throttle in our case. From these examples, a neural network is trained to learn a policy which we will use to make the car drive autonomously.

GATHERING THE DATA

The first thing to do was to get the data in order to train our neural network. The data is composed of an image extracted by the camera in front of the vehicle as we saw in the previous chapter, and the ground truth is the control command of the vehicle.

We tried two different techniques to get those data.

Technique 1

As we explained before it is not possible to get the steering and throttle when we drive the car manually in the simulator. So we had to find a way to overcome this problem. To do that we use different threads as you can see in figure 8. We implemented a keylogger which gives us the ID of the key pressed at time t_k and in parallel we have a thread which logged the images taken by the camera at time t_i . We then get two .csv files containing the images, the ID of the keys pressed and the time when we get each of them. The .csv file looks like the figure 9. And now we can reconstruct the throttle and steering by counting how many times we have pressed the keys in the time between the capture of two images. We tried different formulas to reconstruct the control command:

- $$\begin{cases} \text{throttle} = k_1 \cdot (\text{up} - \text{down}) \\ \text{steering} = k_2 \cdot (\text{right} - \text{left}) \end{cases}$$
 Where up, down, right, left are the number

of times we have pressed the corresponding keys in the time between the capture of two images. And k_1 , k_2 are two positive constants we can set up. The problem with this technique is that we don't know if it is a good

way to reconstruct the throttle and steering and we don't know how to set k_1 and k_2 perfectly, so it was not a very acceptable idea.

- $\begin{cases} steering = k_1 \cdot \text{sgn}(right - left) \\ throttle = K \end{cases}$ Here, we just use a binary steering and

a constant throttle. In that case, the steering can take only three values k_1 , $-k_1$ and 0. So the problem switches from a regression problem to a classification problem as was explained in the first chapter.

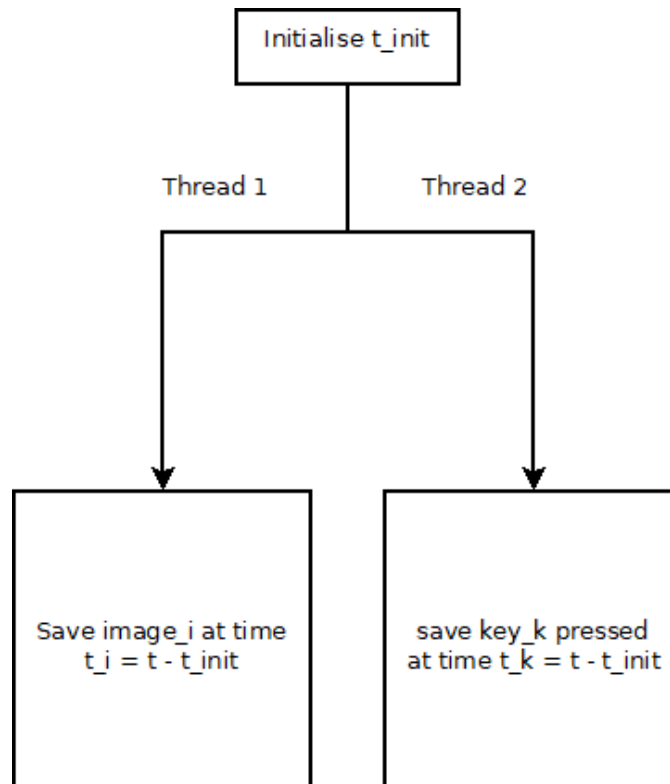


Figure 8: scheme of the multi-threading algorithm used for logging the data

Image	time
Image 1	t _j 1
Image 2	t _j 2

key	time
id 1	t _k 1
id 2	t _k 2
id 3	t _k 3
id 4	t _k 3

Figure 9: data file

Technique 2

The second technique was more accurate because we use the API to drive the car manually. To do that we used a joystick to get the steering value and we use a constant speed instead of a constant throttle. The reason is that a constant throttle can lead to a high speed and high acceleration at the beginning. In this environment, a high speed and a high acceleration make the vehicle drift making it in turn difficult to control. The pseudo code in figure 10 explains the procedure we have used to get the data.

```

1  poses = []
2  command = []
3  speed_ref = 6
4  while( button not pressed)
5      steering = getJoystickPosition() # get the position of the joystick
6      throttle = K*(speed_ref - speed)
7      p = getPose() # get the pose of the vehicle
8      controlCar(steering,throttle) # control the car with steering and throttle
9      command.append(steering)
10     poses.append(p)
11
12  for i in range(len(poses))
13     setPose(poses[i]) # set the vehicle to pose poses[i]
14     image = captureImage()
15     steering = command[i]
16     data = (image,steering)
17     save(data) # save the data

```

Figure 10: Technique 2

One can see that we don't get the images at the first time but we obtain the pose of the vehicle first and then go through all the poses and then extract the images. We do this to facilitate the driving because when we extract an image in the simulator, the frame rate decreases and it can become difficult to drive the car.

SPLITTING THE TRAINING AND TESTING DATA

In any supervised learning algorithm we need to get some data to train our algorithm, but if we train it on the entirety of the data, we can make the model overfit and it will behave very badly on unseen data. To prevent this, we have to separate our data into two sets, namely, a training set and a testing set. We can then train our model on the training set and see if it overfits on the testing set. As we were in possession of only one environment we had to find a way to make these two sets. To do that, we drove the car and saved different positions of the car in the environment. The car can later be set into which positions using the API of the simulator. So different paths were used to obtain training and testing data. Hence we obtain the training and testing tracks. Figure 11 gives an example as to where we separate a track into two subsets for training and testing. In the end, we have gathered around 23 thousand images for training the network and around 4 thousand for testing it.

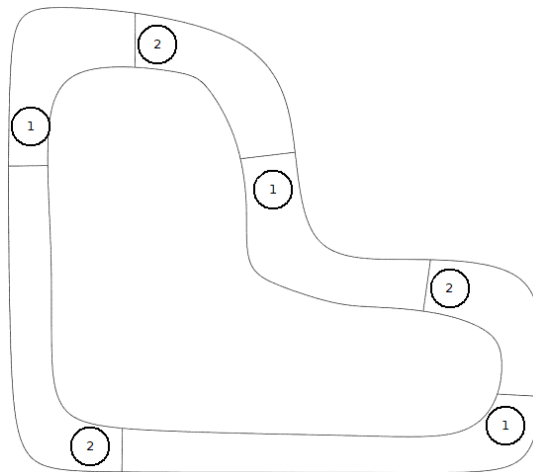


Figure 11 : split train/test

Experiments

For our first experiment, we tried to reconstruct a continuous control command from manual driving as explained before. The first problem was that we used the full architecture described in [8]. As we explained before, this network had more than 4 million parameters to tune so it overfitted rapidly, and we also needed a lot of training data. We then tried different ways to reconstruct the control command but at one point the vehicle was performing badly. It barely managed to navigate a single curve even after providing it with a lot of data.

