



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

BACHELOR THESIS

Degree on Industrial Electronic Engineering and Automation

**STUDY OF THE IMPLEMENTATION OF AN AUTONOMOUS
DRIVING SYSTEM**



Report and Appendices

Author: Marta Basquens Muñoz

Director: Gerard Escudero Bakx

Convocatory: January 2020



Resum

Aquest treball pretén ser una guia introductòria al món de la intel·ligència artificial, aplicada més concretament al món de la conducció autònoma. L'aplicació final en la que es vol implementar tot el coneixement desenvolupat al llarg d'aquest treball no és genèrica, entenent com a tal un cotxe autònom que podria circular pel carrer. Per contra té una finalitat més concreta i menys complexa a nivell de seguretat: l'aplicació a un cotxe de competició de la Formula Student.

Al llarg del document s'explica tant la teoria necessària per aventurar-se en aquest món com també totes les eines necessàries per dur a terme l'entrenament d'una intel·ligència artificial capaç d'aprendre a conduir per ella mateixa. Així mateix, també es descriu tot el procés realitzat per trobar el model més òptim amb les eines utilitzades i comentaris per aprendre a interpretar els resultats.

Cal destacar que la realització d'aquest treball és merament una introducció a aquest món i que, per bé que els resultats obtinguts són esperançadors i possiblement serviran de base per a futurs desenvolupaments sobre el tema, no poden ser aplicats directament a l'aplicació final per falta de complexitat i diversitat de casos en l'entrenament del model.

Resumen

La intención de este trabajo es servir como una guía introductoria al mundo de la inteligencia artificial, aplicada más concretamente al mundo de la conducción autónoma. La aplicación final en la que se pretende implementar todo el conocimiento desarrollado a lo largo de la realización de este trabajo no es genérica, entendiéndose como tal un coche autónomo que podría circular por la calle. Por el contrario, tiene una finalidad más concreta y menos compleja a nivel de seguridad: aplicarse a un coche de competición de la Formula Student.

A lo largo del presente documento se explica tanto la teoría necesaria para adentrarse en este mundo como todas las herramientas necesarias para crear y entrenar una inteligencia artificial capaz de aprender a conducir por ella misma. De igual modo, también se describe todo el proceso realizado para encontrar el modelo más óptimo con las herramientas utilizadas y comentarios para aprender a interpretar los resultados obtenidos.

Cabe destacar que la realización de este trabajo es meramente una introducción a este mundo y que, bien que los resultados obtenidos son buenos y posiblemente servirán como base a futuros desarrollos en el tema, no puede ser aplicados directamente a la aplicación final por falta de complejidad y diversidad en los casos de entrenamiento del modelo.

Abstract

The main aim of this thesis is setting a base on the topic and serve as an introductory guide to the world of artificial intelligence, applied to the topic of autonomous driving. The final application in which the obtained knowledge is intended to be implemented is not a generic one, meaning an street autonomous driving car. Conversely, it has a much more particular and less complex in terms of security: being applied to a Formula Student competition autonomous racecar.

Along the document there are explained the introductory theory on the topic, as well as all the tools needed to create and train an artificial intelligence capable of learning to drive by itself. In the same way, all the processes done in order to find the most optimal model are explained along with all the results commented and interpreted.

It is to be said that the materialization of this thesis is just an introduction to the topic and, although the obtained results are favorable and could possibly be used as a base for further development, they can not be applied directly to the final application due to lack of complexity and diversity on the training cases.



Glossary

Formula Student – Competition between university students all over the world that design, build, develop and compete a single seater car.

AI – Artificial Intelligence

ANN – Artificial Neural Network

SAE – Society of Automotive Engineers

CNN or ConvNet – Convolutional Neural Network

ReLU – Rectifier Linear Unit

RL – Reinforcement Learning

Index

RESUM	I
RESUMEN	II
ABSTRACT	III
GLOSSARY	V
1. PREFACE	1
1.1. Origin.....	1
1.2. Motivation	1
1.3. Previous requirements	2
2. INTRODUCTION	3
2.1. Goals	3
2.2. Scope.....	3
3. INTRODUCTION TO ARTIFICIAL INTELLIGENCE	5
3.1. Types of Artificial Intelligence	5
3.1.1. Reactive machines	5
3.1.2. Limited memory.....	5
3.1.3. Theory of mind.....	6
3.1.4. Self-awareness	6
3.2. Machine learning	6
3.2.1. Types of machine learning.....	6
3.3. Artificial Neural Networks	8
3.3.1. Basic structure.....	9
3.3.2. Activation functions	11
3.3.3. Neural Networks	13
4. AI APPLIED TO AUTONOMOUS DRIVING	16
4.1. Introduction	16
4.2. Levels of autonomy on cars.....	16
4.3. Case of study.....	17
5. CONVOLUTIONAL NEURAL NETWORKS	18
5.1. Basic structure	18
5.2. Types of layers	19

5.2.1.	Convolutional layer.....	19
5.2.2.	Normalization/Non-linearity layer	23
5.2.3.	Pooling layer	23
5.2.4.	Fully connected layer.....	24
6.	DEVELOPMENT ENVIRONMENT SETUP	26
6.1.	Requirements.....	26
6.1.1.	Development machine characteristics.....	26
6.1.2.	NVIDIA GPU card	26
6.2.	Development programs	29
6.2.1.	Anaconda	29
6.2.2.	Python.....	30
6.2.3.	Spyder	32
6.2.4.	Visual Studio and Visual Studio Code.....	33
6.3.	Development libraries.....	34
6.3.1.	General software requirements.....	34
6.3.2.	Installation observations	35
6.3.3.	CUDA.....	35
6.3.4.	cuDNN.....	37
6.3.5.	Tensorflow	40
6.3.6.	Keras	41
6.3.7.	Opencv.....	42
7.	CARLA SIMULATOR	43
7.1.	Introduction	43
7.2.	Features.....	44
7.3.	Preparing the simulator	44
7.3.1.	Prerequisites.....	44
7.3.2.	CARLA Installation	45
7.3.3.	Dependencies	48
7.4.	CARLA execution	49
7.4.1.	Changing the port and enabling server mode	49
7.4.2.	Adjusting the graphics quality	50
7.4.3.	Choosing the map.....	50
8.	MODEL TRAINING	51
8.1.	Training environment	51
8.1.1.	Initial considerations	51

8.1.2. Racetrack.....	51
8.2. Databases.....	52
8.2.1. Learning methodology.....	52
8.2.2. Reinforcement Learning	53
8.3. Training process.....	58
8.3.1. Reinforcement learning implementation.....	58
8.3.2. Previous training test cases	65
8.3.3. Final trained model.....	117
8.4. Conclusions	123
9. MODEL TESTING	124
9.1. Testing environment.....	124
9.1.1. Initial considerations.....	124
9.1.2. Racetracks	125
9.2. Testing process	127
9.3. General performance of the model	128
9.3.1. Testing.....	128
9.3.2. Training vs Testing.....	131
9.4. Conclusions	135
10. ENVIRONMENTAL IMPACT	137
CONCLUSIONS	139
ECONOMIC ANALYSIS AND MATERIAL BUDGET	141
BIBLIOGRAPHY	143
APPENDICE A	147
A1. Tested models.....	147
A1.1 Comparative between models.....	148
A1.1.1 Map02	148
A1.1.2 Map04	153
A1.1.3 Map07	158
A1.2 Comparative between maps.....	164
A1.2.1 2 Conv Layers of 16 neurons each	164
A1.2.2 2 Conv Layers of 32 neurons each	166
A1.2.3 2 Conv Layers of 64 neurons each	168
A1.2.4 3 Conv Layers of 16 neurons each	170

A1.2.5 3 Conv Layers of 32 neurons each.....	173
A1.2.6 3 Conv Layers of 64 neurons each.....	175
A1.3 Final conclusion	177

1. Preface

1.1. Origin

This thesis arises from the Formula Student competition (also known as Formula SAE), an International competition where teams composed by university students all around the globe demonstrate their skills by designing, modeling, manufacturing and assembling a race car. Among other characteristics, this competition specially promotes excellence, as each team competes with its own developed car to be the best one.

Each team can choose whether to apply for any of the 3 possible categories this competition supports, which are the following:

- Driven combustion race car
- Driven electrical race car
- Driverless combustion/electrical race car

As having been part of one Formula Student's team competing in the driven electrical race car category and initiating the will of certain team members to evolve to a driverless car, I decided to get into the field of Deep Learning to start investigating the dimension of the resources needed to solve that problem.

1.2. Motivation

An official communicate from Formula Student Germany (FSG) stated that in the incoming years this specific competition will hold one or more dynamic events where the car will need to be fully driverless, even in a not driverless category.

Although the team will probably not be competing in FSG, this rule change is likely to be spread among other competitions and so, the team will have to be prepared to incorporate this kind of technology in a near future.

All the results obtained in this report will hopefully settle a solid base, a first introduction to the autonomous driving technology, that will allow the future team to be able to compete in a driverless event sometime.

1.3. Previous requirements

Although the problem of developing a driverless car is isolated from the team itself, this topic would not even have arisen if the cars from previous seasons didn't reach a certain stable basis, that is, they didn't have major problems in design and implementation. With more serious problems in mind, nobody raises to do further investigation in topics that are not compatible with the current state of the single seater race car.

Referring to non-competitive topics, there are no strict previous requirements without which it is not possible to start on this topic. Of course, with some previous knowledge about how artificial intelligence and neural networks work things can be done relatively straighter forward but everything can be learned while on the development, just learning as the same time your artificial intelligence does.

However, it has to be pointed out that the interest in this field and a clearer vision on how to introduce this type of problem to artificial intelligence came after taking part in the optative subject "Artificial intelligence applied to engineering".

2. Introduction

2.1. Goals

The main aim of this report is to establish the basis on how to do a prototype of an autonomous driving race car, both in a hardware level (inputs needed, meaning sensors) and in a software level (neural network). The development will be done in a specific environment emulating the real conditions where the final version of this prototype will have to be implemented.

Focusing on the final application, a race car able to take part into any Formula SAE competition, the problem is severely simplified compared to a real autonomous driving car that would be driving around a city or even in a highway, where there exist many other stimuli and also safety is a key element. In the application that concerns us, safety is not that important since the race car will be driving alone in the track, no pedestrians or other cars will interfere with it. And so, the problem can be reduced and focused only on driving well and as fast as possible.

In a hardware level, the following topics will be studied:

- Sensors and actuators needed for input and output data

In a software level, the following topics will be studied:

- Quantity of data required to train the neural network
- Time required to train the model
- Neural network design and implementation
- Different options to approach the problem
- Efficiency of the programming languages used and compatibility with the final application

2.2. Scope

Developing a full study about autonomous driving on city streets would add a little bit more degrees of complexity to the problem, as in those situations several aspects referring to security need to be accurately studied. This added complexity could not be transmitted to the final application where this study is aimed to be implemented so it will be pointless to perform the development thinking about driving on a city. For this reason, the problem has been bounded to the following:

- The problem will be approached from the perspective of the final application, although simplified. The resultant model will have the possibility to be scalable to the final application.
- The datasets used will be extracted from a special simulator, where the conditions for the selected approach will be used.
- Due to lack of time to train the neural network until performing without errors, it will be trained until its accuracy is higher than an acceptable limit.
- No real implementation (physical implementation) of the car will be performed.

3. Introduction to Artificial Intelligence

Artificial Intelligence is a branch of computer science that aims to simulate human intelligence processes by machines (1). This process includes learning, reasoning and self-correction and can be applied to an infinity of fields such as decision making, object classification and detection or speech recognition and translation.

AI can be categorized in either weak or strong. On one hand, weak AI (also known as Artificial narrow intelligence) is designed to resolve a particular problem, which means that their task is very specific and working out of its comfort zone would not grant any correct function. For example, virtual personal assistants are classified in this group. On the other hand, strong AI (also known as Artificial general intelligence) is a system with generalized human cognitive abilities. As opposed to weak AI, this kind of AI is able to find a solution to a wide range of problems even when not trained specifically to do so.

3.1. Types of Artificial Intelligence

Although AI include some subsets which are explained in the following sections, there are four main types of AI according to its performance and applications (2).

3.1.1. Reactive machines

Those are the most basic types of AI which can only react to current situations, generating a “reaction”. They can’t use memories or already learned experiences to influence the decisions taken, they are only programmed for the “here” and “now” situations, not for a “before” and “after”.

They have no concept of its surrounding world and therefore they can only work for the simple function they are programmed. A characteristic of this type of AI is that they will always behave the same, as they do not learn from their experiences.

An example of reactive machine is Deep Blue, a chess-playing super computer created by IBM in the mid-1980s.

3.1.2. Limited memory

This type of AI is comprised by machine learning models that use previously learned data by observation to build experimental knowledge. Although they are more advanced than reactive machines, they already have limitations since they depend on a combination of observational and pre-programmed knowledge.

3.1.3. Theory of mind

Currently this is the most advanced existing form of AI which have decision making ability equal to a human being. Although there are already machines that exhibit a humanlike capability, none of them are fully capable of holding conversations on a human standard level, introducing emotional capacities.

Social interactions are the key point where yet there is still difference between AI and humans, so to make theory of mind machines tangible they would need to identify, understand, retain and remember emotional behaviors while knowing how to respond to them.

3.1.4. Self-awareness

This last level of AI involves machines that have human consciousness, evolved theory of mind machines. This kind of AI doesn't exist yet, but it would be so difficult to create them since they would need to have the capability not only to recognize and replicate humanlike behaviors, but also to think for themselves, understand their own feeling and have desires.

3.2. Machine learning

Machine learning is a science derived from AI that creates systems that are able to learn by themselves as if they were humans: they are able to identify some patterns in a huge volume of data. Those machines make use of algorithms that by analyzing the data have the ability to predict future behaviors. They are always on a constant learning since they accumulate their knowledge over time in an automatic way, without human intervention.

3.2.1. Types of machine learning

According to which method is used to learn, machine learning machines can be divided into three different categories (3).

3.2.1.1. Supervised learning

Within this category the desired output of the network is also provided with the input while training it. By doing so, the network is able to calculate an error based on its target and actual output.

As seen in image below, the supervisor is the responsible of giving the algorithm as many information as it wants to be reflected on the final output.

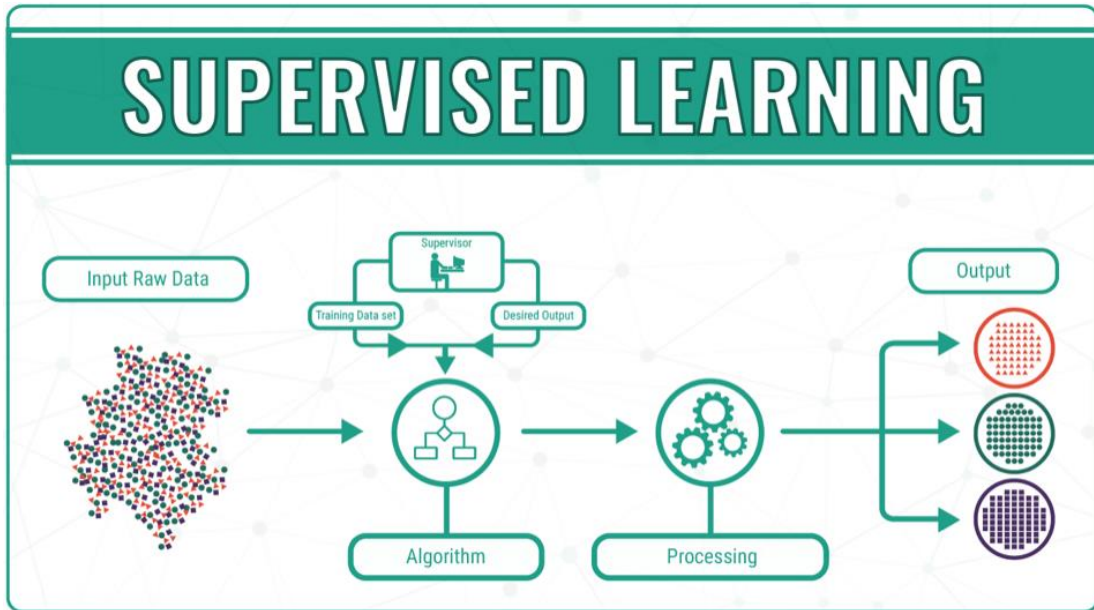


Figure 3.1. Supervised learning scheme. (Source: (4))

This type of learning has mainly two applications:

- Classification: output classes are predicted. All the possible final classification groups are already included in the training dataset.
- Regression: models a target prediction value based on independent variables. Mainly used for finding out the relationship between variables and predictions.

3.2.1.2. Unsupervised learning

In this case the neural network is only given a set of inputs and is the network itself the responsible to find some kind of pattern within the inputs given. This learning doesn't have output labels so it uses the given dataset without any instruction on what to do with it. (4)

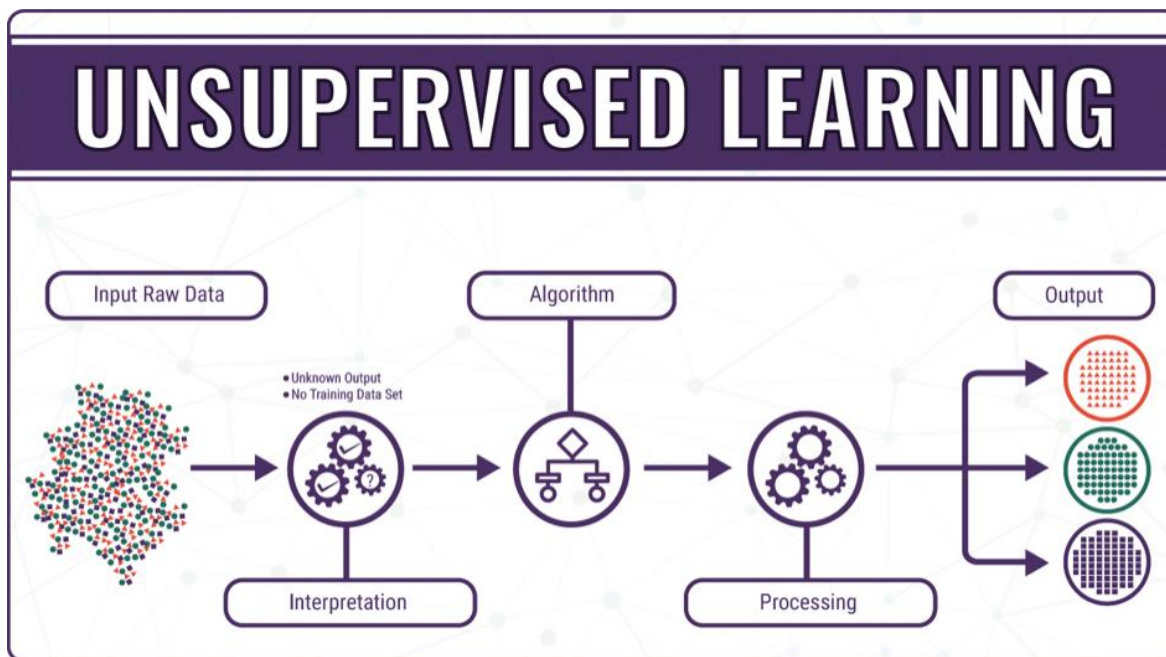


Figure 3.2. Unsupervised learning scheme. (Source: (4))

3.2.1.3. Reinforcement learning

Reinforcement learning is a little bit similar to supervised learning, as the input and its feedbacks are given to the network but instead of having a target output a reward is given depending on how good the system is performing. The main objective of the network in this case is to maximize the output reward received through trial and error.

This type of learning is the most similar to how humans and animals learn.

3.3. Artificial Neural Networks

Artificial Neural Networks (ANN) are computational models inspired by the biological neural networks in the human brain. (5)

Speaking in biological terms, the basic computational unit of the brain is a neuron. The pictures below show (on the left) a drawing of a biological neuron and a common mathematical neuron model (on the right).

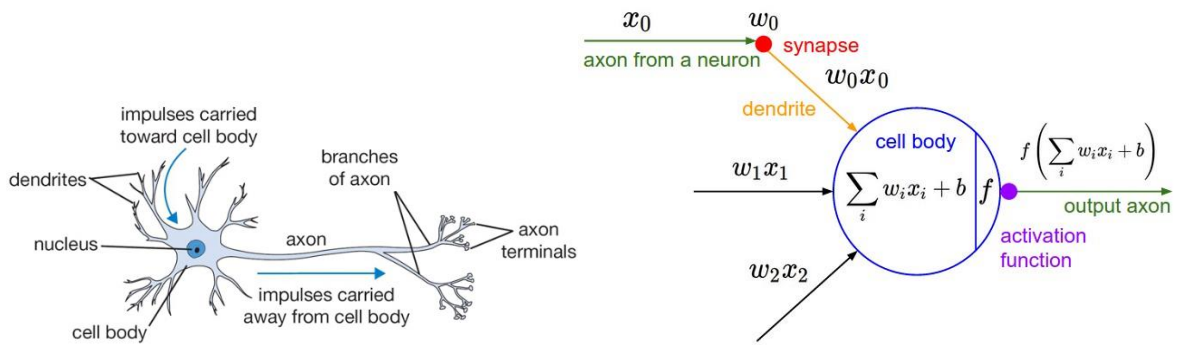


Figure 3.1. Comparison of a biological neuron (left) and its mathematical model (right). (Source: (6))

Each neuron receives input signals and produces output signals along its output channel (axon). This output channel is able to connect with other neurons, who receive the outputs of the previous one as an input.

In the computational model of a neuron, the output signals interact multiplicatively (a certain weight given to the interaction multiplied by the output signal itself) with the input of other neurons based on the strength of that signal. The main idea is that those strengths (weights) are learnable by the network and control the strength of influence of one neuron to another.

On the biological model, the dendrites (input) carry the signal to the body cell where they all get summed. If this sum is higher than a certain threshold, the neuron is fired, sending a spike along its axon (output). On the mathematical model, it is assumed that the duration of the spikes is not from big interest, but it is the frequency of the firings. Based on this rate, it is possible to model the firing rate of the neuron with an activation function, which represents the frequency of the output signal.

3.3.1. Basic structure

Setting aside the biological neuron and getting in depth on the computational model, its basic representation would be, again, the following:

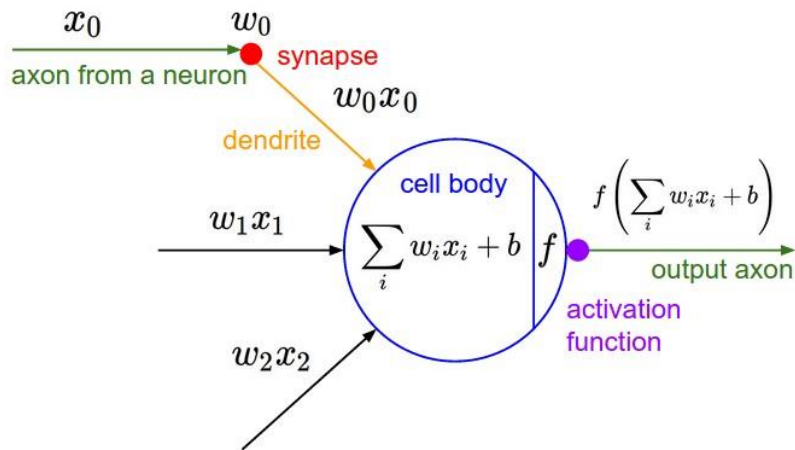


Figure 3.2. Detail of a neuron mathematical model. (Source: (6))

This neuron (also called node or unit in this field) is the basic unit of a neural network. Each neuron receives one or multiple inputs of other nodes and computes an output. Every input received has an associated weight (w), which is assigned based on the relative importance of the other inputs. The neuron internally computes a function which is the weighted sum of all the inputs it receives. Later an f function is applied to that weighted sum in order to introduce a non-linearity to the result. This function is called the Activation Function and its main purpose is, as already said, introducing non-linearity into the output of a neuron in order to learn non-linear representations are often occur on real world data.

In a mathematical equation the neuron is performing the following

$$W(i) = W(i) + a * (T - A) * P(i) \quad (\text{Eq. 3.1})$$

where

$W(i)$ is the weight vector

$P(i)$ is the input vector

a is the learning rate

T is the correct output for a given input

A is the actual output given by the neuron

When an entire pass through all the input training vectors is completed successfully without any deviation between T and A, the neuron has learnt. At this time, if an input vector that was already in the training set is given to the neuron, it will output the correct value. However, if it is given an input vector that was not in the training set, it will respond with an output similar to other training vectors close to the input given.

With this methodology, the neuron is actually adding all the inputs and separating them into 2 categories, those that will cause the neuron to fire and those that won't. In fact, that is drawing a line with a certain threshold. However, not all input vectors can be classified and divided into two categories so easily (not linearly separable) and that is the main limitation of a single neuron. For this reason, the most common and efficient form of performing artificial intelligent operations is with a succession of neurons: the called multi-layered neural networks.

3.3.2. Activation functions

Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions that may encountered in practice, which are the followings:

3.3.2.1. Step function

A step function is defined as follows

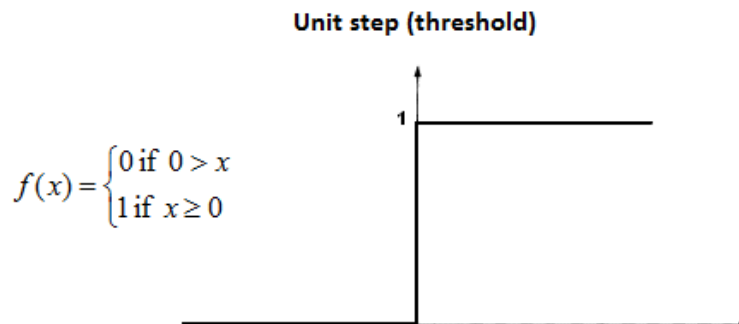


Figure 3.3. Graphical representation of a step function. (Source: (8))

This activation function is very simple and useful for simple networks. However, it has a main problem: it is not differentiable at zero. This is very critic with neural networks that need to perform operations with non-linear data, as they make use of gradient descent approach.

3.3.2.2. Sigmoid function

A sigmoid function or logistic function is defined as follows:

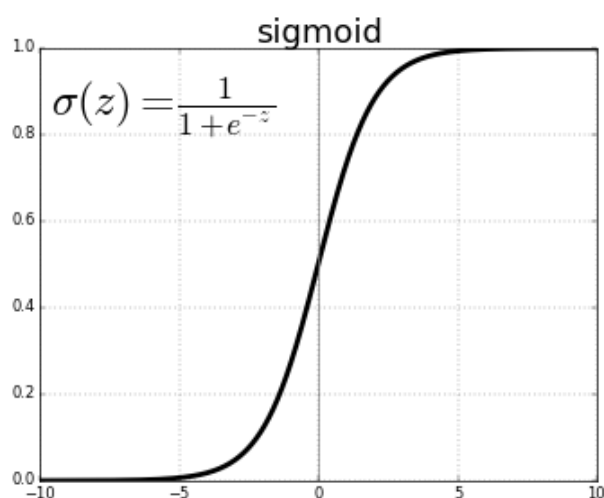


Figure 3.4. Graphical representation of a sigmoid function. (Source: (8))

This function comes to a necessity of correcting the deficiency in the step function. It is able to capture the non-linearity of the data, as it is differentiable in every point. However, this type of function has also a con: it also suffers from a problem of vanishing gradients. This means that its input is reduced to a very small output range $[0,1]$: even a large change in the input produces a very small change on the output. Moreover, this problem grows with an increase in the number of layers used, stagnating the learning of the neural network at a certain level.

3.3.2.3. Tanh function

The tanh function is a reshaped version of the sigmoid function. It has the same curvature but its output range is much higher from $[-1,1]$.

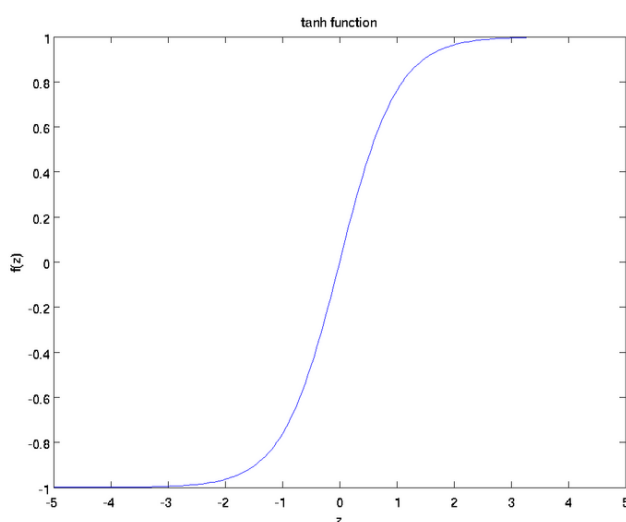


Figure 3.5. Graphical representation of a tanh function. (Source: (8))

The improvement this function represents with respect to sigmoid function is the fact that here since the data is centered around 0, its derivatives are higher. A higher gradient helps with a better learning rate. However, the problem of vanishing gradients is still not solved.

3.3.2.4. ReLU function

The Rectified Linear Unit function is the most used in deep learning models. The function returns a 0 if receives some negative input but for any positive input, it returns the value back.

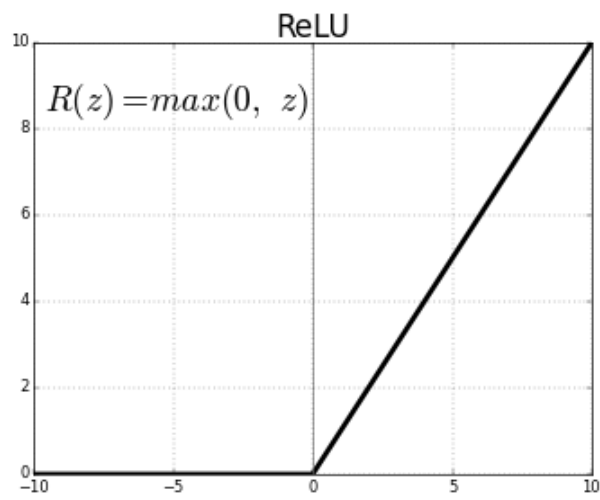


Figure 3.6. Graphical representation of a ReLU function. (Source: (8))

In general practice, this type of activation function has been found to perform better than any of the others.

3.3.3. Neural Networks

A Neural Network consists of an association of neurons arranged in layers. A layer is the union of several individual neurons to form a more stable and computationally higher unit.

The basic representation of a neural network is shown in the following picture.

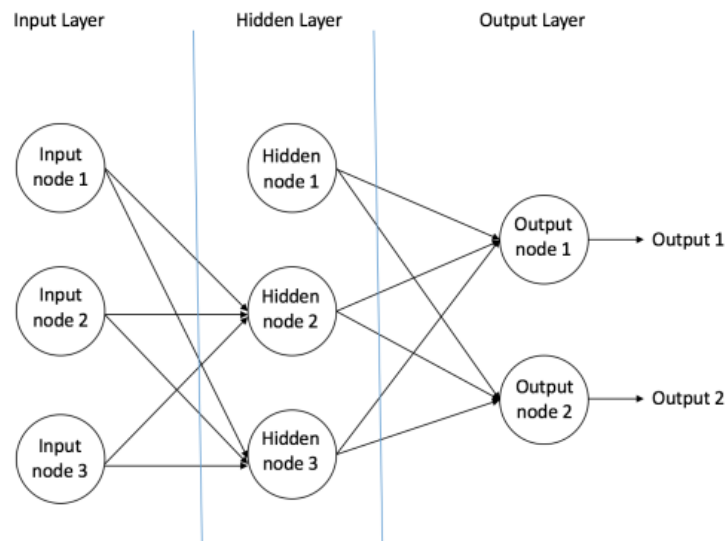


Figure 3.7. Basic representation of a Neural Network with 3 types of layers. (Source: (9))

Note that nodes from adjacent layers have connections between them, all of them with their associated weights previously explained.

There exist three types of layers:

- Input layer – provides information from the outside world to the network. No computation is performed here, it just passes the information through the following layer.
- Hidden layer – has no connection with the outside world (from here derives its name, as it is hidden from the outside). This layer performs computations with the data from the previous layer and transfer the results to the output layer.
- Output layer – perform the final computations and transfer this information to the outside world.

Any neural network has, at least 2 types of layers: 1 input layer and 1 output layer. The number of hidden layers differ between different networks depending on the complexity of the problem that needs to be solved. Generally speaking, the harder the problem, the higher the number of hidden layers used. A neural network having more than one hidden layer is generally referred to as Deep Neural Network.

An important point to note is that every layer is independent to the other. Although it may, for instance, be 3 hidden layers, they can perform different operations with also different activation functions. The choice of the activation function always depends on the problem in question and the data being used as has already been explained above.

There are, however, different types of Neural Networks according to the methodology used to make predictions.

3.3.3.1. Feedforward Neural Network (Multi layered Neural Network)

This is the first and simplest case of a Neural Network in which the input information moves only in one direction: forward. This means that no loops or cycles are performed in the network and there is no correction about the used weights.

3.3.3.2. Back-Propagation Neural Network (10)

Evolving from the feedforward neural network, back-propagation neural network performs a series of iterations through all the network. The information goes forward, backward and repeats this cycle until it has learned.

A neural network needs to make accurate predictions in order to be reliable. In this kind of neural networks, each neuron learns certain weights at every layer through an algorithm called back-propagation.

Back-propagation is the essence of neural net training. It is the practice of correctly and finely tuning the weights of a neural network based on the error rate obtained in the previous iteration. The accuracy of the prediction depends on the proper tuning of the weights, ensuring lower error rates and making the model more reliable by increasing its repetition and generalization.

4. AI applied to autonomous driving

4.1. Introduction

Artificial Intelligence is such a wide field with so many possibilities that can be applied nearly in anything thought. It is not only limited to classical classification patterns, such as face recognition or personalized shopping experience. It can be applied on technology to assist users in daily tasks, improving them and making them easier and more convenient.

This is the case of the automotive industry, where AI can be used just as a part of the total system, applying it to a particular case (for instance assisted parking) or it can be applied to the whole system, as would be the case of driverless autonomous cars.

4.2. Levels of autonomy on cars

According to SAE standards (11) there exist 6 levels of driving automation according to the required human intervention on the driving action. Although each level has subtle changes from one level to another, there is a clear frontier from the first three levels to the last three: going from a driver support feature to an automated driving feature.

All those levels and its explained features can be found on image below.



SAE J3016™ LEVELS OF DRIVING AUTOMATION

	SAE LEVEL 0	SAE LEVEL 1	SAE LEVEL 2	SAE LEVEL 3	SAE LEVEL 4	SAE LEVEL 5
What does the human in the driver's seat have to do?	You are driving whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering			You are not driving when these automated driving features are engaged – even if you are seated in “the driver’s seat”		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
What do these features do?	These are driver support features			These are automated driving features		
	These features are limited to providing warnings and momentary assistance	These features provide steering OR brake/acceleration support to the driver	These features provide steering AND brake/acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met	This feature can drive the vehicle under all conditions	
	<ul style="list-style-type: none"> • automatic emergency braking • blind spot warning • lane departure warning 	<ul style="list-style-type: none"> • lane centering OR • adaptive cruise control 	<ul style="list-style-type: none"> • lane centering AND • adaptive cruise control at the same time 	<ul style="list-style-type: none"> • traffic jam chauffeur 	<ul style="list-style-type: none"> • local driverless taxi • pedals/steering wheel may or may not be installed 	<ul style="list-style-type: none"> • same as level 4, but feature can drive everywhere in all conditions
Example Features						

Figure 4.1. SAE six levels of driving automation. (source: (11))

4.3. Case of study

Already knowing the range of each level and aiming to develop a fully automated car, the greatest aspiration of the developed model is to reach level 5. Although all the development will be done keeping the eyes on that goal, the truth is that even if the car is able to learn how to drive perfectly on all the tested situations, it would never fit on level 5 standards. The development is bounded to a very specific situation (a racecar) and thus would never be able to “drive everywhere in all conditions”, as it will be trained just to fulfill certain specifications. The developed model will never be suitable for normal driving conditions, as the training scenario does not include other vehicles, pedestrians, objects on the road, traffic lights and other agents associated to driving.

For this reason, the implementation of the study developed on this thesis is not suitable for its direct application to *street cars* and further actions and considerations would need to be taken into account if this would be the final desired application.

5. Convolutional Neural Networks

A Convolutional Neural Network (ConvNet or CNN) is a Deep Learning algorithm widely used for volume data (2D or 3D) and specially for image (2D) recognition and processing. It is capable of assigning importance (weights and biases) to various aspects on an input image in order to differentiate between different types of images. (12)

CNN, as neural networks, are made up of individual neurons that receive several inputs, take a weighted sum of all the analyzed aspects, pass it through an activation function and finally responds with an output. The whole network has still a loss function and everything applied to Neural Networks still plays a role here.

CNN is able to successfully capture spatial and temporal dependencies through the application of relevant filters that will be explained further on. When it comes to images, the primary function of CNNs is to reduce significantly its size into a form that is easier to process, but without losing any feature critical for getting a good prediction.

5.1. Basic structure

The essence behind every image is that it can be represented by a matrix of pixel values. For color images, those pixels are a combination of every primary color channel (red, green and blue) while on grey images there is just one channel. Each channel provides a 2D matrix stacked over each other with pixel range from 0 to 255.

CNNs are mainly composed of three or four types of hidden layers that go between the input and the output. The basic structure of this network is shown in figure below.

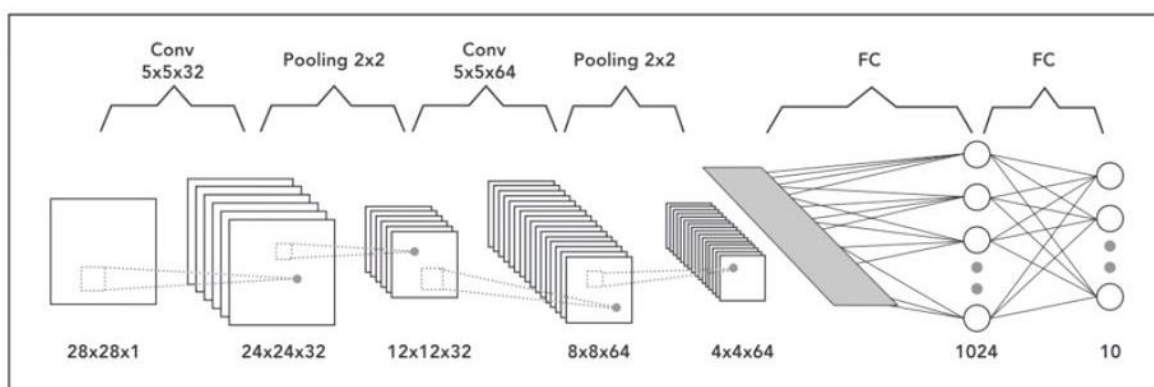


Figure 5.1. Example of a convolutional neural network with 2 convolutional layers, 2 pooling layers and 2 fully connected layers. (Source: (13))

The assets and operations of every layers will later be discussed in more depth, but here there are 3 basic steps that occur in each Convolutional Neural Network.

Immediately after an input image is received a convolutional operation is applied to that image, generating a subset on new images each of them with some different extracted features. After that, a pooling operation is applied in order to reduce the dimensionality of the previous images, hence reducing the computational load. These two steps can be repeated as many times as desired, according to how precise the network should perform. The final step is to apply a fully connected layer, which will connect all the neurons from its neighbor layers in order to perform a final classification of the image with all the information extracted before.

5.2. Types of layers

5.2.1. Convolutional layer

This first layer gives the name to Convolutional Neural Networks as it carries the main portion of the network's computational load. It is used to extract the high-level features of an images, such as edges, curves, lines or dots.

Mathematically this layer performs a dot product between two matrices, being one matrix the set of learnable parameters (known as kernel) and being the other matrix a channel of the input image. For a color RGB images, this operation will be performed for every color.

For a better understanding, imagine an input image which has a of 5 (height) x 5 (width) x 1 (number of channels) matrix, as the one depicted below.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Figure 5.2. Example of an input image with dimensions 5x5x1. (Source: (14))

Also, consider a Kernel matrix of 3x3x1, for example this one.

1	0	1
0	1	0
1	0	1

Figure 5.3. Example of a kernel with dimensions 3x3x1. (Source: (14))

The convolution of these two matrices would then be computed as shown in the following image.

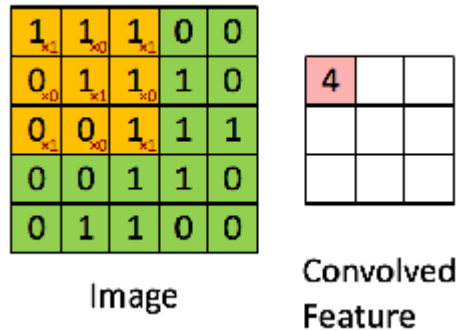


Figure 5.4. Representation of the dot product operation between the input image and the kernel (left) and its corresponding result (right). (Source: (14))

The Kernel matrix slides over the input image matrix depending on a parameter known as stride (by 1 pixel at a time in this example) for every position of the matrix computing the dot product of these two matrices forming an output matrix called Feature Map. The resulting Feature map for the input image and the Kernel used would be the following.

4	3	4
2	4	3
2	3	4

Figure 5.5. Output matrix obtained from the dot product operation between the input image and the kernel. (Source: (14))

This would be for a one channel image. For an RGB image the process is the same, but the Kernel is applied to each channel individually and the weight of each of them is then summed up to obtain the final Feature Map.

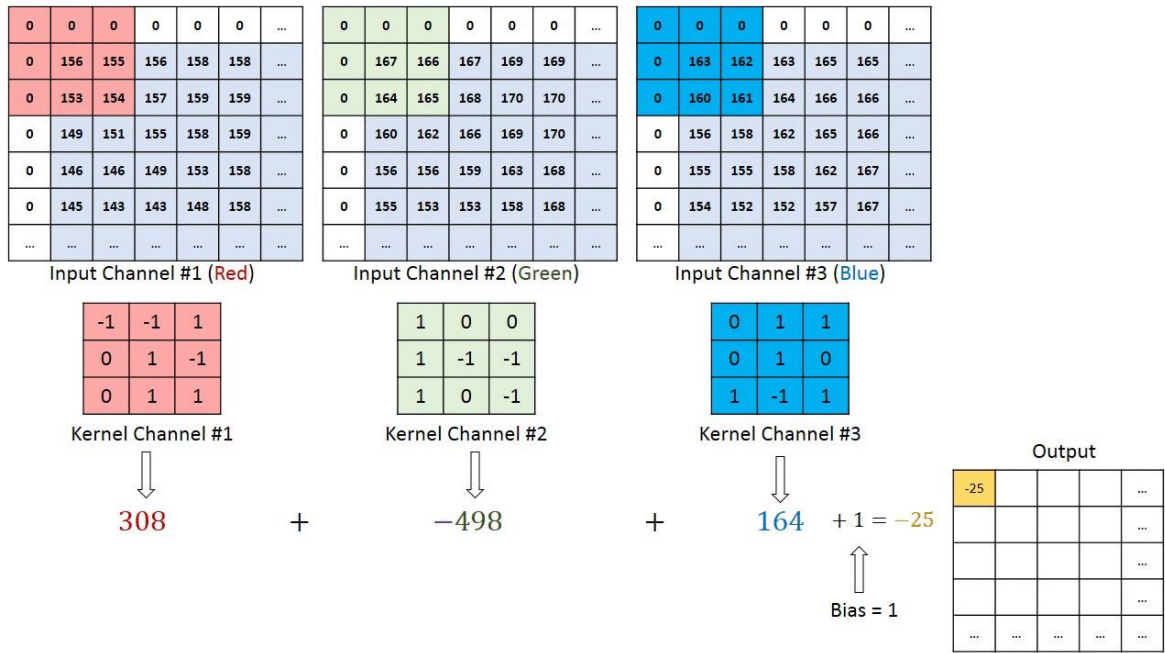


Figure 5.6. Computation of the dot product operation for a 3 channel RGB image. (Source: (15))

It is important to note that a Kernel acts as a filter for feature detector from the original input image. This means that different types of Kernels can be used according to which detection the layer is intended to do, and so different types of Feature Maps can be obtained from the same input image. This can be more clearly seen in the image below.







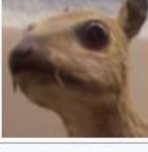
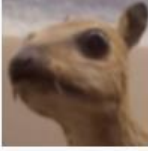
Operation	Kernel ω	Image result $g(x,y)$
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3 x 3 (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	
Gaussian blur 5 x 5 (approximation)	$\frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

Figure 5.7. Different types of kernels and its effect to the same input image. (Source: (16))

As every Kernel is capable of learning different aspects of an image, the CNN is not only limited to just one convolutional layer. The first layer is responsible for a Low-Level feature detection and by adding more layers the architecture adapts to a High-Level feature detection, as shown in figure below. In practice, all the Kernels used by every layer are initialized randomly on the network and it learns its values during its own training process.