Then we realized two things. The first thing is that the network used in [8] is used to navigate through a city with a lot more information than we have in our environment. For example road signs, pedestrians or other vehicles. So it was a good idea to reduce the size of the network to extract only the necessary features like the shape of the road or the slope of it. The second thing is that when we tried to reconstruct the control command, we had no idea if it was the command we used when we drove the car. So instead of continuing using this technique, we switched to a new technique.

Now instead of controlling the throttle and the steering of the vehicle, we resorted to just controlling the steering with a constant throttle. And to label the images we just considered three options: going straight, going left or going right. Which results in the steering only taking three values. It switches the problem from a regression problem to a classification problem. We also decided to change the architecture of our network and decrease its complexity. We then made different experiments to get the architecture we needed. The main problem was the overfitting. We were always able to get the network to fit the training data perfectly but then it behaved really badly for the testing data. And as soon as there was some drifting we were no longer able to follow the training data. So even on a training track where we gathered data for training the network, the vehicle behaved badly. One reason is that because we couldn't use a constant throttle when driving the vehicle manually, we needed at times, to increase the throttle to make the vehicle move. But with our keylogger, we couldn't log two keys at the same time. And sometimes when we needed to go left or right, we also had to accelerate and the logger just logged the acceleration and then labeled the image as "go straight" instead of "go left" or "go

right’. Because of this, there was a lot of incoherent data that were difficult to distinguish.

We decided then to use another technique to gather the data, the second technique that was described before, using the API and a joystick. Using this technique we switched again to a regression problem but it was much easier to gather the data, and the results were much better.

Fine tuning of hyperparameters

After we tried different architectures for the network we finally observed that the car was able to drive by itself satisfactorily, but to get the best results we needed to find a fine-tuning for the hyperparameters. The first parameter we had to set was the learning rate.

To find the best learning rate, we tried different learning rates and watched the effect on the training. We can see the different learning curves in figure 12 and each graph is also in Annex 1 with a better resolution.

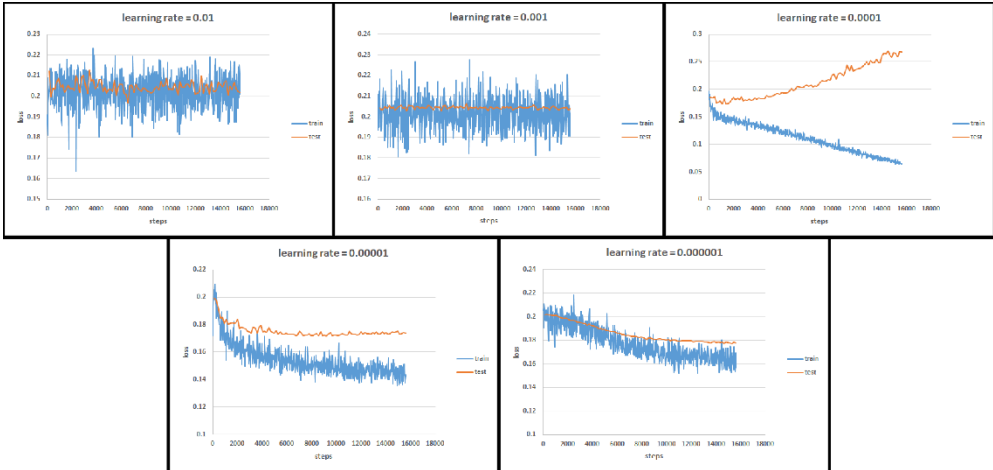


Figure 12: effect of learning rate onto learning process

We plot the value of the loss function for training and testing data (Y-axis) for different training steps (X-axis). We see that for a learning rate of 0.01 or 0.001 the network almost doesn't learn anything and for a learning rate of 10^{-6} , it learns really slowly. So we were left with the choices of a learning rate of 10^{-4} or 10^{-5} . Although, for the second one we observed that the network stops leaning after few epochs. But for a learning rate of 10^{-4} , even if it overfits - the network continues to learn. To choose a good learning rate we have to see what the minimum loss value for the testing data is. This is what we see in figure 13. We see that there is just a small difference between a learning rate of 10^{-4} and one of 10^{-5} . So, we decided upon the best learning rate being 10^{-4} .

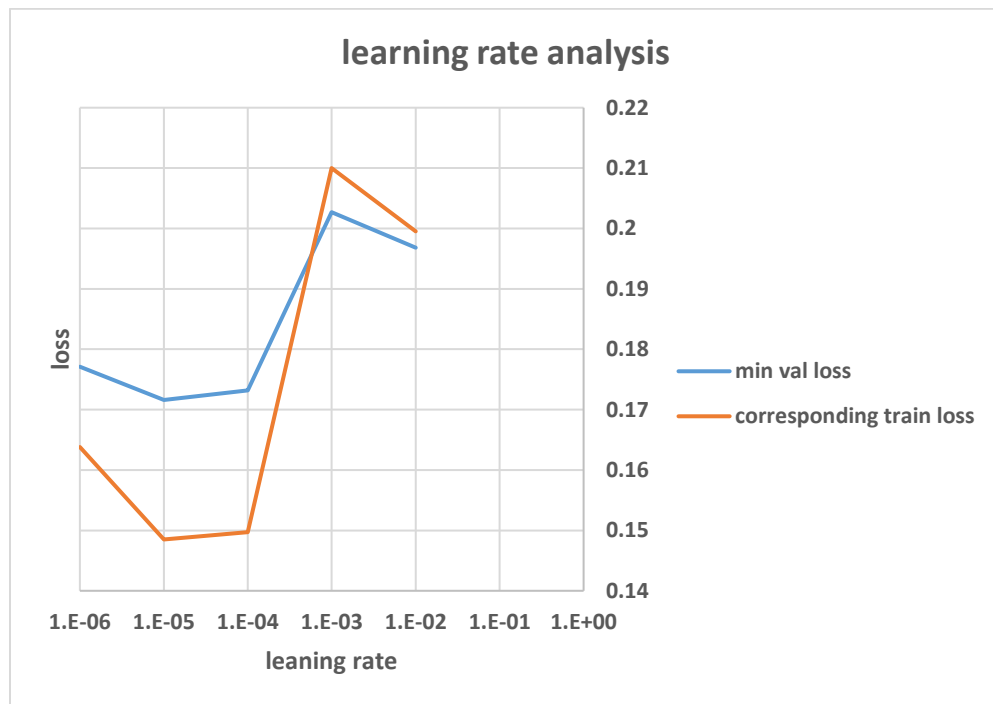


Figure 13: learning rate comparison

After choosing a good learning rate we needed to choose a good *keep_prob* parameter for the dropout layer. It is the parameter which decides if we keep a

neuron or not. We did the same experiment as before, we choose a learning rate of 10^{-4} and then change the *keep_prob* parameter and watch the effect on the learning process, as one can see on figure 14 and again each graph is on Annex 2 with a better resolution.

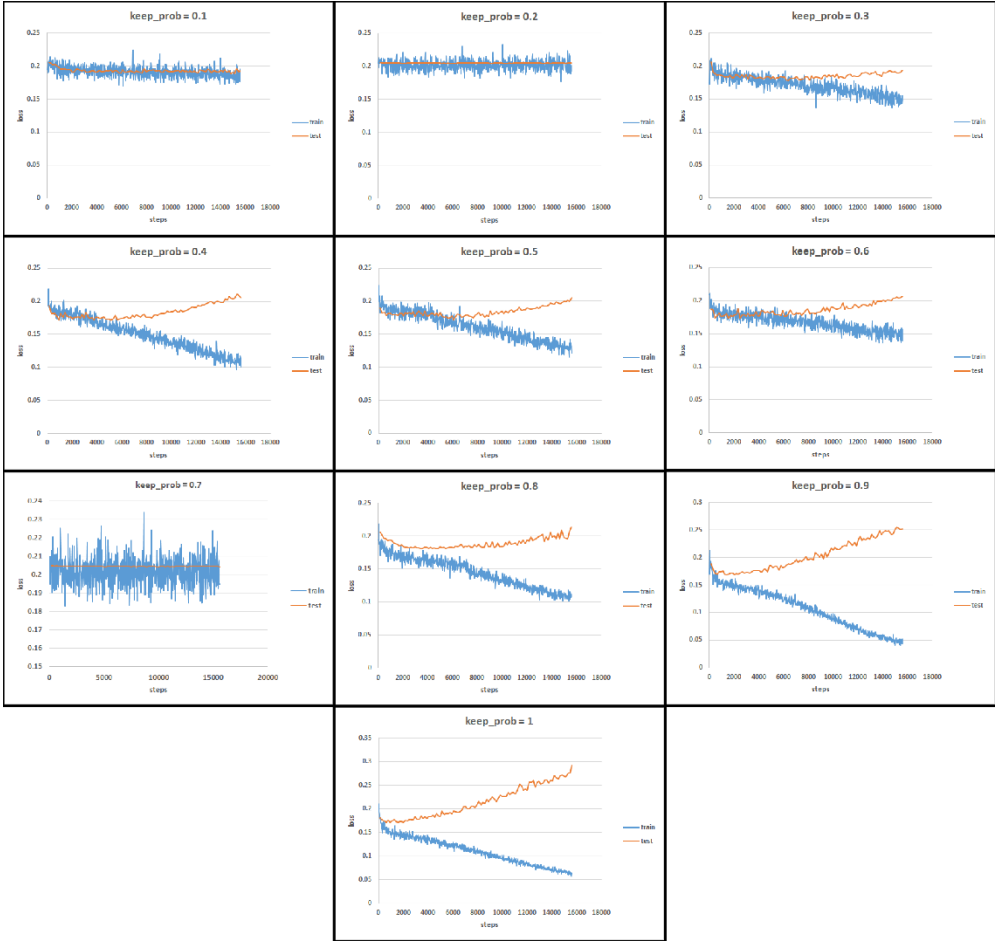


Figure 14: effect of *keep_prob* on the learning process

We see that if *keep_prob* is too slow, the network doesn't learn enough and if it is too big it overfits too much. The figure 15 is the same as figure 13 but for the *keep_prob* parameter. This helps us to choose the best parameter and here the best *keep_prob* seems to be 0.9.

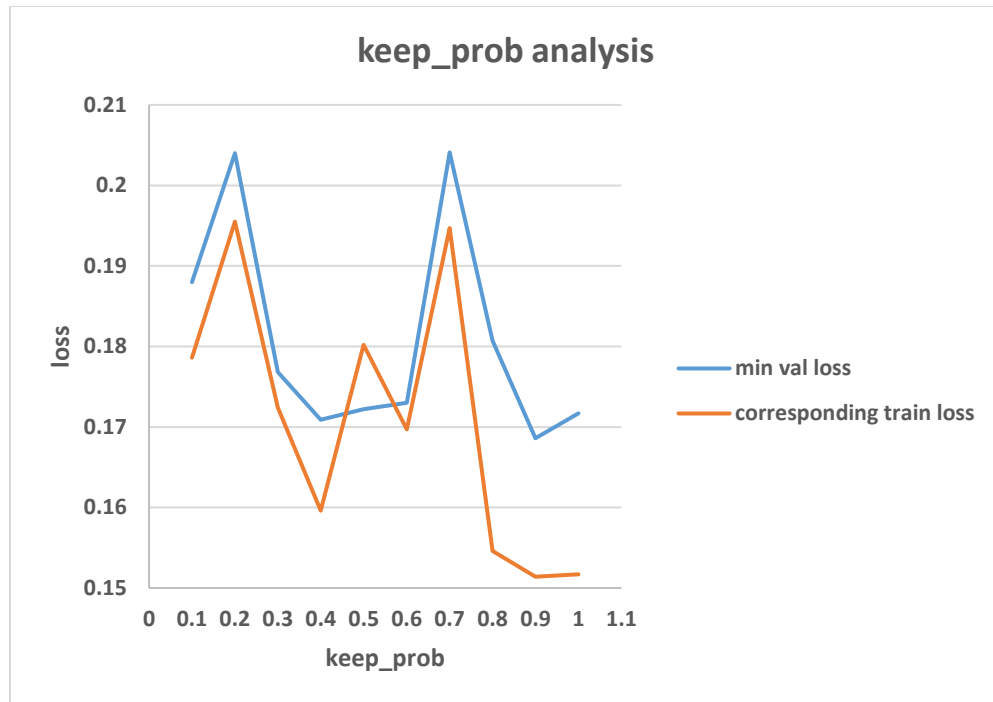


Figure 15: *keep_prob* comparison

Now we have these two parameters so we can train the network to get the most optimal result. The figure 16 shows the learning process with the loss value for training and testing data and the red line shows the best step to save our network.