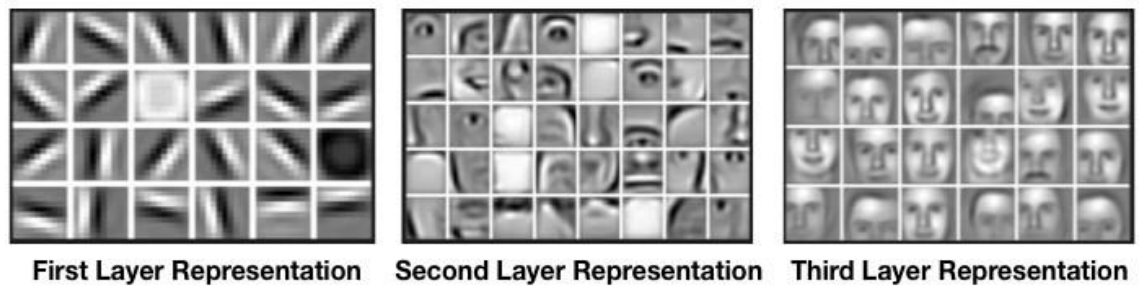


Figure 5.8. Extraction produced in every convolutional layer as the effect of using 3 concatenated convolutional layers. (Source: (17))

Taking a look to the first layer, through back propagation it has tuned itself to become blobs of something that can be interpreted as edges. Going deeper into the Neural Network to the second and third layer, the Kernels are doing dot products to the input of the previous layer, taking the forms of the first layer and making larger pieces out of them. This means that each neuron is just connected to a small portion of the initial input image and all the neurons have the same connection weights, a concept that is known as Local Connectivity.

Local Connectivity is the concept of each neuron connected only to a subset of the input image. This helps with the faster performance of the neural network as it reduces the number of parameters in the whole system.

5.2.2. Normalization/Non-linearity layer

Since convolution is a linear operation and images are everything but linear, those kinds of layers are often placed right after the convolutional layer to introduce some non-linearity to the Feature Map. The different types of activation functionals have already been discussed in Activation functions section so it is already known that the most used due to its performance is ReLU.

5.2.3. Pooling layer

This layer is the responsible for progressively reducing the spatial size of the Feature Map, decreasing the computational power required to process the data while maintaining the most important information. It is also useful for extracting dominant features which are rotational and positional invariant, which means that an object will be recognized regardless where it appears on the global image or its rotation.

Mathematically it replaces the output of the network at certain places by deriving a summary statistic of the nearby outputs. There are some pooling functions such as average pooling and max pooling but

the most used is the max pooling as it also acts as a noise suppressant. This type of pooling returns the maximum output from the portion of image covered by the used Kernel.

For a better understanding of this layer, imagine now a 4x4x1 Rectified Feature Map and a Kernel of 2x2x1 with stride 2. For a max pooling, the Kernel will travel along the matrix moving 2 pixels at a time and computing the maximum number of each window. For an average pooling, the computation will be the average of the window instead of the maximum number. The comparison between is type of pooling can be seen in the image.

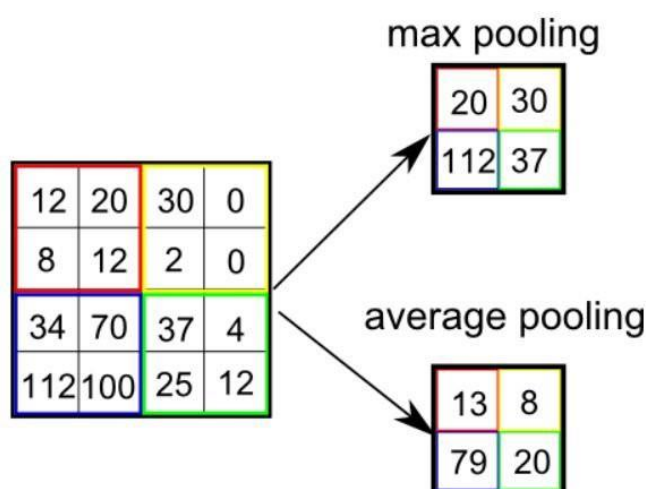


Figure 5.9. Result of applying a max pooling layer (top) and average pooling layer (bottom) on the same input data matrix. (Source: (18))

Again, this works for a one channel matrix. For RGB images the pooling operation is applied separately to each of the resulting Feature Maps for each channel.

5.2.4. Fully connected layer

This layer is always the last one in a CNN and is a traditional Multi Layer Perceptron that uses a Softmax Activation Function in the output layer. The neurons in a fully connected layer have total connection with all the neurons in the preceding and succeeding layer as in regular Neural Networks.

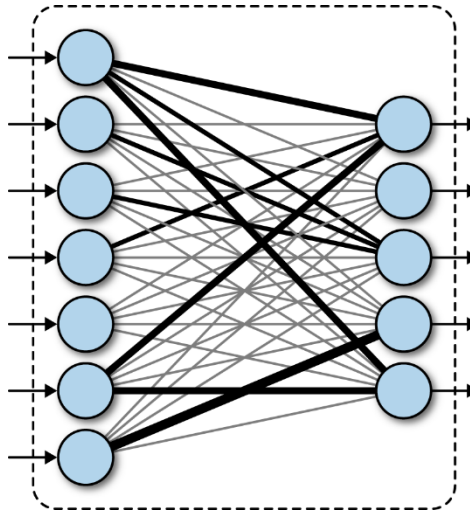


Figure 5.10. Basic structure of a fully connected layer. (Source: (19))

From the previous layers, the network has obtained high-level features and the main aim of the fully connected layer is to use these features to classify the input image into various classes. Apart from that, this final layer is able to combine the learned features of the two previous types of layers to introduce some non-linearity to the network.

6. Development environment setup

6.1. Requirements

6.1.1. Development machine characteristics

Any operative system is compatible for the development of machine learning, as all the libraries and software are available for every platform. However, the capacity of the computer where the development is performed may be determinant if many tasks need to be simulated, as well as the time it lasts to do so.

The development computer used during this thesis is not the best one as it lacks RAM memory and has a low range GPU card, but for an initial, introductory development can do the fact. The features of the development computer are the followings:

- Operative system: Windows 10 x64 bits
- RAM: 8 GB
- Processor: Intel Core i7-6700HQ CPU @ 2,60 GHz
- Geforce GTX 950M NVIDIA card

Although a lower level performance computer might be compatible for machine learning, refer to the 7.3.1 section for the used simulator as it required a specific hardware capacity.

6.1.2. NVIDIA GPU card

Although it is not essential for the development, a GPU card is a useful tool to enhance the performance of the trained model.

In order to know which GPU card is installed on the computer (if any), open the "Device Manager". In this new window as the one shown below, unfold the "Screen adapters" section and the installed GPU card model will pop up.

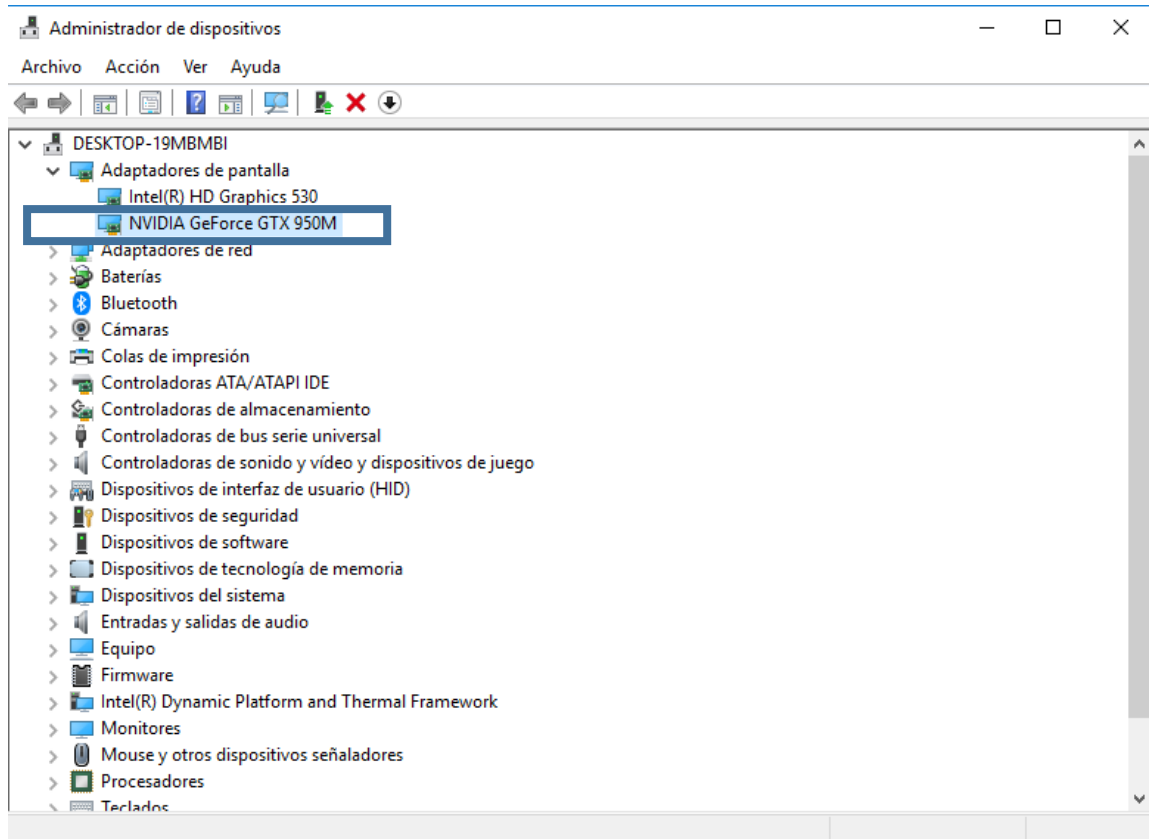



Figure 6.1. GPU’s model verification through “Device Manager” window. (Source: own)

In order to accelerate the development, some special libraries that work on an NVIDIA GPU card are used. Note that not all GPU cards will work fine as they need a special hardware requirement, which is described in [Tensorflow GPU’s installation guide](#).

At the time of writing this report, the minimum Compute Capability which a GPU card needs to have in order to work with Tensorflow is 3,5. So first thing to do is check if our GPU card fulfills the requirements by visiting the [NVIDIA CUDA GPUs webpage](#). Navigate to the appropriate option according to the type of NVIDIA card installed in the computer and find the name of the GPU card. In this case, the name of the card seen in

Figure 6.1, which is the one marked in figure below.



CUDA-Enabled GeForce and TITAN Products

GeForce and TITAN Products		GeForce Notebook Products	
GPU	Compute Capability	GPU	Compute Capability
NVIDIA TITAN RTX	7.5	GeForce RTX 2080	7.5
GeForce RTX 2080 Ti	7.5	GeForce RTX 2070	7.5
GeForce RTX 2080	7.5	GeForce RTX 2060	7.5
GeForce RTX 2070	7.5	GeForce GTX 1080	6.1
GeForce RTX 2060	7.5	GeForce GTX 1070	6.1
NVIDIA TITAN V	7.0	GeForce GTX 1060	6.1
NVIDIA TITAN Xp	6.1	GeForce GTX 980	5.2
NVIDIA TITAN X	6.1	GeForce GTX 980M	5.2
GeForce GTX 1080 Ti	6.1	GeForce GTX 970M	5.2
GeForce GTX 1080	6.1	GeForce GTX 965M	5.2
GeForce GTX 1070	6.1	GeForce GTX 960M	5.0
GeForce GTX 1060	6.1	GeForce GTX 950M	5.0

Figure 6.2. Compute Capability of the GPU card used for the development. (Source: own)

If willing to reproduce the final development code, ensure the computer has a GPU card with at least a Compute Capability of 3,5 or higher. In this case the GPU card used is compatible, since the Compute Capability is 5,0.

Make sure your software is up to date, otherwise some of the following installation steps might fail. Check that the following items are installed:

- NVIDIA GeForce Experience
- NVIDIA driver
- PhysX system software

As a final check, go to the path where all the software is installed and check for the NVSMI folder. In this particular case

```
C:/Program Files/NVIDIA Corporation/NVSMI
```

Inside that folder, execute the command

```
nvidia-smi
```

If a valid GPU is installed and up to date, the output of this last command should look like figure below.

```

Microsoft Windows [Versión 10.0.18362.356]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\martax>cd C:\Program Files\NVIDIA Corporation\NVSMI

C:\Program Files\NVIDIA Corporation\NVSMI>nvidia-smi
Thu Sep 26 21:25:36 2019

+-----+
| NVIDIA-SMI 436.30      Driver Version: 436.30      CUDA Version: 10.1     |
+-----+-----+
| GPU   Name           TCC/WDDM | Bus-Id      Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
+-----+-----+
|  0   GeForce GTX 950M  WDDM  | 00000000:01:00:0 | Off      | N/A |
| N/A   36C    P8      N/A /  N/A |  37MiB /  2048MiB |    0%    | Default |
+-----+-----+

Processes:
GPU      PID   Type   Process name                      GPU Memory
Usage
+-----+
| No running processes found |
+-----+

```

Figure 6.3. Command prompt with the information about the installed GPU in the development computer. (Source: own)

In case any other output is returned, the GPU’s software is either not up to date or the GPU is not compatible. Read the output in the command prompt for further information.

6.2. Development programs

6.2.1. Anaconda

Anaconda is a packet manager, an environment manager, a Python/R distribution and a collection of many open source packages. It is basically the platform where all the code will be written, compiled and executed.

It includes a bunch of useful libraries by default, as well as the possibility to install new ones in an easy way thanks to its own command prompt. No need to run pip commands.

6.2.1.1. Installation procedure

Download the installation packet though its [Webpage](#).

Choose the appropriate packet depending on the used operative system. In this case, Windows is used. Download the latest version available by clicking the Download button shown in the picture below.

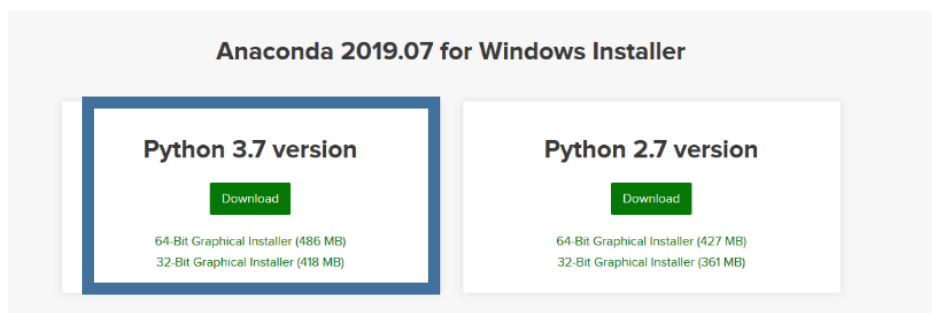


Figure 6.4. Chosen installation packet for the Anaconda environment with python 3.7. (Source: own)

Once the download is complete, the installation can begin by executing the downloaded executable. Leave all options by default.

Note: The python version can later be downgraded or upgraded according to necessities. However, it is not recommended to downgrade python to 3.6 in order to avoid Tensorflow warning. The downloaded and used version of CARLA (later seen on section 7) is only compatible with python 3.7 and it won't work with python 3.6 or other versions.

6.2.2. Python

Python is a programming language and the one used by the Anaconda environment. It is widely used in scientific and numeric computing, having so many packages and libraries specifically designed for the field.

6.2.2.1. Installation procedure

As having installed Anaconda, Python is already installed by default. The version will be the one specified in the Anaconda packet chosen, as can be seen in above Figure 6.4. However, it is also recommended to download the specific python packet for the version installed in anaconda in order to incorporate the `py` command for future uses in the command line.

To download it, just go to [Python webpage](#). Chose the version installed through anaconda. If not sure which exact version is in use (Python3.7.x) just open the window's search bar and type "Anaconda Prompt". Open it and just type

```
conda list
```

A list with all the installed libraries will show up. Search for python, as shown in the following picture

```

Seleccionar Anaconda Prompt (Anaconda3)
pysocks 1.7.0 py37_0
pytables 3.5.2 py37h1da0976_1 anaconda
pytest 5.0.1 py37_0
pytest-arraydiff 0.3 py37h39e3cac_0
pytest-astropy 0.5.0 py37_0
pytest-doctestplus 0.3.0 py37_0
pytest-openfiles 0.3.2 py37_0
pytest-remotedata 0.3.1 py37_0
python 3.7.3 h8c8aaf0_1
python-dateutil 2.8.0 py37_0
python-libarchive-c 2.8 py37_11
pytz 2019.1 py_0
pywavelets 1.0.3 py37h8c2d366_1
pywin32 223 py37hfa6e2cd_1
pywinpty 0.5.5 py37_1000
pyyaml 5.1.1 py37he774522_0
pymzmq 18.0.0 py37ha925a31_0
qt 5.9.7 vc14h73c81de_0
qtawesome 0.5.7 py37_1
qtconsole 4.5.1 py_0
qtpy 1.8.0 py_0
requests 2.22.0 py37_0
rope 0.14.0 py_0
ruamel_yaml 0.15.46 py37hfa6e2cd_0
scikit-image 0.15.0 py37ha925a31_0
scikit-learn 0.21.2 py37h6288b17_0 anaconda
scipy 1.2.1 py37h29ff71c_0
seaborn 0.9.0 py37_0
send2trash 1.5.0 py37_0
setuptools 41.0.1 py37_0
    
```

Figure 6.5. Output from the conda list command, showing the python version installed in the anaconda environment. (Source: own)

The version in use here is 3.7.3 so it would be necessary to choose this version. At the end of the python version page, on the “Files” section, chose the correct download packet. If using a x64 bits system, as in this case, chose the packet called *Windows x84-64 executable installer* as shown below.

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		68111671e5b2db4aef7b9ab01bf0f9be	23017663	SIG
XZ compressed source tarball	Source release		d33e4aae66097051c2eca45ee3604803	17131432	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	6428b4fa7583daff1a442cba8cee08e6	34898416	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	5dd605c38217a45773bf5e4a936b241f	28082845	SIG
Windows help file	Windows		d63999573a2c06b2ac56cade6b4f7cd2	8131761	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	9b00c8cf6d9ec0b9abe83184a40729a2	7504391	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	a702b4b0ad76debdb3043a583e563400	26680368	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	28cb1c608bbd73ae8e53a3bd351b4bd2	1362904	SIG
Windows x86 embeddable zip file	Windows		9fab3b81f8841879fda94133574139d8	6741626	SIG
Windows x86 executable installer	Windows		33cc602942a54446a3d6451476394789	25663848	SIG
Windows x86 web-based installer	Windows		1b670cfa5d317df82c30983ea371d87c	1324608	SIG

Figure 6.6. Option to download the executable installer for the installed version of python on the anaconda environment, compatible with a windows x64 operative system. (Source: own)

Once downloaded, execute the executable file. Make sure to check the “Add Python to PATH” option in the first window.

6.2.2.2. Pip

Install pip in order to be able to use it in the command line. Download the *get-pip.py* file from [this webpage](#) by simply clicking the text that says this name.

On the windows search bar type “Anaconda Prompt” and open it as administrator. Navigate to the download folder, where the file *get-pip.py* is located. In this case it is located in `C:/Users/marta/Downloads` so the command to get there would be

```
cd C:/Users/marta/Downloads
```

Once there, execute the following command

```
py get-pip.py
```

Note: Usually for python3 operations the command used is `pip3` instead of `pip`. However, as working inside the anaconda environment the default command is `pip`.

6.2.3. Spyder

Spyder is a python development environment, an IDE. It offers a combination of advanced editing, analysis, debugging and some other useful functionalities for scientists and engineers.

6.2.3.1. Installation procedure

The Anaconda installation also grants the Spyder installation. In order to check that everything is correctly installed, search for Anaconda Navigator in the window’s search bar and open it.

A panel as the one shown in Figure 6.7 with a list of associated extensions to Anaconda will be displayed. Check that Spyder is already installed by ensuring the corresponding button says “Launch”. Otherwise the button will say “Install” and that will mean it wasn’t installed during Anaconda’s installation. In this case, press the button to manually install it.

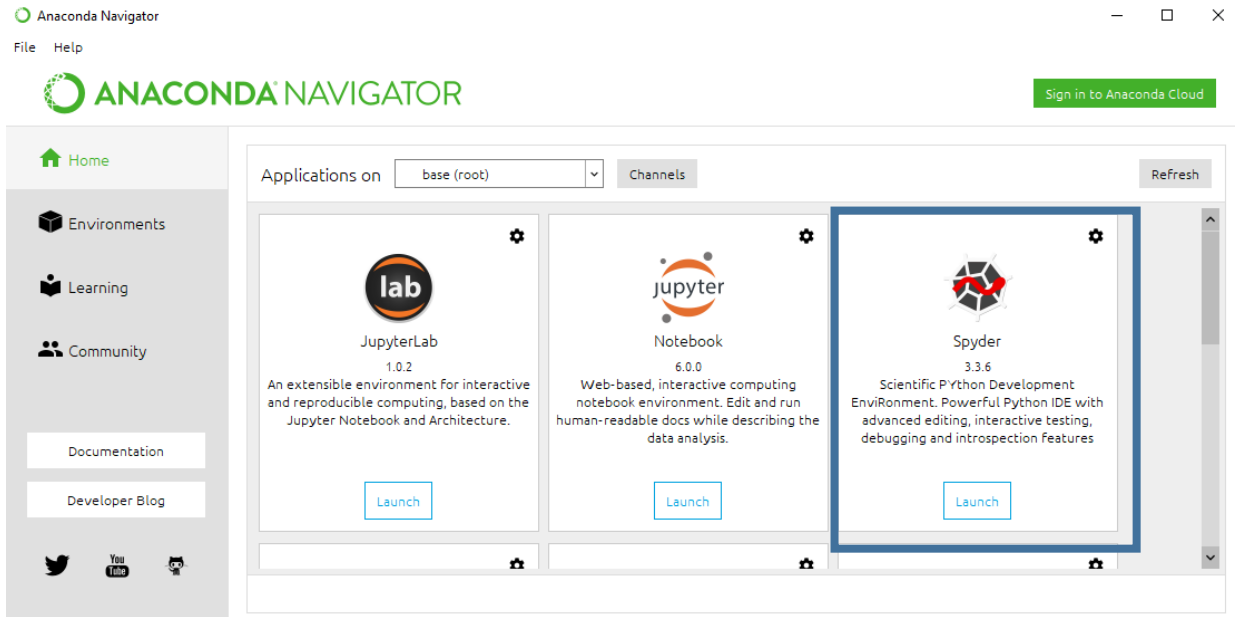


Figure 6.7. Anaconda’s Navigator panel, showing that Spyder is already installed. (Source: own)

6.2.4. Visual Studio and Visual Studio Code

Visual Studio is a set of tools and software development technologies. Its IDE is also used as a software development environment as it integrates multiple programming languages and the possibility to compile the code.

In this development it will be used as a tool for CUDA library and some other external programs, as the main code will be written in the Spyder IDE.

6.2.4.1. Installation procedure

Search again for Anaconda Navigator in the window's search bar and open it. Find the VS Code section as shown in Figure 6.8 and press the "Install" button. Once finished, the button will change from "Install" to "Launch".

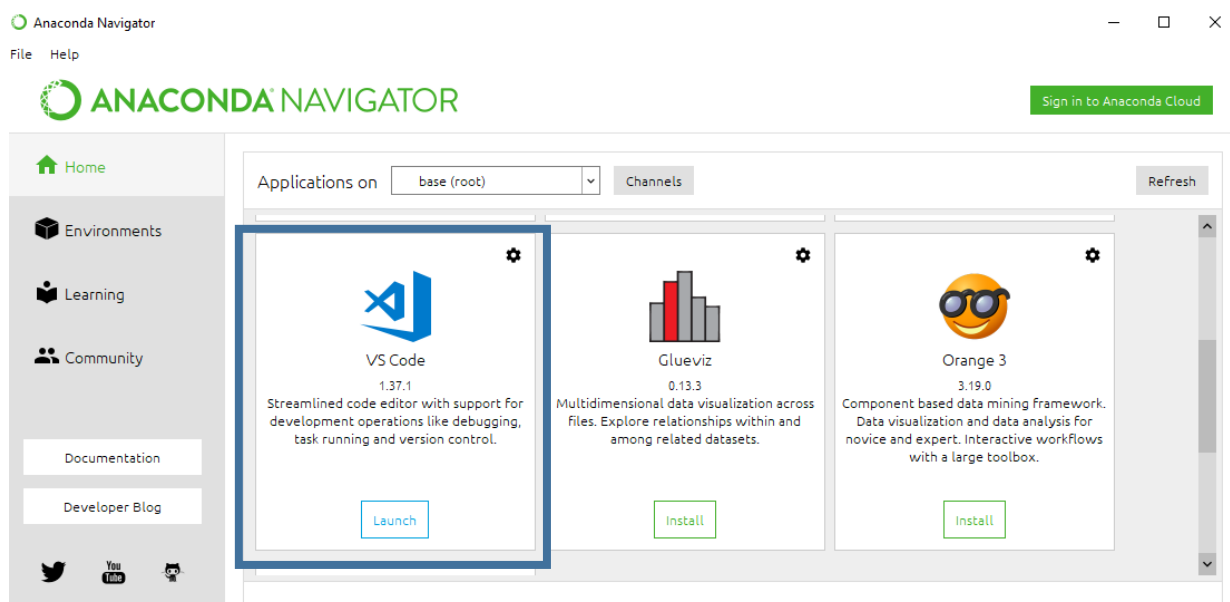


Figure 6.8. VS Code selection in Anaconda's Navigator. (Source: own)

Next download the full Visual Studio environment. Do not download the latest Visual Studio version, since the only compatible one with the used libraries Visual Studio 2017. Instead, go to [Visual Studio's older downloads page](#) and download the only current compatible version which is 2017.

Once the download is finished, install it and select the Python development environment.

6.3. Development libraries

6.3.1. General software requirements

The hardware requirements for Tensorflow and some other libraries are already explained in the NVIDIA GPU card section but since it is a software library it obviously also needs to fulfill some other kind of requirements (versions and compatibilities between libraries) which are also explained in [Tensorflow GPU's installation guide](#). To summarize them up for this development and taking into account that the Tensorflow version used is 1.14.0, the compatibilities are the followings:

- Python 3.7

- CUDA v10.0
- cuDNN's version at least 7.4.1 (used 7.6.3)
- NVIDIA GPU drivers' version at least 410.x (used 436.30)
- Visual Studio 2017

6.3.2. Installation observations

All used libraries will be installed using Anaconda's Command Prompt, which is automatically installed with the Anaconda packet. It allows to easily install software packets directly onto the development environment of Anaconda. (20)

Access to this terminal by searching "Anaconda Prompt" on the window's search bar. It is important to execute it as administrator, right click and select option "Execute as administrator", to have full permission for all the installations.

6.3.3. CUDA

6.3.3.1. Installation procedure (21)

Once knowing the hardware is compatible with CUDA (remember that the Compute Capability must be at least 3,5), it is necessary to check the compatibilities with Tensorflow in order to know which version is required. To do so, check the "Software Requirements" section in the [Tensorflow GPU support documentation](#). At the time of this report, the CUDA 10.0 is supported and required by Tensorflow.

In order to download the NVIDIA CUDA Toolkit, visit the [NVIDIA webpage](#) and choose the packet that fulfills your computer requirements. In this case, the packet downloaded would be the one with the options shown in the image below.

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System: Windows, Linux, Mac OS X

Architecture: x86_64

Version: 10, 8.1, 7, Server 2019, Server 2016, Server 2012 R2

Installer Type: exe (network), exe (local)

Download Installer for Windows 10 x86_64

The base installer is available for download below.

Base Installer [Download (2.5 GB)]

Installation Instructions:

1. Double click cuda_10.1.243_426.00_win10.exe
2. Follow on-screen prompts

The checksums for the installer and patches can be found in [Installer Checksums](#).
For further information, see the [Installation Guide for Microsoft Windows](#) and the [CUDA Quick Start Guide](#).

Figure 6.1. Parameter combination for the download of CUDA in a x64 windows 10 operative system. (Source: own)

Running the downloaded installer will produce the CUDA setup package window, as show in **Figure 6.9**.

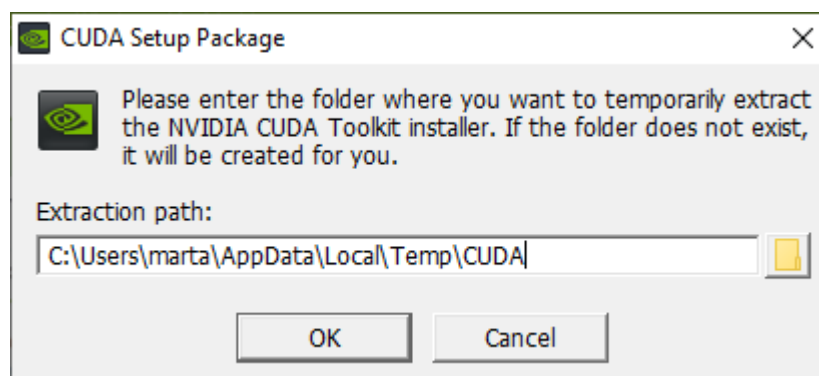


Figure 6.9. CUDA installation path windowed. (Source: own)

The CUDA installer will be extracted to the PC and once finished, it will start with the installation of the NVIDIA CUDA Toolkit. The installer itself will guide the user through the installation and there is no need to modify the default options.

With this installation, the CUDA driver and some other useful tools to create, build and execute a CUDA application will be obtained.

Last step is to verify the correct installation of this software by writing on the command prompt the command

```
nvcc -V
```

6.3.4. cuDNN

6.3.4.1. Installation procedure (22)(23)

As already done with CUDA, check the required version compatible with Tensorflow in the “Software Requirements” section in the [Tensorflow GPU support documentation](#). In this case, the cuDNN version is required to be at least 7.4.1.

The first step to download cuDNN ist to register into the NVIDIA Developer’s Program, which is totally free of charge. With the membership achieve, visit the NVIDIA Developer webpage and click the Download cuDNN button, as shown in Figure 6.10.

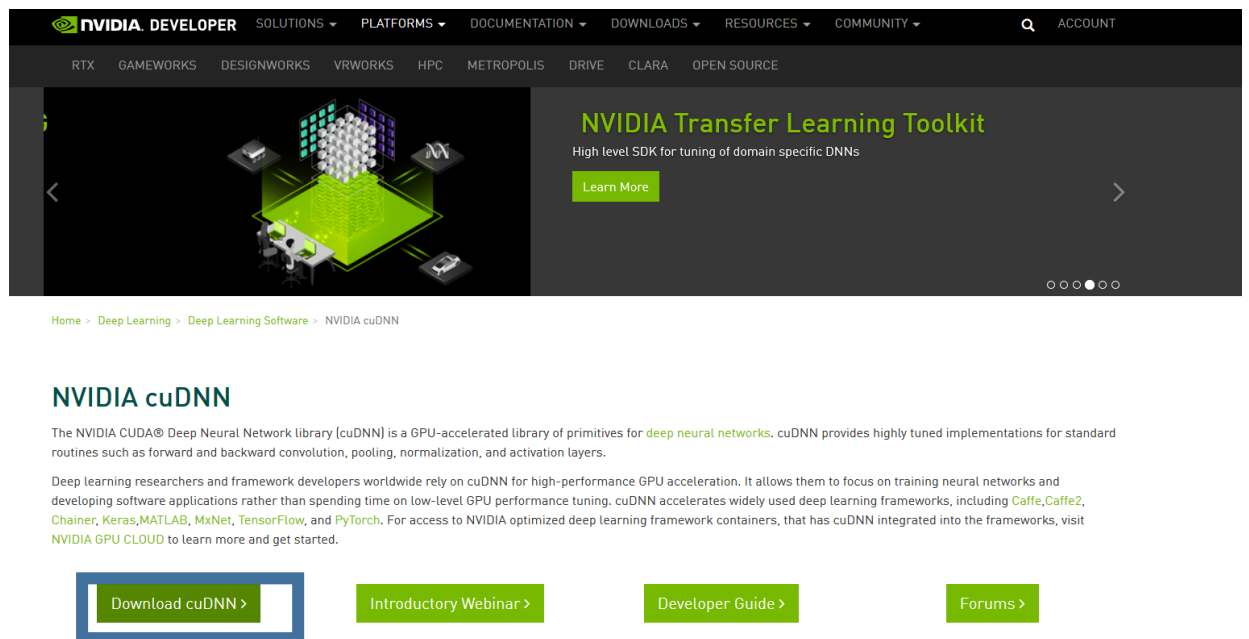


Figure 6.10. Button to Download cuDNN in the NVIDIA Developer webpage. (Source: own)

A page with all the possible versions for cuDNN Will be displayed. Choose the appropriate combination according to the CUDA version installed. In this case, it will be the package compatible with CUDA 10.0.

cuDNN Download

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

I Agree To the Terms of the [cuDNN Software License Agreement](#)

Note: Please refer to the [Installation Guide](#) for release prerequisites, including supported GPU architectures and compute capabilities, before downloading.

For more information, refer to the cuDNN Developer Guide, Installation Guide and Release Notes on the [Deep Learning SDK Documentation](#) web page.

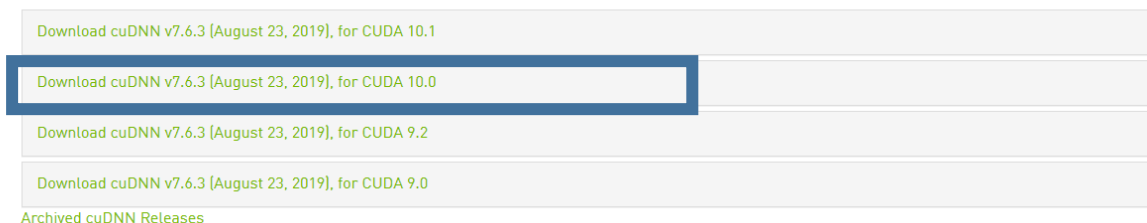


Figure 6.11. cuDNN download selection window. Compatible version with CUDA 10.0 (previously installed) selected. (Source: own)

Unzip the cuDNN files in the download folder and, in this case the directory path will be

```
C:\Users\martas\Downloads\cudnn-10.0-windows10-x64-v7.6.3.30
```

Leave a window open on that path, since there are 3 files that need to be copied from it. Open a new window on the directory path where the CUDA Toolkit is installed, in this particular case it would be

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0
```

All files need to be copied from the cuDNN path to the CUDA path, always respecting the folder. The files to be moved are the followings:

- **cuda_10.0.176_420.30_cuda10-windows.exe**
- **cuda_10.0.176_420.30_cuda10-windows.exe**
- **cuda_10.0.176_420.30_cuda10-windows.exe**

From C:\Users\martas\Downloads\cudnn-10.0-windows10-x64-v7.6.3.30\cuda\bin

To C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\bin

- **cuda_10.0.176_420.30_cuda10-windows.exe**
- **cuda_10.0.176_420.30_cuda10-windows.exe**
- **cuda_10.0.176_420.30_cuda10-windows.exe**

From

C:\Users\martas\Downloads\cudnn-10.0-windows10-x64-v7.6.3.30\cuda\include

To C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\include

- **cuda_10.0.176_420.30_cuda10-windows.exe**
- **cuda_10.0.176_420.30_cuda10-windows.exe**
- **cuda_10.0.176_420.30_cuda10-windows.exe**

From

C:\Users\martas\Downloads\cudnn-10.0-windows10-x64-v7.6.3.30\cuda\lib\x64

To C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.0\lib\x64

The final step is to verify whether CUDA environment variables are set or not in Windows. Go to *Control Panel -> System and Security -> System -> Advanced System settings*. At this point, a new window will pop up. Click on the button marked in figure below.

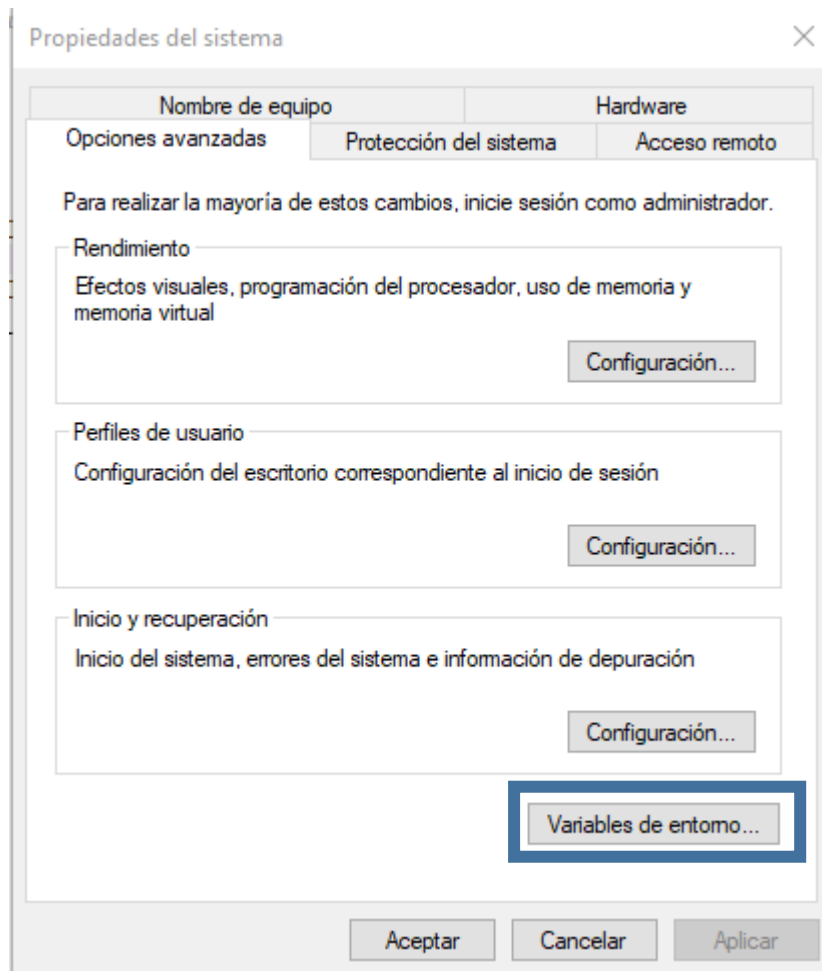


Figure 6.12. Advanced System settings with the button to access the Environment Variables window selected. (Source: own)

This button will again pop up a new window like the following one.

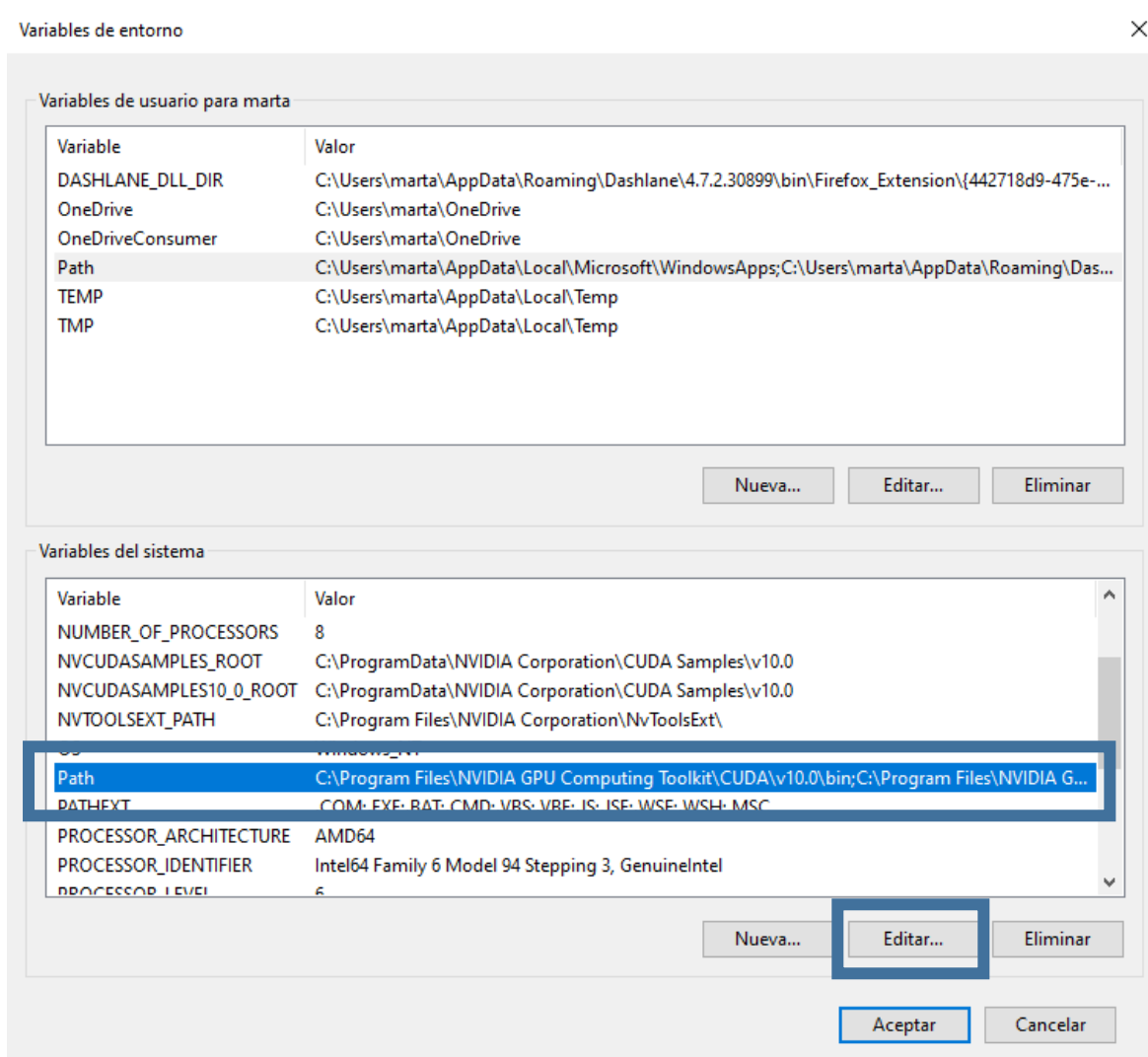


Figure 6.13. “Path” system variable selected in the Environment variable’s window. Button “Edit” to modify the parameter also selected. (Source: own)

Check that the variable “Path” has been assigned to following CUDA installation paths by clicking on the Edit button:

```
C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v10.0/bin
```

```
C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v10.0/libnvvp
```

6.3.5. Tensorflow

Tensorflow is an open source platform for machine learning. It allows to easily create machine learning models in order to train and test them using python.

6.3.5.1. Installation procedure

With the previous installation of Python, Tensorflow may have already been installed. As here it is necessary to switch from Tensorflow to tensorflow-gpu, perform the following commands to ensure the good functioning of the library.

```
conda uninstall tensorflow
conda install tensorflow-gpu
```

6.3.5.2. Installation test

Open the Spyder IDE and work with the IPython terminal. Introduce the command to import tensorflow library

```
import tensorflow as tf
```

In order to test CUDA support for the Tensorflow installation, run the command

```
tf.test.is_built_with_cuda()
```

If a False is returned, the installation is wrong at some point. If a True is returned, go ahead with the next step by running the command

```
tf.test.is_gpu_available(cuda_only=False,min_cuda_compute_capability=None)
```

This last command takes a little bit longer to answer. If it returns a False, go back to NVIDIA GPU card 6.1.2 NVIDIA GPU card section and check that all the software is up to date. If it returns a True instead, Tensorflow is already prepared to run with its GPU version.

6.3.6. Keras

Keras is a Python Deep Learning library, a high-level neural network API which runs on top of Tensorflow.

6.3.6.1. Installation procedure

In order to install this library, run the following command on the Anaconda Command Prompt.

```
conda install -c anaconda keras-gpu
```

6.3.7. Opencv

OpenCV is a library specifically designed to solve computer vision problem. It is extremely necessary for this project, as the main sensor used will be a camera and therefore the processing of the obtained image needs to be done prior interacting with it.

6.3.7.1. Installation procedure

On Anaconda Command Prompt running as administrator, navigate to OpenSSL folder (24). The name of the folder may vary according to which version is installed.

```
C:\Users\\Anaconda3\pkgs\openssl-1.1.1a-he774522_0\
```

Inside that folder, continue navigating until root \Library\bin is found. That is, the final path would be

```
C:\Users\\Anaconda3\pkgs\openssl-1.1.1a-he774522_0\Library\bin
```

Once on that directory, run the command

```
pip install opencv-python
```


7. CARLA simulator

7.1. Introduction

CARLA (25) is an open-source simulator which has been specially developed for autonomous driving research. It offers the user the possibility to configure its own development map, with different types of obstacles and items that go from cars and pedestrians to traffic lights and buildings, as well as including weather conditions control. It also includes a powerful API which allows to control the main car and all the other elements on the map with a Python script.

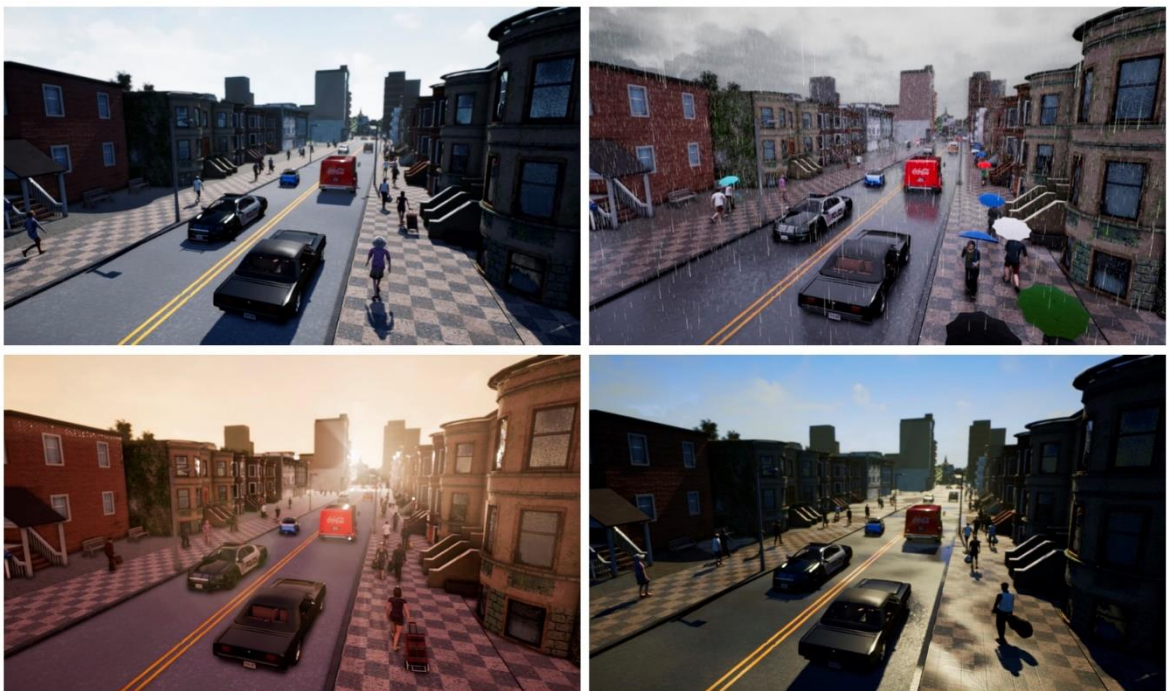


Figure 7.1. Capture of the same scene on CARLA simulator with different weather conditions. (source: (25))

However, the most interesting thing about this simulator is all the variety of sensors it offers, along with the possibility to control steering angles, throttle and brake. There is no longer a need to do a car prototype with expensive sensors to test all the neural networks in a first prototyping version.

CARLA is still under development, which means that there are still some interesting things to come, but it already offers some stable versions which has been tested in order to ensure that no crashes will occur during simulation.

7.2. Features

This open-source simulator includes many assets to fully simulate the real world yet they are not necessary for the development of this thesis, as just a partial capacity of it will be used.

The most interesting assets are, without no doubt, the range of sensors it has to offer that may be really expensive if bought specially for testing without any previous expertise, which are:

- RGB camera
- Depth camera
- Semantic Segmentation camera
- LIDAR
- Collision sensor
- GNSS
- Lane invasion sensor

Additionally, it also allows to govern some car actuators in order to control:

- Steering angle
- Throttle
- Brake

7.3. Preparing the simulator

7.3.1. Prerequisites

7.3.1.1. Hardware

CARLA runs on Unreal Engine 4 and so it needs the followings recommended requisites to work well. Note that although those are recommended specifications, a computer with lower performance may also be able to execute and run CARLA simulator but with a slower performance. (26)

- Quad-core Intel or AMD processor, 2,5 GHz or faster
- NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series card or higher
- 8 GB RAM
- 10GB of hard drive space for the simulator setup

7.3.1.2. Software

If working on Windows, CARLA simulator requires Windows 7 64-bit or later to run. It also needs the following programs:

- Visual Studio (2017, already installed to work with CUDA library)
- Python (at least version v3.5)
- [Unreal Engine](#) (v4.22.x)

7.3.2. CARLA Installation

CARLA can be installed using two different methods, depending on the skills and the necessities of the user.

7.3.2.1. Precompiled version

This is the easiest way to install CARLA and the traditional sequence. Just go to [CARLA download page](#) and choose any of the released version. I would recommend not to choose the newest versions, since it may probably have some bugs and it may also happen that it is not available for Windows users.

For this development, the used [CARLA version is 0.9.5](#).

Once chosen the version, download the packet (CARLA_0.9.5.zip in this case). Create a folder named “Carla Simulator” to any working directory, then extract the content of the download onto it. The windows root path for this project would then be

```
C:/Users/marta/Desktop/Carla Simulator
```

7.3.2.1.1 Limitations of this kind of installation

Note that choosing a precompiled version will probably be conflictive with the Python version previously installed (it is really important that they are the same, otherwise the simulator won't work), as it is not sure for which one the simulator was compiled. In order to check that, go to

```
<root path for CARLA simulator>/PythonAPI/carla/dist
```

Check that inside that folder there is a file with .egg extension. Something like

```
carla-0.9.5-py3.7-win-amd64.egg
```

The name of that file is itself explanatory of which CARLA version is in use and the Python version used for the compilation. In this case it was luckily build for Python 3.7 (which is used for the development)

but if it was the case that those versions didn't match, it would be necessary to build the project from source which is explained in the next section.

7.3.2.2. Building from source

This is the more advanced level of installation, which requires the command terminal. (27)

7.3.2.2.1 Prerequisites

First of all, make sure the following commands are already installed in the PC

- [Git](#)
- [Make](#)
- [Cmake](#)

And also make sure the following programs are already installed:

- Visual Studio 2017
- Unreal Engine
- Python3

7.3.2.2.2 Environment setup

Once everything is installed, open Visual Studio x64 Native Tools Command Prompt by typing "vs" on the window's search bar. If not able to find it, go to the Microsoft Visual Studio folder and find the file VsDevCmd. The approximate path would be

```
C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/Common7/Tools
```

Open this command terminal and enable a 64-bit Visual C++ Toolset on the Command line (28). To do so, first go to the path where "vcvarsall.bat" is

```
C:/Program Files (x86)/Microsoft Visual  
Studio/2017/Community/VC/Auxiliary/Build
```

Then execute it. On the used computer, the necessary command is the following, but make sure to check the previous link

```
vcvarsall.bat x64 8.1 -vcvars_ver=14.0
```

The command terminal will look something like this

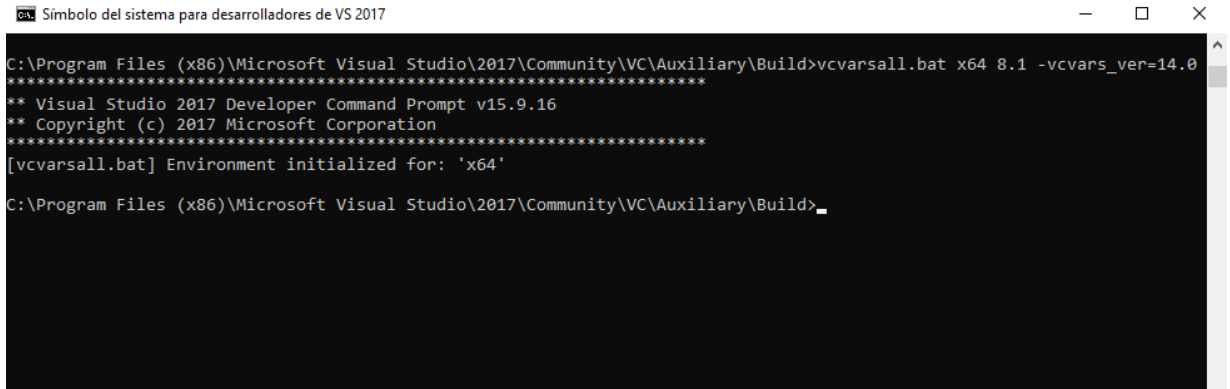


Figure 7.2. Visual Studio command terminal with x64 environment initialized. (Source: own)

If an error occurs saying that “vcvarsall.bat” is not recognized, it is probably because some features of Visual Studio are not installed. To solve that go to Control Panel > Uninstall a Program and search for Visual Studio 2017. Click on it, and instead of “Uninstall” click on “Modify”. The Visual Studio installer window will pop up. On the right section, unfold the section that refers to C++ and make sure the following two options are selected and installed, as shown in figure below:

- Windows’s SDK 8.1
- VC++ 2015 Tools

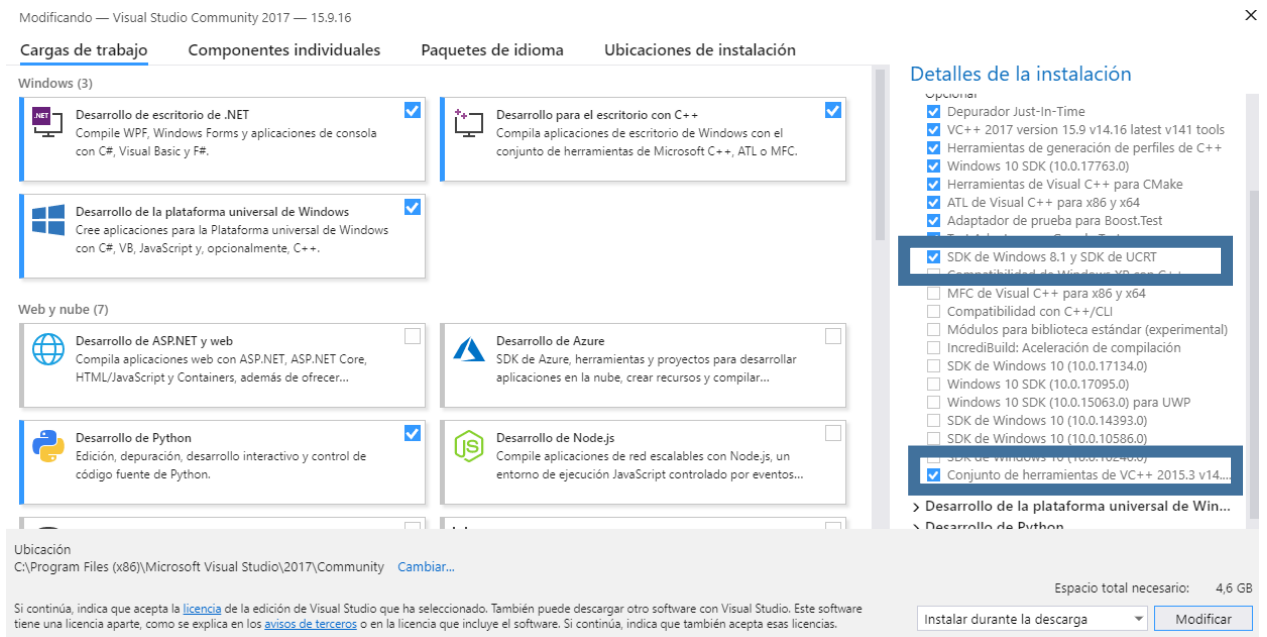


Figure 7.3. Features to install to enable the correct operation of vcvarsall.bat. (Source: own)

7.3.2.2.3 Building

Go to the path where it is desired to install CARLA, for example a folder named Carla Simulator on the desktop, which will be `<CARLA path>`

```
C:/Users/marta/Desktop/Carla Simulator
```

Use the git command to download the content of the repository

```
git clone https://github.com/carla-simulator/carla.git
```

And get into the created folder by typing

```
cd carla
```

Go to CARLA folder and open the following file

```
<CARLA path>/Util/ContentVersions.txt
```

This file explains how to download the contents corresponding to every version. Depending on the CARLA version needed, use a link or another. For version 0.9.5

http://carla-assets-internal.s3.amazonaws.com/Content/20190404_c7b464a.tar.gz

Next, extract those files to (if the path doesn't exist, create it)

```
<CARLA path>/Unreal/CarlaUE4/Content/Carla
```

The final step is to build CARLA. Execute the following command on the terminal

```
make launch
```

This step takes a little bit longer than the other, expect around 20 to 40 minutes. Just as an advice, it is really hard to make this process right the first time as some errors may occur during the process. If so, check [the Building on Windows thread on GitHub](#).

7.3.3. Dependencies

The CARLA API needs some dependencies in order to work. Install them all by navigating to the path for all the "requirements.txt" files, which are the followings:

```
<CARLA path>/PythonAPI/carla
```

```
<CARLA path>/PythonAPI/docs
```

```
<CARLA path>/PythonAPI/examples
```

```
<CARLA path>/PythonAPI/test
```

```
<CARLA path>/PythonAPI/util
```

And in the command line execute the following command inside each folder

```
py -m pip install -r requirements.txt
```

7.4. CARLA execution

Note that it is very important to work inside the CARLA project directory so the carla library will be reached by the system.

For better performance, always open CARLA with the command terminal. To do so, navigate to the root path of the CARLA directory. In this case

```
C:/Users/marta/Desktop/Carla Simulator
```

To execute CARLA, just enter the command

```
CarlaUE4.exe -windowed -resx=800 -resy=600
```

It should open a new window (resized according to resx and resy parameters) with the simulator.

7.4.1. Changing the port and enabling server mode

CARLA is running on port 2000/2001 by default and it may cause some conflicts with other programs (such as antivirus) running already in the PC. To solve that, just open CARLA in a different port. In this particular case port 5000 is used, but whatever other port is totally fine. The command would be

```
CarlaUE4.exe -windowed -resx=800 -resy=600 -carla-world-port=5000
```

Next thing to take into account is to run CARLA in server mode, which means that it will wait for a python client to connect with it. Otherwise the code written in order to train the model won't be able to interact with the simulator. It is important to know that by enabling the server mode CARLA simulator will wait until the Python client is connected. This means that the nothing will appear on the window (it will be apparently frozen) until the client is connected, so make sure to always open the Python project prior running the simulator.

Again, navigate to the CARLA directory and execute the command

```
CarlaUE4.exe -windowed -resx=800 -resy=600 -carla-server -carla-world-  
port=5000
```

7.4.2. Adjusting the graphics quality

Depending on the GPU card used and its capacity, it won't be able to both run CARLA in its best graphics quality level and perform the operations for the neural network with the Tensorflow-gpu library. In those cases, it is recommended to downgrade the quality of the simulator's graphics by adding the command *-quality-level=Low*. The full command to execute CARLA would then be

```
CarlaUE4.exe -windowed -resx=800 -resy=600 -quality-level=Low -carla-server  
-carla-world-port=5000
```

7.4.3. Choosing the map

The map can either be selected prior executing the simulator or once it is already loaded. For better performance, it is recommended to select it before executing CARLA by adding the name of the map right next to the executable path. An example with map "Town02" would be