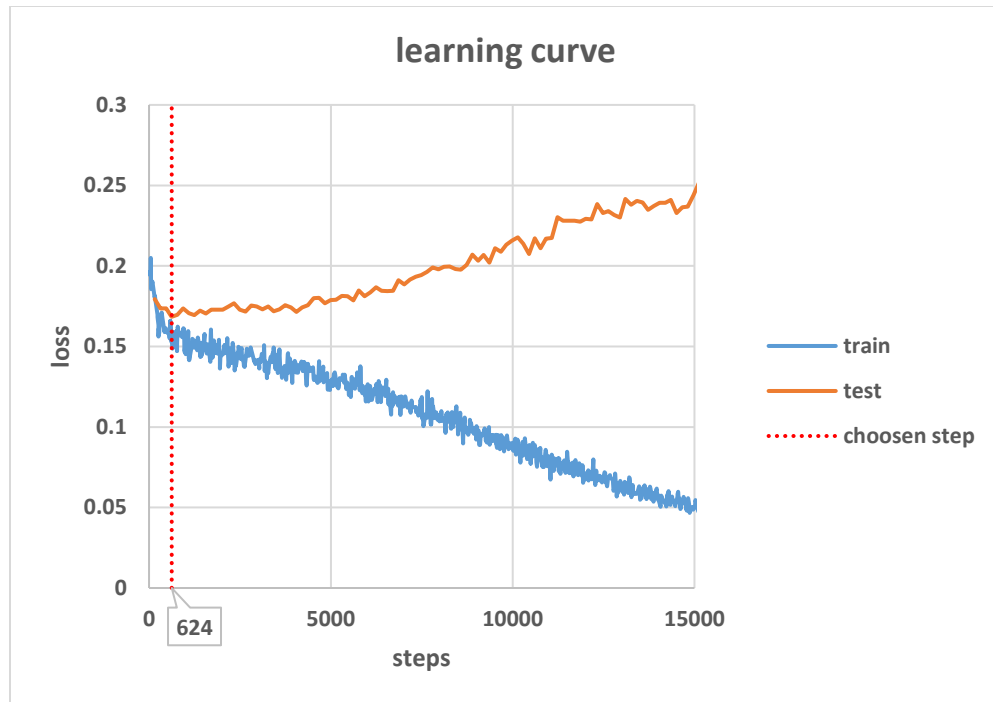


Figure 16: final learning curve

Results

So now we have a control policy for our car, with loss value of 0.1525 for training data and 0.1684 for testing data, we need to see how it behaves. In table 3 one can see the performance of our algorithm. To get these results we drove the car from different parts of the track for hundreds of episodes. The way to compute the distance of the car from the center of the road is explained in the next chapter. We see that the car stays mostly in the center of the road but as there are not a lot of

examples where the car is in a difficult situation, the network has some problems to manage these situations.

We also see that the car travels 1 km most of the time. We explained before that we separate the track into training and testing parts. Each part is around 400 m long. Since there is at least one curve in each part, we can be hopeful that our car is able to navigate through curves. And as a result of this, the car can drive for 3 minutes autonomously without crashing in most of the cases.

mean distance from the center of the road	≈8 pixels
mean distance traveled	≈1 km
mean time traveling	≈3 minutes

Table 3: results for imitation learning

CHAPTER 4

REINFORCEMENT LEARNING

In this chapter we will go through the second technique we have used to make the car autonomous. It is a reinforcement learning algorithm which tries to make the car learn how to drive by itself using trial and error experiments.

Principle

Reinforcement learning is a technique of artificial intelligence which was inspired by how animals can learn by trial and errors. For example, when people try to teach a dog to do some tricks like seating, they ask the dog to seat and when it seats they give him treats or caress, then the dog will know that when its owner asks him to sit it will be rewarded if it does. A connection in its brain will be made between the order to seat and the action to sit. This is how reinforcement learning works. For each action, the agent receives a reward or a penalty (the pain in our example) and then back propagates this error to be able to know what to do the next time it encounters this state. And we have used neural networks in this algorithm to generalize those situations.

In this project, we used a deep Q-learning algorithm. In this algorithm, the actions are discretized. In our case we have three actions:

1. Go left
2. Go straight
3. Go right

So we try to learn the following function

$$Q(s_t, a_t) : (s_t, a_t) \rightarrow a_{t+1}$$

Where:

- s_t state at time t
- a_t action taken at time t (straight, left, right)
- a_{t+1} action taken at time t+1

And this function is following the Bellman's equation described by (4.1)

$$(4.1) \quad Q(s_t, a_t) = r_t + \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

R e w a r d

For a reinforcement learning algorithm, the reward is the most important feature. Designing a good reward can dictate if our agent will learn what we want it to learn. Let's go back to the example we used before with the dog. If instead of treat or caress one reprimand it, the dog will never seat when its owner asks it to sit. Hence, we can see that using the wrong reward can change the behavior of our agent completely.

In this project, we decided to use the distance of the car from the center of the road as a reward. As we could not directly obtain the coordinates of the center of the road, we have used the semantic segmentation feature from the simulator. We first get a binary image where all the pixels of the road are 255 and the rest is 0. We used the equation (4.2) to compute the center of the road for each row of pixels, where:

- i is the row i
- j is the column j
- $P_{i,j}$ is the value of the pixel at row i and column j
- c_i is the center of the road for the row i

$$(4.2) \quad c_i = \frac{\sum_j j \cdot P_{i,j}}{\sum_j P_{i,j}}$$

Once we have the center of the road for each row of the image, we take only a few rows in front of the vehicle and compute their mean. As we know that our image has a size of 120x120, if the vehicle is in the center of the road, then this value should be 60. So the distance of the car from the center of the road is just $d = |C - 60|$ where C is the center of the road. So if d is greater than 30 it means that the car is starting to go out of the road and when d is 60 it means that the car is completely out of the road.

We now have to design a reward from this value. We could directly use $-d$ as a penalty but if we train it with different image sizes, this penalty can be higher or smaller. To prevent this we decided to clip the reward between 1 and -1 with the following formula (4.3) where:

- d is the distance from the center of the road
- d_{\max} is the maximum value d can take, here it is $\frac{120}{2} = 60$

$$(4.3) \quad r = 2 \cdot \left(\left| 1 - \frac{d}{d_{\max}} \right| - 0.5 \right)$$

We can see that if $d = \frac{d_{\max}}{2}$ which means that as the car starts to go out of the road the reward is 0 and if $d = 0$ which means the car is perfectly at the center of the road, the reward is 1 and finally if $d = d_{\max}$ which means that the car is completely out of the road, the reward is -1. So our agent should learn to stay as close as possible from the center of the road.

Algorithm

Here we will explain how to implement the reinforcement learning algorithm. For this project, we have used an already implemented algorithm [14] and changed it to fit our needs.

The first important thing to implement is the replay memory. The replay memory is a buffer containing experiences that the agent has encountered during the learning process. During this learning process we will save each tuple (s_t, a_t, r_t, s_{t+1}) in this replay memory where:

- s_t is the state at time t
- a_t is the action taken for state s_t
- r_t is the reward we get for taking action a_t for state s_t
- s_{t+1} is the state at time $t + 1$

After each episode, we will extract a random set of tuples from the replay memory and apply the Bellman equation to train the network. This replay memory makes the network more robust by learning from a bigger amount of examples and not only from the few examples it has encountered in the previous episode. To create the replay memory we used the class that has already been implemented.

Now we have to implement the algorithm itself. It is described by the pseudo code in figure 17. We can see that there is a probability epsilon that the agent takes a random action. It helps the agent discover new states during the training process.


```

1  for i in range(number of episodes)
2      s0 = extract 120x120 image from the simulator
3      while(episode is not done)
4          e = random number in [0,1]
5          if (e < epsilon)
6              a = random action
7          else
8              a = action predicted by the network for state s0
9              r = reward obtained for taking action a for state s0
10             s1 = extract 120x120 image from the simulator
11             save (s0,a,r,s1) in replay memory
12             if (vehicle collided with something or vehicle completely out of the road)
13                 episode is done
14         extract sample from replay memory
15         update Qvalues by computing the bellman's equation on those samples
16         train network with the updated Qvalue
17

```

Figure 17: Pseudo code for DQN

The network we used has the same architecture as in imitation learning. The difference is that it outputs a vector with three values and then we take the $\arg \max$ of this vector to decide if we wish to go straight, left or right. An example being: 0: left; 1: right, 2: straight. It is a classification problem, but instead of using the cross-entropy loss function we use the mean square as a loss function. This way we can directly train the value of the policy.

Finally, we have to update the Q value according to the bellman's equation, and for that, we have tried different techniques.

TECHNIQUE 1: SIMPLE DQN [15]

For the simple DQN, we use only one network. We first compute $Q(s_t, a_t)$ by feeding the state s_t into the network, we then get 3 values for each action (left, right, straight) and then we take the value corresponding to the action a_t . Secondly we compute $\max_{a_{t+1}} (Q(s_{t+1}, a_{t+1}))$ by feeding the state s_{t+1} into the network and by taking the $\arg \max$ of the 3 values we get. Which leaves us with just having to update $Q(s_t, a_t)$ according to the Bellman's equation.

TECHNIQUE 2: DOUBLE DQN [15]

This technique is similar to the previous one but we use two networks here. The main one, to compute $Q(s_t, a_t)$ and the second one - to compute $\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$. Here, we train only the first network and we update the weight of the second one to the values of the first one. This technique should help in preventing the network towards overestimating some states and hence makes the learning process more efficient.

TECHNIQUE 3: DOUBLE DUELING DQN [16]

This technique is the same as the second one but we have here, changed the architecture of our network. So for this technique, we split the network into two branches at the end. One branch is supposed to decide how good it is to stay in a given state, while the second one is supposed to decide how good an action is compared to the others. And finally, we combine these two branches at the end to output our Q values.

COMPARISON OF THESE TECHNIQUES

We tried these three techniques for thousands of episodes and observed the learning process.

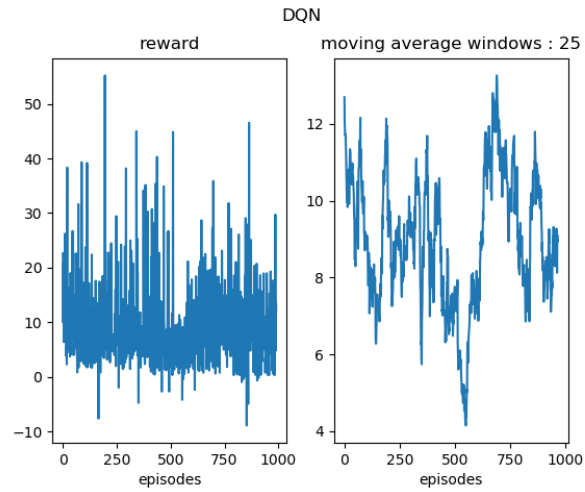


Figure 18: DQN learning process

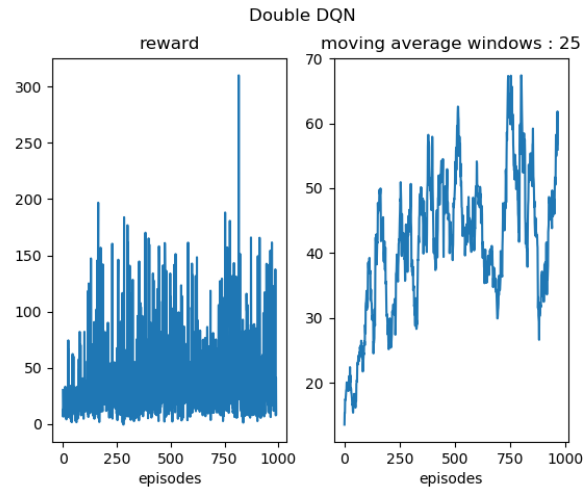


Figure 19: double DQN learning process

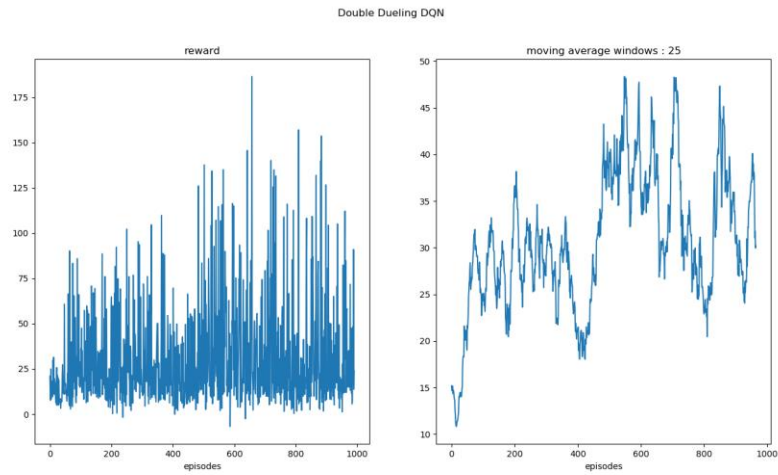


Figure 20: double dueling DQN learning process

In figures 18, 19, 20 we can see the learning process of these three techniques. For each figure, we can see the total reward we get after each episode (left) and the moving average of total reward with a window of 25 episodes (right). We observe this in order to ascertain if the network manages to learn something.