```
CarlaUE4.exe Town02 -windowed -resx=800 -resy=600 -quality-level=Low -  
carla-server -carla-world-port=5000
```


8. Model training

8.1. Training environment

8.1.1. Initial considerations

This training model is applied to a very specific problem. In this case training an autonomous racecar doesn't require many added security elements as compared to a real world, understood as city or highway conduction. This makes the model easier, so applying this training model wouldn't be reliable on another type of autonomous cars.

For this study, the car will be trained alone in a special racetrack: no added pedestrians, no added racecars, no added objects. Just the trained racecar alone and the own limits of the track.

8.1.2. Racetrack

From the seven different maps offered by the simulator the most suitable for the training conditions is Map07, as it has a complete external perimeter that could do the fact to simulate a complete race track. The top view of that map is shown in the figure below.

All the other six maps have been discarded for the training as they were either too simple (comparing them to the average racetrack from the final application) or too big in distance (they are more like highways). The chosen map is yet not ideal as it has "blank" space in the middle that is not suitable for training the model but choosing the right path for the training can make a difference in its effectiveness.



Figure 8.1. Top view of Map07 offered by Carla Simulator. Three points marked as A, B and C that are used later on the training phase. (Source: own)

8.2. Databases

There are many online autonomous car databases for research purposes, however they are specially focused on commercial autonomous cars, not racecars. For this reason, CARLA will be used to collect all the necessary data to train this model.

8.2.1. Learning methodology

The selection of which type of learning method is a very critical point when it comes to more advanced problems. It has not only to suit the database in use, but also give the agent the capacity to face the problem with profit.

From the three types of machine learning techniques mentioned in section 3.2.1 the two only suitable ones here would be either supervised learning or reinforcement learning.

By collecting the data by driving the car around the racetrack by myself, it could be possible to train the model with supervised learning as it would have the outputs classified either as racetrack or racetrack limits. However, this type of learning has been discarded due to the time consuming of gathering the correct dataset to train correctly the model. Furthermore, the model would learn with a non impartial dataset, as it would learn with my own driving style which could eventually lead to a non-accurate trained model.

So, in order to make a more reliable model with different types of driving styles, it has been chosen to train in with reinforcement learning. By choosing this type of learning it is left on the algorithm's hand to learn its own patterns with a trial and error method, the data is gathered as the agent goes ahead. This basically means that the agent will be colliding with everything until it learns how to avoid it and thus, learn how to drive correctly.

8.2.2. Reinforcement Learning

Reinforcement Learning (RL) in neural networks is the parallelism to how humans, and all living creatures in fact, learn: a learning supported on the trial and error method.

Imagine a robot (agent) that is trying to learn how to pick-up an object. With this type of learning, it will perform many failed attempts until it is finally able to get that object. Once it has reached it successfully, it will have learnt how-to pick-up objects. In the particular case of an autonomous car it will learn how to drive by previously hitting so many times the walls and other objects or limits, until it is able to drive within the road limits. Although this is not a really interesting learning method if the car is being trained in a real scenario, since CARLA simulator will be used for the entire development this is not a problem. Furthermore, as every race track is supposed to be extremely different from each other, being in a constant learning is certainly the best option.

8.2.2.1. Basics

There are 2 main actors in reinforcements learning: the agent and the environment where the agent is trained. These two concepts interact between them, in both directions, through 3 key terms, which are the followings:

- State: describes the current situation of the agent. For instance, a possible state would be at which position the agent is
- Action: describes what the agent can do in the current situation (state). An example for an action would be move forward.
- Reward: feedback about how good or bad was the chosen action in a given state.

The graphical relationship between all these concepts is shown in figure below. The agent outputs and actions to relate with the environment and the environment gives back a feedback to the agent about the state and the reward so it can, again, perform another action to adapt to the situation according to the previous generated response.

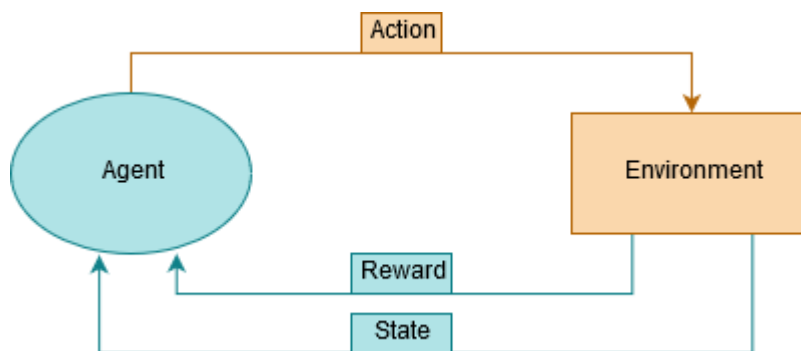


Figure 8.2. Relationship between the agent and the environmental. (Source: own)

To relate all these terms between them in an example, imagine the car we are about to train. The car will be in an initial position (which will be the state) and will need to perform some kind of action (for now let's just assume three basic actions: either turn right, turn left or move forward). For each action taken, the agent will receive a reward previously decided by the programmer (it is not random, neither can be learned by the neural network). An example of rewards are the followings:

- Driving between the road limits: +10 points
- Colliding: - 100 point
- Driving fast: +20 points

The main aim of the agent is to perform a series of actions along all of its states in order to maximize the final reward achieved, that is, learn what the programmer wants it to learn. The neural network will learn the patterns and be more or less accurate which his job at the end of the training according to the predefined rewards, so it is important to set significant items and values to them. An irrelevant reward, for instance, rewarding the agent with +15 points if the car is blue, may certainly lead to an extremely inaccurate model.

Every time the agent does an action and receives a reward, it updates its memory with the value of the reward at that state. This is called the *policy*, which “*defines the learning agent's way of behaving at a given time*” and is really “*a mapping from perceived states of the environment to actions to be taken when in those states*”. Generally speaking, the policy represents a huge table with all the information stored for each state, including all possible actions for every state and the associated rewards given to those pair state-action.

This sounds right but at first the agent has no experience (the policy is empty) and has to take a certain number of random actions before it is able to populate a first table to start working with. When the agent has already a bunch of data it starts learning by adjusting its predicted rewards for specific pairs state-action toward the received reward. Those predicted values are known as Q-values and are an indicated on how good that state is and thus, how much the agent wants to get into that state. The highest the Q-value, the highest the willing to get into that state.

Again, this might sound right and easy but leads to two situations that can turn into problems. Here comes the duality *Exploitation – Exploration*: is it better to continue learning on a known behavior or explore a new way to perform? This is basically the problem of gathering the data on-line: the taken actions affect the future data the agent will see. In practice many factors affect the policy and there is no better solution than the other. For this reason, normally a combination of both (~60-70 % exploitation – ~ 30- 40 % exploration) is used.

To exemplify this duality, imagine the following policy table with the start state 0. As it is the initial state, all the Q-values will be 0.

State	Initial (0)		
Action	Move forward	Turn right	Turn left
Q-value	0	0	0

Table 8.1. Q-table with the possible actions and Q-values associated to the initial state.

Since it is the initial state, every action has an equal chance to be chosen (33,3%). The agent choses randomly to move forward and updates the table according to the given rewards (suppose we give +1 point if the agent doesn't collide).

State	1		
Action	Move forward	Turn right	Turn left
Q-value	1	0	0

Table 8.2. Q-table with the possible actions and Q-values associated to the state number 1.

8.2.2.1.1 Exploitation

Here the agent is faithful to what it has already learned and stored in its memory. It makes the best decision according to current information, based on the best decisions made on the past.

Transferring those words into the table, that means that every time the agent resets to state 1 it will choose the action move forward since it is the one with highest Q-value. This is called the Greedy policy, where the agent picks the highest q-value. Brought into practice, the agent will never choose any of the other two actions.

8.2.2.1.2 Exploration

Here the agent is encouraged to find new pair state-actions that may lead to better solutions in the future, although it may not be in the current or near state. It gathers more information that might lead to better decisions on the future.

Going back to the example of the table, the agent will choose, again randomly, a possibly action of the table without being affected by the q-value table.

8.2.2.2. Deep Q-Learning

The problem up to now with the basics about reinforcement learning explained above is that the agent is only able to look one step ahead. This means that there is no global vision about all the rewards, but just the immediate after. Here is where one of Bellman's equations plays a role.

The following equation is called the Future Cumulative Discounted Reward and is used to obtain future rewards according to the actual and future states. It works for infinite series. (29)

$$R_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \gamma^3 \cdot r_{t+4} + \dots = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} \quad (\text{Eq. 8.1})$$

Where

R is the return, the term used for cumulative future rewards.

t is the current state

r is the reward associated to an specific state

γ is the discount factor (between 0 and 1). This factor determines the importance given to future actions. With a 0, the agent will give priority to achieving a higher reward in a short term. The other way around, a 1 will make the agent search for a higher reward in a long term.

k is the number of steps into the future where the reward was received

And the above equation is then transformed to fit the Deep Q-Learning problem as follows (30):

$$Q_t(s_t, a_t) = (1 - \alpha) * Q(s_{t-1}, a_{t-1}) + \alpha * (r_t + \gamma * \max Q(s_{t+1}, a_{t+1})) \quad (\text{Eq. 8.2})$$

Where

Q is the value of the accumulated reward

t is the current state

s is an specific state

a is an specific action

α is the learning rate, a parameter for setting the strategy exploration – exploitation (between 0 and 1)

r is the reward associated to an specific state

γ is the discount factor

With this equation, now the agent links up all the information learned resulting in a batch of memory that can be used for improving the efficiency of its learning.

8.3. Training process

8.3.1. Reinforcement learning implementation

8.3.1.1. Algorithm

The following flowchart illustrates the main process of the implemented algorithm.

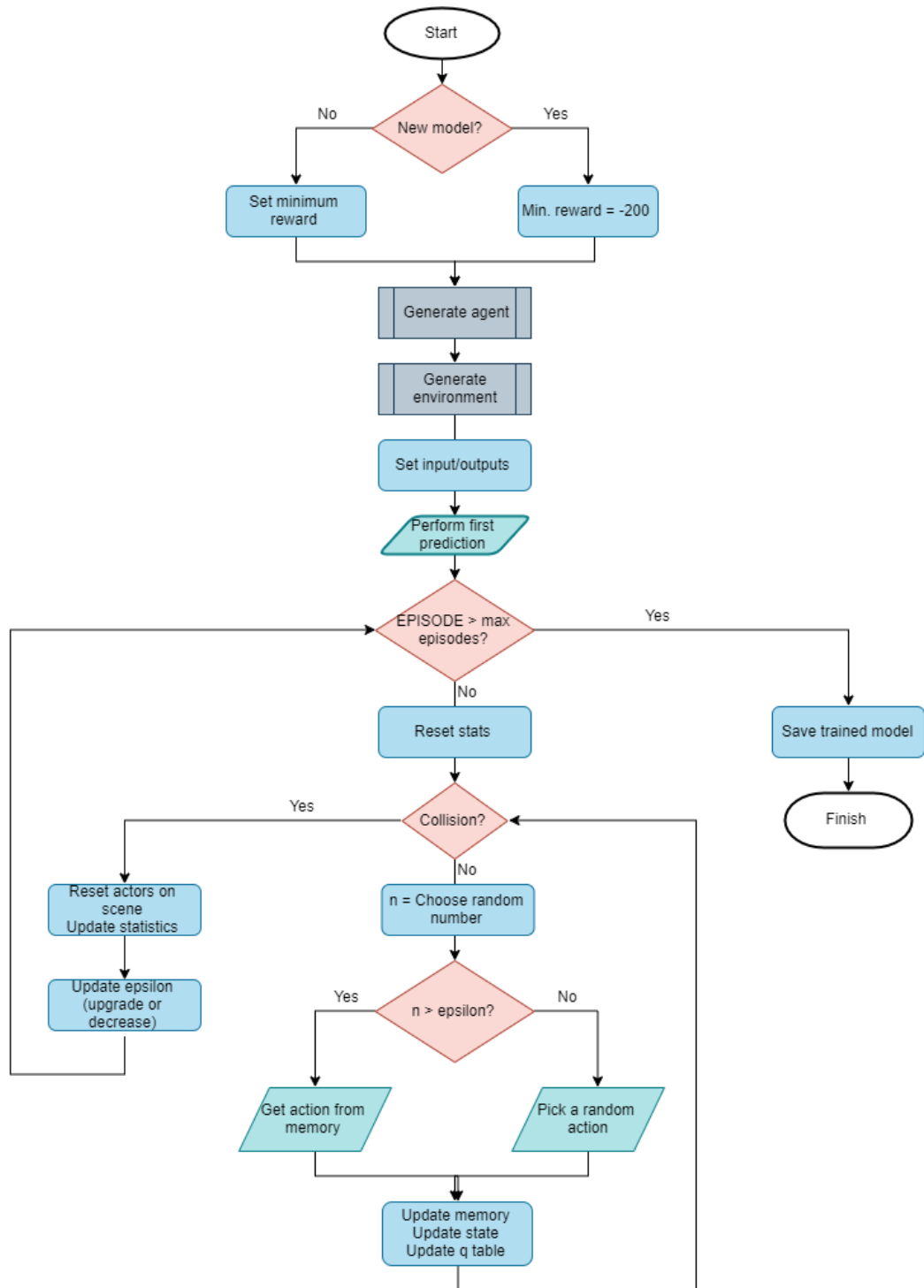


Figure 8.3. Flowchart of the main process which trains the model.

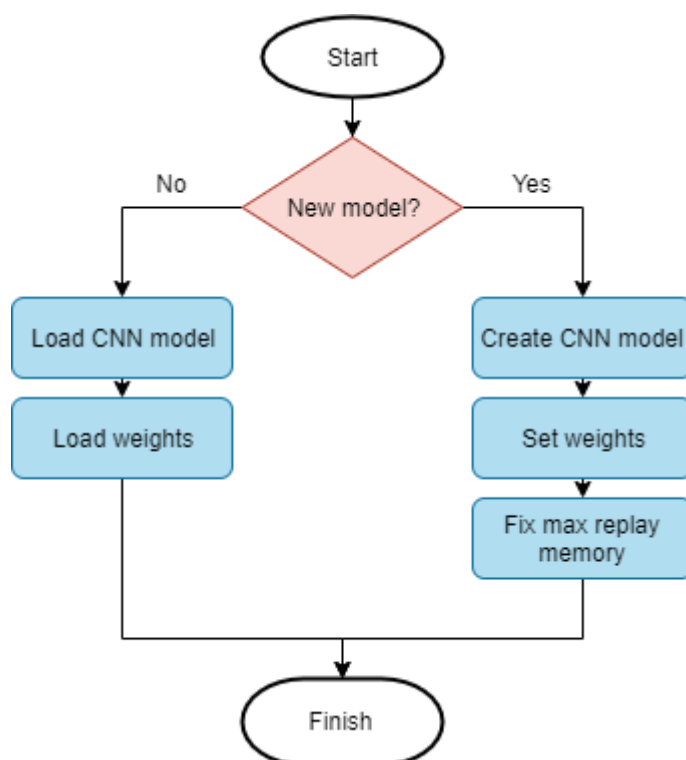


Figure 8.4. Flowchart of the subprocess "Load agent" on the main process.

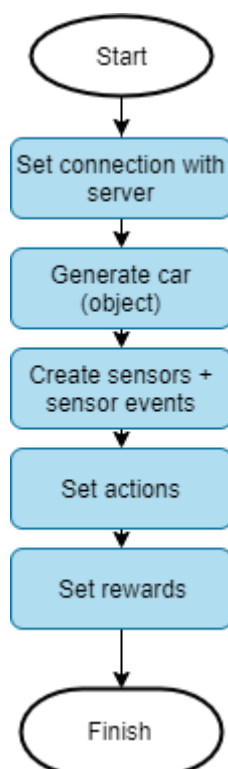


Figure 8.5. Flowchart of the subprocess "Load environment" on the main process.

8.3.1.2. Neural network

The premise for choosing an optimum neural network for reinforcement learning is not complicating it in excess, as this learning methodology itself is such complex that adding a degree of complexity on the neural network is counterproductive. This is shown in some previous developed models (for example in (31)), so the main objective is to keep the neural network as simple as possible to maximize the quality of learning.

After performing some tests under the same conditions on three different maps with 9 different types of neural networks, the best performance has been seen with a Neural Network consisting of three convolutional layers of 64 neurons. The details of those tests are found on Appendice A.

The chosen Convolutional Network for this training is as represented in the image below.

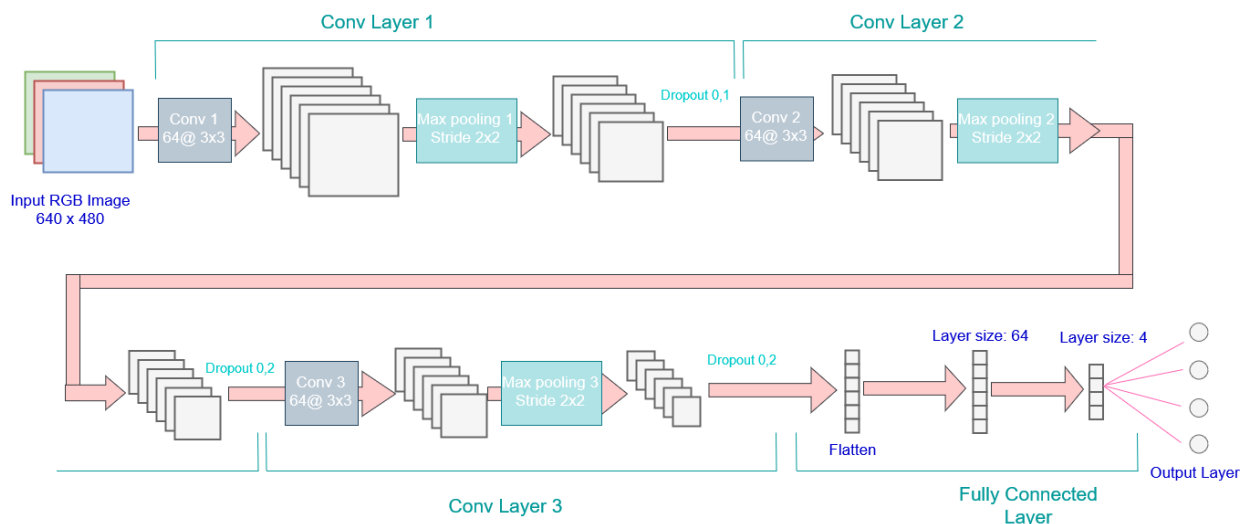


Figure 8.6. Structure of the Neural Network used for the final testing. (Source: own)

It consists of

- 3 convolutional layers with 64 neurons each + ReLu activation function. The kernel used for those layers is 3x3.
- 3 dropouts layers following the convolutional networks to avoid overfitting. (32)
- 3 pooling layers performed with max average. The kernel used for those layers is 2x2.
- 1 flatten layer to convert the vector to 1D
- 1 dense layer with 64 neurons
- 1 dense layer with 5 neurons (5 possible actions) with linear activation function.

8.3.1.3. Actions and rewards

A series of actions and rewards have been defined at first by just thinking about how could the car perform better. However, after several training, some others conditions have been added in order to correct the behavior observed in the previous training.

The complete list is shown below

ACTIONS			
Pretraining (brainstorming)			
State	Num	Condition	Description/Justify
Added	1	Turn right (90º)	Basic movement.
Added	2	Turn left (90º)	Basic movement.
Added	3	Go straight	Basic movement.
Added	4	Accelerate	Does the same as action (3) but with a larger module. Once the model is stable, the goal is to make it able to drive fast.
Added	5	Brake	Allows the movement rectification.
Added	6	Turn right (variable angle)	Possibility to adjust the turn making it a bit smoother.
Added	7	Turn left (variable angle)	Possibility to adjust the turn making it a bit smoother.
During training (correction)			
State	Num	Condition	Description/Justify
No change	1	Turn right (90º)	Maintained the basic movement.
No change	2	Turn left (90º)	Maintained the basic movement.
No change	3	Go straight	Maintained the basic movement.

Removed	4	Accelerate	The succession of multiple action (3) does the same.
No change	5	Brake	Maintained the basic movement for correction.
Removed	6	Turn right (variable angle)	An accurate turn is not critical.
Removed	7	Turn left (variable angle)	An accurate turn is not critical.

Table 8.3. Actions described to model the agent.

REWARDS				
Pretraining (brainstorming)				
State	Num	Condition	Points	Description/Justify
Added	1	Collision	-10	During pre-training phase (deciding the Neural Network), this value seemed to lead to good results.
Added	2	Driving fast (> 50 km/h)	+1	During pre-training phase (deciding the Neural Network), this value seemed to lead to good results.
Added	3	Driving slow (< 50 km/h)	-1	During pre-training phase (deciding the Neural Network), this value seemed to lead to good results.
Added	4	Long training episode (>10s)	+5	Prevent short and unproductive training cycles with not smart decisions.
Added	5	Driving to an end point	+10	Ensure the car goes to a certain point (that is, finishes a lap)
Added	6	Turning smoothness	-1 * yaw_acc	Prevent the model to do abrupt movements while turning.

Added	7	Efficiency	-0.05 * time to finish the lap	The main goal of this model is to drive correctly and fast, so this reward promotes its time efficiency.
During training (correction)				
State	Num	Condition	Points	Description/Justify
Modified	1	Collision	-20	Incremented collision penalty to avoid premature crashes due to bad decisions
Modified	2	Driving fast (> 50 km/h)	+1	
Modified	3	Driving slow (< 50 km/h)	-1	
Modified	4	Long training episode (> 20 seconds)	+5	Incremented the time to be considered "long episode". Recalculated taking into account the time it takes to spawn and start taking decisions.
Deleted	5	Driving to an end point	-0.1 * distance to the point	Ensure the car goes to a certain point (that is, finishes a lap)
No change	6	Turning smoothness	-1 * yaw_acc	Prevent the model to do abrupt movements while turning.
Deleted	7	Efficiency	-0.05 * time to finish the lap	The main goal of this model is to drive correctly and fast, so this reward promotes its time efficiency.
Added	8	Driving really slowly (<5km/h) or not moving	-8	Prevent the car to be stuck in an infinite loop of braking actions. Also

				cuts the training if the car goes really slow.
Added	9	Short training episode (<20 seconds)		Added to complement reward number 4. This way the agent is expected to learn quicker that needs to train for longer.
Added/Deleted	10	Driving on track center	-0.1 * distance to track center	Ensure the agent drives between the track limits.

Table 8.4. Rewards described to model the agent.

8.3.2. Previous training test cases

During the training process the model has undergone several modifications in order to be able to find the best one. Each Test Case describes the actual parameters used, as well as the actions and rewards defined.

As a summary, the following table described the sensors used in each training and the commands passed to the agent to actuate on a certain actuator.

Test Case	Sensors	Actuators
TC.01	Collision sensor RGB camera	Torque Brake Steering
TC.02	Collision sensor RGB camera	Torque Brake Steering
TC.03	Collision sensor RGB camera	Torque Brake Steering
TC.04	Collision sensor RGB camera	Torque Brake Steering
TC.05	Collision sensor RGB camera	Torque Brake Steering
TC.06	Collision sensor RGB camera	Torque Brake Steering

TC.07	Collision sensor RGB camera	Torque Brake Steering
TC.08	Collision sensor Camera (applying Semantic Segmentation)	Torque Brake Steering
TC.09	Collision sensor Camera (applying Semantic Segmentation)	Torque Brake Steering
TC.10	Collision sensor Camera (applying Semantic Segmentation)	Torque Brake Steering
TC.11	Collision sensor Camera (applying Semantic Segmentation)	Torque Brake Steering
TC.12	Collision sensor Camera (applying Semantic Segmentation)	Torque Brake Steering
TC.13	Collision sensor Camera (applying Semantic Segmentation)	Torque Brake Steering
Further test cases	Collision sensor Camera (applying Semantic Segmentation) ???	Torque Brake Steering

Table 8.5. Relation between sensors used and actions on actuators on each Test Case.

8.3.2.1. TC.01 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	3h 21m 25s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Random		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -10 points Reward 2: Velocity > 50 km/h: +1 point Reward 3: Velocity < 50 km/h: -1 point		

	Reward 4: > 10 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$
Sensors used	RGB Camera Collision detection sensor

General comments of the model

This is the first modified model after choosing the neural network, with just few modifications (added action 4 and rewards 4 and 6). The main aim of this TC is to see how the chosen model performs in order to modify the rewards to make them a little bit more accurate.

Statistics and performance on track

On the statistics below are shown 2 records of this same model, one trained 180 episodes and the other one trained 300 episodes.

The average accuracy of the model is not quite good being just at 50%, as shown below. Furthermore, the accuracy from an episode to another is quite disparate. Looking at the average reward there is a visible tendency to stagnate around $-100 \sim -50$, which is not a good result but predictable as 3 of the possible 5 rewards are negative. From the other reward statistics (max and min) there is not a clear tendency to any point with so few trained episodes. Maybe some more episodes would have to be trained to see if this model could lead to any good results.

However, by looking at the performance on track, it was clear that some other conditions needed to be taken into account in order to upgrade the model. In some situations, the agent decided to choose action number 4 (brake) for long time periods (more than 10 seconds) and then restart the movement, which lead to fake punctuation episodes (the agent was awarded with a long training episode although it wasn't doing anything).

The overall conclusion of this Test Case is that it is a good start point, but the model can improve a lot by adding some other rewards. For this reason, there is no point on training the agent under these conditions for longer episodes.

Graphical statistics

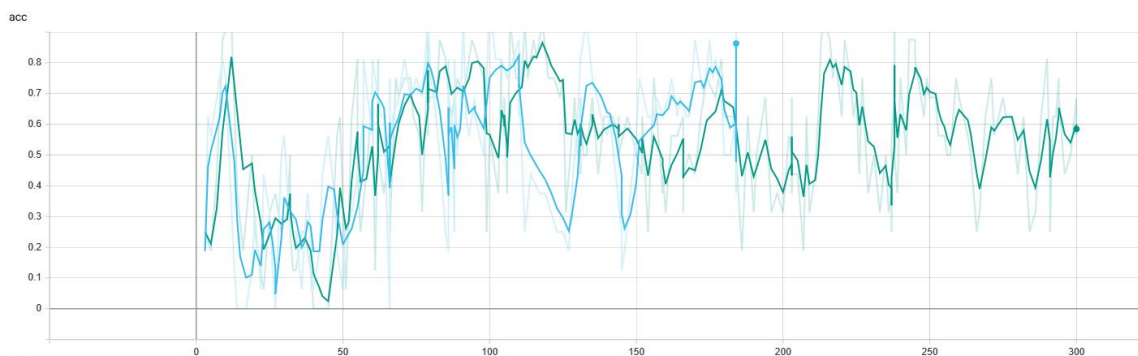


Figure 8.7. Accuracy traces on light blue (model trained less than 200 episodes) and on a darker blue (model trained during 300 episodes) for the model defined in TC.01.

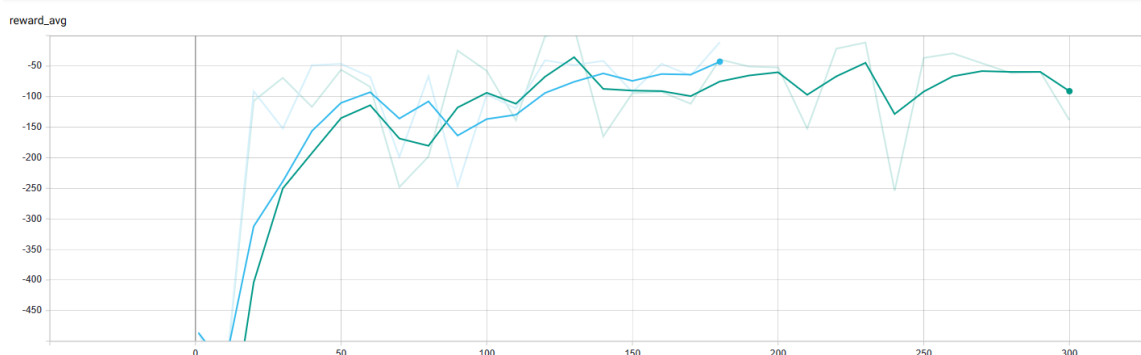


Figure 8.8. Average reward traces on light blue (model trained less than 200 episodes) and on a darker blue (model trained during 300 episodes) for the model defined in TC.01.

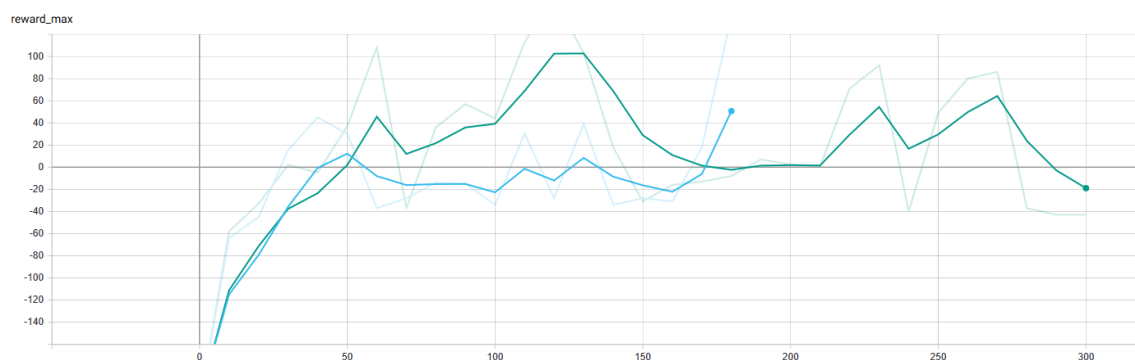


Figure 8.9. Maximum reward traces on light blue (model trained less than 200 episodes) and on a darker blue (model trained during 300 episodes) for the model defined in TC.01.

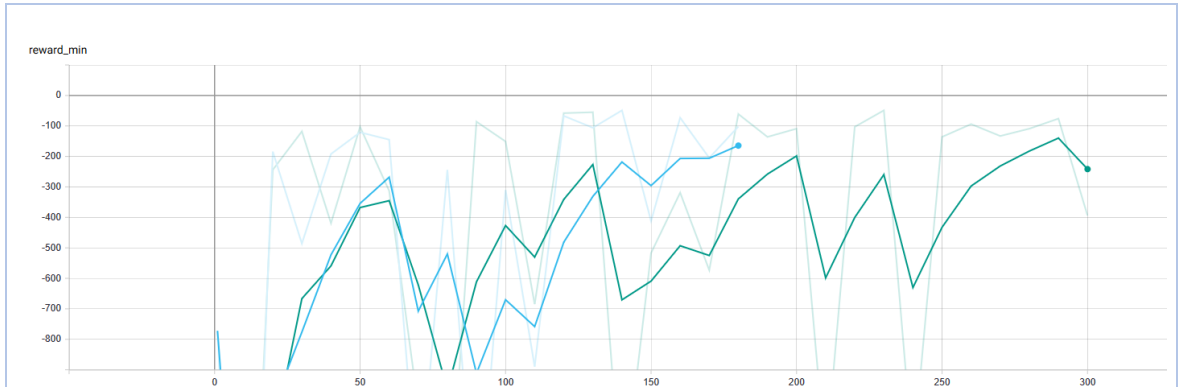


Figure 8.10. Minimum reward traces on light blue (model trained less than 200 episodes) and on a darker blue (model trained during 300 episodes) for the model defined in TC.01.

8.3.2.2. TC.02 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	9h 41m 44s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Random		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -10 points Reward 2: Velocity > 50 km/h: +1 point Reward 3: Velocity < 50 km/h: -1 point Reward 4: > 10 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$ Added Reward 8: No movement: -8 points		
Sensors used	RGB Camera Collision detection sensor		

General comments of the model

With the base of TC.01, a new reward has been introduced in order to avoid the main problem of the previous model: no movement periods. All the other conditions have been maintained, as no apparent problem with them has been found.

Statistics and performance on track

This test is divided in two phases. In the first phase the model is trained during 300 episodes and is represented below in the graphs with just a grey trace. This model seemed to had opportunities if trained for longer (at the end of the 300 episodes the tendency seemed to be ascendant, at least on the reward. The accuracy seemed to decrease), so the decision was made.