We see that simple DQN is not really efficient, it almost doesn't learn anything. The double dueling DQN is learning during the process, but it is not as efficient as the double DQN. So this is the technique we have decided to use for this project.

R e s u l t s

After training the network for 2000 episodes we get a policy and then evaluate it the same way as we did in the case of imitation learning. We can see these results in table 4.

What we can say about this result is that they could very certainly be better if we train our network for a longer time period. But the training process itself is very time-consuming. We will talk more about these results in the next chapter where we will compare both imitation learning and reinforcement learning.

mean distance from the center of the road	≈13 pixels
mean distance traveled	≈0.5 km
mean time traveling	≈2 minutes

Table 4: results for reinforcement learning

IMITATION LEARNING VS REINFORCEMENT LEARNING

Here we exhibit a comparison between the results for the imitation learning algorithm and the reinforcement learning approach.

Our first conclusion is that even though the reinforcement learning approach seems easier because there is no data to gather and moreover it takes less time- it is actually not the case. Firstly, when we started our experiments with the reinforcement learning algorithm we already had a good network architecture which made the problem easy. But since we don't provide our agent with an expert policy and it has to learn it by itself, it takes a lot of time to see the first results. And because it takes a lot of time to see some results, it is difficult to tune all the hyperparameters used in the algorithm.

	Imitation learning	Reinforcement learning
Distance from the center of the road	8 pixels	13 pixels
Distance traveled	1 km	0.5 km
Time of traveling	3 minutes	2 minutes
Total reward	100	63

Table 5: imitation learning vs reinforcement learning

In table 5 we see a comparison between imitation learning and reinforcement learning. We can readily observe that imitation learning is better but as we have said before, the results of reinforcement learning can be improved by going through more episodes.

First, we see that in reinforcement learning the controlled vehicle drives further from the center of the road than in the case of imitation learning. We can understand the reason behind this by looking at how the car is driven. For imitation learning, we use a continuous steering which is less affected by the frame rate. But for reinforcement learning, we use a binary steering which is more affected by the frame rate. It is the same situation with discrete-time systems in control theory, where a system can be unstable if the sample time is too large. This can be solved by an increase in the number of actions. But even if the reaction time is too high the reinforcement learning algorithm should be more robust, because it has seen more states where the car is in a difficult situation than in the imitation learning algorithm which leads us to conclude that if we train our network for a longer time it should be more suitable for such a task.

FUTURE WORK

In this chapter, we will see what can be done to improve our results.

Using semantic segmentation for imitation learning

The first idea we can think of is to use two different networks for the imitation learning technique.

The first network would be trained only on semantically segmented images. We saw that it is possible to get these kind of images in the simulator. And using only these images would make the task easier for the network as we already have extracted the important features of these images to drive the car i.e. the road position.

But because we wish to put the policy in a real car in the future, we need a second network to do the semantic segmentation. There is already a lot of semantic segmentation networks online that we can use for this purpose. But we still have to train those networks for our task. To do that we can combine the two networks, the one for driving the car and the other one for obtaining semantic segmentation, and train this big network as we did before using the pictures from a camera in front of the car. In the project when we had unacceptable results with imitation learning, we tried to use only the segmented images and the results looked promising but we didn't have the time to pursue this idea.

Adding some memory

In this project, every control action the neural network took, was made according to an image taken at some instant t . But we did not take into account the previous states. There was no memory of what happened before. So we can think of adding some memory by using a recurrent neural network to improve our results, in which way the actions would be taken not only according to what the vehicle sees but also with a consideration towards what happened before so that the controller may predict the inertia of the vehicle or the slope of the track. The difficulty lies in the fact that training a recurrent neural network is more difficult. To provide a workaround to this, we can first try to feed the network with a bunch of consecutive images instead of a single image, which is another way to provide some memory to the network.

Train also the throttle of the vehicle

In both the reinforcement learning and imitation learning approach, we trained only the steering of the vehicle, and we use a constant speed. Problems arise if for example, the slope is too high or if there is a tight curve. But if we also think about training the throttle of the vehicle resulting in decreasing it when we are about to take a tight curve or increase it if the slope is too high, we should see even more promising results.

CONCLUSION

In this paper we went through two different techniques to make a car autonomous, in an environment like a countryside track which is a little bit less structured than usual roads, using the power of artificial neural networks, and the realism of simulators.

We first had a look at the composition of the neural network we used. It was inspired by the network used in [8]. In the end, we changed the different layers to simplify it and make it fit our task better.

Then we looked at the different simulators we could choose and compared them in order to make the best choice. We chose the simulator Airsim developed by Microsoft which provide different environments and a wide range of sensors.

After that, we tried our first technique which was the imitation learning algorithm. It is a supervised learning technique where we had to first gather an expert policy, where we had to manually drive the car through the simulator while gathering the control command of the vehicle, and then we used this data to train a neural network to behave the same way as the trainer did. After some experiments to find the best architecture of this network we managed to make the car drive by itself for around 1 km in the simulator.

Finally, we tried the second technique which was the reinforcement learning algorithm. For this technique the car learned by itself, the strategy to drive through the simulator. It makes a lot of trial and error experiments and then learns from its mistake. For any action we have a reward or a penalty. And we back propagate this reward to find the appropriate action to take when the agent is in a given state. But this learning process is really time-consuming especially since we want our policy to converge to the optimal policy. After training it for 2000 episodes our car was able to drive for half a kilometer by itself. But these results can be improved by training it for a longer time.

We finally could compare these techniques and at this point, the imitation learning had better results, even though the reinforcement learning seemed more robust as it has seen more unusual states where the car is in difficult situations.

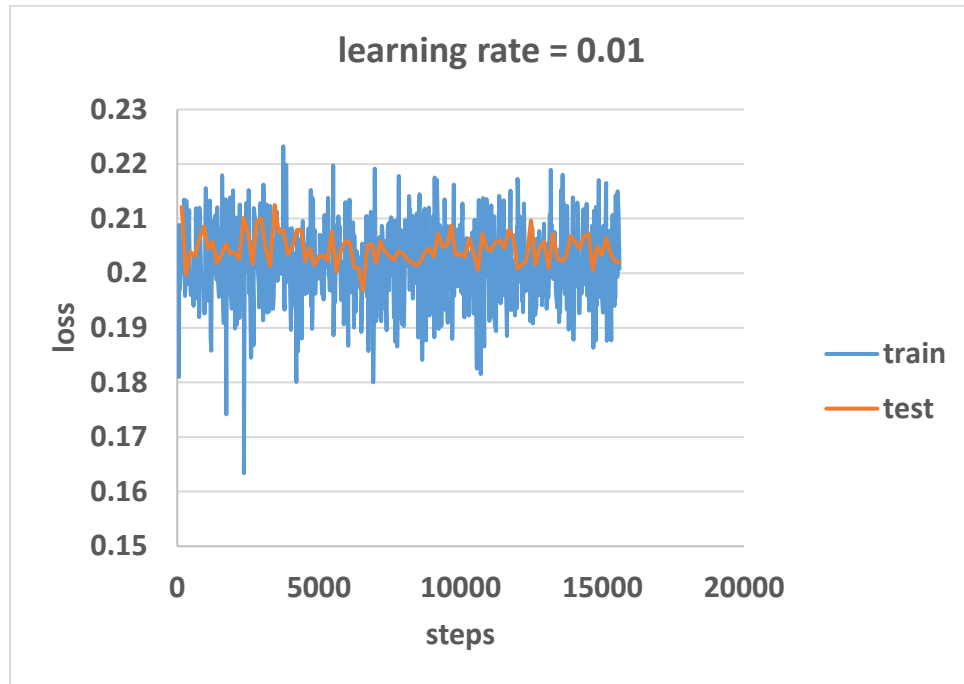
We have mentioned some ideas to improve these results. One of them being, for example, by adding some memory to our network to make it more capable of predicting the inertia of the car for example.

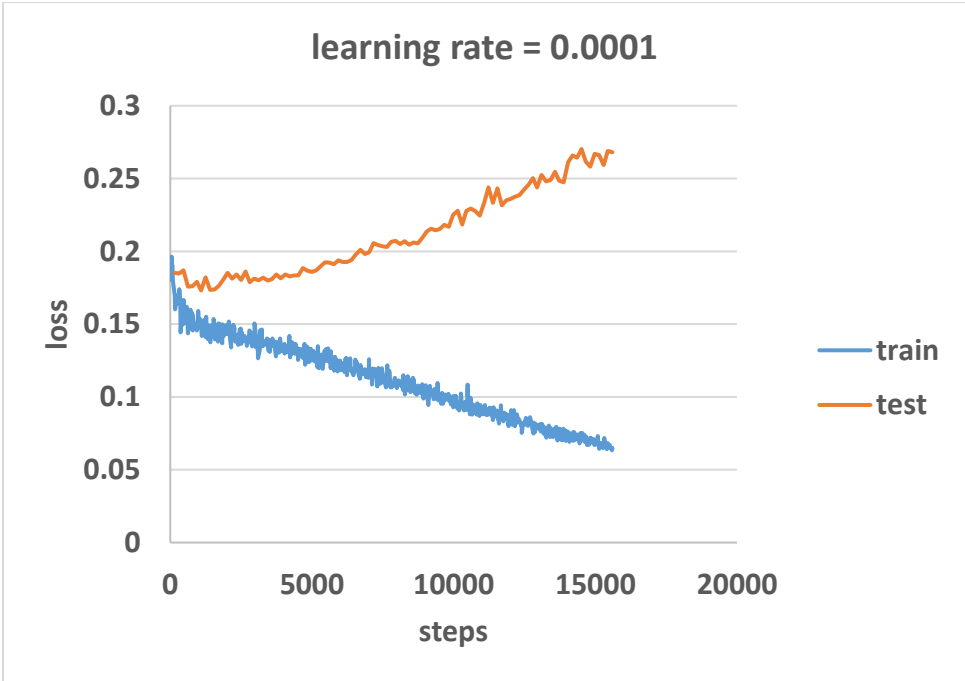
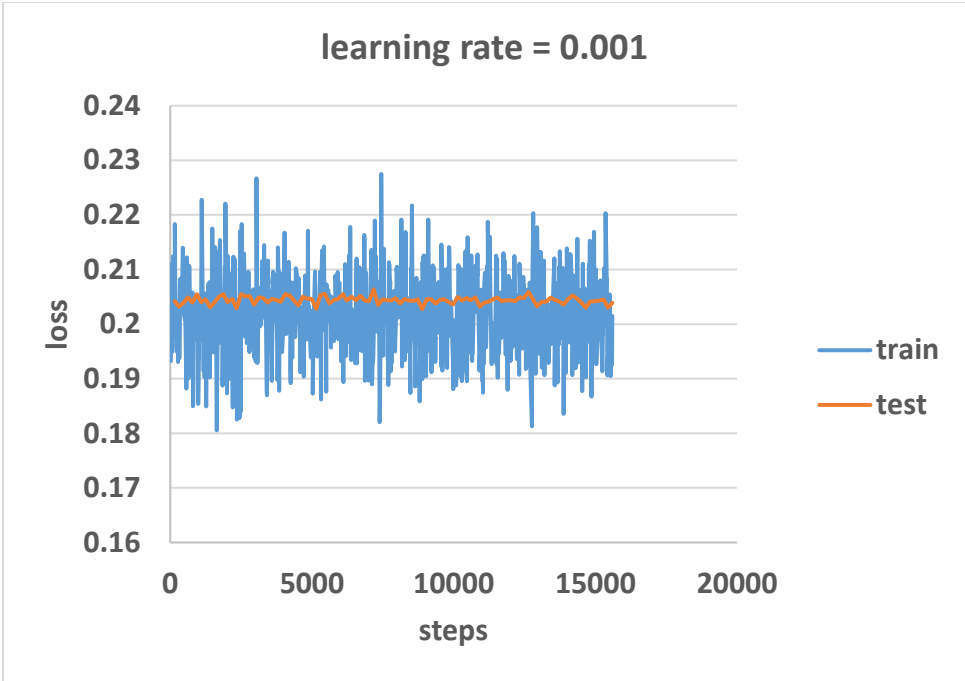
BIBLIOGRAPHY