Below each graph with just a grey trace, there is another one with this same trace and the one obtained from training the model for a little bit longer (x5 times the base episodes). While on the accuracy the model seemed to perform much better at a long-term and with a positive tendency it is important to notice that the maximum accuracy is $\sim 0,7$, leaving a large range of improvement. The average reward, however, doesn't show a good result at a long-term but at a mid-term. The performance of the model seems to be good and positive until it hits 1k episodes and then progressively worsens. Although this only affects average and maximum reward (the minimum reward remains stable), the loss is extremely significant if it is also complemented with the performance on road. If the agent is spawned at the start of a long straight road, it decides to accelerate until it collides at the end of this road (always an intersection where it could have avoided colliding by turning either left or right), not a smart option. Furthermore, after doing this for several times, the agent doesn't seem to learn that this is a really bad behavior. Also, this behavior leads to fake results, as it accumulates positive rewards for going fast on the straight road and the penalty for colliding is not significant compared to the gain obtained by the velocity.

The overall conclusion of this Test Case is that the statistics don't go along with the model performance and that more restrictive penalties need to be applied to the collision.

Graphical statistics

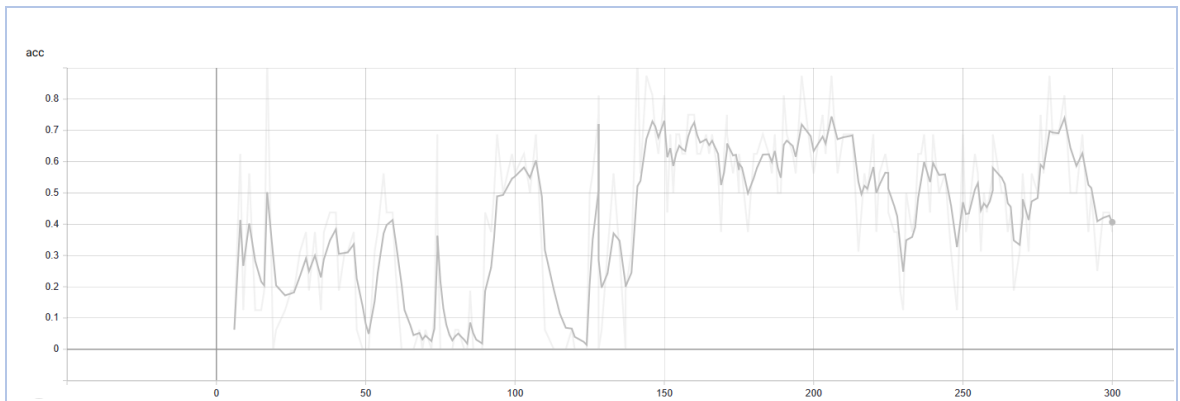


Figure 8.11. Accuracy trace for the model defined in TC.02 trained during 300 episodes.

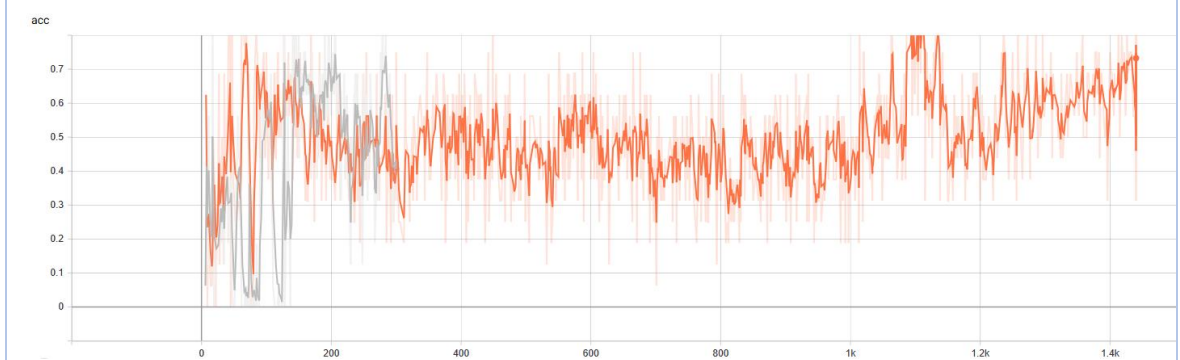


Figure 8.12. Accuracy traces on grey (model trained for 300 episodes) and on orange (model trained for 1500 episodes) for the model defined in TC.02.



Figure 8.13. Reward average trace for the model defined in TC.02 trained during 300 episodes.

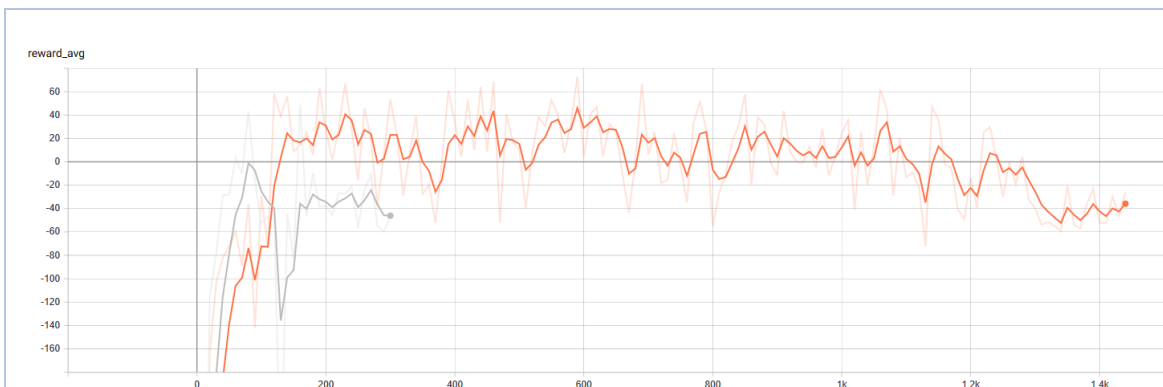


Figure 8.14. Average reward traces on grey (model trained for 300 episodes) and on orange (model trained for 1500 episodes) for the model defined in TC.02.

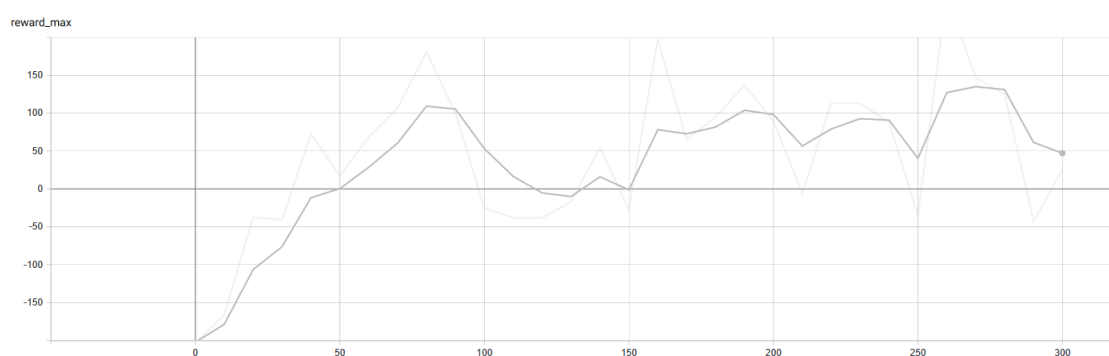


Figure 8.15. Maximum reward trace for the model defined in TC.02 trained during 300 episodes.

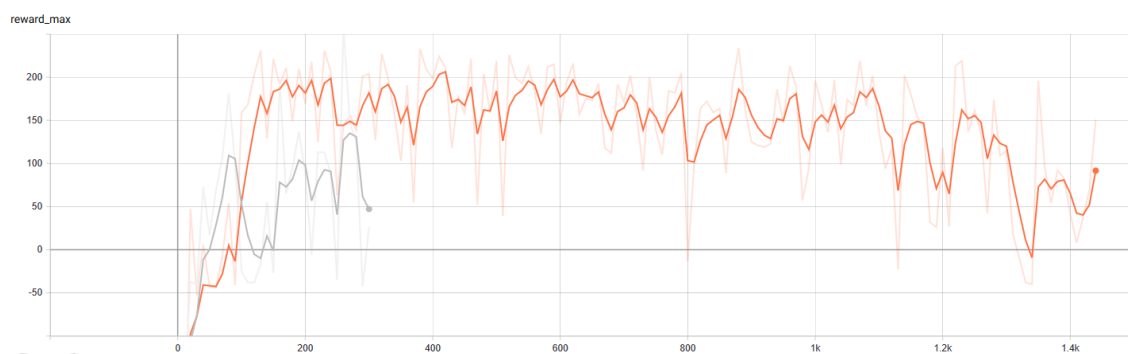


Figure 8.16. Maximum reward traces on grey (model trained for 300 episodes) and on orange (model trained for 1500 episodes) for the model defined in TC.02.

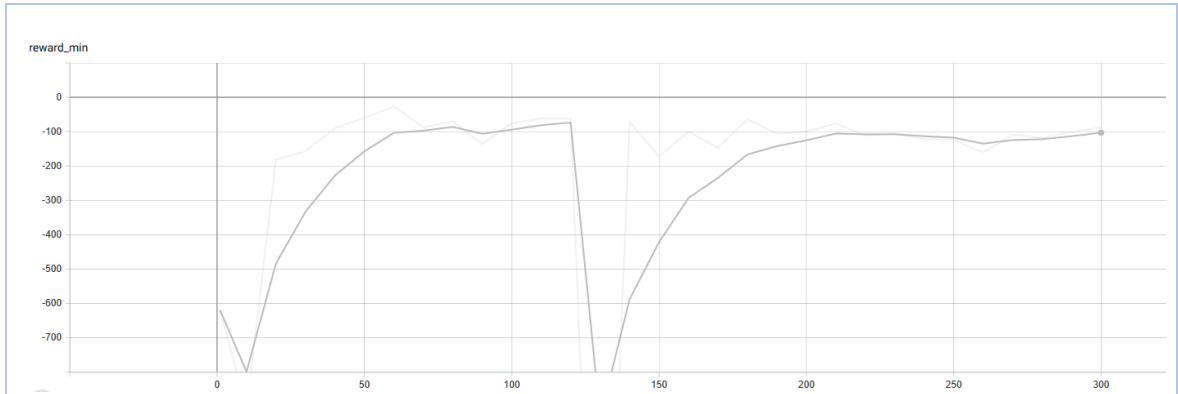


Figure 8.17. Minimum reward trace for the model defined in TC.02 trained during 300 episodes.

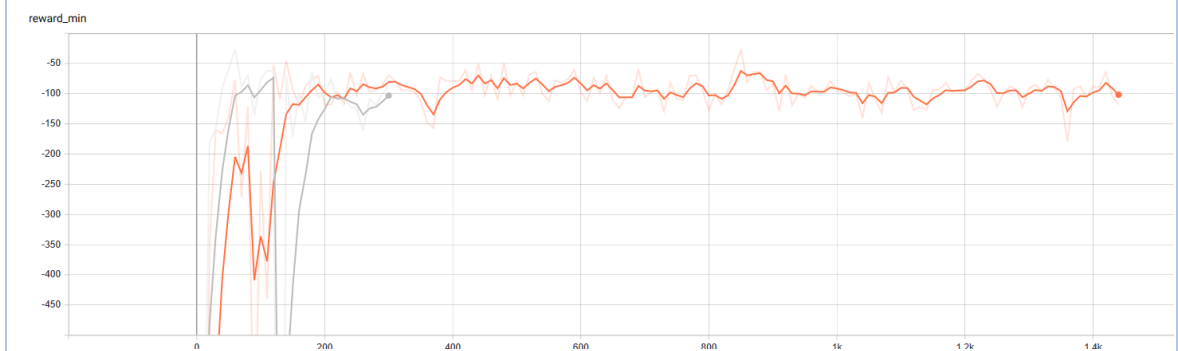


Figure 8.18. Minimum reward traces on grey (model trained for 300 episodes) and on orange (model trained for 1500 episodes) for the model defined in TC.02.

8.3.2.3. TC.03 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	1h 7m 36s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Random		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Incremented penalty Reward 1: Collision: -20 points Incremented reward Reward 2: Velocity > 50 km/h: +3 point Incremented penalty Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 10 seconds without collision: +5 points		

	<p>Removed reward 6</p> <p>Reward 8: No movement: -8 points</p>
Sensors used	<p>RGB Camera</p> <p>Collision detection sensor</p>

General comments of the model

In order to isolate the problem in which the agent goes straight until colliding, the reward for abrupt turns has been deleted to promote turns. Also, the penalty for colliding has been doubled and the rewards involving velocity have also increased.

Statistics and performance on track

With the corrections made the agent seems to perform a little better, although the accuracy doesn't surpass 0,7 and the fluctuation is quite significant (from 0,2 to 0,7). Comparing the two models trained only for 300 episodes, the one in TC.03 is better in average accuracy.

The average and maximum reward are inferior compared to TC.02 (result of not so many straight road crashes, which is good) but with a positive tendency with could lead to better results than TC.02 if trained for longer time. The minimum reward remains the same as the previous model.

The performance on track is a little better than TC.02 as not having so many straight road end collisions but this behavior is still present in too many training episodes. The agent decides so many times to go fast instead of exploring the map.

The overall conclusion of this Test Case is that neither the statistics nor the performance of the agent are the desired ones and some other rewards can be introduced to be tested prior to giving opportunities to this model for longer training episodes.

Graphics statistics

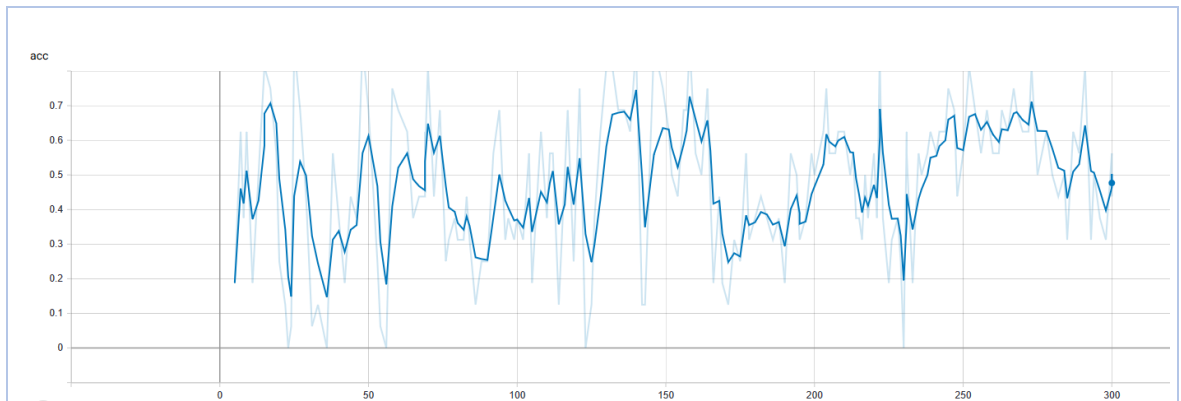


Figure 8.19. Accuracy trace for the model defined in TC.03 trained during 300 episodes.

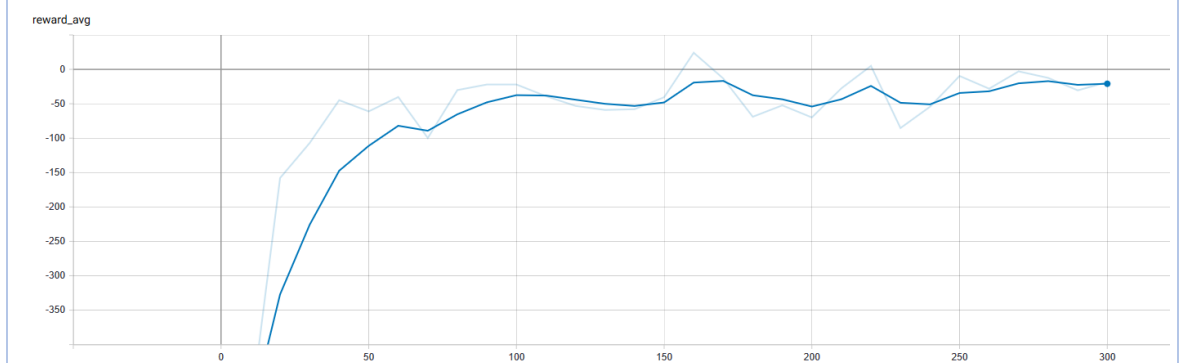


Figure 8.20. Reward average trace for the model defined in TC.03 trained during 300 episodes.

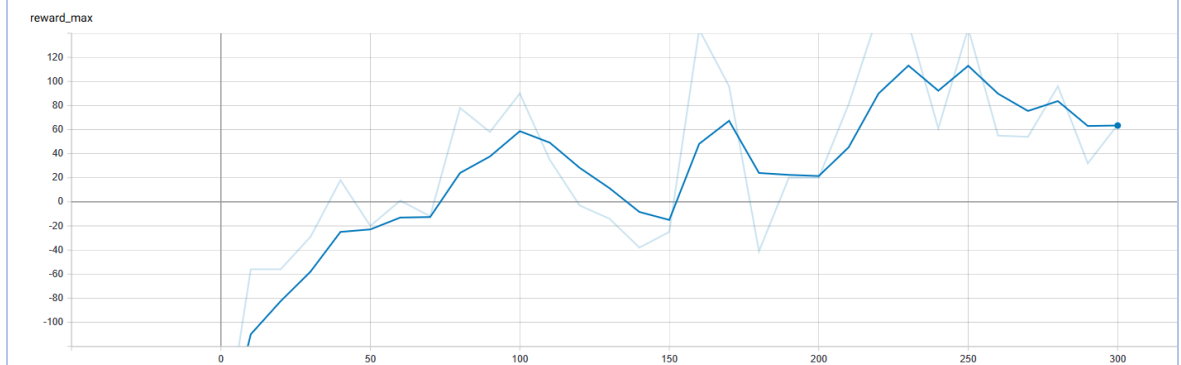


Figure 8.21. Minimum reward trace for the model defined in TC.03 trained during 300 episodes.

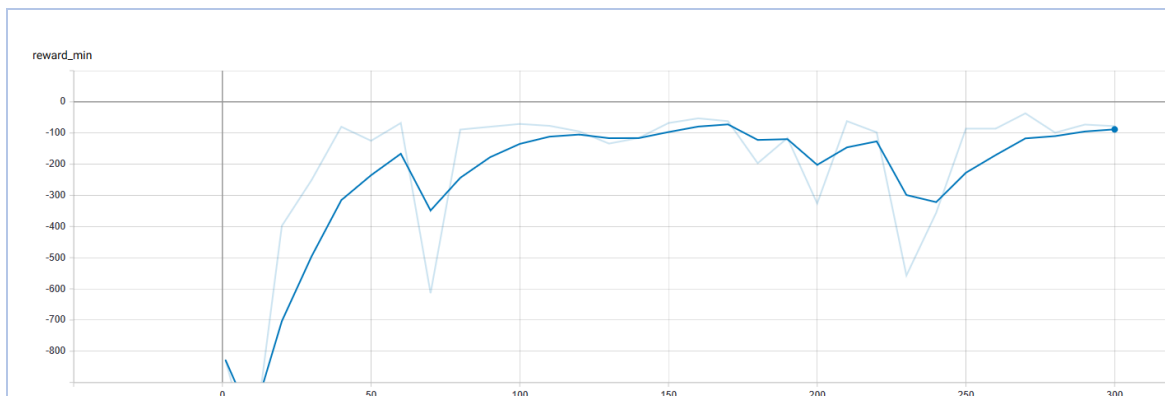


Figure 8.22. Maximum reward trace for the model defined in TC.03 trained during 300 episodes.

8.3.2.4. TC.04 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	4h 48m 16s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Random		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Modified condition Reward 4: > 20 seconds without collision : +5 points Added Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Added Reward 9: Short training episode (< 20 seconds): -10		
Sensors used	RGB Camera Collision detection sensor		

General comments of the model

The reward condition referring to long training episodes has been modified, increasing the time considered to be “long” as from previous Test Cases (01, 02 and 03) it has been observed that 10 seconds is already the time it takes to spawn and start taking decisions, that is, the average time from all the episodes is 10 seconds. To complement this reward, another one has been introduced to penalize short episodes. The aim of this reward is to make the model learn quicker that it needs to take decisions that increment the training time without collision.

A final modification has been made; reintroducing reward number 6.

Statistics and performance on track

The first training for this Test Case has been held during 300 episodes as a previous approach to test the new conditions applied. The accuracy of the model has improved in general, but the fluctuations are quite huge. The maximum accuracy, however, surpassed the previous maximum accuracy (which was at 0,7) reaching the value of 0,8 and the average of 0,5. Looking at all the rewards’ statistics, the tendency seems to be favorable if trained longer, which a maximum reward tendency quite positive. The average reward, although still negative, seems to have possibilities to reach a stable positive value.

Encouraged by the previous results, the seconds training for this Test Case has been held longer, during 1000 episodes. According to accuracy, the model performs better incrementing its average reward. Nevertheless, looking at the rewards the model is not able to evolve and become better because after 400 episodes the statistics start decreasing and are no further recovered. The hope of surpassing the 0 line on the average reward is frustrated as before reaching a positive value the model starts obtaining much more significative negative rewards.

The performance on track doesn’t seem either to improve, as the agent still collides frontally on long straight roads.

The overall conclusion of this Test Case is, again, that neither the statistics nor the performance of the agent are the desired ones. The random spawn point doesn’t seem to grant any advantage to the learning of the model, as it is not able to concentrate in a particular task because every time it collides it has a very different starting situation which can not be learned in such short training episodes. Maybe in a much longer training (more than 10 thousand episodes, for instance) this model could learn to drive better.

Graphical statistics

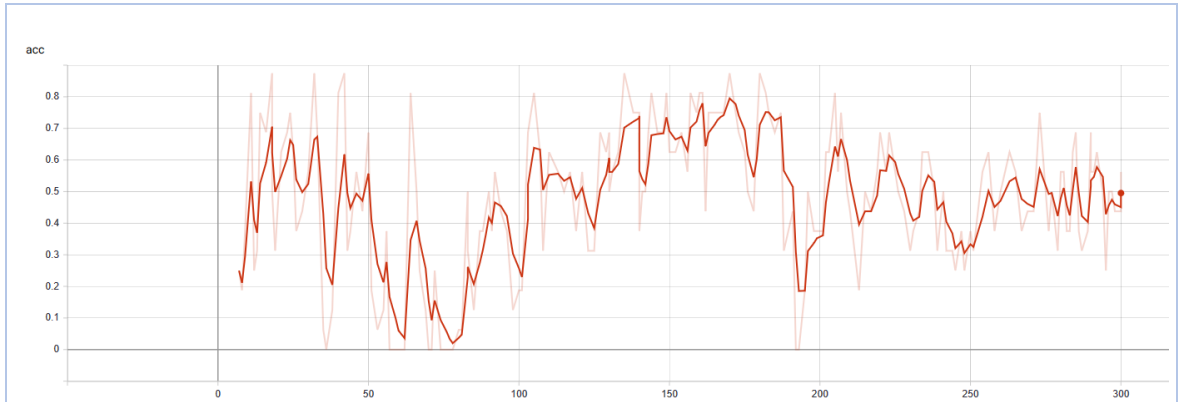


Figure 8.23. Accuracy trace for the model defined in TC.04 trained during 300 episodes.

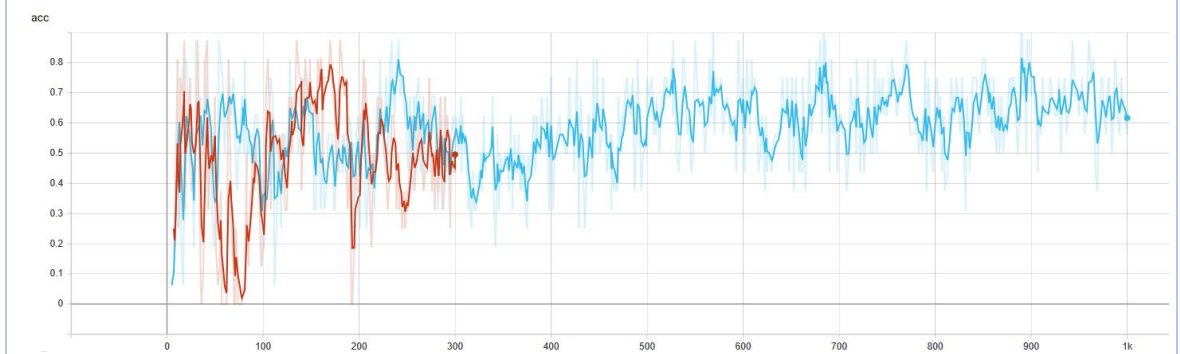


Figure 8.24. Accuracy traces on red (model trained for 300 episodes) and on blue (model trained for 1000 episodes) for the model defined in TC.04.

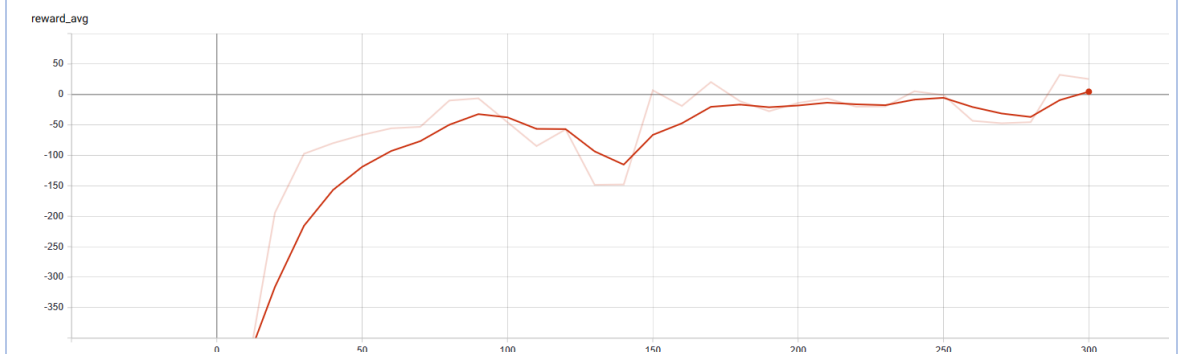


Figure 8.25. Average reward trace for the model defined in TC.04 trained during 300 episodes.

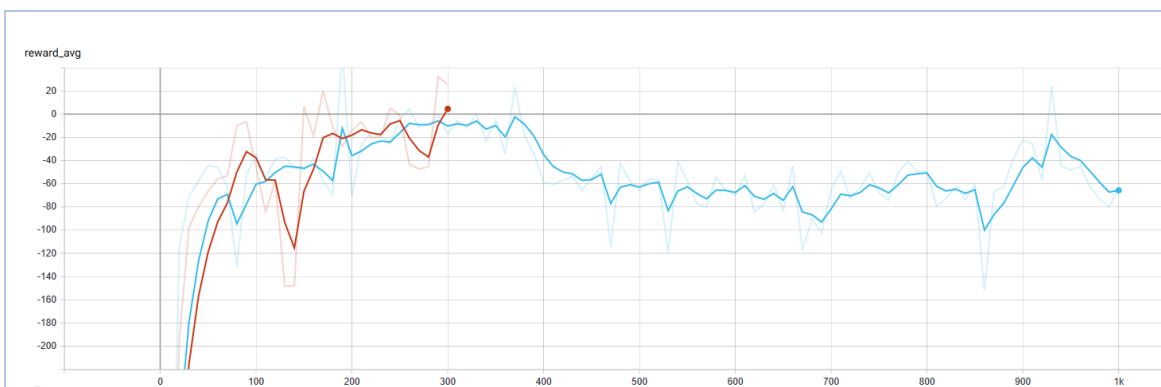


Figure 8.26. Average reward traces on red (model trained for 300 episodes) and on blue (model trained for 1000 episodes) for the model defined in TC.04.

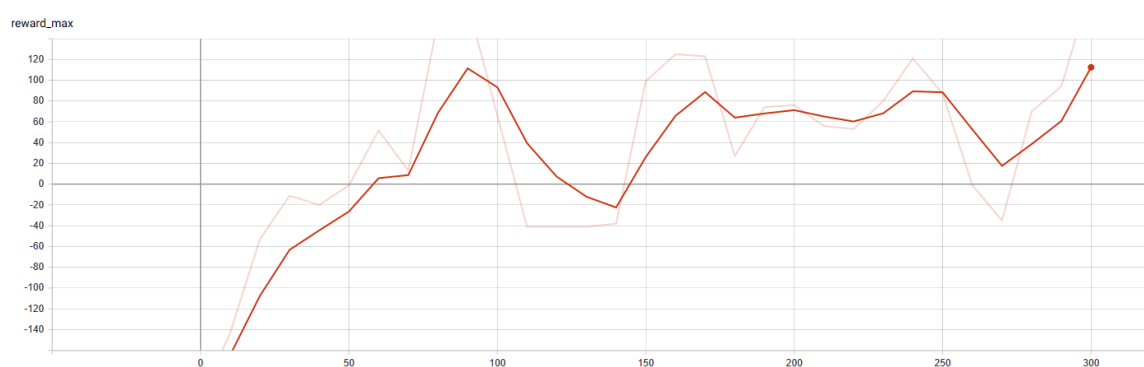


Figure 8.27. Maximum reward trace for the model defined in TC.04 trained during 300 episodes.

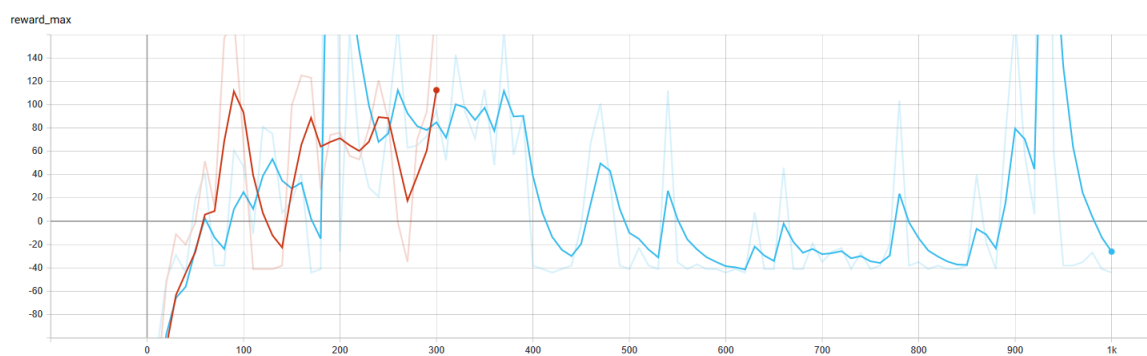


Figure 8.28. Maximum reward traces on red (model trained for 300 episodes) and on blue (model trained for 1000 episodes) for the model defined in TC.04.

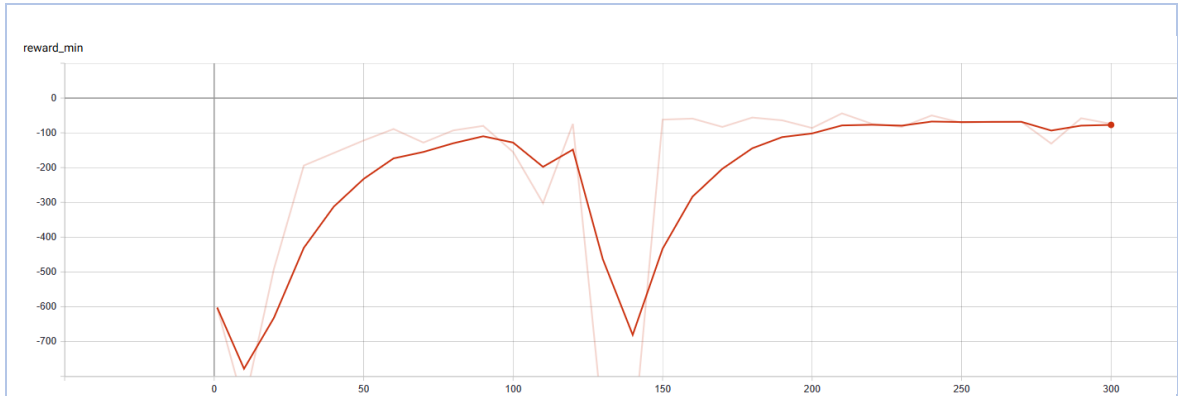


Figure 8.29. Minimum reward trace for the model defined in TC.04 trained during 300 episodes.

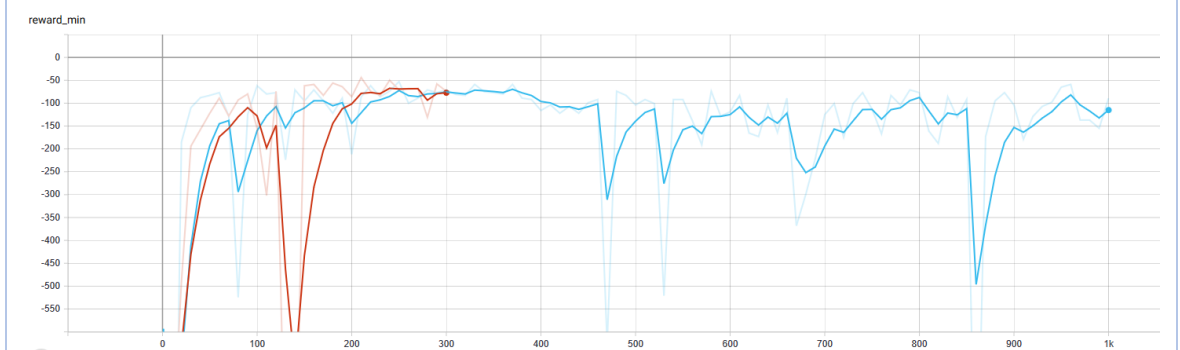


Figure 8.30. Minimum reward traces on red (model trained for 300 episodes) and on blue (model trained for 1000 episodes) for the model defined in TC.04.

8.3.2.5. TC.05 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	2h 31m 20s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Modified Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 20 seconds without collision: +5 points		

	Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (< 20 seconds): -10
Sensors used	RGB Camera Collision detection sensor

General comments of the model

From the bad experience obtained during all the four previous Test Cases while spawning on a random point, the final decision of setting a fixed spawn point is finally made in this fifth Test Case. By setting a fixed spawn position the main aim is not only to allow the agent to focus on learning step by step on a particular situation, but also to start putting the agent on “real” conditions as the racetrack will always have a fixed start point.

All the other parameters, including rewards and actions remain the same as in TC.04.

Statistics and performance on track

During this test case a previous training for about 300 episodes to see the general performance of the model has been made prior training it for longer. As this previous training seemed to work more or less (the statistics are not quite bad and by the performance on track the model doesn't really showed a bad behavior, or at least it was not clear), a second round previewed for 1000 episodes was made.

At least according to accuracy, the model seems to perform a little bit better than in the previous Test Case. However the average reward is significantly lower as on the fixed spawn point the agent is not able to accelerate on a straight line (which granted several reward points on the previous Test Case) so it seems the statistics start reflecting in a more accurate way the performance of the agent on track.

Even though the model was expected to train for 1000 episodes, by looking at the performance on track the model reached so easily a point where already had the learning rate so low that it stopped the exploration phase and continued learning on base of the already learned values. This was certainly a problem, as the model learned that the best combination of movements was, instead of going ahead with the direction of the road, doing a loop and going backwards. This led to a much later crash but incorrect driving behavior and without possibility to recover even if trained for longer

due to the low learning rate value. For this reason, the Test Case was stopped at the half of the expected training as it would be pointless to waste more time with this obvious problem.

The overall conclusion of this Test Case is that the statistics start reflecting what is really going on the performance of the model on track. The negative conclusion is that the learning rate is decreased really fast (which is bad for long training episodes but good for testing on short ones as 300 episodes) and also that it seems to be necessary to force the agent to go forward or it will choose a much easier road.

Graphical statistics

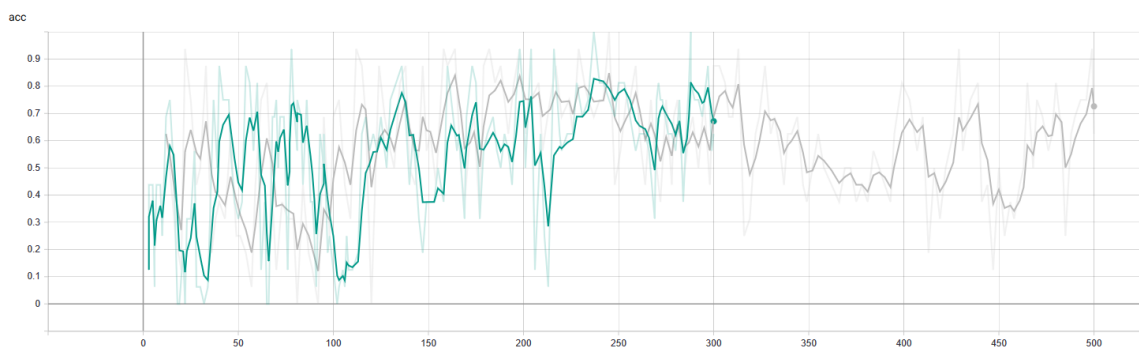


Figure 8.31. Accuracy traces on blue (model trained for 300 episodes) and on grey (model trained for 500 episodes) for the model defined in TC.05.

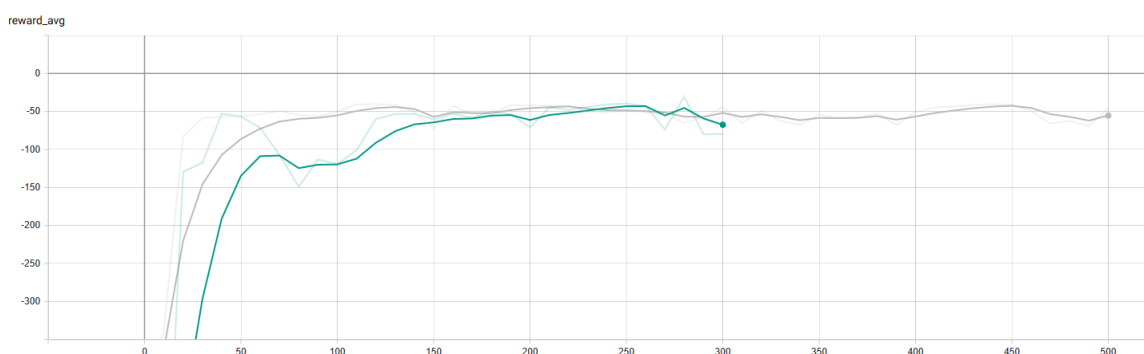


Figure 8.32. Average reward traces on blue (model trained for 300 episodes) and on grey (model trained for 500 episodes) for the model defined in TC.05.

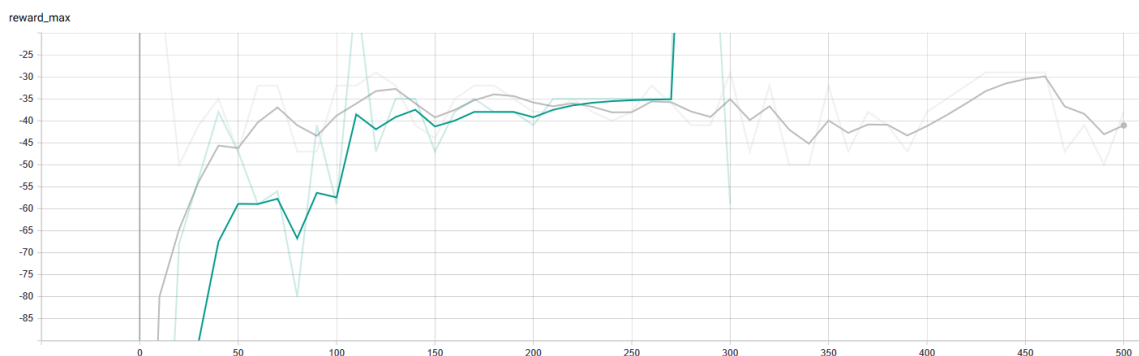


Figure 8.33. Maximum reward traces on blue (model trained for 300 episodes) and on grey (model trained for 500 episodes) for the model defined in TC.05.

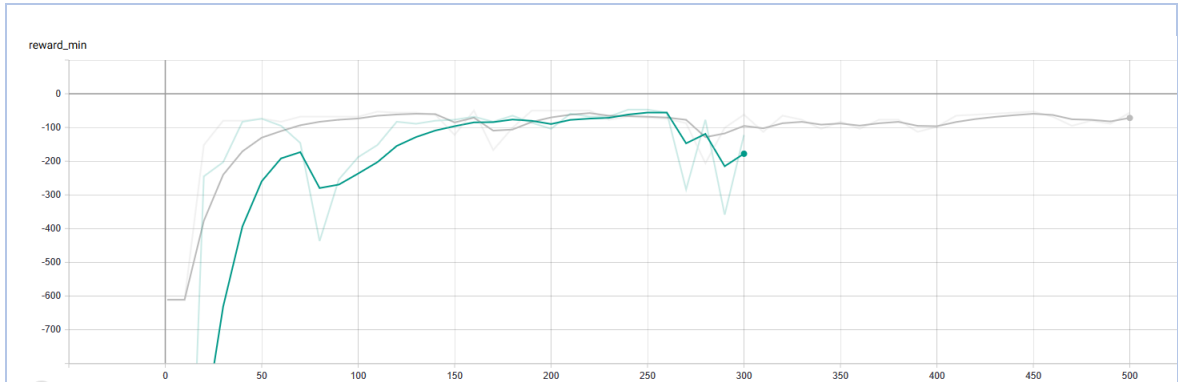


Figure 8.34. Minimum reward traces on blue (model trained for 300 episodes) and on grey (model trained for 500 episodes) for the model defined in TC.05.

8.3.2.6. TC.06 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	3h 10m 48s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Modified condition Reward 4: > 30 seconds without collision: +5 points Added Reward 5: Driving to an end point: $-0,1 * \text{distance to point}$ Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Modified condition Reward 9: Short training episode (< 25 seconds): -10		
Sensors used	RGB Camera Collision detection sensor		

General comments of the model

In order to solve one of the main problems of the previous Test Case, which was that the agent chose to go backwards to find a much attractive path to drive in, a new reward is added to force it to go to a certain point (point B) that can be accessed easily (more easily than doing a turn and going backwards) by going forward.

Statistics and performance on track

As having introduced a reward that is always negative (of course the magnitude of the value depends on how far away from the destination point is the agent) it is clear that all the statistics will be lower than in the previous models. For this reason, although by looking at the graphs the model seems to be the worst by far, in this Test Case the statistics are not so important but the performance on track to see if this new reward is able to solve the problem previously observed.

Unfortunately, the destination point doesn't solve the problem as the agent continues going backward because apparently by the design of the map itself the distance from point A to B going forward and backward seems to be more or less the same. An added point is that by going forward the agent collides faster than by going backwards, so it accumulates a higher penalty which is then translated on learning that the forward path is not good.

The overall conclusion of this Test Case is that fixing a destination point may be useful if the path to arrive there is unique, as the agent will respect the natural circulation direction of the road. However in a real scenario this is not an applicable reward as the destination point might not be known or accessible.

Graphical statistics

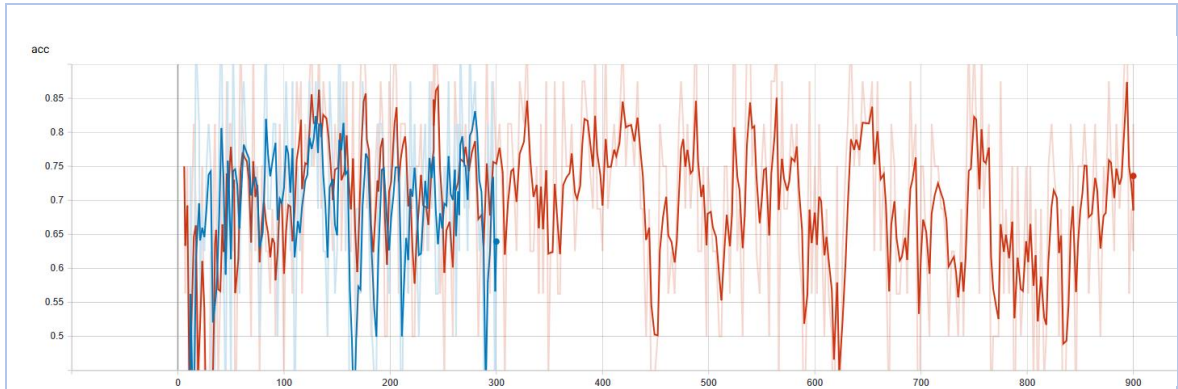


Figure 8.35. Accuracy traces on blue (model trained for 300 episodes) and on red (model trained for 900 episodes) for the model defined in TC.06.

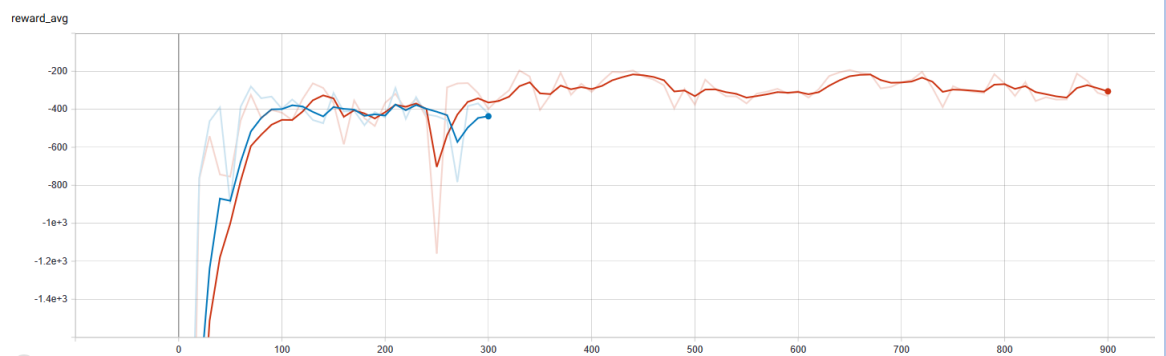


Figure 8.36. Average reward traces on blue (model trained for 300 episodes) and on red (model trained for 900 episodes) for the model defined in TC.06.

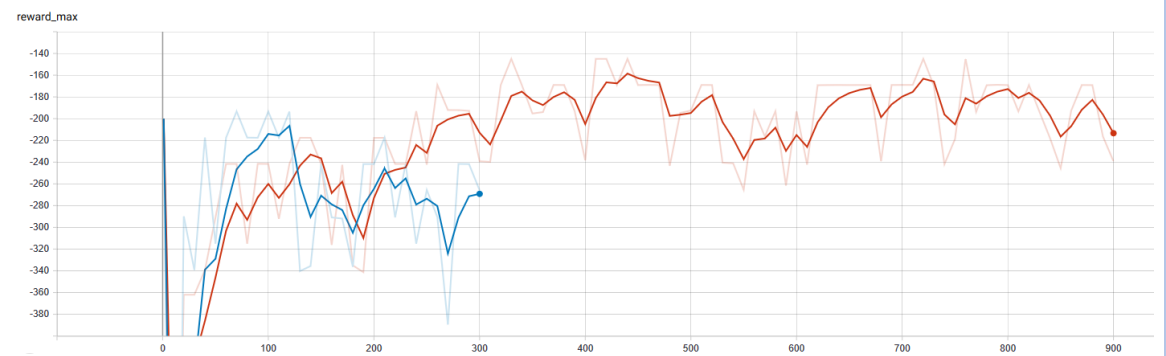


Figure 8.37. Maximum reward traces on blue (model trained for 300 episodes) and on red (model trained for 900 episodes) for the model defined in TC.06.

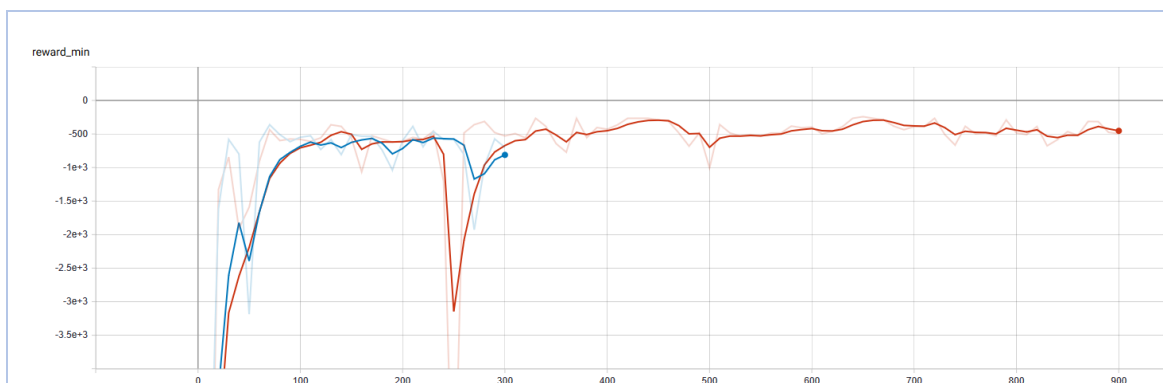


Figure 8.38. Minimum reward traces on blue (model trained for 300 episodes) and on red (model trained for 900 episodes) for the model defined in TC.06.

8.3.2.7. TC.07 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	48m 10s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 30 seconds without collision: +5 points Reward 5: Driving to an end point: $-0,1 * \text{distance to point}$ Reward 6: Turn smoothness: Reward 8: No movement: -8 points Reward 9: Short training episode (< 25 seconds): -10 Added Reward 10: Driving on track center: $-0,1 * \text{distance to track center}$		
Sensors used	RGB Camera Collision detection sensor		

General comments of the model

This Test Case can be considered as an independent test. The main aim of it is to test if it is possible to redirect and make a little bit smoother the trace of the agent by promoting to go to the middle of the track by introducing a new reward which is detailed in reward number 10.

Statistics and performance on track

The statistics show a real bad performance as the base model is the same as TC.06 which already showed to have low statistics. However, comparing both models, by introducing in TC.07 reward number 10 it seems to perform a little bit better as it forces a little bit more the model to go to a certain path and direction.

By looking at the performance of track a clear limitation of the simulator shows up: the track center is not calculated for all along the track so at certain points the track center is marked really far away from where the agent is training. Nevertheless, leaving that aside, the agent doesn't seem much to respect that condition as it continues driving on the limits of the track

The overall conclusion of this Test Case is that although the idea might seem interesting on a final application, for a prototype and in a real initial training phase is quite a restrictive condition that doesn't allow the agent to correctly learn. This condition would have a better effect if it is applied once the model already knows how to drive even if it is in a very rough way.

Graphical statistics

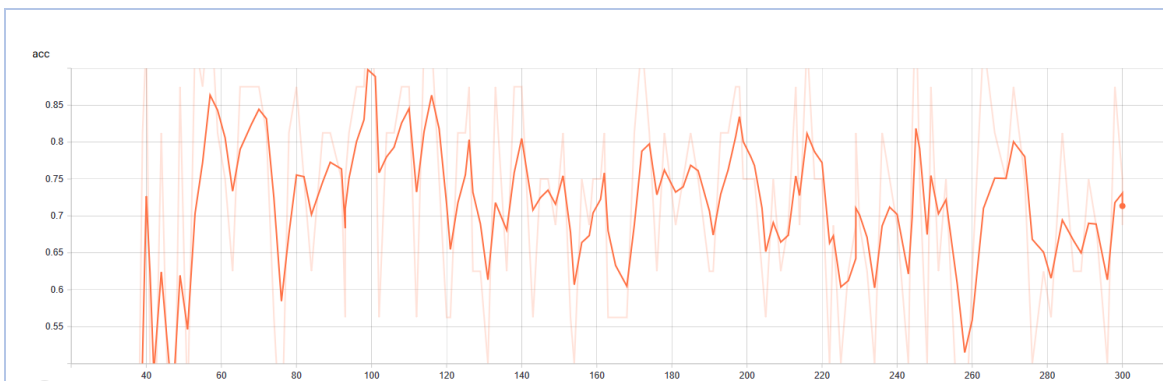


Figure 8.39. Accuracy trace for the model defined in TC.07 trained during 300 episodes.

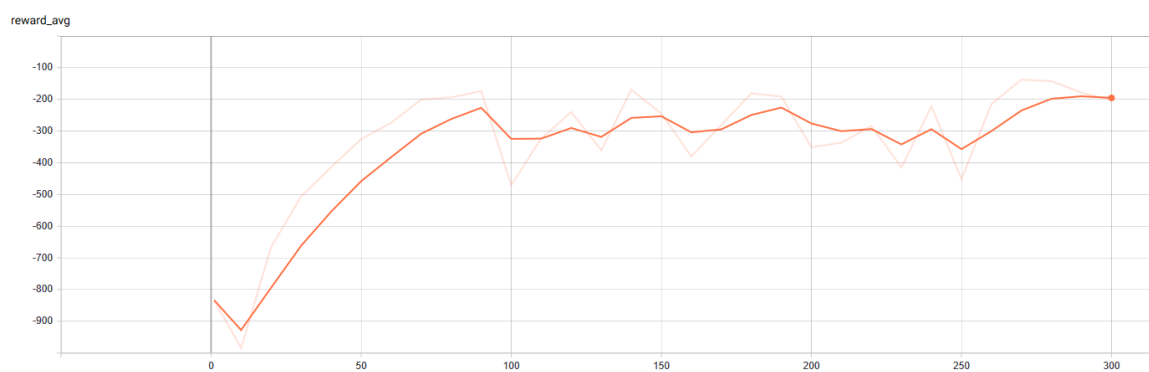


Figure 8.40. Average reward trace for the model defined in TC.07 trained during 300 episodes.

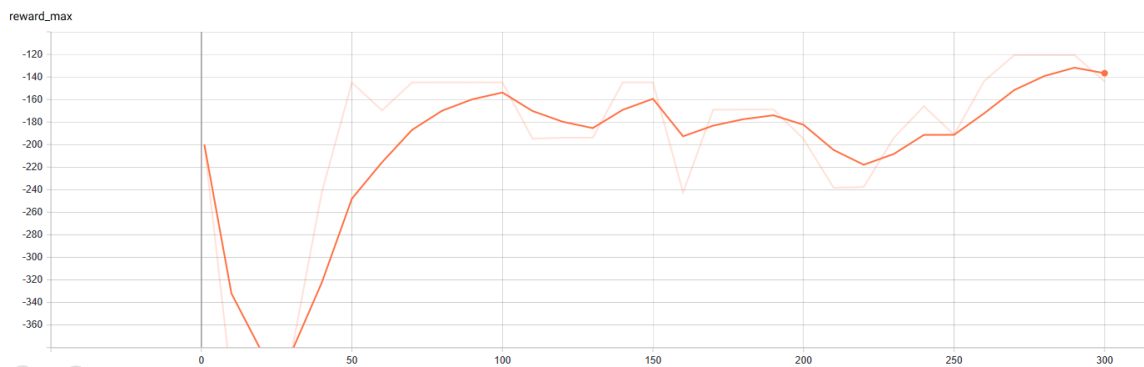
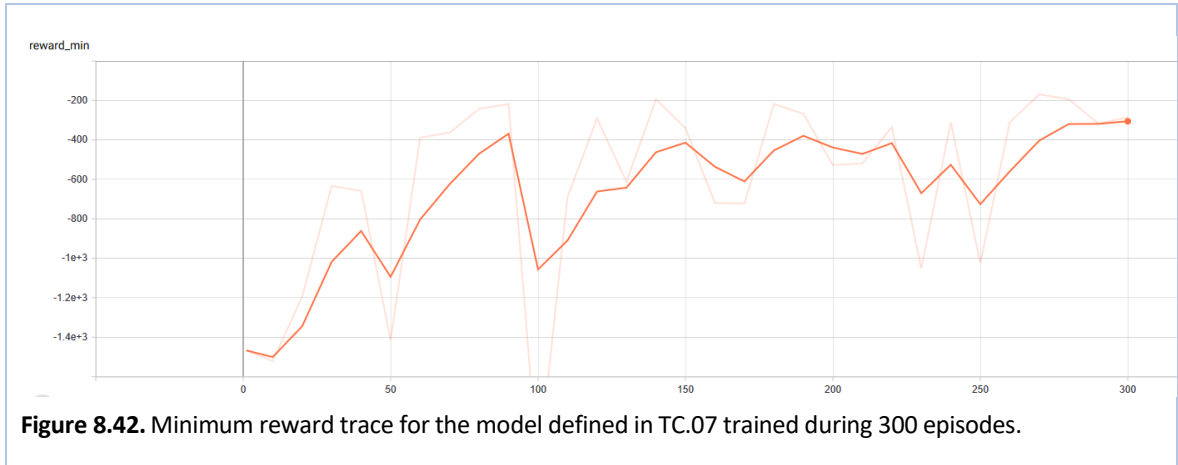


Figure 8.41. Maximum reward trace for the model defined in TC.07 trained during 300 episodes.



8.3.2.8. TC.08 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	56m 30s
Discount factor	0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 30 seconds without collision: +5 points Removed reward 5 Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (< 25 seconds): -10 Removed reward 10		
Sensors used	Removed RGB camera Added Semantic Segmentation camera Collision detection sensor		

General comments of the model

As the model used in TC.06 didn't seem to have much more room for improvement other than changing the destination point or modifying the learning rate and additionally the sensor used is also limiting the performance as no image treatment is performed, a radical change is done in this Test Case.

The main RGB camera is changed for the same camera but applying a Semantic Segmentation to the captured image. This will allow the agent to distinguish between the road and the outside.

Also, the destination point is removed and the driving on the middle of the road condition is permanently deleted. As a comparison point, this Test Case is similar to TC.05 only changing the used sensor.

Statistics and performance on track

In order to judge if this model can lead to an improvement of the model, a direct comparison with the model used in TC.05 has to be done. The first item to compare is accuracy: on one hand in TC.08 the fluctuation between one episode and another is quite significant although the minimum and maximum levels are acceptable (from approximately 0,40 to 0,85). On the other hand, in TC.05 the fluctuation is even more exaggerated (from 0,1 to 0,8) and the maximum accuracy level is worse. So, in terms of accuracy, the change on the input sensor data tested on TC.08 seems to work better.

Comparing the reward traces, the average reward seems to be more or less the same even though the model from TC.08 doesn't show a clear tendency due to its lack of training episodes. Concerning the max reward, on TC.08 there is an evident positive tendency while on TC.05 the tendency was to either be stable or go down.

The performance on track of this model seemed to be more hopeful as it ceased to choose to go backwards or colliding while going straight. The accuracy really starts reflecting what the agent does.

The overall conclusion of this Test Case is that switching from one sensor to another does the fact. Seems that with the right training parameters this input data has much to offer.

Graphical statistics

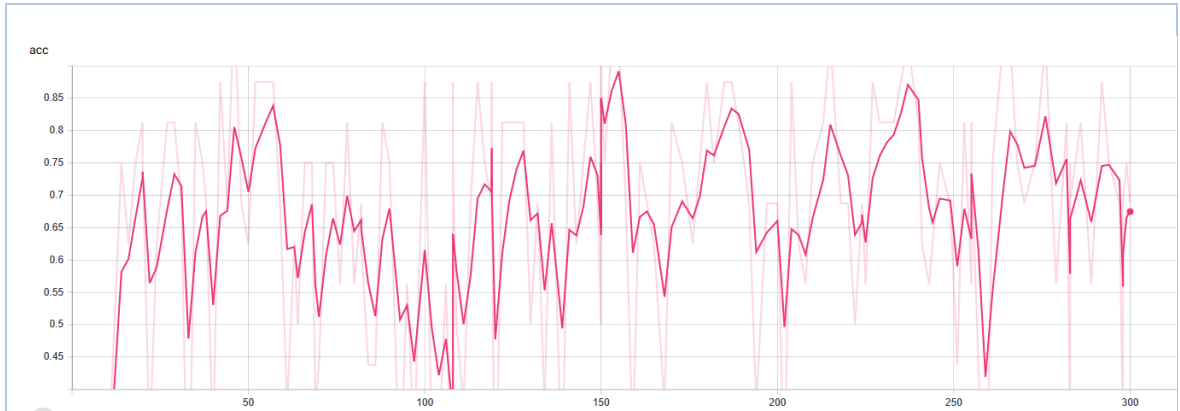


Figure 8.43. Accuracy trace for the model defined in TC.08 trained during 300 episodes.

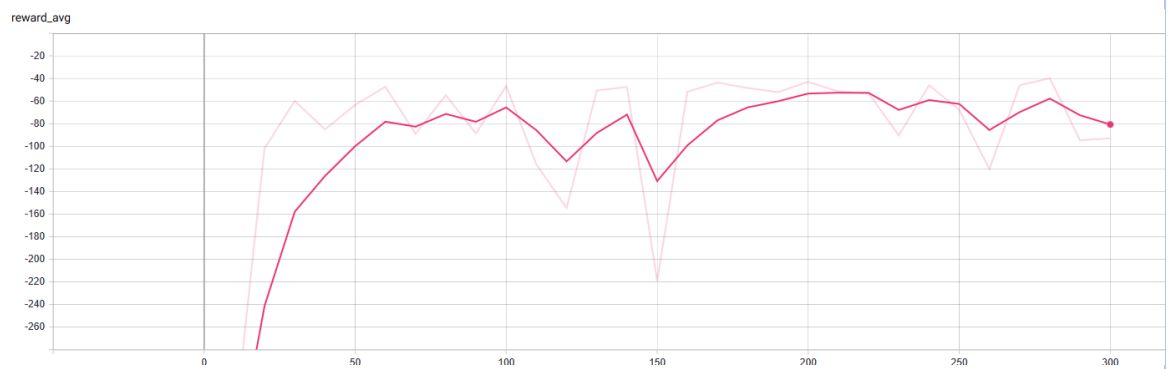


Figure 8.44. Average reward trace for the model defined in TC.08 trained during 300 episodes.

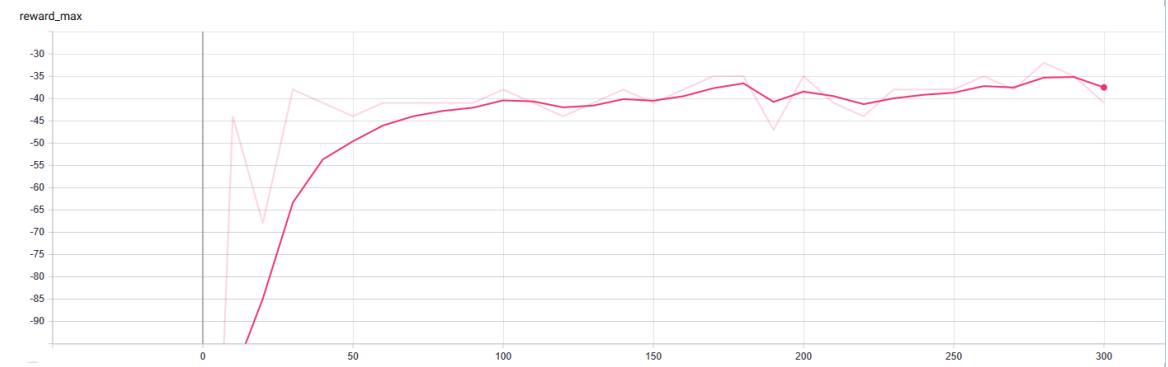


Figure 8.45. Maximum reward trace for the model defined in TC.08 trained during 300 episodes.

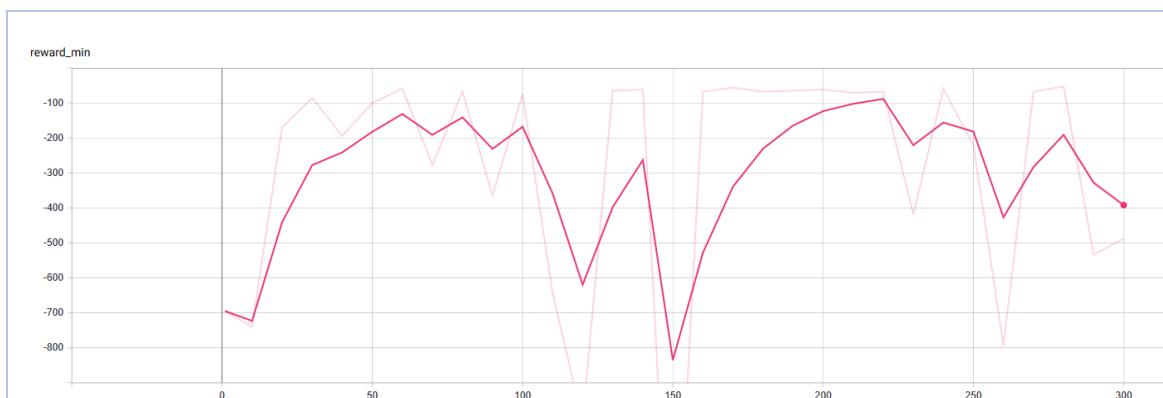


Figure 8.46. Minimum reward trace for the model defined in TC.08 trained during 300 episodes.

8.3.2.9. TC.09 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	1h 15m 48s
Discount factor	Modified 0,97	Learning rate (α)	Modified Downgrading at a rate: α * 0,9975
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 30 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (< 25 seconds): -10		
Sensors used	Semantic Segmentation camera Collision detection sensor		

General comments of the model

Having already checked that Semantic Segmentation camera works better than the normal RGB camera, it is now time to start playing with the training parameters. In this particular Test Case, the discount factor has been increased to a value of 0,97 and also the downgrade for the learning rate has been increased to a value of 0,9975. By doing so, the model is expected to give even more weight to becoming a good reward at a long term and delay the exploitation phase.

Statistics and performance on track

Comparing the accuracy from this model to the previous one, they look quite similar: same minimum and maximum values and still the high fluctuation is present. There are no improvements in this aspect.

Looking at the reward traces, however, this model seems to be much more stable than the previous (remember that the traces on TC.08 were like a roller coaster) and with a clear positive tendency while only trained for a tiny episode number. This fact is a good indicator that by upward modifying the discount factor (just by a 2%), the model focuses on long term results while already taking care to make a progressive learning. Unfortunately, this progressive learning is not reflected on the performance on track, as the desired behavior doesn't show up at the end of the training.

The overall conclusion of this Test Case is that it is a stable save point that can be recovered later on if other Test Cases with different values doesn't bring on good results both in a statistics level and on track.

Graphical statistics

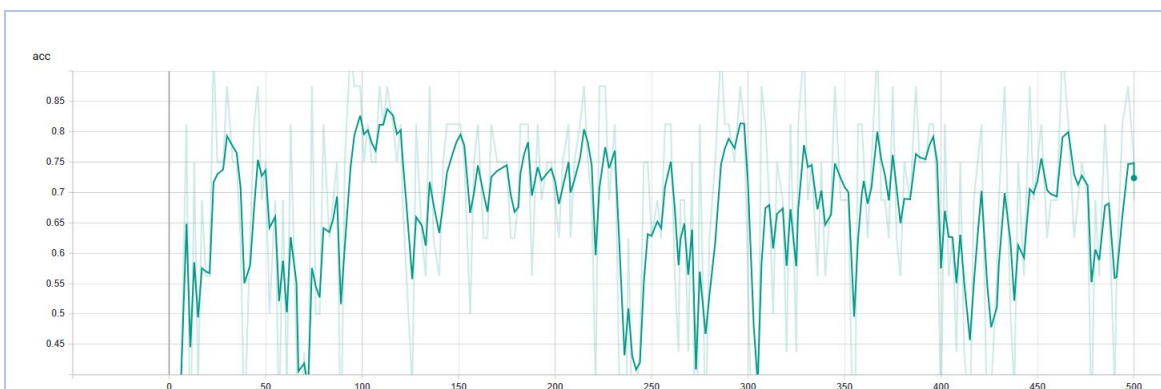


Figure 8.47. Accuracy trace for the model defined in TC.09 trained during 500 episodes.

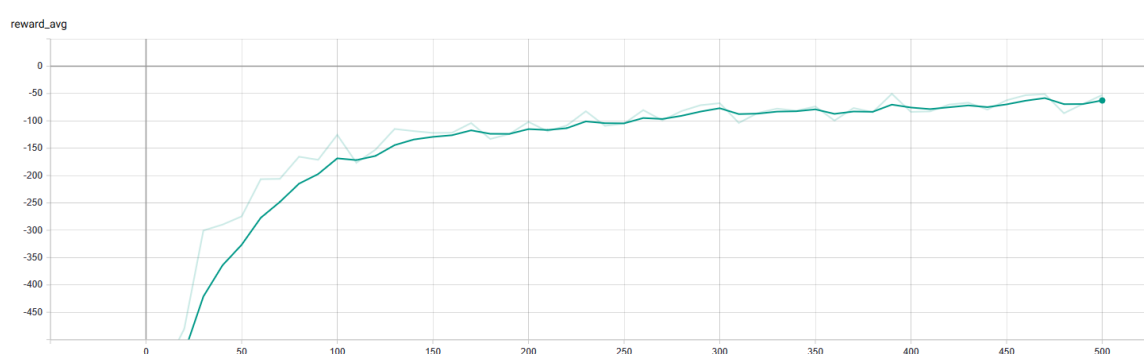


Figure 8.48. Average reward trace for the model defined in TC.09 trained during 500 episodes.

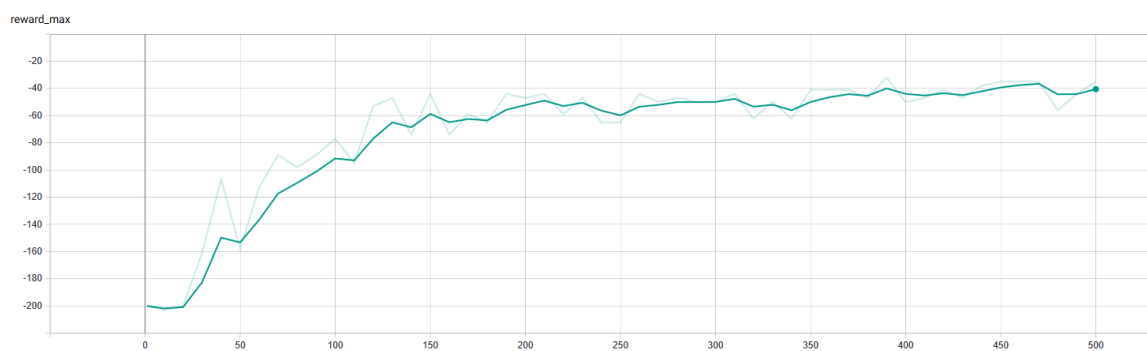


Figure 8.49. Maximum reward trace for the model defined in TC.09 trained during 500 episodes.

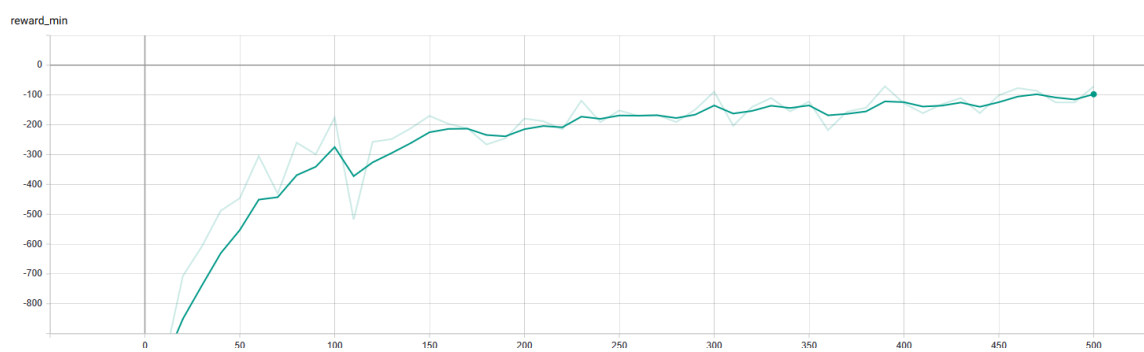


Figure 8.50. Minimum reward trace for the model defined in TC.09 trained during 500 episodes.

8.3.2.10. TC.10 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	7h 54m 13s
Discount factor	Modified 0,95	Learning rate (α)	Downgrading at a rate: $\alpha * 0,9975$