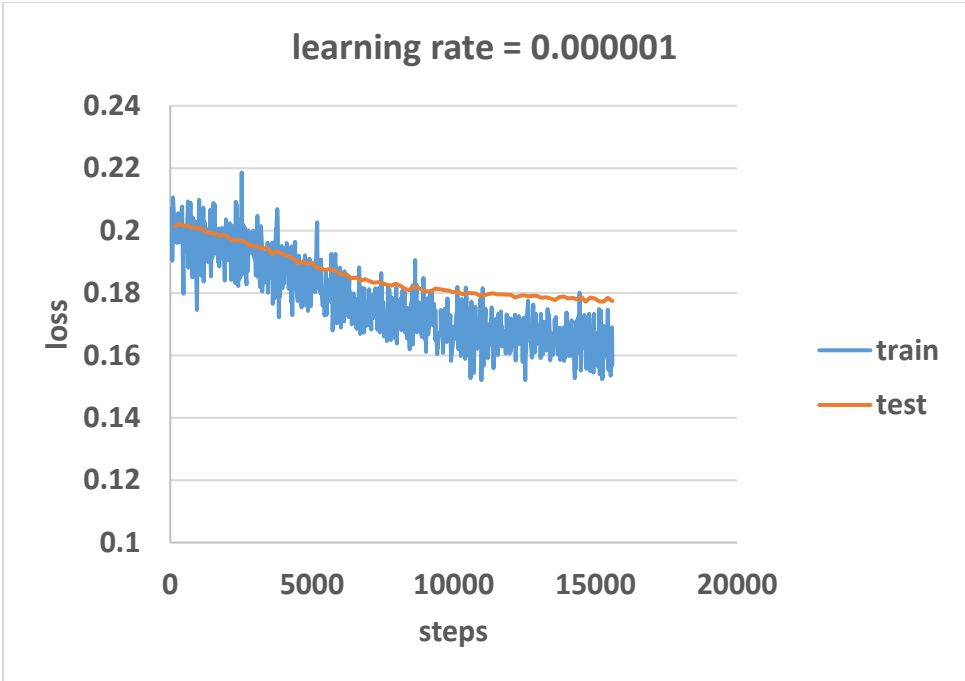
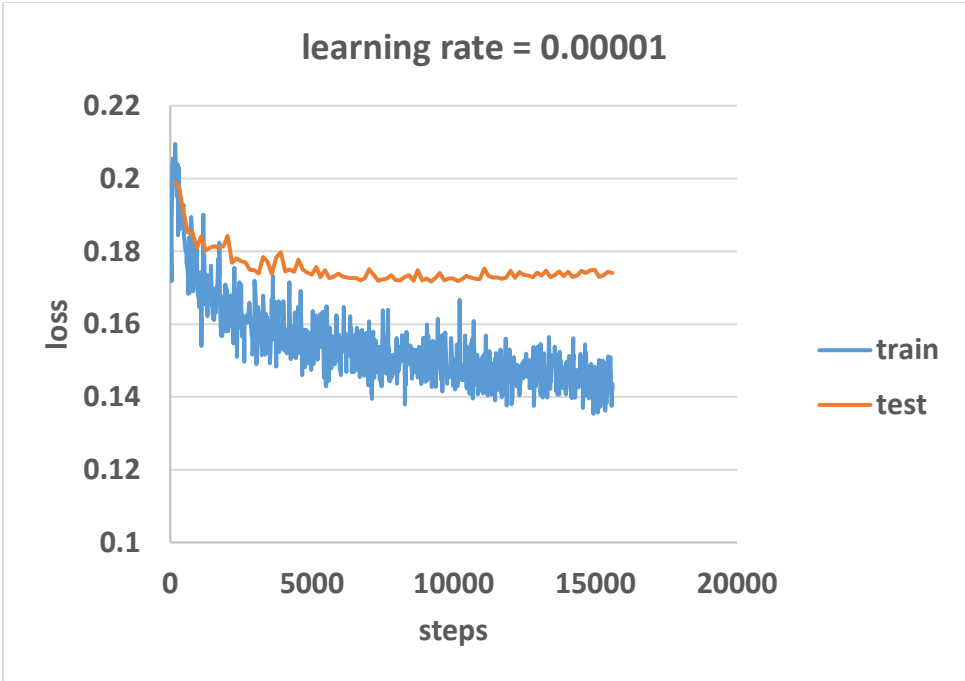
- [1] . M. Piotr, K. G. Matthew , . M. Mateusz, M. H. Karl , A. Keith , T. Denis , S. Karen , K. Koray , . Z. Andrew and H. Raia , "Learning to Navigate in Cities Without a Map," 2018.
- [2] H. Matteo , M. Joseph , v. H. Hado , S. Tom , O. Georg , D. Will , H. Dan , P. Bilal , A. Mohammad and S. David , "Rainbow: Combining Improvements in Deep Reinforcement Learning," 2017.
- [3] S. David , S. Julian , S. Karen , A. Ioannis , H. Aja , G. Arthur, H. Thomas , B. Lucas , L. Matthew , B. Adrian , C. Yutian , L. Timothy, H. Fan , S. Laurent , v. d. D. George , G. Thore and H. Demis , "Mastering the Game of Go without Human Knowledge," *Nature volume*, no. 550, 19 October 2017.
- [4] S. David, H. Aja , J. M. Chris , G. Arthur , S. Laurent , v. d. D. George , S. Julian , A. Ioannis , P. Veda , L. Marc , D. Sander , G. Dominik , N. John , K. Nal , S. Ilya , L. Timothy , L. Madeleine , K. Koray , G. Thore and H. Demis , "Mastering the game of Go with deep neural networks and tree search," *Nature volume*, no. 529, 28 January 2016.
- [5] C. Felipe , M. Matthias , . L. Antonio, K. Vladlen and D. Alexey , "End-to-end Driving via Conditional Imitation Learning," 2017.

- [6] . J.-R. Matthew, B. Charles , M. Rounak , N. S. Sharath , R. Karl and V. Ram , "Driving in the Matrix: Can Virtual Worlds Replace Human-Generated Annotations for Real World Tasks?," 25 Feb 2017.
- [7] S. Nitish , H. Geoffrey , K. Alex , S. Ilya and S. Ruslan , "Dropout: A Simple Way to Prevent Neural Networks from," 2014.
- [8] B. Mariusz, D. T. Davide , D. Daniel , F. Bernhard , F. Beat , G. Prasoon , . D. J. Lawrence, M. Mathew , M. Urs , Z. Jiakai , Z. Xin , Z. Jake and Z. Karol , "End to End Learning for Self-Driving Cars," 25 Apr 2016.
- [9] . P. K. Diederik and . B. Jimmy, "Adam: A Method for Stochastic Optimization," 2014.
- [10] "<https://www.tensorflow.org/>," [Online].
- [11] "<http://tflearn.org/>," [Online].
- [12] D. Alexey , R. German , C. Felipe , . L. Antonio and K. Vladlen , "CARLA: An Open Urban Driving Simulator," 2017.
- [13] S. Shital , D. Debadepta , L. Chris and . K. Ashish, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," 2017.
- [14] "<https://github.com/awjuliani/DeepRL-Agents/blob/master/Double-Dueling-DQN.ipynb>," [Online].
- [15] v. H. Hado , G. Arthur and S. David , "Deep Reinforcement Learning with Double Q-Learning".
- [16] W. Ziyu , S. Tom , H. Matteo , v. H. Hado , L. Marc and d. F. Nando , "Dueling Network Architectures for Deep Reinforcement Learning," 2016.

ANNEX 1: LEARNING RATE ANALYSIS







ANNEX 2: KEEP_PROB ANALYSIS