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 30 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (< 25 seconds): -10		
Sensors used	Semantic Segmentation camera Collision detection sensor		

General comments of the model

Starting from the base model trained on TC.08 and with the actually good performance on TC.09, this Test Case pretends to isolate the parameter Learning Rate to see whether it is the one that plays a major role on developing a good training performance or not.

The downgrade of the Learning Rate is not modified, being again at a value of 0,9975 and the Discount Factor is downgraded to its original value of 0,95.

Statistics and performance on track

This model seems to have a good start point until 200 episodes trained, as its average accuracy is around 0,65, but it is further reduced as it follows a negative tendency as the model gets deeper in episode number which didn't happen on the previous Test Case (TC.09). All the other traces referring to reward doesn't seem to follow the same tendency as accuracy.

Making a comparison between TC.09 and TC.10, having adjusted the value of Discount Factor doesn't seem to affect the model either in a positive or a negative way as the results of both Test Cases are similar. Nonetheless the performance on track has been observed to be better on this last Test Case although it could not be due to the change on Discount Factor value but affected by the randomness of the model itself. It has to be pointed out that even this model performs better on track, it stills does bad movements which are going straight and colliding frontally at a high velocity without even exploring other possibilities.

In order to discard the random factor on the effectiveness of the model, a second model running under the TC.10 conditions has been trained a little bit longer (defined as base model). This model has been set to be the base to train 2 more times, again under TC.10 conditions with just few modifications, to exploit the model designed in this Test Case and choose whether to discard it or go ahead on this way.

The statistics for the three mentioned models can be seen below but the general performance of each and every one of them seems to discard this model by itself, as the most common movement which has already been learned before, is not modified. This entails that the model stagnates at a certain point and is not able to evolve in any situation even if trained for much longer. Besides that, the statistics doesn't seem either to be favorable.

The overall conclusion of this Test Case (although it has already been seen along the training with the other Test Cases) is that the learning rate plays a major role in the training of the models and can not be reduced drastically and not further recovered. If done so, the agent will learn a certain pattern in the exploration phase and once the learning rate is so small that it no longer explores the environment it will start recovering its past knowledge to exploit what it already knows. This means that once learned a wrong sequence, it will continue to repeat it as it won't have the opportunity to go back and modify it to a better one. The agent will go on and on with the same movement sequence until the end of the training and it won't have learned anything valuable.

For this reason, several modifications will come on the next Test Case.

Graphical statistics

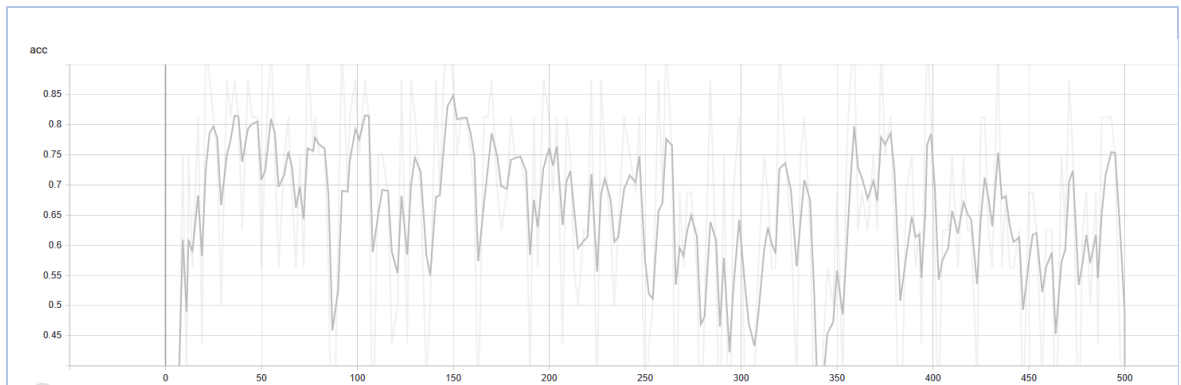


Figure 8.51. Accuracy trace for the model defined in TC.10 trained during 500 episodes.

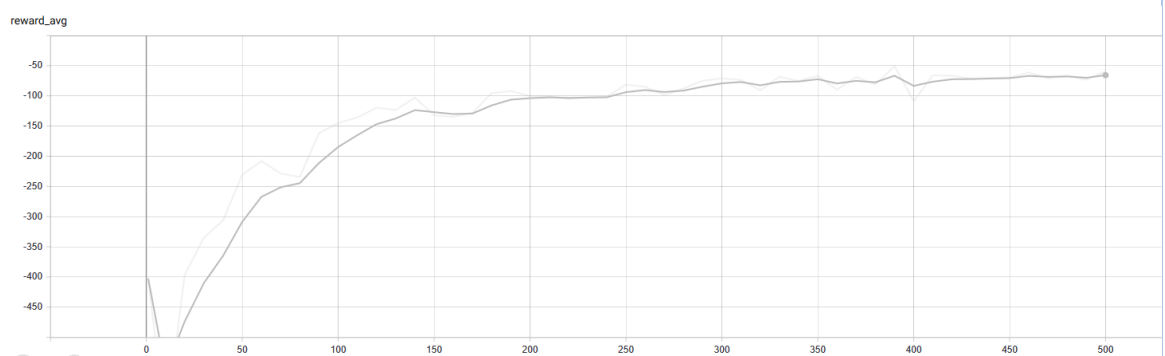


Figure 8.52. Average reward trace for the model defined in TC.10 trained during 500 episodes.

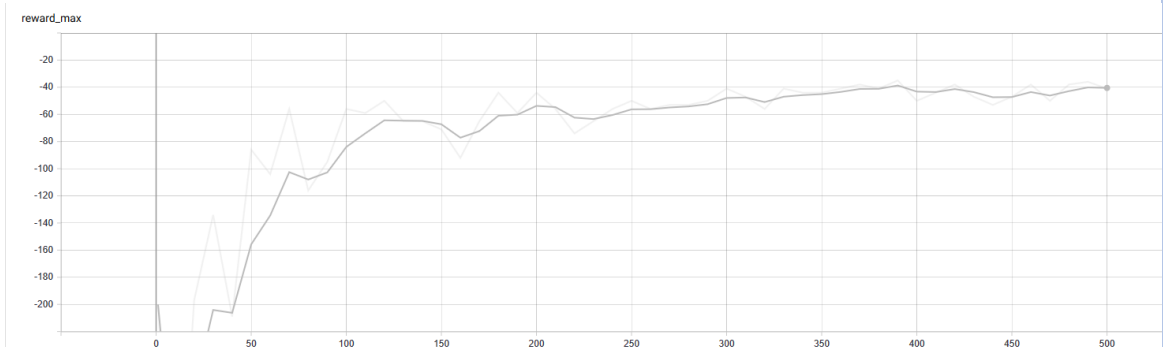


Figure 8.53. Maximum reward trace for the model defined in TC.10 trained during 500 episodes.

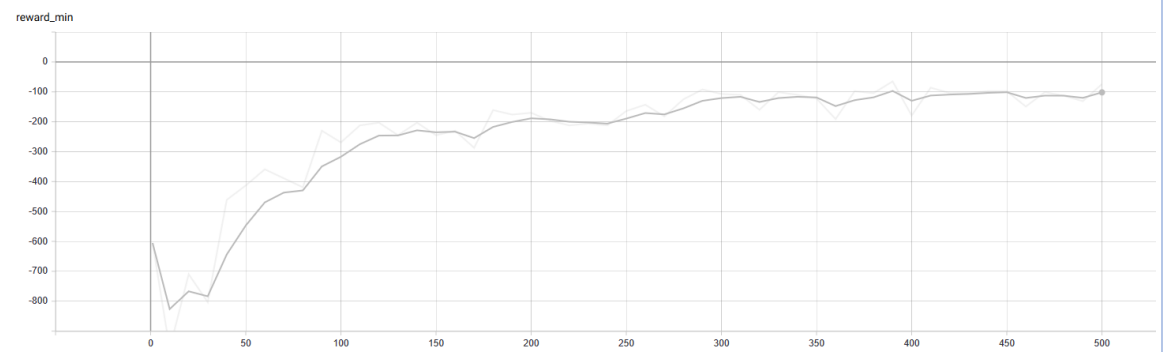


Figure 8.54. Minimum reward trace for the model defined in TC.10 trained during 500 episodes.

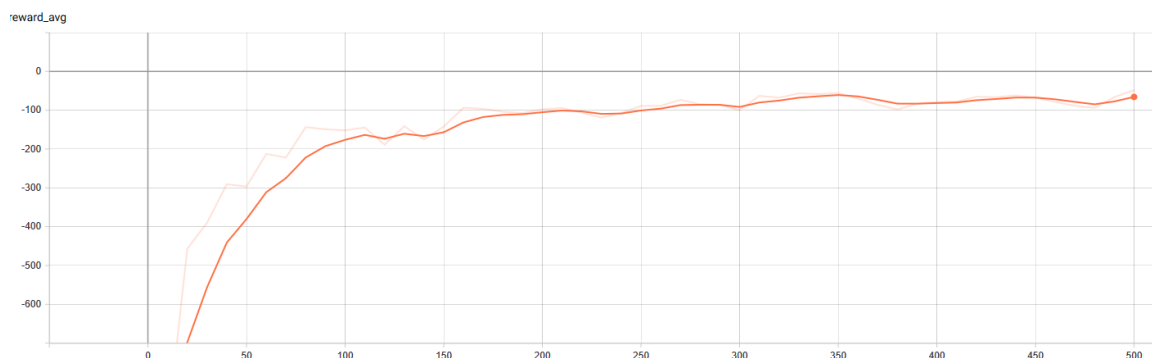


Figure 8.55. Average reward trace for the model trained from the base model defined in TC.10 trained during 500 episodes.

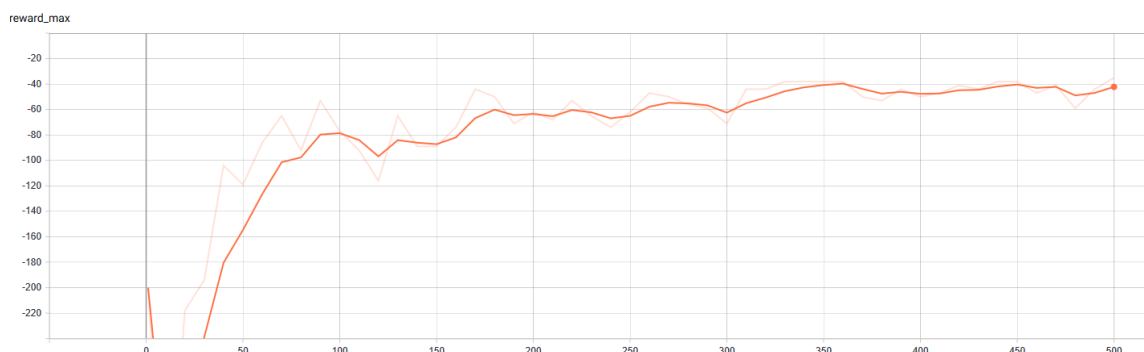


Figure 8.56. Maximum reward trace for the model trained from the base model defined in TC.10 trained during 500 episodes.

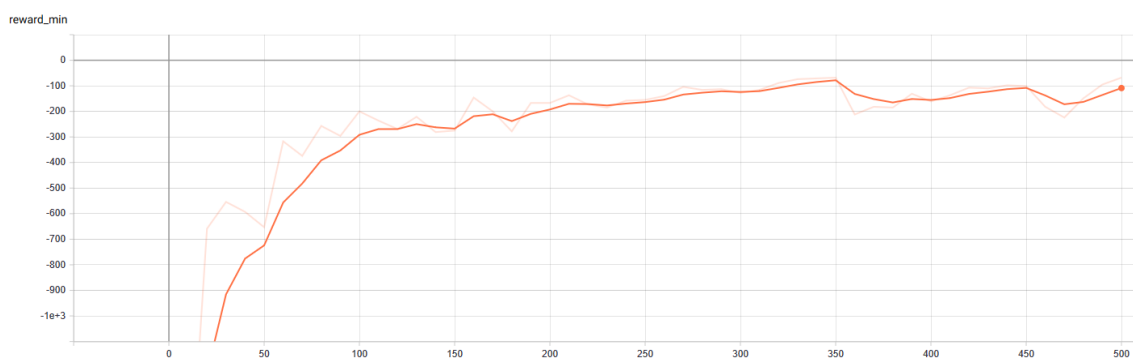


Figure 8.57. Minimum reward trace for the model trained from the base model defined in TC.10 trained during 500 episodes.

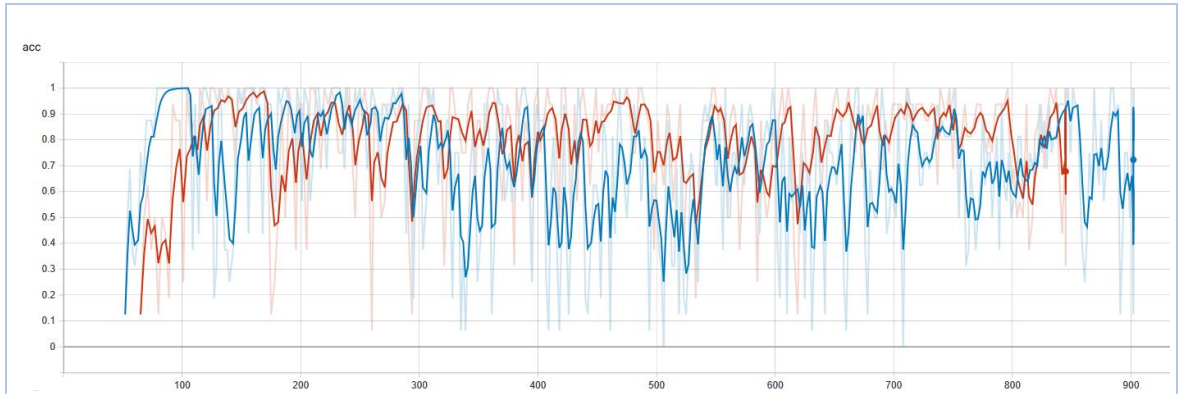


Figure 8.58. Accuracy traces on blue (variation 1 - model trained 900 episodes) and on red (variation 2 - model trained during 850 episodes) for the models trained after the base model.

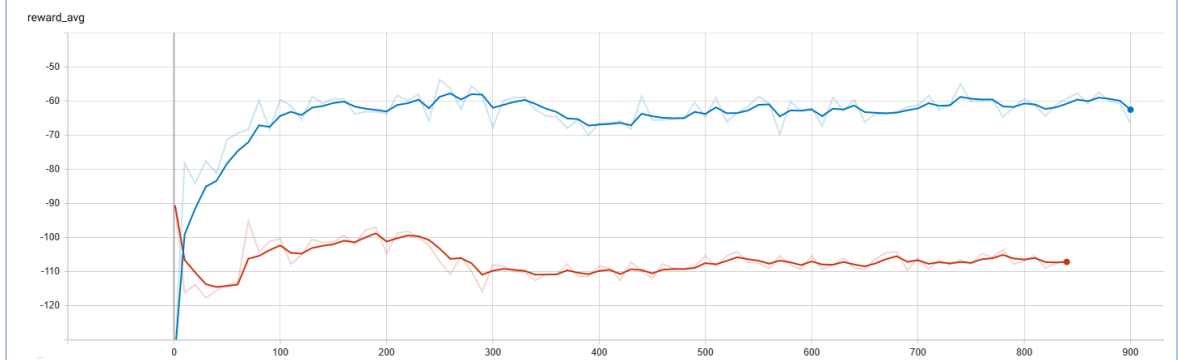


Figure 8.59. Average reward traces on blue (variation 1 - model trained 900 episodes) and on red (variation 2 - model trained during 850 episodes) for the models trained after the base model.

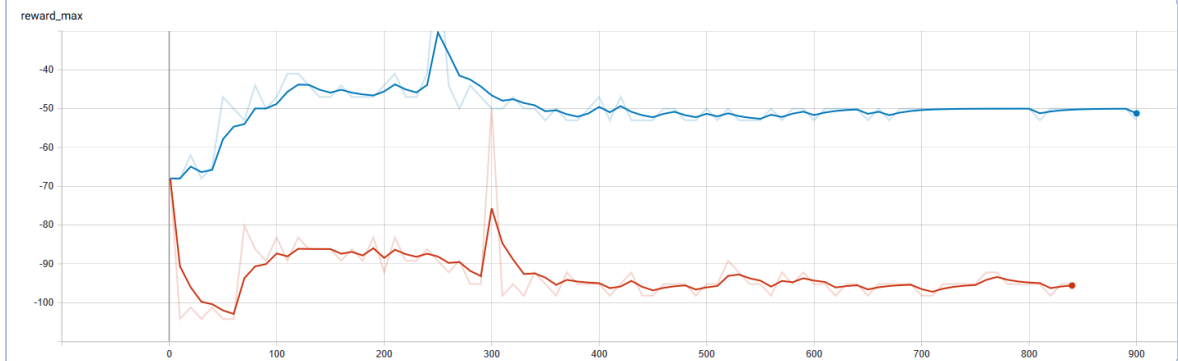


Figure 8.60. Maximum reward traces on blue (variation 1 - model trained 900 episodes) and on red (variation 2 - model trained during 850 episodes) for the models trained after the base model.

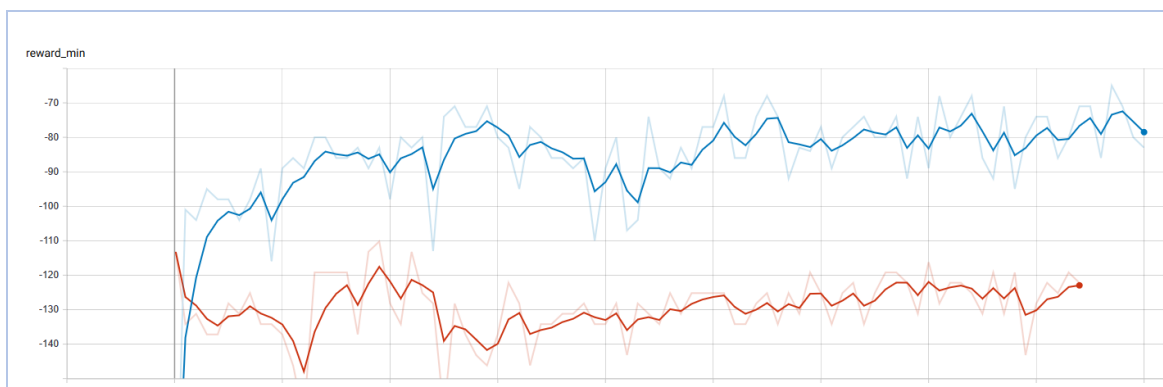


Figure 8.61. Minimum reward traces on blue (variation 1 - model trained 900 episodes) and on red (variation 2 - model trained during 850 episodes) for the models trained after the base model.

8.3.2.11. TC.11 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	7h 59m 41s
Discount factor	0,95	Learning rate (α)	Special condition, see general comments of the model.
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 30 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (< 25 seconds): -10		
Sensors used	Semantic Segmentation camera Collision detection sensor		

General comments of the model

The most critical point on all the previous Test Cases has been observed to be the learning rate. In this Test Case, it will not only decrease but also be able to increase to avoid the stagnation of the model on a wrong sequence. The learning rate will now behave like that:

1. If it decreases, it will decrease at a rate $\alpha * 0,99975$
2. If it increases, it will increase at a rate $\alpha * 1,025$ (notice that it recovers faster than decreases)
3. The agent will start with $\alpha = 1$ and will decrease until one (or both) of the following conditions are true:
 1. $\alpha < 0,4$ and minimum reward < -50
 2. minimum reward < -50 and current episode $> \text{total episodes}/2$

By doing so, the learning rate will be dynamically updated if the agent does not perform as expected and will be able to correct itself without any interaction.

Statistics and performance on track

By looking at the graph of the learning rate, it can be observed that under the test conditions the trigger level in which this parameter starts recovering is never met, as this value doesn't reach the set value of 0,4. However it do can be observed that, by decreasing the learning decay, so the model can explore during much longer, the model starts performing better on track.

As for the statistics the conditions under TC.11 the model seems to reach a point of stability at the end of the training episodes. Visibly the model is not able to evolve from that point as the curve gets into a flat tendency (which clearly doesn't happen on TC.10) but eventually this tendency could be broken if trained for longer and the learning rate recovery is activated. As a positive point of view, even if the model gets stuck on a flat tendency, at least it doesn't follow a decreasing tendency.

Observing the car performance on track, it has been decided to change the reward related to "long time" episode in order to encourage the agent in further more correct decisions: currently the agent is never able to reach the set "long time" reward. Even though no more rewards are expected to be modified, the "velocity" reward is under study for the next test cases as if the "long time" reward is decreased in order to fit the behavior of the model the velocity set is not reached in most cases.

The overall conclusion of this Test Case is that although its main purpose has not been successfully reached, it can establish a better base for further test cases. All in all, the test cases not always come

out the desired way but as long as they serve to point out improvements on the model they are useful.

Graphical statistics

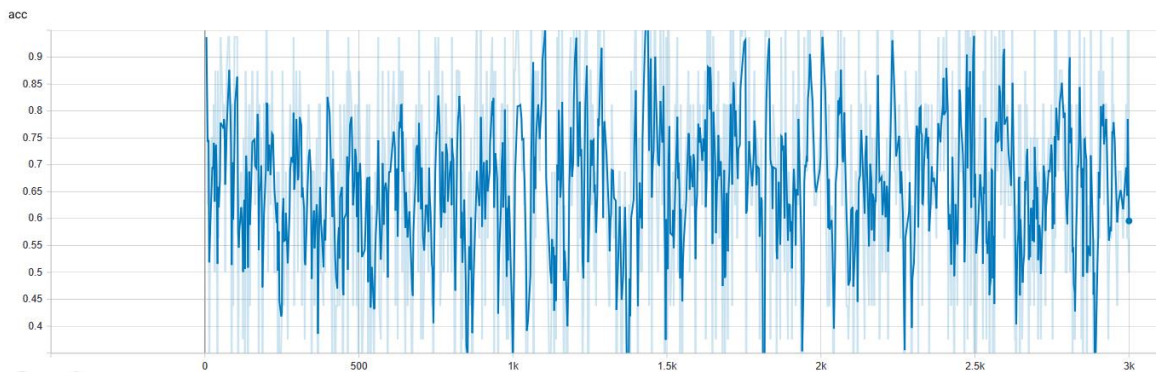


Figure 8.62. Accuracy trace for the model defined in TC.11 trained during 3k episodes.

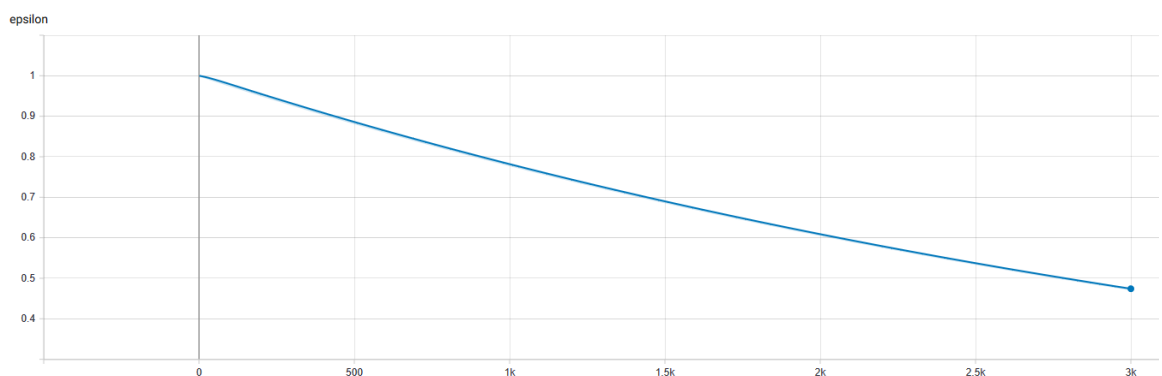


Figure 8.63. Learning rate trace for the model defined in TC.11 trained during 3k episodes.



Figure 8.64. Average reward trace for the model defined in TC.11 trained during 3k episodes.

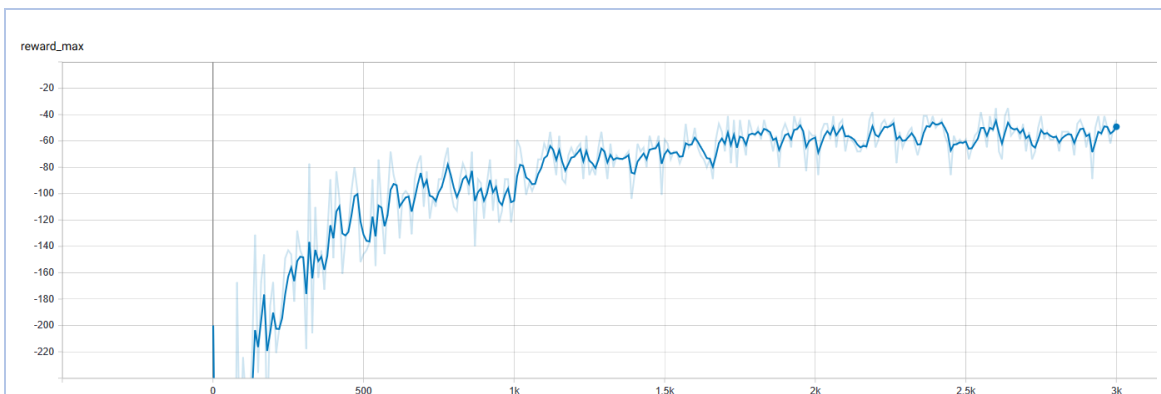


Figure 8.65. Maximum reward trace for the model defined in TC.11 trained during 3k episodes.



Figure 8.66. Minimum reward trace for the model defined in TC.11 trained during 3k episodes.

8.3.2.12. TC.12 – Map07

Training conditions			
GPU fraction used	0,4	Total trained time	2d 19h 6m 43a
Discount factor	0,95	Learning rate (α)	Special condition, see general comments of the model.
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Modified Reward 4: > 10 seconds without collision: +5 points		

	Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Modified Reward 9: Short training episode (≤ 10 seconds): -10
Sensors used	Semantic Segmentation camera Collision detection sensor

General comments of the model

The main problems observed in the previous Test Case are expected to be solved in TC.12 by doing the following:

1. The learning rate condition is modified, see below.
2. It has been observed during TC.11 that the average frontier time that reflects if the agent performs good or bad (colliding in the first stage) is 10 seconds. Above that, it is able to wander more racetrack. According to that observation, rewards 4 and 9 have been modified.
3. The velocity reward is being overlooked in case it is necessary to reduce the threshold.

According to point 1, the learning rate now behaves as follows:

1. If it decreases, it will decrease at a rate $\alpha * 0,99975$ (no modification)
2. If it increases, it will increase at a rate $\alpha * 1,000025$
3. The agent will start with $\alpha = 1$ and will decrease until one (or both) of the following conditions are true:
 - i. $\alpha <$ (lower threshold, varies depending on the model) and minimum reward < -80
 - ii. minimum reward < -80 and current episode $>$ total episodes/2
4. The learning rate will only be allowed to vary between a minimum and maximum value, which will be adjusted on every model running under TC.12 conditions.
 - i. Variation 1 – from 0,4 to 1
 - ii. Variation 2 – from 0,25 to 1
 - iii. Variation 3 – from 0,25 to 1

Statistics and performance on track

This Test Case tries to isolate and correct all the problems observed in TC.11, which are basically the learning rate algorithm. For this reason, three different variations are made based on the results of the previous ones.

The first variation starts from the same learning rate range (0,4 to 1) but increasing the decay so it can reach the minimum value within the trained episodes. Observing the graphs below (Figure 8.68) a periodical tendency is acquired: once the minimum value is reached it goes back to its maximum value again. During this recovery transition, the model is truncated as seen in the reward traces (Figure 8.70) as if it started the learning from zero, resetting it. This is clearly not a good performance of the model as it is not able to apply its learning, losing it.

In the second variation the learning rate range (0,25 to 1) is modified and the recovery is decreased. By applying these modifications, it is expected to give more space to the model so it can recover its knowledge for a little bit longer and increase its average reward, delaying the learning rate recovery phase. However, the expected demeanor of the model is not fulfilled, as the model is always under the average reward threshold. In fact, the same periodical tendency as in variation 1 is achieved, but delayed in time. This delay leads to a better performance both on track and on a statistics levels as the truncation of the model is also postponed.

In the last variation, the third one, the learning rate range remains constant and so does its recovery factor. The one modification made to the model is the average reward threshold that triggers the recovery phase, which has been decreased in order to prevent its premature recovery. Nevertheless, by having a look again on the graphs, the threshold is still not correct as the same periodical tendency is still present.

The last point to comment can be observed on the loss graph (Figure 8.69). All three variations of the model under test follow have the same curve: increasing the loss as the model gets trained for longer. This is clearly a bad sign that the model is overfitted, surely caused by the abrupt recovery phase, and needs to be corrected.

The overall conclusion of this Test Case is that although the recovery phase on the learning rate may be positive, it needs to be restricted to a certain range to prevent model's truncation and overfitting.

Graphical statistics

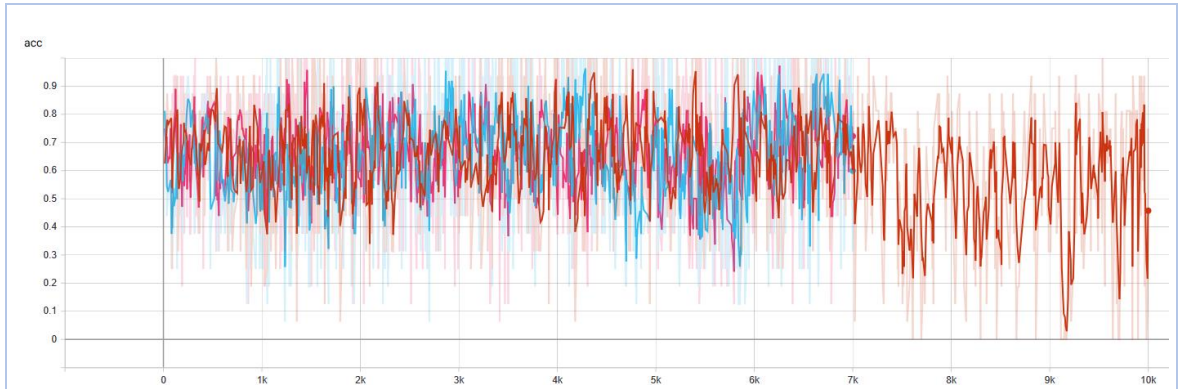


Figure 8.67. Accuracy traces on red (variation 1 – trained during 10k episodes), on blue (variation 2 – trained during 7k episodes) and on pink (variation 3 – trained during 7k episodes) for the model defined on TC.12.

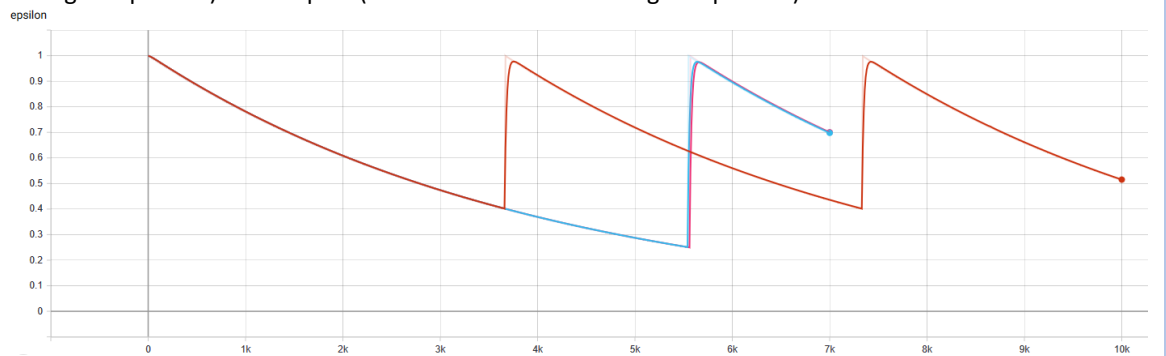


Figure 8.68. Learning rate traces on red (variation 1 – trained during 10k episodes), on blue (variation 2 – trained during 7k episodes) and on pink (variation 3 – trained during 7k episodes) for the model defined on TC.12.

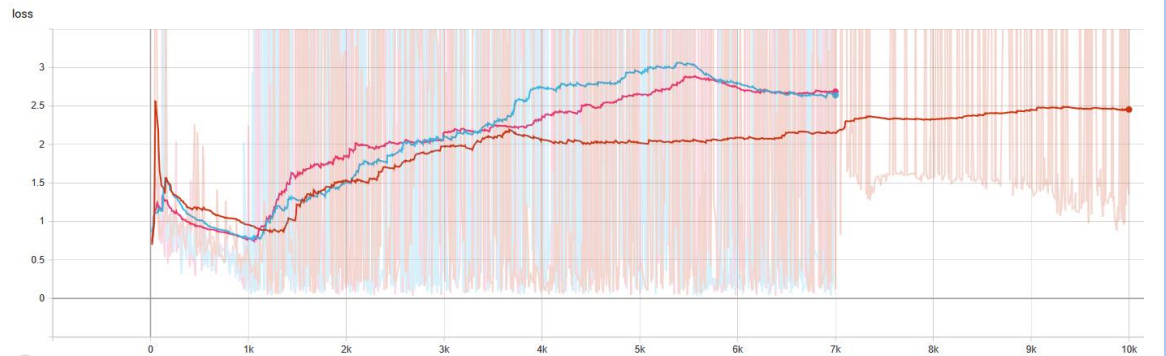


Figure 8.69. Loss traces on red (variation 1 – trained during 10k episodes), on blue (variation 2 – trained during 7k episodes) and on pink (variation 3 – trained during 7k episodes) for the model defined on TC.12.

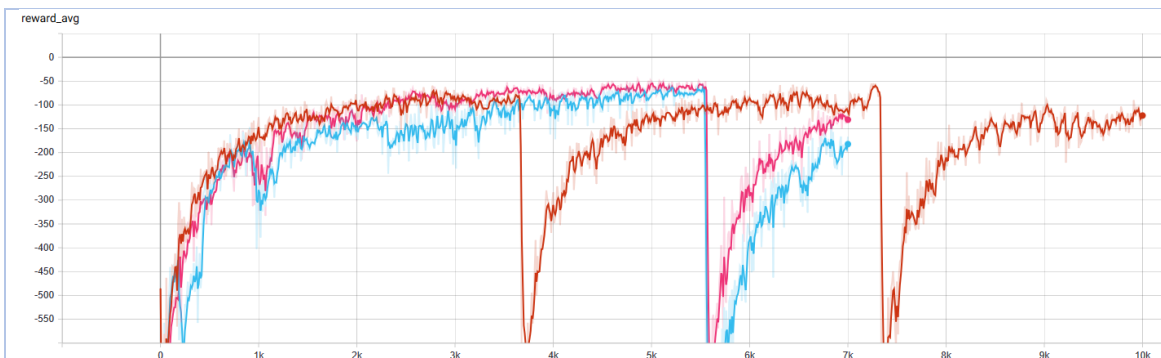


Figure 8.70. Average reward traces on red (variation 1 – trained during 10k episodes), on blue (variation 2 – trained during 7k episodes) and on pink (variation 3 – trained during 7k episodes) for the model defined on TC.12.

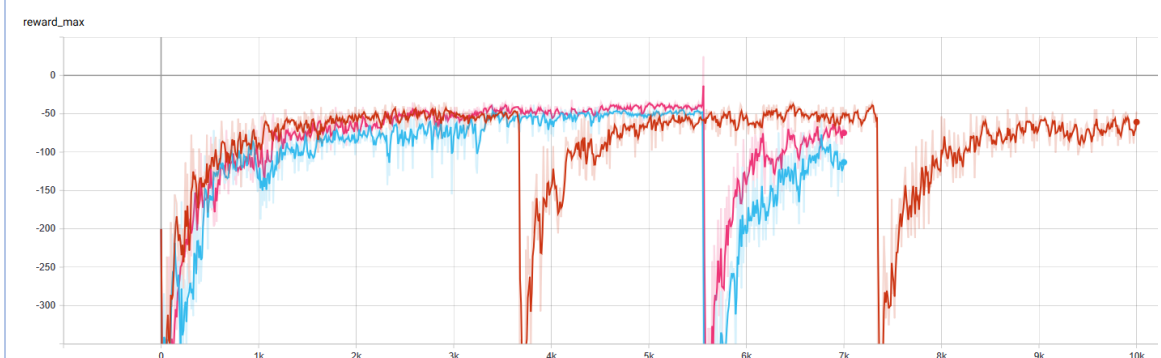


Figure 8.71. Maximum reward traces on red (variation 1 – trained during 10k episodes), on blue (variation 2 – trained during 7k episodes) and on pink (variation 3 – trained during 7k episodes) for the model defined on TC.12.

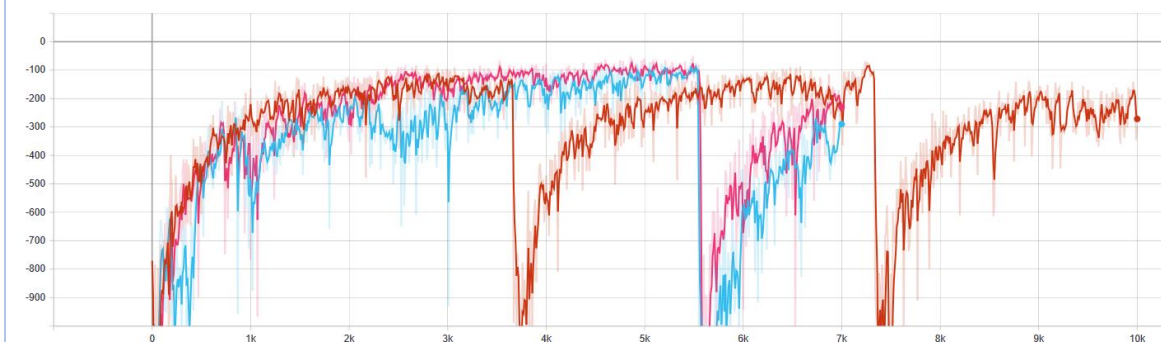


Figure 8.72. Minimum reward traces on red (variation 1 – trained during 10k episodes), on blue (variation 2 – trained during 7k episodes) and on pink (variation 3 – trained during 7k episodes) for the model defined on TC.12.

8.3.2.13. TC.13 – Map07

Training conditions

GPU fraction used	0,4	Total trained time	3d 4h 12m 29s
-------------------	-----	--------------------	---------------

Discount factor	0,95	Learning rate (α)	Special condition, see general comments of the model.
Spawn point	Fixed point (point A)		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 10 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (≤ 10 seconds): -10		
Sensors used	Semantic Segmentation camera Collision detection sensor		

General comments of the model

With the intention to improve the model defined on TC.12, the learning rate recovery phase has been modified, not allowing a total recovery. It will now behave as follows:

1. If it decreases, it will decrease at a rate $\alpha * 0,99975$ (no modification)
2. If it increases, it will increase at a rate $\alpha * 1,000025$
3. The agent will start with a fixed α (not 1) and will decrease until one (or both) of the following conditions are true:
 - i. $\alpha <$ (lower threshold, varies depending on the model) and minimum reward < -80
 - ii. minimum reward < -80 and current episode $>$ total episodes/2
4. The learning rate will only be allowed to vary between a minimum and maximum value, which will be adjusted on every model running under TC.13 conditions.
 - i. Variation 1 – starting at 0,3, from minimum 0,25 to maximum 0,35
 - ii. Variation 2 – starting at 0,4, from minimum 0,35 to maximum 0,45
 - iii. Variation 3 – starting at 0,45, from minimum 0,3 to maximum 0,65

The model defined in TC.13 with all of the variation has been trained from the previous model found in TC.12 – variation 3. For this reason, the starting learning rate is not 1, as the model already accumulates previous knowledge.

Statistics and performance on track

The first variation to this model is the most restrictive one, as the range is set to 0,2 - 0,35. This learning rate range assumes that the previous trained model is good enough to work with, with no more exploration and just driving by already acquired knowledge. By looking at the graphs it can be corroborated that, as expected, the model reaches soon a flat tendency and is not able to evolve as the agent continues to drive as previously learned. However, the sought correction (not truncating the model) is fulfilled in this range.

The second variation tries to raise the level of the learning rate fluctuation (not to enlarge the range), setting it to oscillate between 0,35 and 0,45. By incrementing the learning rate it is expected that the agent can perform a little exploration, not entirely relying on the previous knowledge. This could lead to a better performance on long term, but as observed on the graphs below, the model still gets stuck on the same tendency. Even if there is a higher probability it gets new paths to explore, it is still not relevant on the overall of the model.

From the first two variations the most relevant points are that the model doesn't get truncated (which is a good sign) and that the learning rate fluctuation can not be set to low levels if it is desired to continue training the model. For this reason, a third variation is introduced to the model.

In the third and last variation the range of the learning rate is enlarged, setting it from 0,3 to 0,65. Even if it can be observed on its statistics below that the model does not perform better as it is not evolving and it can also be seen that the model gets truncated again, this variation helps to point out the following:

1. If the recovery is done fast, the model gets abruptly truncated.
2. Higher the maximum level until fully recovery, higher the descent of the average reward level.
3. The decrease needs to be done more finely, delaying it over time. Once the recovery phase plays a role, the model is more likely to not be able to get better over time.
4. A low learning rate level and range can only be applied to testing purposes.

The overall conclusion of this Test Case is that the learning rate recovery phase has been isolated from the model performance, limiting and stabilising the limit in which the model gets or not truncated.

From now on, two paths are defined:

- Either new sensors are introduced to the model.
- Apply some corrections to the problems found in this test case and train a final model.

Graphical statistics

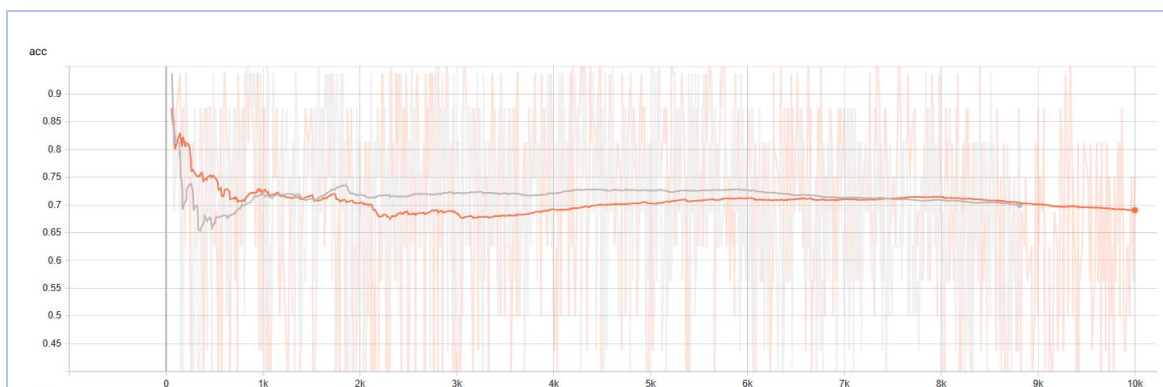


Figure 8.73. Accuracy traces on grey (variation 1 – trained during 10k episodes) and on orange (variation 2 – trained during 10k episodes) for the model defined on TC.13.

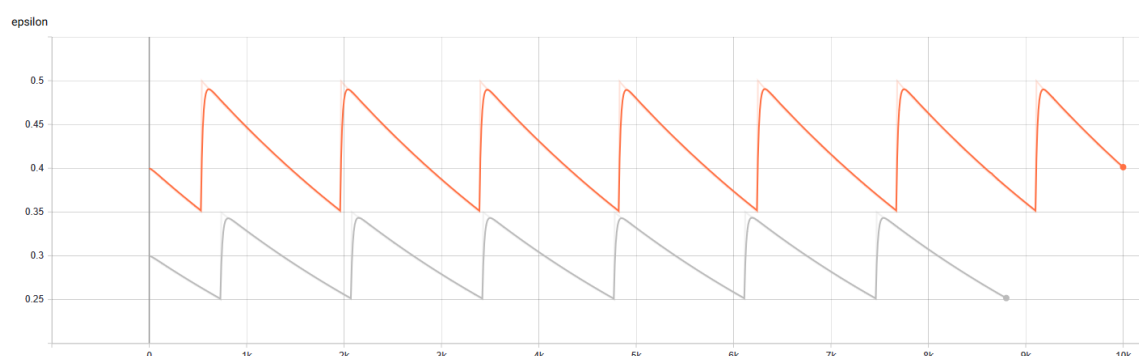


Figure 8.74. Learning rate traces on grey (variation 1 – trained during 9k episodes) and on orange (variation 2 – trained during 10k episodes) for the model defined on TC.13.

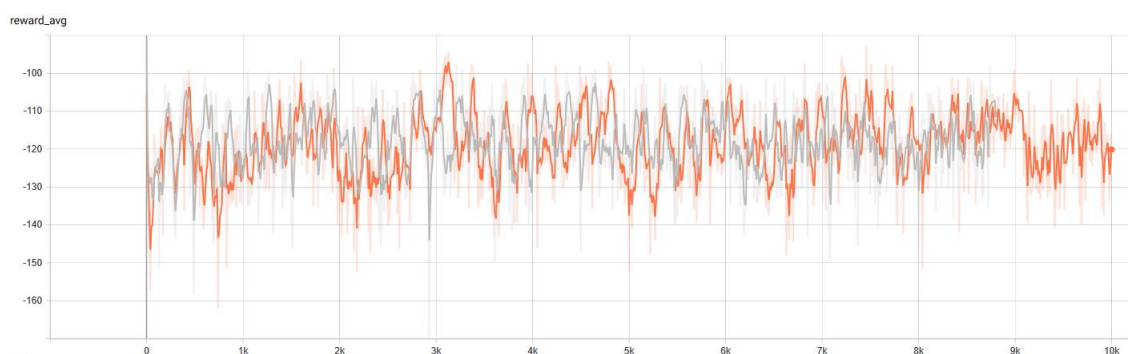


Figure 8.75. Accuracy traces on grey (variation 1 – trained during 10k episodes) and on orange (variation 2 – trained during 10k episodes) for the model defined on TC.13.

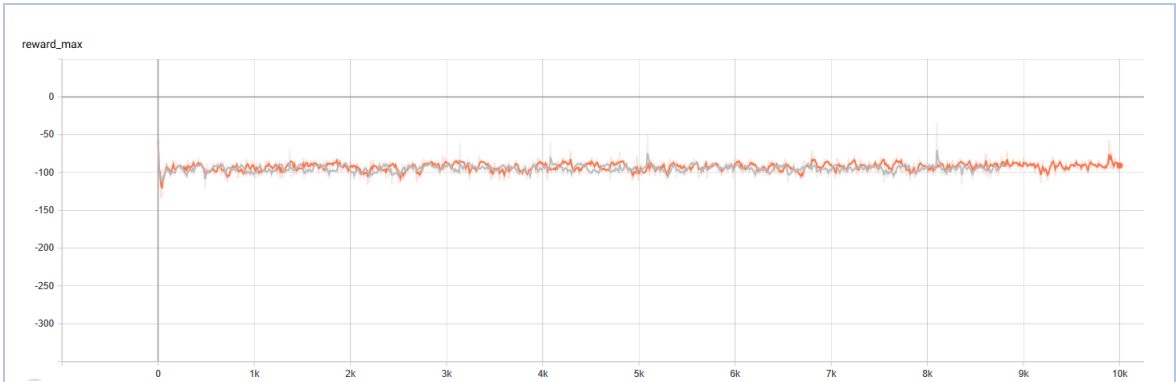


Figure 8.76. Maximum reward traces on grey (variation 1 – trained during 10k episodes) and on orange (variation 2 – trained during 10k episodes) for the model defined on TC.13.

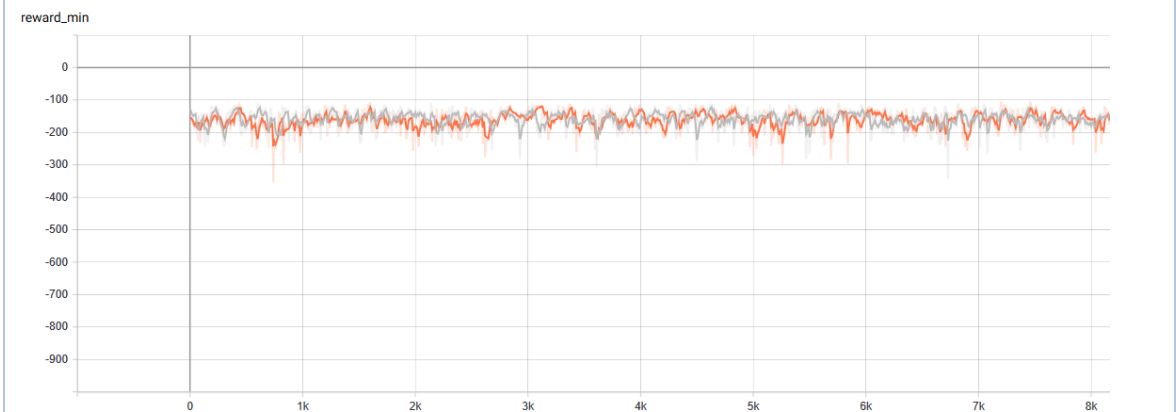


Figure 8.77. Minimum reward traces on grey (variation 1 – trained during 10k episodes) and on orange (variation 2 – trained during 10k episodes) for the model defined on TC.13.

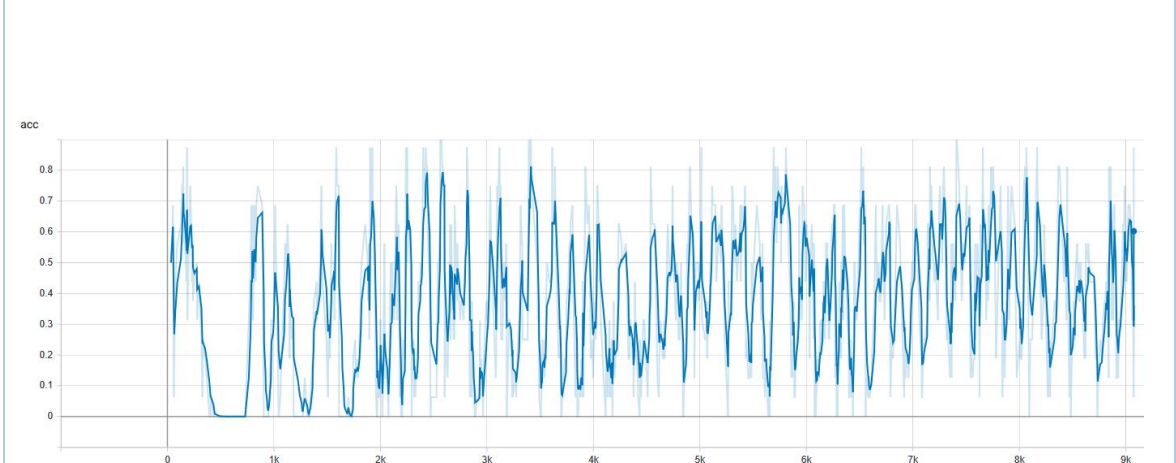


Figure 8.78. Accuracy trace (variation 3 – trained during 10k episodes) for the model defined on TC.13.

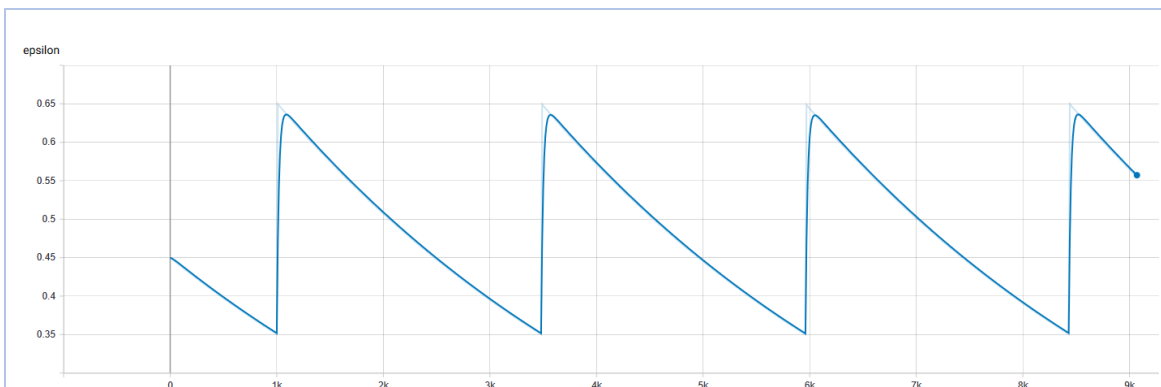


Figure 8.79. Learning rate trace (variation 3 – trained during 10k episodes) for the model defined on TC.13.

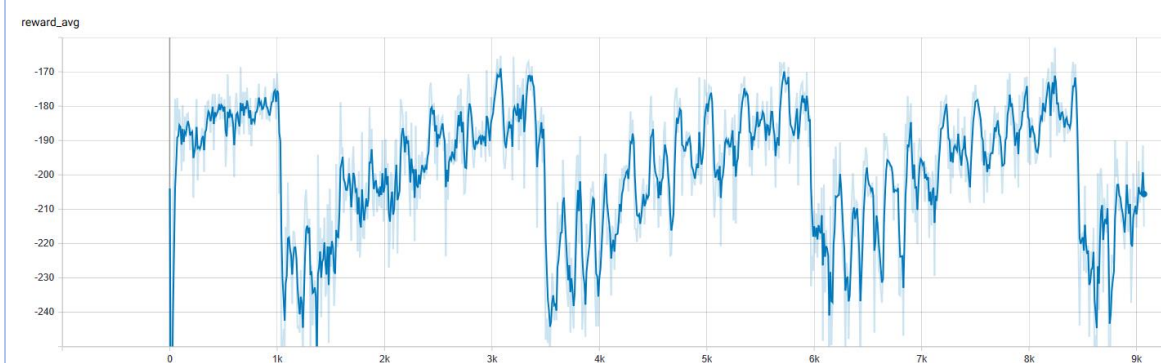


Figure 8.80. Average reward trace (variation 3 – trained during 10k episodes) for the model defined on TC.13.

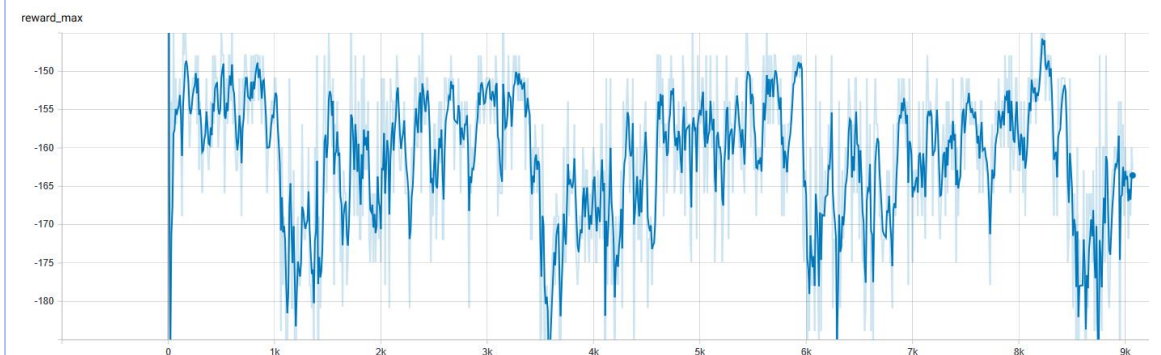
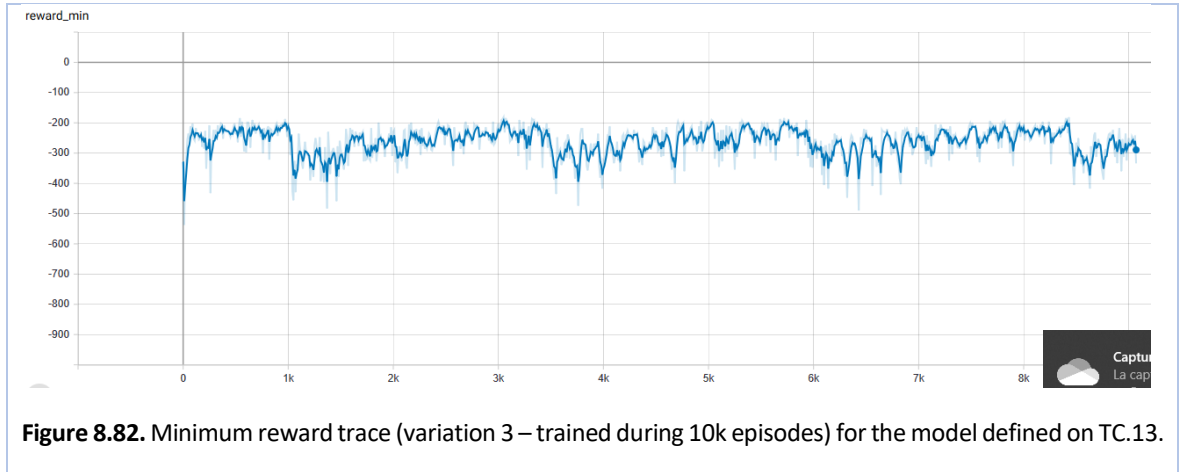


Figure 8.81. Maximum reward trace (variation 3 – trained during 10k episodes) for the model defined on TC.13.



8.3.2.14. Further test cases

Being CARLA such a powerful simulator that allows the usage of multiple complementary sensors that would eventually lead to a full and complete driverless experience, some further test cases can be performed in order to find the best model for its final implementation.

Two main reasons have been the cause for not moving forward looking for the best model.

The first one is due the tight thesis schedule, having in mind that up to this point the time consumed searching for a good model is insignificant compared to the time the final model will consume in order to be trained. A generous margin should be considered.

The second one is that, unfortunately, the capacity of the development computer is not enough to withstand the use of multiple parallel sensors. For this reason, it has not been possible to test more sensors than the used ones on all the previous test cases, leaving a large space for future improvement.

8.3.3. Final trained model

8.3.3.1. General considerations

After all the pretraining performed on section 8.3, it has been chosen a final, more or less accurate, model which is considered valid for training it longer than the others, as it will be considered the main trained model.

This model is the evolution of all previous thirteen test cases and has not evolved randomly but it carries all the problems and adopted solutions throughout the whole training.

The most critical part which is certainly the exploration strategy (the learning rate) that it has finally been chosen to do the following:

1. The agent will start at the beginning of the trained episodes at a learning rate value of 1, since it is useless to give this parameter a lower value if the agent still hasn't got any learned values for exploitation.
2. The learning rate will eventually be decreased at a constant rate as the episode cycles pass, in order to decrease and cease at a certain point the exploration part.
3. Fix a margin where the value of the learning rate can fluctuate in order to prevent and avoid overfitting on wrong decisions (the agent could not be able to explore once its learning rate is low), and let the possibility to relearn bad patterns. This margin is fixed at 0,35 – 0,55.

As a summary, the learning rate will decrease until its minimum value (0,35) where it will be kept constant unless some bad conditions are present on the agent, in which case the learning rate will increase until its maximum value (0,55). Eventually, the learning rate will be able to decrease again if the agent is performing good on track.

8.3.3.2. Training strategy and conditions

The model will be trained under 2 different starting conditions, in order to cover a wider range of possibilities but always on the same map (Map07).

1. The first starting point (point A) will be trained for longer as it is the most difficult starting point and the most desirable situation in which the car is able to drive correctly.
2. The second starting point (point C) will be trained half of the time just to give another approximation to the model of a desired track.

In total, the agent will be trained during 100k episodes. During 70k episodes on starting point A and during 30k episodes on starting point C.

For each starting point, the starting conditions for the training will be reset but not the model. This means that the knowledge of the agent will have continuity between the two parts, but the initial conditions (the learning rate and all the reward and accuracy statistics) will be set to initial values. This is made to give equal learning opportunities to all the environments.

Training conditions			
GPU fraction used	0,4	Total trained time	
Discount factor	0,93	Learning rate (α)	Special condition, see general comments of the model.

Spawn point	Fixed point (point A) during 70k episodes Fixed point (point C) during 30k episodes
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake
Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 10 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (≤ 10 seconds): -10
Sensors used	Semantic Segmentation camera Collision detection sensor

Table 8.6. Summary of the training conditions used.

General comments of the model

The learning rate behaves as follows:

1. If it decreases, it will decrease at a rate $\alpha * 0,999975$
2. If it increases, it will increase at a rate $\alpha * 1,000025$
3. The agent will start with $\alpha = 1$ and will decrease until one (or both) of the following conditions are true:
 - i. $\alpha < 0,4$ and average reward < -150
 - ii. average reward < -100 and current episode $> \text{total episodes}/1,5$
4. The learning rate will only be allowed to vary between a minimum and maximum value, adjusted at minimum 0,35 and maximum 0,55.

8.3.3.3. Performance on track and statistics

It is worth mentioning that although from starting point A the training has been longer, the whole training on that point has been split into two independent trainings. For this reason, three traces can be observed on the graphs below, as the training sequence has been the following:

- Trained during 40k episodes on starting point A (orange trace)

- Trained during 30k episodes on starting point C (blue trace)
- Trained during 30k episodes on starting point A (grey trace)

This training sequence has been done mainly for two reasons:

1. Not to overfit the model, giving space between each training and another on the same starting position.
2. Avoid extra-long training periods in a row in order to avoid computer problems related to fatigue that could lead to slower and less efficient responses.

All the graphs representing the performance of the model are shown down below.

Rewards traces are important to know how good the model is doing according to the prefixed conditions; however, the loss information is also crucial to determine if the training strategy turns out good or not. The graph below shows the loss traces for all the 3 final trainings of the model and even if the final decision on whether the model has been overfitted, underfitted or fitted good has to be done with both training and testing traces it can already be seen that training traces apparently do not show signs of a bad training strategy, since in all cases the trace keeps decreasing (even if the variation is very little). It can also be observed that during the trace on orange, the training has been stopped on the correct moment, as from 40k on the loss trace starts increasing again.

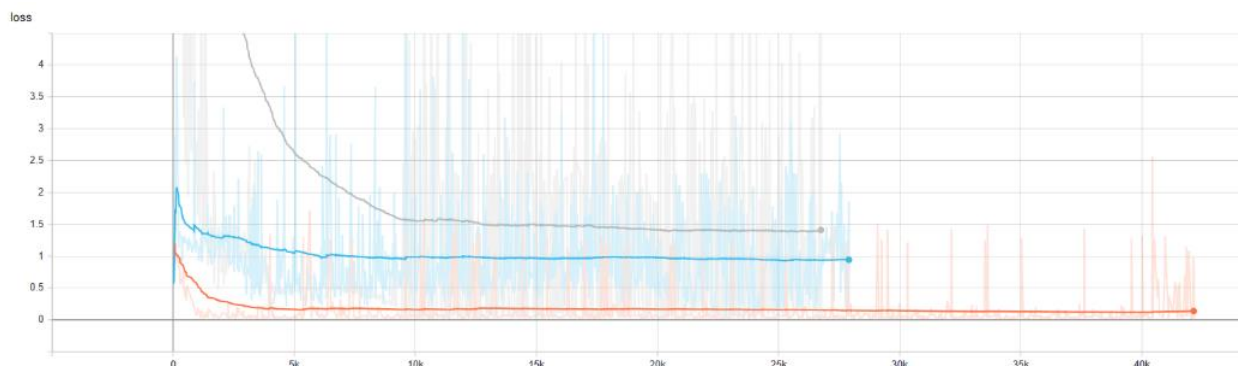


Figure 8.83. Loss traces for the final model, on starting position A during 40k episodes (trace on orange), on starting position C during 30k episodes (trace on blue) and on starting position A during 30k episodes (trace on grey).

The following reward graphs show how good has been the capacity of the agent to react to the inputs of its environment in order to fulfill all the expectations described in the model through actions and rewards. A common fact in all of them (average reward, maximum reward and minimum reward graphs) is that the first training (starting point A – trace on orange) performs way worse than the other two, having a stable tendency (thus not improving) and even undergoing a decreasing (worsening)

tendency. Being this the initial model and starting without previous knowledge, it is clear to expect from it the worse behavior.

Nonetheless, the initial training is able to set a good foundation as the other two trainings that follow are able to evolve and obtain a progressive positive trace. On all the reward traces, the grey trace (starting point A, last training) is the one that obtains the best statistics, being that an indicative that the agent achieves the capacity to improve its driving skills over time.

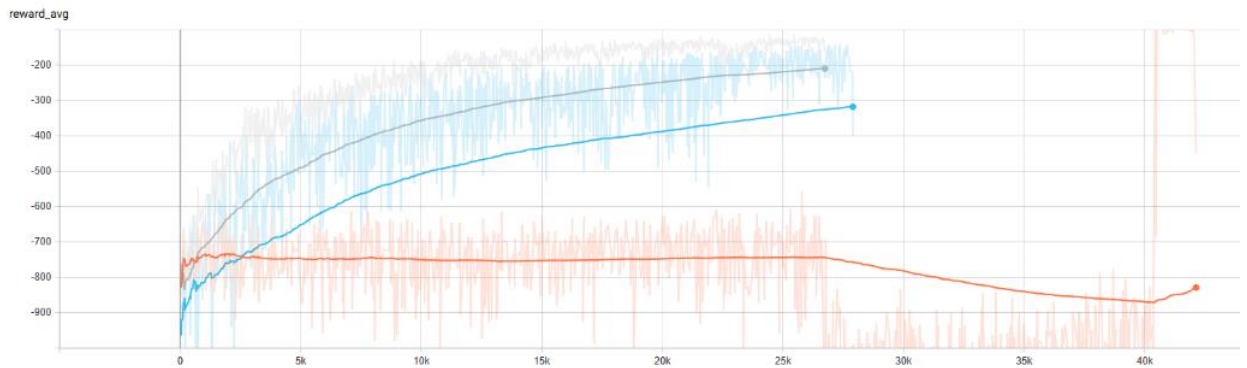


Figure 8.84. Average reward traces for the final model, on starting position A during 40k episodes (trace on orange), on starting position C during 30k episodes (trace on blue) and on starting position A during 30k episodes (trace on grey).

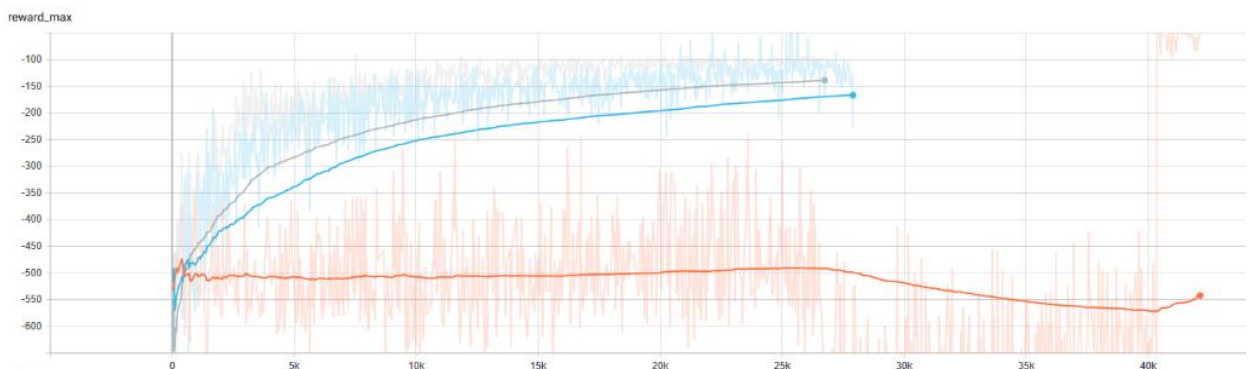


Figure 8.85. Maximum reward traces for the final model, on starting position A during 40k episodes (trace on orange), on starting position C during 30k episodes (trace on blue) and on starting position A during 30k episodes (trace on grey).

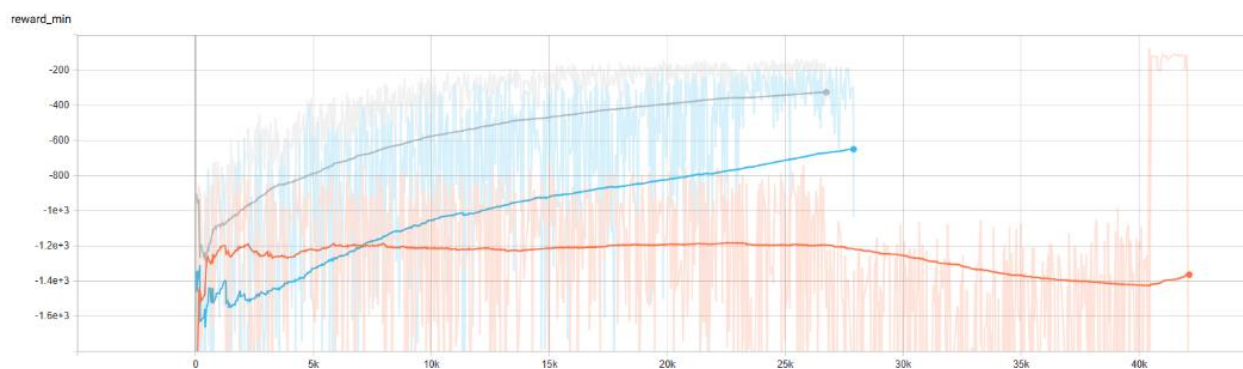


Figure 8.86. Minimum reward traces for the final model, on starting position A during 40k episodes (trace on orange), on starting position C during 30k episodes (trace on blue) and on starting position A during 30k episodes (trace on grey).

Last but not least, the accuracy shown throughout all the training journey seems to remain quite stable on the first two trainings (starting points A and C), while on the third one (starting point A again) seems to have accumulated enough experience to improve the agent's accuracy, meaning that the agent is driving better over time.

On the first training experience (starting point A – orange trace), the accuracy starts from a good point but keeps decreasing as the agent keeps training. This is due the fact that this is the starting model, with absolutely zero experience and needs to try harder than the other two models to learn from the environment. Later on, it can be observed that on the other two models the accuracy does not decrease but increase, especially on the last training (starting point A – grey trace).

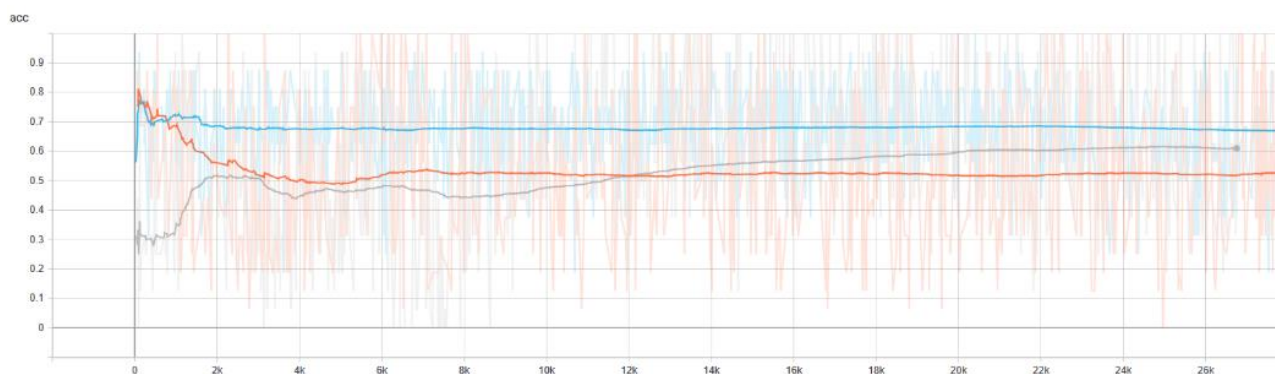


Figure 8.87. Accuracy traces for the final model, on starting position A during 40k episodes (trace on orange), on starting position C during 30k episodes (trace on blue) and on starting position A during 30k episodes (trace on grey).

Leaving aside the statistics and focusing on what the agent actually does on track and can be seen while playing the simulator, the car seems to react not so accurately so it could be applied immediately to the final application but quite good having in mind that the used sensometry is modest and the training time is tight.

On starting point A, which is by far the most critical part of the circuit as it has a narrow driving zone and a curvy track, is where most of the times the car is not able to surpass the first curve. However, that doesn't mean that it never reaches a further point, as sometimes it is able to reach the third curve. Observing that occasionally the agent is able to drive through the first zone of the track without problem, this case could be exploited until this casuistic is repeated most of the times. This is a good starting point as the has been seen that the agent has the capacity to learn that pattern.

On starting point C, the agent has way much more space (not precisely track), as it is an open field space. For this reason, this starting point is not as difficult as the previous one since it has more driving possibilities prior a collision yet being them an off-limits path. Even so, the agent learns eventually to drive on the track and repeats this casuistic sometimes. Again, as already said on starting point A, this could be exploited until the agent learned to drive more or less accurate under this situation.

8.4. Conclusions

After performing several different test cases in which major problems have been resolved, a stable and quite accurate and efficient model has been found according to the capacity of the development computer. Although adding some more sensors could have led to a better result (or not, maybe the model would have become very complex), the basic sensometry used during the development is neither disappointing.

During all the training the statistics and the performance on track observed on the simulator have been compared to see whether the values of the model were really representative of the knowledge learned by the agent, which in most cases didn't. On the final trained model, however, the statistics turned out to be quite representative, as the more the agent entered the training on different situations the more it incremented its rewards.

It has to be said that, although the model has been trained during several days and that could seem a lot, the reality is that with the volume of training done this model could not be applied to the final application, not even in a prototype. It is not trivial to decide when to stop the training process, but as long as the agent doesn't have a high accuracy ratio, it has to be kept on training. Unfortunately, due to the lack of time, the training process has been cut in a very premature state but with already showing a good behavior. With the observed results, the obtained model could eventually evolve to a more accurate one if the training is maintained, always under some different situations to avoid overfitting.

9. Model testing

9.1. Testing environment

9.1.1. Initial considerations

The model that will be loaded to the algorithm is considered to be a final prototype and thus, it is important that it recovers its learned knowledge where it was already learned to drive. This is why the learning rate will be now frozen to a value of 0,4, where it mostly regains its knowledge but also leaves an empty room for it to learn to adapt to the new situation. There will be no strategy applied to this parameter as in the training process, either upgrading or downgrading the value, as the most interesting part in the testing process is to check whether the training was done right or not.

Another important point to notice is that the discount factor, the parameter which sets the priorities of the model to obtain a higher reward either to short-term or long-term, has been decreased to 0,65 to permit the model focus on a more mid-term objective and not wait until nearly the end of the testing phase to show its abilities.

The algorithm used during training will be the same used here for the testing. This means all the actions, rewards and sensors used will be held constant. The summary of the testing conditions is shown below.

Testing conditions			
GPU fraction used	0,4	Total trained time	
Discount factor	0,65	Learning rate (α)	0,4
Spawn point	Fixed point (point A on Map07) during 7k episodes Fixed point (point B on Map07) during 7k episodes Fixed point (point C on Map07) during 7k episodes Fixed point (point A on Map02) during 7k episodes Fixed point (point A on Map04) during 7k episodes		
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90° Action 4: Brake		

Rewards	Reward 1: Collision: -20 points Reward 2: Velocity > 50 km/h: +3 point Reward 3: Velocity < 50 km/h: -3 point Reward 4: > 10 seconds without collision: +5 points Reward 6: Turn smoothness: $-1 * a_{yaw}$ Reward 8: No movement: -8 points Reward 9: Short training episode (≤ 10 seconds): -10
Sensors used	Semantic Segmentation camera Collision detection sensor

Table 9.1. Summary of the testing conditions used.

9.1.2. Racetracks

As the model that will be tested has been prior trained on Map07, the testing needs at least to be done on another different map. For this reason, two additional maps have been chosen to check the versatility of the model. Map02 and Map04 will be used, as they have already been used during the pre-training phase when deciding the most accurate neural network due to its features.

All the maps are shown down below, as well as the starting points where the model will be forced to start. On Map07, where the model has already been trained, all the three starting points have been maintained in order to test if the model has really learned to drive correctly under those situations. On the rest of the maps, just one point for each of them has been marked, one of them as a reference of straight road and the other one as a reference of a curvature.



Figure 9.1. Top view of Map02 offered by Carla Simulator. One point marked as A is used later on the testing phase. (Source: own)



Figure 9.2. Top view of Map04 offered by Carla Simulator. One point marked as A is used later on the testing phase. (Source: own)



Figure 9.3. Top view of Map07 offered by Carla Simulator. Three points marked as A, B and C that are used later on the testing phase. (Source: own)

9.2. Testing process

In order to test the pre-trained model in more than one situation to check whether it can apply the learned driving strategy in different scenarios, the following training process has been defined:

1. Map07 from starting point B
2. Map07 from starting point A
3. Map07 from starting point C
4. Map02 from starting point A
5. Map04 from starting point A

In every part, the model will be tested during 7k episodes. Every part will be tested with the resulting model of the previous test, concatenating the models. This means that in every new testing situation the model will have learned a little bit more than in the previous one. The final pre-tested model will just be used raw in the first situation, as in the second testing situation the model will also include the knowledge learned from the first one.

For this reason, the order of the testing is especially important, as the first testing situations are in the same map where the final model has been trained. This has been chosen to be done so to detect overfittings in the model, as if the first testing situations are in the same conditions than the testing situations the model wouldn't have time to reprogram its knowledge by seeing other maps and starting positions.

9.3. General performance of the model

9.3.1. Testing

All the graphs representing the five testing situations are shown below, overlaid on the same graph. Just to sum up, the previously defined testing situations correspond to the following colors:

1. Map07 from starting point B - **Orange**
2. Map07 from starting point A - **Strong blue**
3. Map07 from starting point C - **Red**
4. Map02 from starting point A - **Light blue**
5. Map04 from starting point A - **Pink**

First thing to point out is that every testing position is completely different and so the agent will have to adapt to each of them almost from zero. For this reason, in all the graphs of this section there will be better results than others, even if the base model is the same.

Starting the analysis with the accuracy, it seems that in three particular situations the model is quite accurate and in the other two not quite. However, at the end of the training the average accuracy is not that bad, with a value of 53,75%. This accuracy can not be considered to be acceptable for a final application but even with the limitations on the model training explained in section 8.3.2, this result can be considered more than admissible. If just with one sensor the agent is able to learn in such a way that its average accuracy is higher than 50%, adding more sensors could raise this value to a very good one. However, translating this value into the performance of the agent on track, it can be observed that in most situations the decisions chosen do not fit the necessities of the track as the agent prematurely crashes with something.

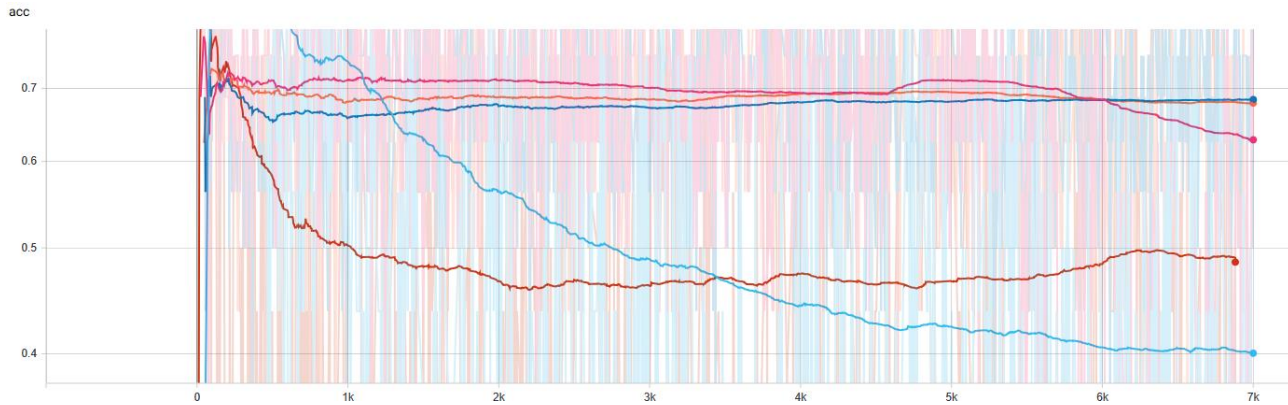


Figure 9.4. Accuracy traces for the testing cases. On map07 on starting position A (trace on strong blue), on starting position B (trace on orange), on starting position C (trace on red), on map04 on starting position A (trace on light blue) and on map02 on starting position A (trace on pink).

Referring to loss traces, in general, they do not show any particular behavior that could be considered as erroneous: just a particular case that will be discussed later on seems to be wrong. The most critical part here is that in the tested episodes the models don't seem to converge anywhere, thus there would have needed a little bit more cycles to be sure the model is not overfitted.

Nevertheless, the loss traces will be compared later on, on the two particular points where the training and the testing have coincided: point A and point C, both on Map07.

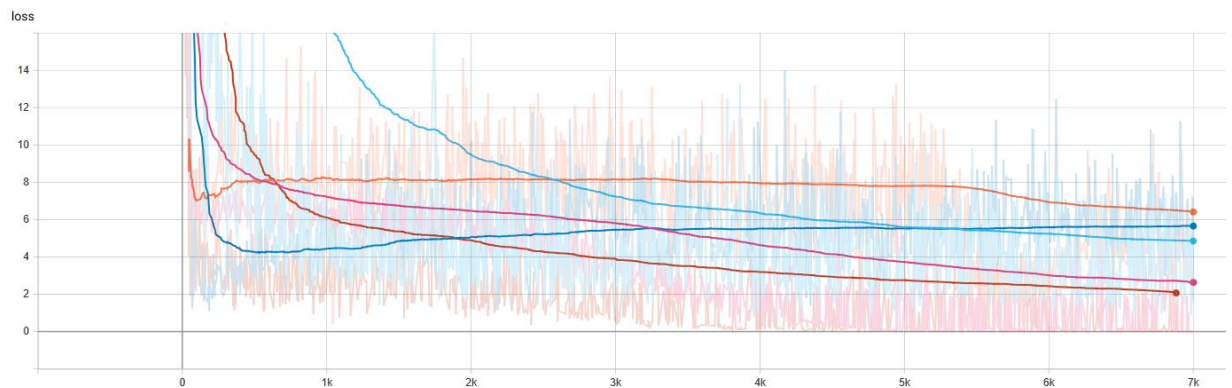


Figure 9.5. Loss traces for the testing cases. On map07 on starting position A (trace on strong blue), on starting position B (trace on orange), on starting position C (trace on red), on map04 on starting position A (trace on light blue) and on map02 on starting position A (trace on pink).

Last taking a look at the reward traces, indistinctly if they represent average, maximum or minimum, a clear tendency can be seen in all of them. The model adapts relatively fast to the new environment (which doesn't grant a good performance by the way), as it stabilizes its reward before 2k episodes in most of the situations. Just in starting position C on Map07, the model seems to struggle finding a good way to fit the expectations entrusted to it. This is probably caused by the own nature of this starting point, as the agent has multiple possible paths.

Going back to the concept of the fast adaptability of the model, on one hand the statistics show that the agent quickly finds a solution that it considers to be good according to the given rewards. A quick adaptation to the environment is important in all cases to reduce accidents, but specially in the final application it is extremely important as the race track will always be unknown and there will be limited laps to adapt to it. The model seems to be fast if compared with the global testing period, however it would not be enough for a real environment as it is a long adaptability period.

On the other hand, a quick adaptability doesn't have to go hand in hand with accuracy as can be seen on the performance on track. The agent finds a way to drive on a certain situation, which is always the same, but it eventually and prematurely crashes. Is it true that the reward traces stabilize, but on a quite bad level as the average reward is still far away from a positive value.

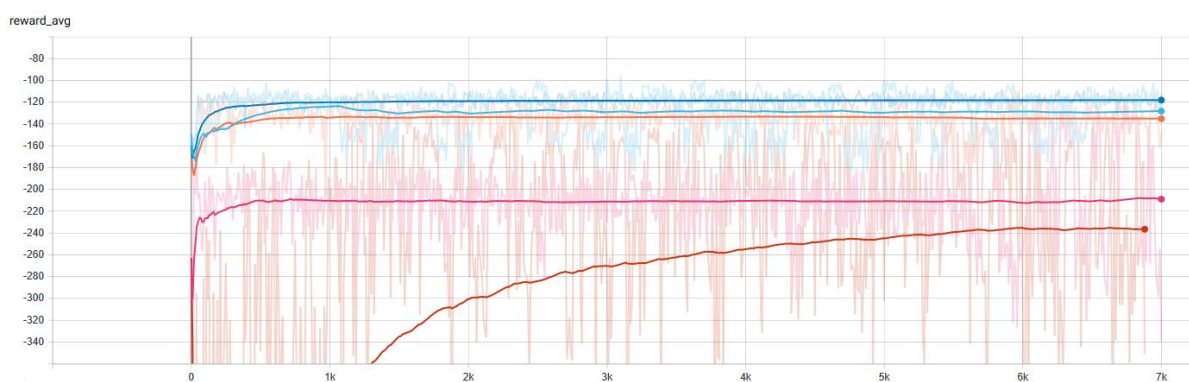


Figure 9.6. Average reward traces for the testing cases. On map07 on starting position A (trace on strong blue), on starting position B (trace on orange), on starting position C (trace on red), on map04 on starting position A (trace on light blue) and on map02 on starting position A (trace on pink).

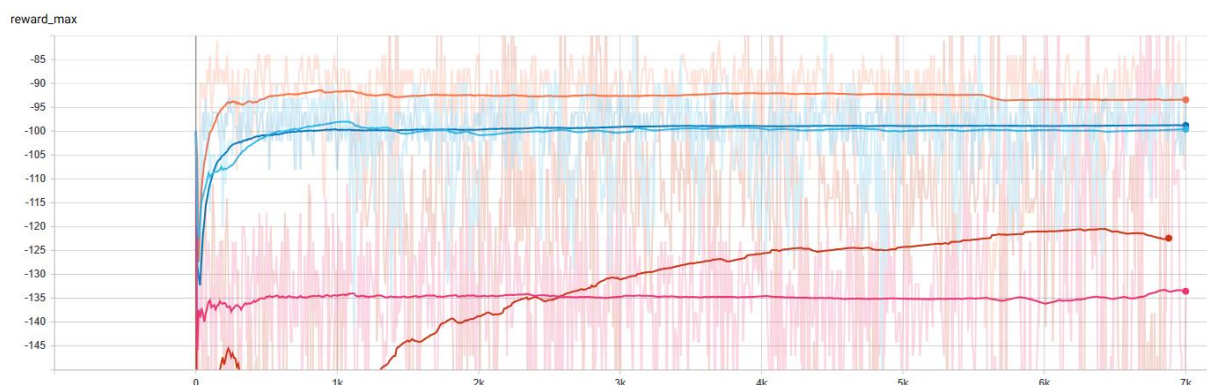


Figure 9.7. Maximum reward traces for the testing cases. On map07 on starting position A (trace on strong blue), on starting position B (trace on orange), on starting position C (trace on red), on map04 on starting position A (trace on light blue) and on map02 on starting position A (trace on pink).

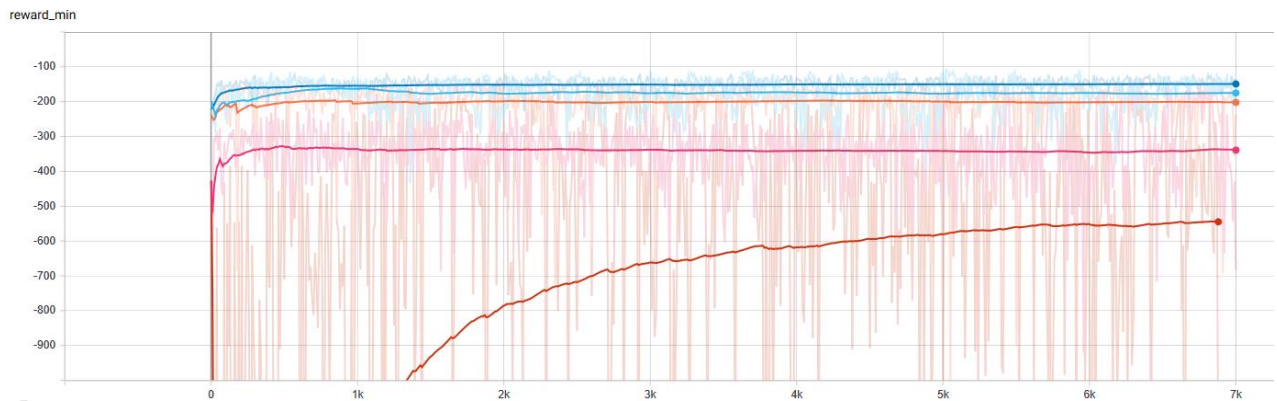


Figure 9.8. Minimum reward traces for the testing cases. On map07 on starting position A (trace on strong blue), on starting position B (trace on orange), on starting position C (trace on red), on map04 on starting position A (trace on light blue) and on map02 on starting position A (trace on pink).

9.3.2. Training vs Testing

Just to analyze the trained model in a more particular and controlled scenario, two points where the model has already been trained have been chosen also for testing: point A and point C on Map 07.

9.3.2.1. Map07 – Point C

Starting from the accuracy, the validation shows off that the model does not perform good on this situation, although in the training it seemed to fit quite good. The validation result was not the one expected for this starting point, although it is compressible since it is a point where more than one path can be chosen. If the model didn't chose the same path where the model was trained and granted its high accuracy, it is clear that the validation accuracy would be a lot lower.

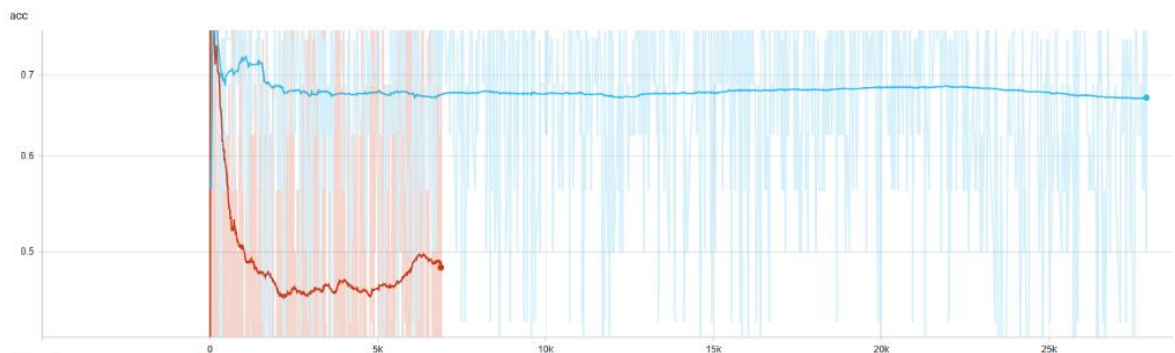


Figure 9.9. Comparison between training (trace on light blue) and testing (trace on red) accuracy traces on map07 – point C.

The loss trace comparison shows clearly that the chosen training strategy has been enough for this starting position. The validation loss keeps decreasing over episodes, although with the set episodes for every training position it can be observed that it is not enough for the model to reach its minimum, as it clearly is stopped before finding some stable point.

With the plotted scale the training trace appears to not have any improvement along episodes, which is completely wrong as in plot from Figure 8.83 the improvement can be seen as the scale is different.

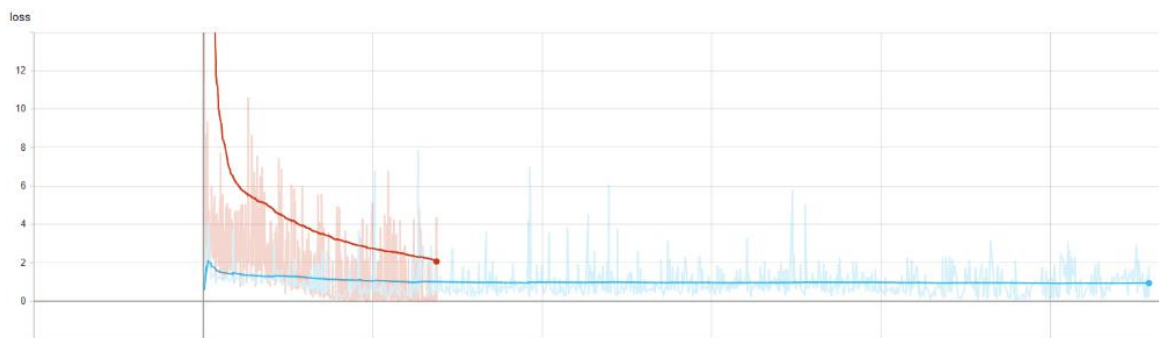


Figure 9.10. Comparison between training (trace on light blue) and testing (trace on red) loss traces on map07 – point C.

All the following graphs refer to reward (average, maximum and minimum) and as usual, the tendency in all of them is quite similar. It can be appreciated that in the validation the agent is more likely to get a higher reward and also a lot faster than in the training, although this may be caused by the difference in the discount factor in both situations.

These graphs come in contrast with the one showing accuracy (Figure 9.10), as the accuracy behaves in reverse as the reward values do. This can be caused by two main reasons: the priority set to finding a good value soon (that is, the discount factor reduced from 0,95 to 0,65) and the variation in the validation dataset (the agent finding different driving strategies in the validation than in the training).

However, the testing model seems to reflect better what happens in reality: with a negative average reward value the accuracy is certainly not above 70% as shown in the testing scenario.

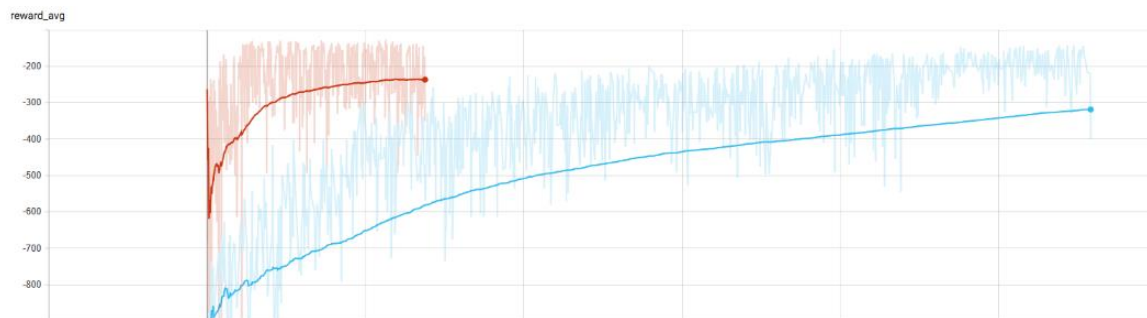


Figure 9.11. Comparison between training (trace on light blue) and testing (trace on red) average reward traces on map07 – point C.

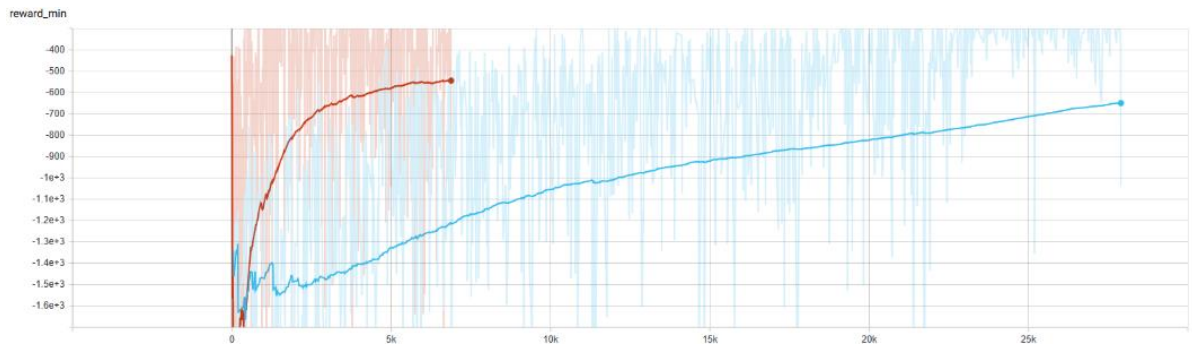


Figure 9.12. Comparison between training (trace on light blue) and testing (trace on red) minimum reward traces on map07 – point C.

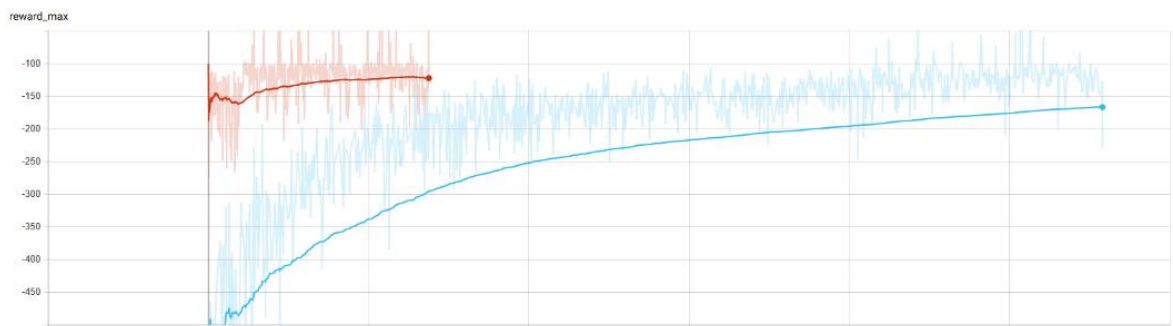


Figure 9.13. Comparison between training (trace on light blue) and testing (trace on red) maximum reward traces on map07 – point C.

9.3.2.2. Map07 – point A

Unlike in the previous case, on point A the validation model performs better than the testing one. This may be caused by the long training done on that point (half of the training) and the agent memorizing most of the path. This behavior may announce a possible case of overfitting on that special point.

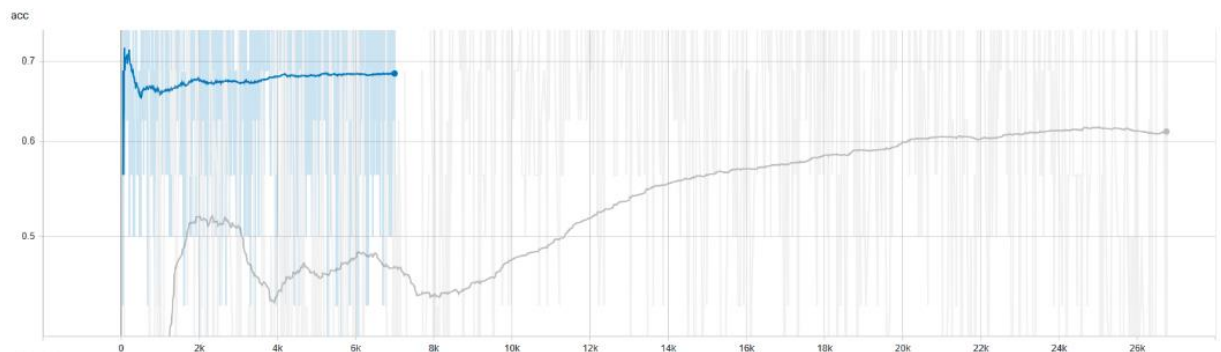


Figure 9.14. Comparison between training (trace on grey) and testing (trace on strong blue) accuracy traces on map07 – point A.

In order to confirm if there is overfitting on point A, the loss graph below needs to be analyzed. As can be seen, the validation trace decreases until its minimum and immediately starts rising again which means a clear overfitting of the model on point A, also the training and testing loss traces are far away, being the testing trace much higher than the other one.

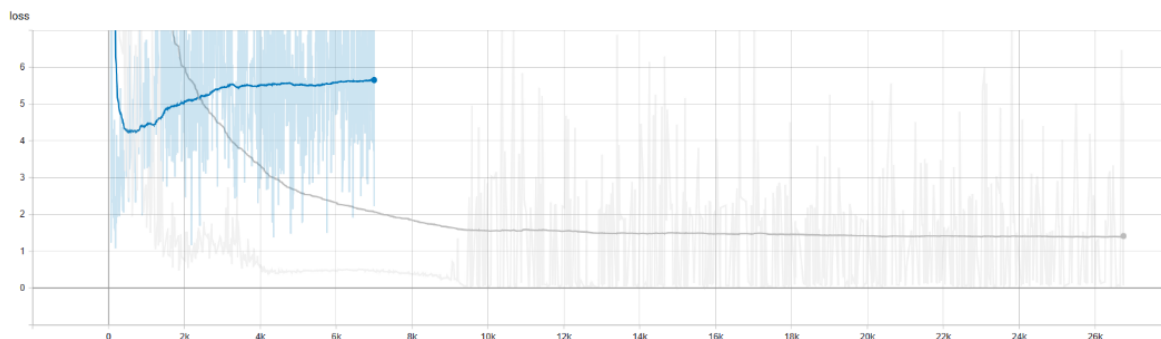


Figure 9.15. Comparison between training (trace on grey) and testing (trace on strong blue) loss traces on map07 – point A.

The reward traces, even if they continue showing the same tendency as on previous case on starting position C where the validation trace was much higher than the training one, they also show a faster progression until hitting its maximum value. This accentuates even more the fact that there is overfitting on this point, as the model knows suspiciously fast where it needs to go: in this case the adaptation period is less than 300 episodes with no further appreciable improvement.

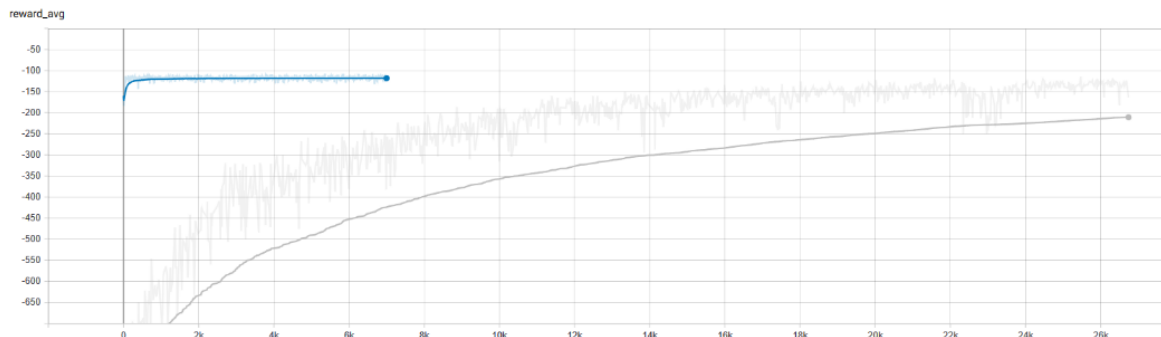


Figure 9.16. Comparison between training (trace on grey) and testing (trace on strong blue) average reward traces on map07 – point A.

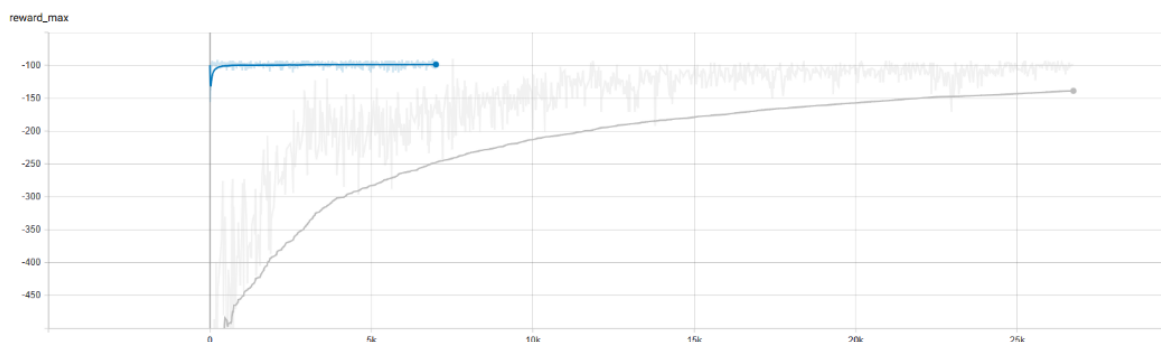


Figure 9.17. Comparison between training (trace on grey) and testing (trace on strong blue) maximum reward traces on map07 – point A.

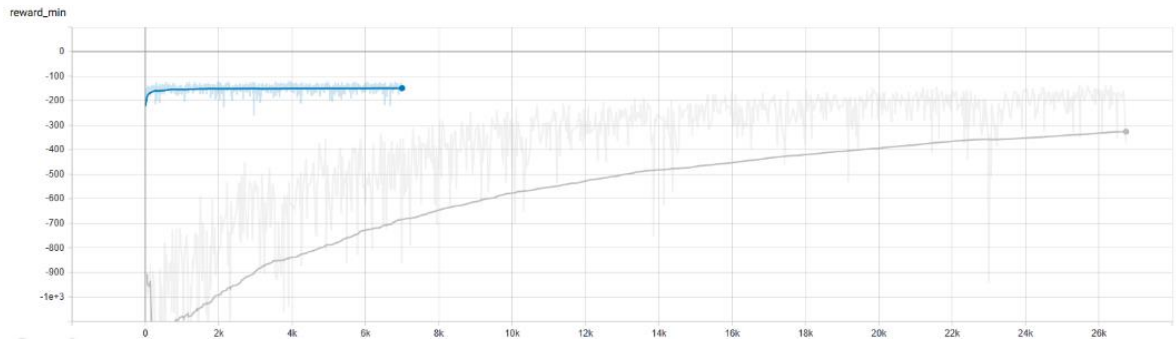


Figure 9.18. Comparison between training (trace on grey) and testing (trace on strong blue) minimum reward traces on map07 – point A.

Comparing point C, where there was no overfitting, and point A where there is a clear overfitting, the training difference has been a number of 40k episodes. These two results give a range of possible efficiency of training while maximizing performance. For the set and used training conditions, the range of episodes oscillate between 30k and 70k, maybe it would be better to cut it a little bit earlier that last value just to prevent possible overfittings.

If this experiment would have to be reproduced exactly the same (identical training conditions), for every desired starting position the favorable training episodes would be chosen to 50k.

9.4. Conclusions

Through testing and testing it has been demonstrated that the used training strategy is correct for most situations, even if there is overfitting on the most trained point. However, encountering this error is not a negative result as it can lead to further improvement. By the validation made a range of training episodes can be set with a clear maximum value that cannot be surpassed with the same training conditions.

It is really important to emphasize that the range of training episodes found is only applicable to the same training conditions (actions, rewards, sensors, reinforcement learning parameters, ...) as if any of the parameters that play a role on the agent learning procedure change, all the other results vary too. Artificial Intelligence and the way the agent learn is random and can not be predicted, even the results vary a little with the same parameters by playing the model learning on different times.

Analyzing the perform on track, which is really the one that matters for a final application, the final conclusion is that it is clearly much more improvable. On average on all the testing situations, the agent is only capable of driving during 10 seconds before colliding with the limits of the track or any object present on the surroundings making it a not so accurate model. Furthermore, most of the decisions

made are not clever, as in most situations it chooses to turn (either left or right) excessively. It has also to be said that the training strategy consisted on restarting the agent on the same spawn point every time it crashed, facilitating the ability to learn always on the same starting position. On further analysis, it would be recommended to not reset the car on the same starting point on each collision, but letting the agent continue from the crashing point. By doing so, the agent would also learn to readapt to a bad situation and find a drivable path on a closed space, learn to not get stuck.

10. Environmental impact

As this thesis is a study with no associated physical implementation, its associated environmental impact is practically null. The full development has been done with a unique computer, so the only impact of performing this theoretical study is the one caused by the production of the electrical energy used to power on the computer.

However, this study has been designed to end up on a final physical implementation, which is an electrical race car for the formula student competition. All the impact generated by this final application is not considered in this thesis but it will eventually need to be taken into account as this study is useless if not implemented.

Considering the car will be an electrical one, the major environmental impact caused by it is associated to the whole pack of energy production. This car has a battery composed by several individual electrical cells with a limited lifetime. This is the major source of carbon emissions, as its production generates a significant carbon footprint as well as during its recycling or destruction. Apart from that, the only associated carbon emission during the usage of the car is the electrical energy consumed by the batteries to power on the car.

It is also important considering that this car will not be driven every day, just occasionally, and in a very short time range. In this way, the carbon emission associated to the final prototype is not as significant as in other applications would be.

Conclusions

At the very beginning of this thesis it was hard to think that getting such deep on the field would be possible, specially having a limited time, budget and access to specific material. Along the development, however, it has been proved that even without the best hardware it is possible to develop something quite rigorous. Even if for a final application the obtained results and accuracy would not be sufficient, as putting something into movement into the real life without a proper control of it is quite a lot dangerous, the obtained results are actually optimistic.

As commented on the goals of this thesis, the main purpose of this report is to establish a base on how to do a prototype for an autonomous driving system. Furthermore, it was also essential to do it as a guided tutorial so anyone could then use it for further development on the topic. For this reason, the accuracy of the final model developed is not as important as having showed how to train an artificial intelligence that can learn to drive by itself without the necessity of using predefined databases. Showing a different and not so common approach on the learning methodology that already fulfills the necessities of the agent has also been an important part of the study.

Although the specific objectives set for this thesis have been met and also a functional model capable of learning some patterns has been trained, there is still continuity and a wide range of improvement. From this study, different paths can be chosen in order to improve a model that is able to achieve a higher accuracy with the given program, even exploring different programming languages with the same idea. The trained model can be exported to be implemented into other languages, such as Matlab, to be applied to a final application.

All in all, artificial intelligence and autonomous driving cars are topics very present in the current technological world. They have still their golden years ahead and certainly much more research on the field can be made.

Economic analysis and material budget

As this is a mere study of the viability to implement and autonomous driving system, all the associated costs refer to the engineering part (design of the model tested and training the model) and the material needed for the training itself.

Item	Quantity	Unitary cost	Percentage	Subtotal
Engineering	[Hours]	[€/Hour]	[%]	[€]
Brainstorming	20	20	2,78	400
Theoretical analysis	90	20	12,5	1800
Practical implementation	70	20	9,72	1400
Model training	450	20	62,5	9000
Documentation	90	20	12,5	1800
TOTAL engineering (no VAT)	720	-	100	14400
VAT	-	-	21%	3024
TOTAL engineering (VAT)				17424 €
Material	[units]	[€/unit]	[%]	[€]
ASUS R510V	1	1047	100	1047
CARLA simulator	0	0	0	0
Software	0	0	0	0
TOTAL material (VAT)	-	-	100	1047

TOTAL (no VAT)	15447 €
TOTAL (VAT)	18471 €

An important point from this budget can be observed in the model training, which is the most consuming part in terms of hours and of course, in terms of costs. This point is significantly different from the others since although is the longest part, it doesn't require a full-time attention as would be with the other points. This means that, although this time is necessary, the engineer can still be performing other tasks in parallel.

If this study were taken to a physical level for its implementation, the associated costs for all the sensors and actuators that need to be used for the agent will need to be added separately which would certainly trigger the final cost of the prototype.

Bibliography

1. *What is AI-Artificial-Intelligence?* [en línia]. Disponible a: <https://searchenterpriseai.techtarget.com/definition/AI-Artificial-Intelligence>.
2. *4 Main Types of Artificial Intelligence* [en línia]. Disponible a: <https://learn.g2.com/types-of-artificial-intelligence>.
3. *AI: Neural Network for beginners* [en línia]. Disponible a: <https://www.codeproject.com/Articles/16508/AI-Neural-Network-for-beginners-Part-of>.
4. *Supervised vs Unsupervised machine learning* [en línia]. Disponible a: <https://medium.com/@chisoftware/supervised-vs-unsupervised-machine-learning-7f26118d5ee6>.
5. *For Dummies — The Introduction to Neural Networks we all need ! (Part 2)* [en línia]. Disponible a: <https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-part-2-1218d5dc043>.
6. *CS231n Convolutional Neural Networks for Visual Recognition* [en línia]. Disponible a: <http://cs231n.github.io/neural-networks-1/>.
7. Philosophy, D.O.F. et al. EFFECTS OF LONG - TERM FOREST FIRE RETARDANTS ON FIRE INTENSITY , HEAT OF COMBUSTION OF THE FUEL AND FLAME EMISSIVITY. A: . núm. September 2009.
8. *Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning* [en línia]. Disponible a: <https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90>.
9. *A Quick Introduction to Neural Networks* [en línia]. Disponible a: <https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/>.
10. *How Does Back-Propagation in Artificial Neural Networks Work?* [en línia]. Disponible a: <https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7>.
11. *SAE international releases updated visual chart for its «levels of driving automation»* [en línia]. Disponible a: <https://www.sae.org/news/press-room/2018/12/sae-international-releases-updated-visual-chart-for-its-“levels-of-driving-automation”-standard-for-self-driving-vehicles>.
12. *The best explanation of Convolutional Neural Networks on the Internet!* [en línia]. Disponible a: <https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8>.
13. *Convolutional Neural Network – In a Nut Shell* [en línia]. Disponible a: <https://engmrk.com/convolutional-neural-network-3/>.
14. *4764cc8c207e9cff6519fc8ad5ba711c214b8e34 @ ujjwalkarn.me* [en línia]. Disponible a: <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>.

15. *a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53* @ *towardsdatascience.com* [en línia]. Disponible a: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
16. *Kernel (image_processing)* [en línia]. Disponible a: [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing)).
17. *cdn images* [en línia]. Disponible a: https://cdn-images-1.medium.com/max/1600/1*I75ghqLA1CbKeyqz3j-yGA.jpeg.
18. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way* [en línia]. Disponible a: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
19. *Tensorflow for deeplearning* [en línia]. Disponible a: <https://www.oreilly.com/library/view/high-performance-spark/9781491943199/ch04.html>.
20. *Installing keras tensorflow using anaconda for machine learning* [en línia]. Disponible a: <https://towardsdatascience.com/installing-keras-tensorflow-using-anaconda-for-machine-learning-44ab28ff39cb>.
21. *Index @ Docs.Nvidia.Com* [en línia]. Disponible a: <http://docs.nvidia.com/gameworks/index.html#developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm?Highlight=occupancy>.
22. *Docs.nvidia.com* [en línia]. Disponible a: <https://docs.nvidia.com/deeplearning/sdk/cudnn-install/>.
23. *Installing Tensorflow with CUDA, cuDNN and GPU support on Windows 10* [en línia]. Disponible a: <https://towardsdatascience.com/installing-tensorflow-with-cuda-cudnn-and-gpu-support-on-windows-10-60693e46e781>.
24. *OpenCV pypi.org* [en línia]. Disponible a: <https://pypi.org/project/opencv-python/>.
25. *CARLA Simulator. A:* [en línia]. [Consulta: 14 novembre 2019]. Disponible a: <http://carla.org/>.
26. *Coursera CARLA Setup for Ubuntu. A:* [en línia]. Disponible a: <https://usermanual.wiki/Document/CARLASETUPGUIDEWINDOWSX64.2140065550.pdf>.
27. *Carla Simulator Documentation (readthedocs)* [en línia]. Disponible a: https://carla.readthedocs.io/en/latest/how_to_build_on_windows/.
28. *How to enable a 64 bit visual cpp toolset on the command line* [en línia]. Disponible a: <https://docs.microsoft.com/en-us/cpp/build/how-to-enable-a-64-bit-visual-cpp-toolset-on-the-command-line?redirectedfrom=MSDN&view=vs-2019>.
29. *Understanding RL: The Bellman Equations* [en línia]. Disponible a: <https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/>.
30. *Q-Learning introduction and Q Table - Reinforcement Learning w/ Python* [en línia]. Disponible a: <https://pythonprogramming.net/q-learning-reinforcement-learning-python-tutorial/>.

31. *Longer-term model results - Self-driving cars with Carla and Python* [en línia]. Disponible a: <https://pythonprogramming.net/trained-model-self-driving-autonomous-cars-carla-python/>.

32. Learning, D. in (Deep) M. *Dropout in (Deep) Machine learning* [en línia]. Disponible a: <https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5>.

Appendice A

A1. Tested models

Three different maps, all of them with the same parameters for the convolutional neural networks, have been tested. Below is explained the comparative and the performance of each model in Town02, Town04 and Town07.

The common parameters for them all are listed in the table below.

Training conditions			
Episodes	300	Batch size	4
Time per episode	10 s	GPU fraction used	0,4
Learning rate (α)	Downgrading at a rate: $\alpha * 0,95$	Discount factor	0,99
Actions	Action 1: Go straight Action 2: Turn left at 90° Action 3: Turn right at 90°		
Rewards	Reward 1: collision: -200 points Reward 2: velocity > 50 km/h: +1 point Reward 3: velocity < 50 km/h: -1 point		
Sensors used	RGB Camera Collision detection sensor		

Table 0.1. Set parameters for the first iteration in order to select the most favorable Convolutional Network to use in the main training.

The Neural Network use from one model to another follows a similar structure, just varying the number of layers and the neurons used in each of them. The main structure of the Neural Network is shown in figure below.

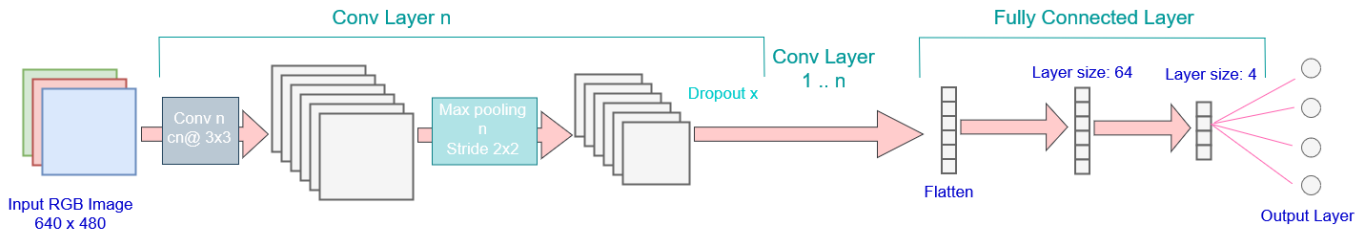


Figure 0.1. Structure of the Neural Network used for the previous selection of the final Neural Network used in the model trained. (Source: own)

The basic unit of each Convolutional Layer consists of:

6. A convolutional layer with cn neurons and a 3×3 kernel. cn corresponds to the number of neurons given to each layer
7. A Max Pooling layer with a 2×2 stride
8. A final dropout x , which is always 0,1 for $n=1$ and 0,2 for $n > 1$.

This unit is repeated in a sequential mode according to the number of layers applied to each model prior entering the Fully Connected Layer which has:

9. A flatten layer
10. A fully connected layer of size 64
11. A fully connected layer of size 4 (the same size as the actions introduced to the model)

A1.1 Comparative between models

A first comparative is done between different models on the same map in order to compare its performance on the same environment.

A1.1.1 Map02

This map is expected to be the easiest one and therefore the data from the models tested in this map won't have so much weight in the final decision.



Figure 0.2. Map02 top view in CARLA simulator.

The number of convolutional layers used and the neurons present on each of them are described in the following table.

Map	Map02		
ConvNet used	Number and distribution of neurons		
2 Conv Layers (equals)	16 x2	32 x2	64 x2
3 Conv Layers (equals)	16 x3	32 x3	64 x3
	128 x3		
3 Conv Layers (diff.)	16 + 32 + 64	16 + 32 x2 + 64	32 x2 + 64 x2

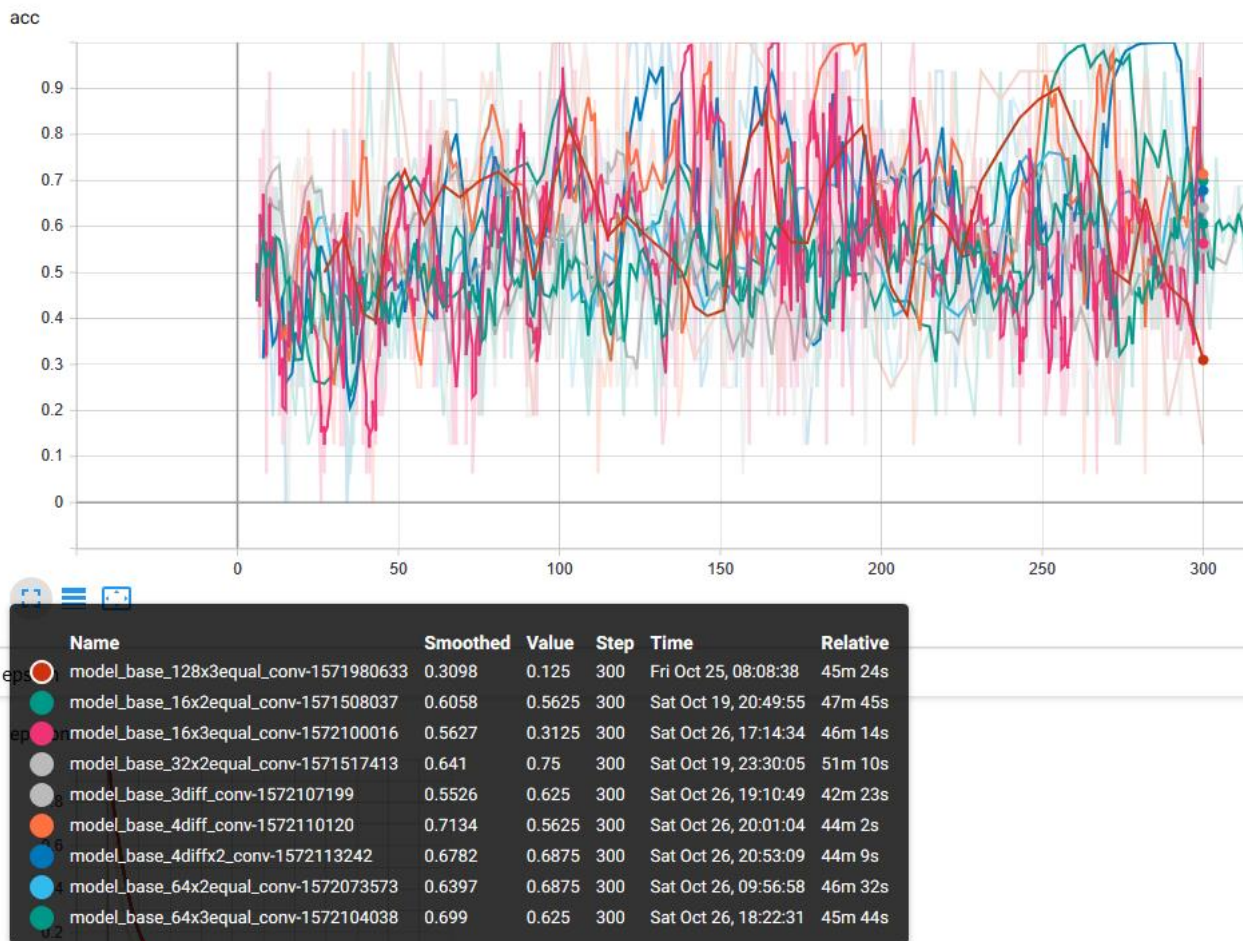
Table 0.2. Distribution of the convolutional layers used to train the model on Map02.

Figure 0.3. Accuracy traces for all the tested models according to Table 0.2. on Map02. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey - 32x2equal_conv), with 64 neurons (on light blue - 64x2equal_conv), the models set with three identical convolutional layers with 16 neurons (pink - 16x3equal_conv), 64 neurons (green – 64x3equal_conv), 128 neurons (red – 128x3equal_conv), the model set with three convolutional layers each one with 16, 32 and 64 neurons respectively (on grey – 3diff_conv), the model set with four convolutional layers each one with 16, 32, 32 and 64 neurons respectively (on orange – 4diff_conv) and the model set with four convolutional layers each one with 32, 32, 64 and 64 neurons respectively (on dark blue – 4diff2_conv).



Figure 0.4. Average reward traces for all the tested models according to Table 0.2. on Map02. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey - 32x2equal_conv), with 64 neurons (on light blue - 64x2equal_conv), the models set with three identical convolutional layers with 16 neurons (pink - 16x3equal_conv), 32 neurons (on orange – 32x3equal_conv), 64 neurons (green – 64x3equal_conv), 128 neurons (red – 128x3equal_conv), the model set with three convolutional layers each one with 16, 32 and 64 neurons respectively (on grey – 3diff_conv), the model set with four convolutional layers each one with 16, 32, 32 and 64 neurons respectively (on orange – 4diff_conv) and the model set with four convolutional layers each one with 32, 32, 64 and 64 neurons respectively (on dark blue – 4diff2_conv).

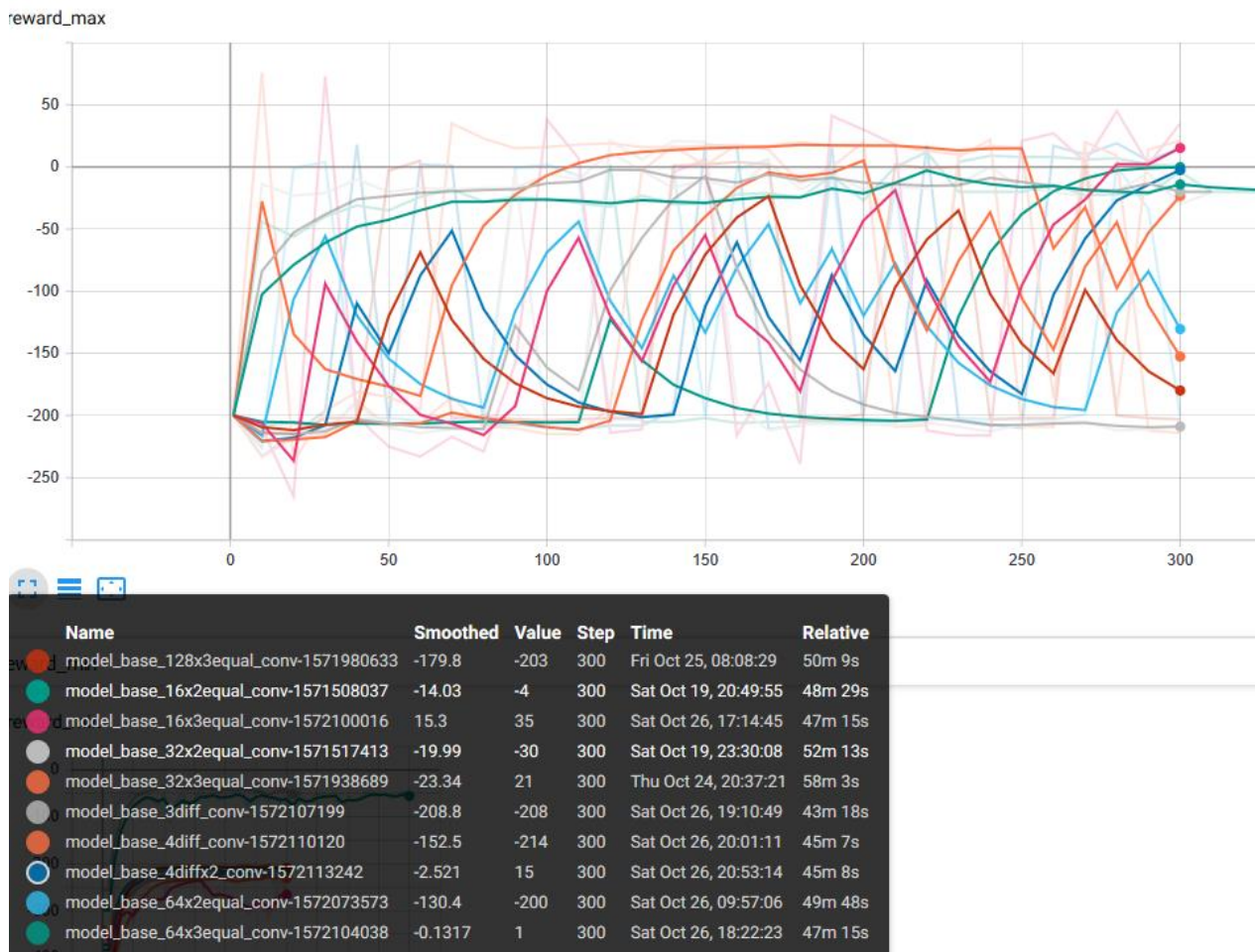


Figure 0.5. Maximum reward traces for all the tested models according to Table 0.2. on Map02. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey - 32x2equal_conv), with 64 neurons (on light blue - 64x2equal_conv), the models set with three identical convolutional layers with 16 neurons (pink - 16x3equal_conv), 32 neurons (on orange – 32x3equal_conv), 64 neurons (green – 64x3equal_conv), 128 neurons (red – 128x3equal_conv), the model set with three convolutional layers each one with 16, 32 and 64 neurons respectively (on grey – 3diff_conv), the model set with four convolutional layers each one with 16, 32, 32 and 64 neurons respectively (on orange – 4diff_conv) and the model set with four convolutional layers each one with 32, 32, 64 and 64 neurons respectively (on dark blue – 4diff2_conv).

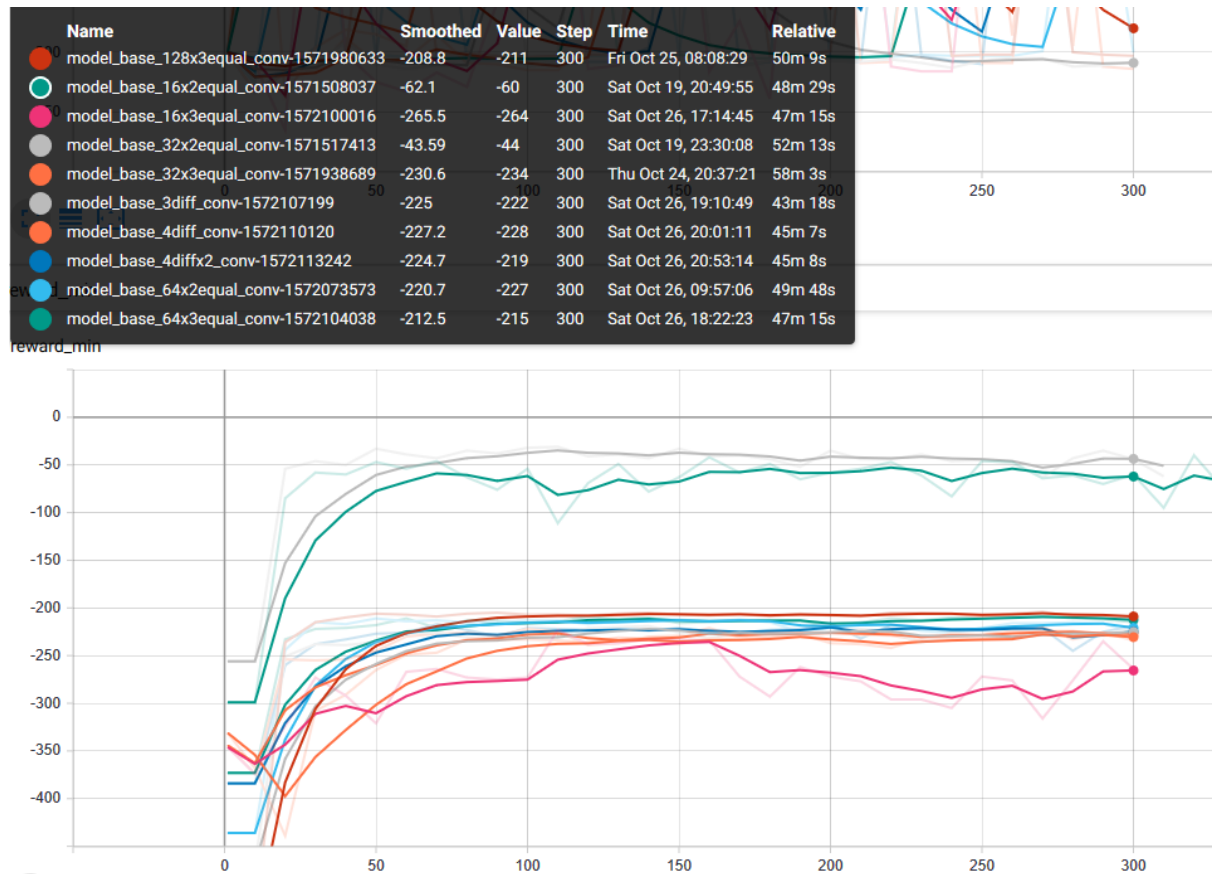


Figure 0.6. Minimum reward traces for all the tested models according to Table 0.2. on Map02. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey - 32x2equal_conv), with 64 neurons (on light blue - 64x2equal_conv), the models set with three identical convolutional layers with 16 neurons (pink - 16x3equal_conv), 32 neurons (on orange – 32x3equal_conv), 64 neurons (green – 64x3equal_conv), 128 neurons (red – 128x3equal_conv), the model set with three convolutional layers each one with 16, 32 and 64 neurons respectively (on grey – 3diff_conv), the model set with four convolutional layers each one with 16, 32, 32 and 64 neurons respectively (on orange – 4diff_conv) and the model set with four convolutional layers each one with 32, 32, 64 and 64 neurons respectively (on dark blue – 4diff2_conv).

A1.1.2 Map04

This map has been chosen due to its large dimensions and the possibility to let the agent drive “free” without any narrow and limited streets. Although this map can also be considered easy for its characteristics and its high difficulty of the agent crashing prematurely, the reward statistics will show the real performance as the only positive reward is given for driving at high speed. For some reason, the agent tends to learn to drive on circles, so it will very rarely earn positive points.

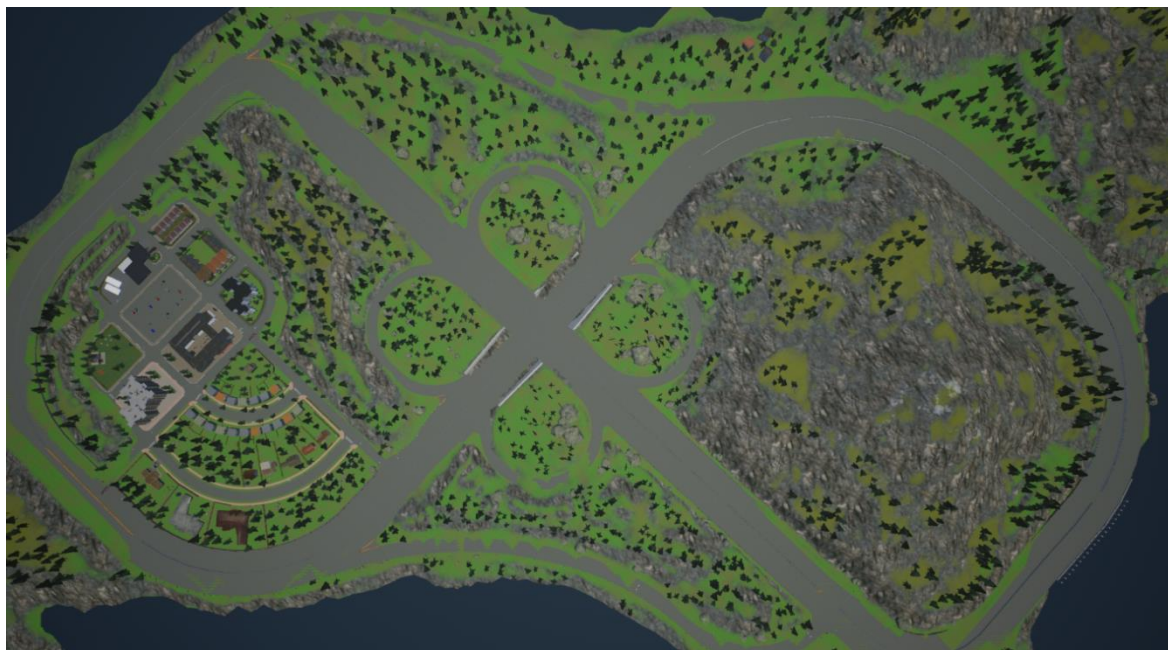


Figure 0.7. Map04 top view in CARLA simulator.

The number of convolutional layers used and the neurons present on each of them are described in the following table.

Map	Map02		
ConvNet used	Number and distribution of neurons		
3 Conv Layers (equals)	16 x3	32 x3	64 x3
2 Conv Layers (equals)	16 x2	32 x2	64 x2
	128 x2		

Table 0.3. Distribution of the convolutional layers used to train the model on Map04.

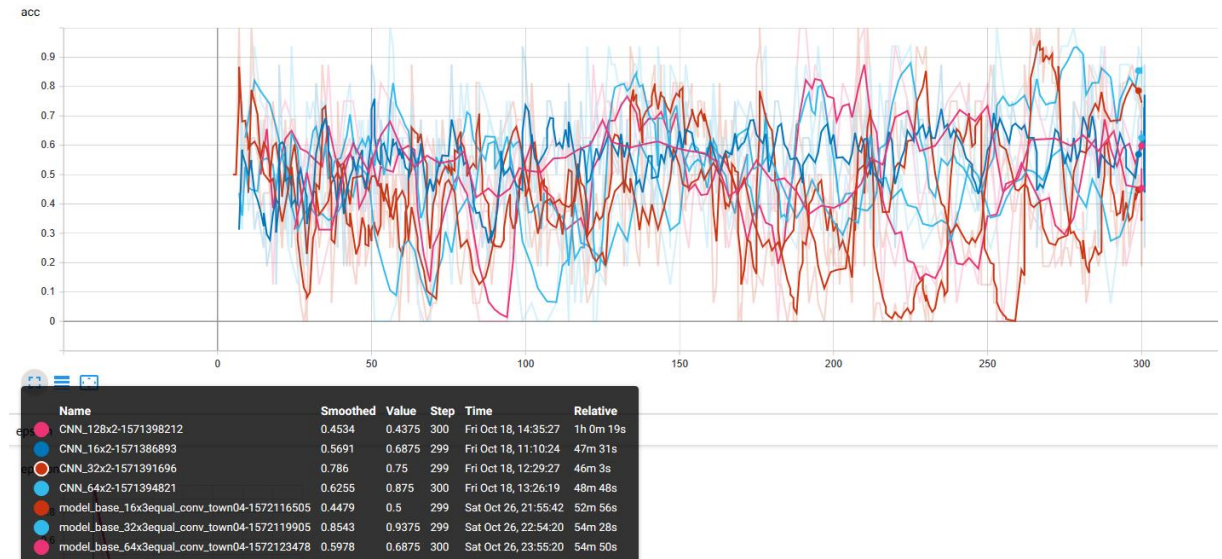


Figure 0.8. Accuracy traces for all the tested models according to Table 0.3. on Map04. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on dark blue – CNN_16x2), with 32 neurons (on red - CNN_16x2), with 64 neurons (on light blue - CNN_16x2), with 128 neurons (on pink - CNN_128x2) and the models set with three identical convolutional layers with 16 neurons (on red - 16x3equal_conv), 32 neurons (on light blue – 32x3equal_conv), 64 neurons (on pink – 64x3equal_conv).

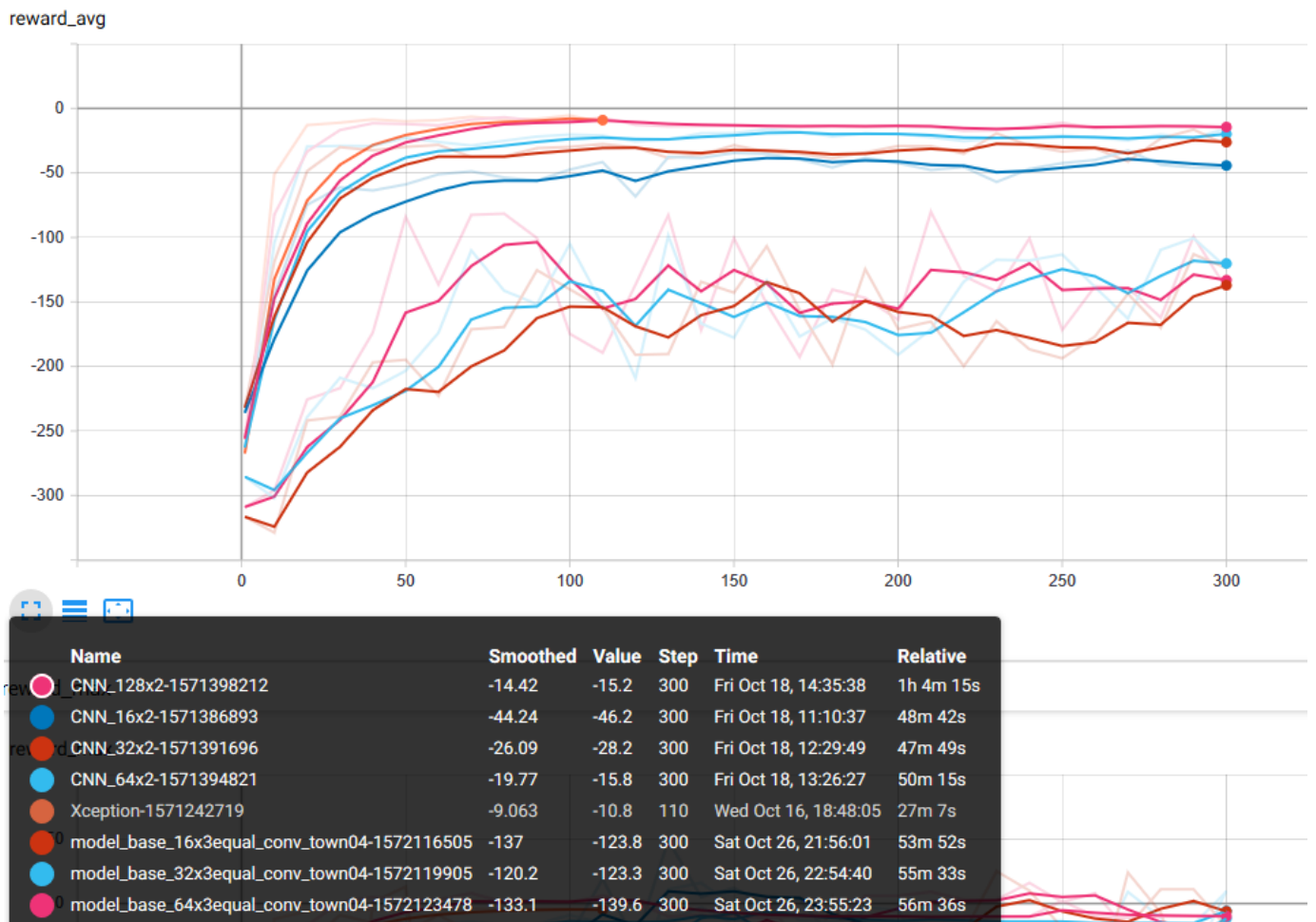


Figure 0.9. Average reward traces for all the tested models according to Table 0.3. on Map04. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on dark blue – CNN_16x2), with 32 neurons (on red - CNN_16x2), with 64 neurons (on light blue - CNN_16x2), with 128 neurons (on pink - CNN_128x2) and the models set with three identical convolutional layers with 16 neurons (on red - 16x3equal_conv), 32 neurons (on light blue – 32x3equal_conv), 64 neurons (on pink – 64x3equal_conv).

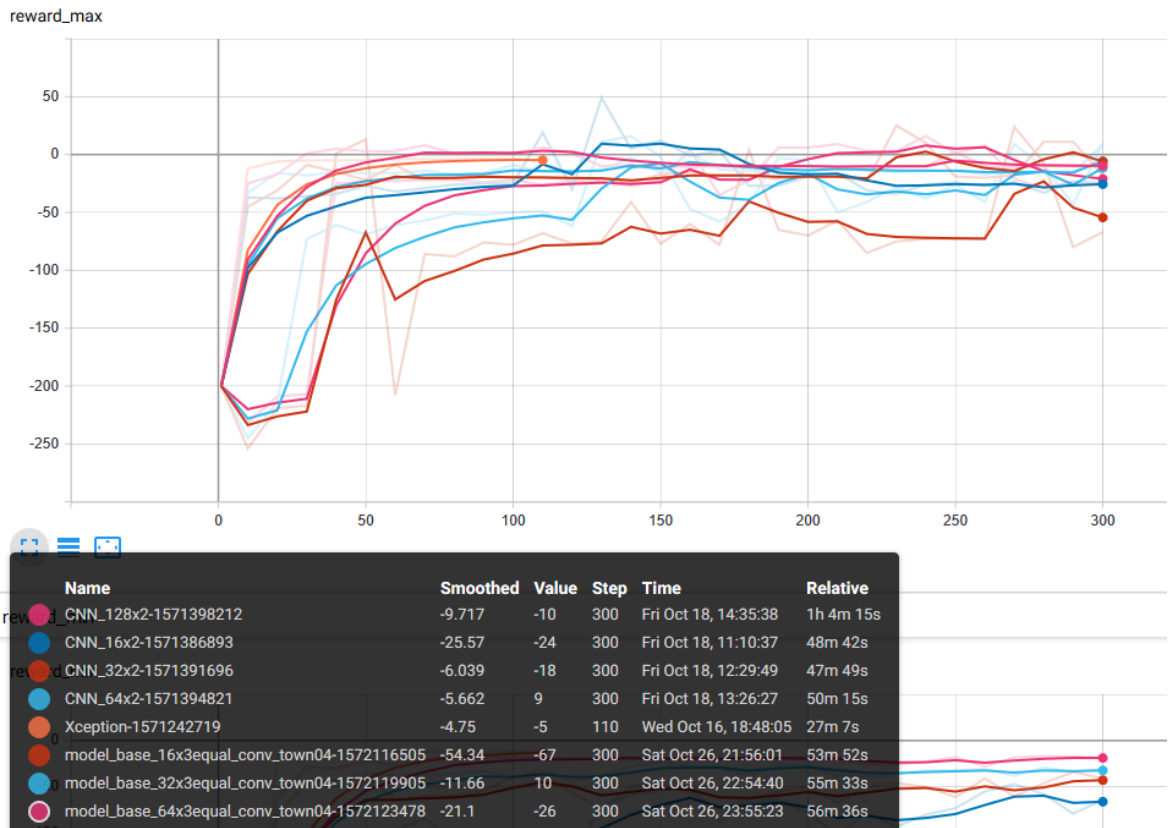


Figure 0.10. Maximum reward traces for all the tested models according to Table 0.3. on Map04. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on dark blue – CNN_16x2), with 32 neurons (on red - CNN_16x2), with 64 neurons (on light blue - CNN_16x2), with 128 neurons (on pink - CNN_128x2) and the models set with three identical convolutional layers with 16 neurons (on red - 16x3equal_conv), 32 neurons (on light blue – 32x3equal_conv), 64 neurons (on pink – 64x3equal_conv).

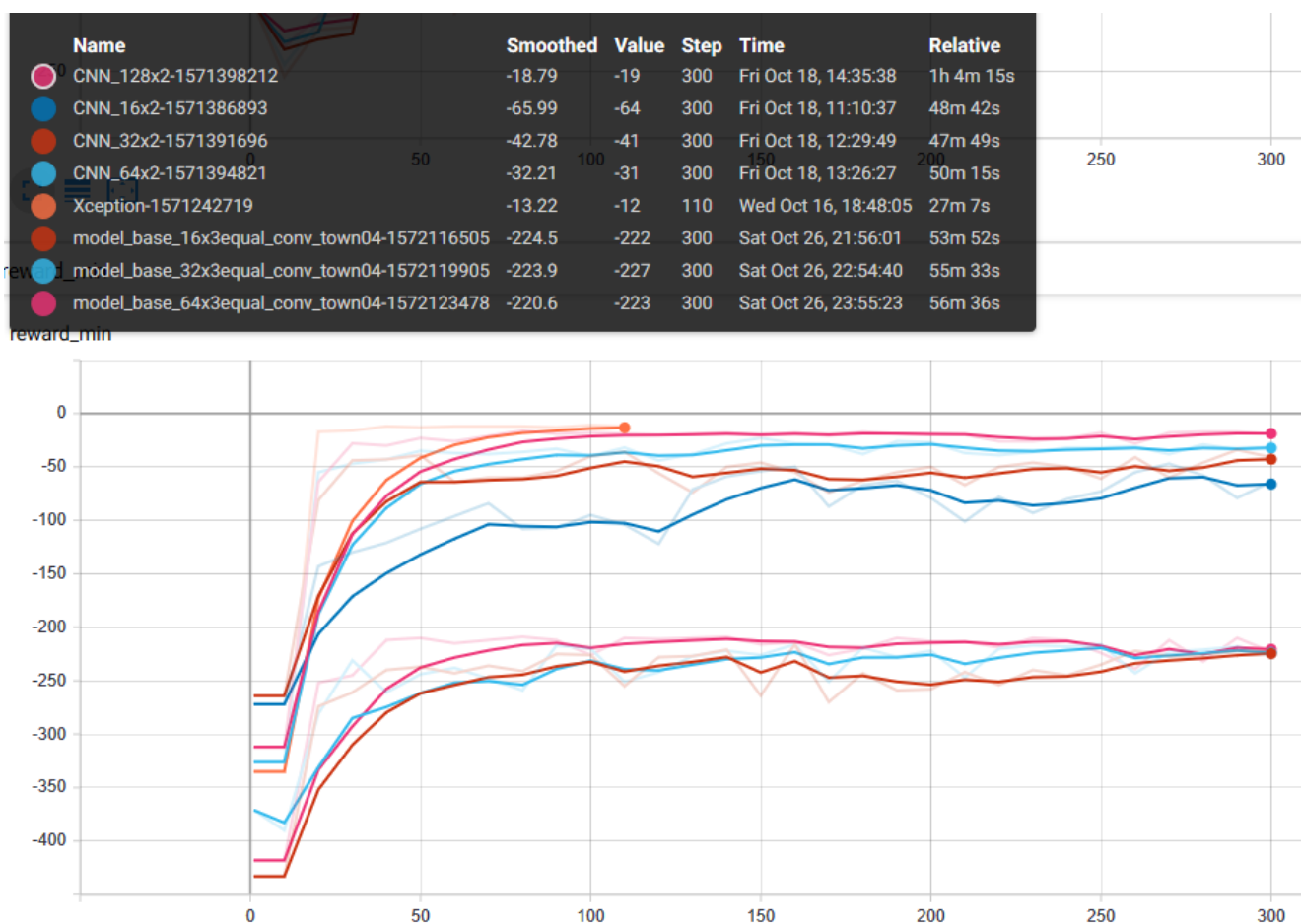


Figure 0.11. Minimum reward traces for all the tested models according to Table 0.3. on Map04. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on dark blue – CNN_16x2), with 32 neurons (on red - CNN_16x2), with 64 neurons (on light blue - CNN_16x2), with 128 neurons (on pink - CNN_128x2) and the models set with three identical convolutional layers with 16 neurons (on red - 16x3equal_conv), 32 neurons (on light blue – 32x3equal_conv), 64 neurons (on pink – 64x3equal_conv).

A1.1.3 Map07

This map is expected to be the hardest one as it is the closest to a “real racetrack”. For this reason, the data from the models tested in this map will have more weight in the final decision.



Figure 0.12. Map07 top view in CARLA simulator.

The number of convolutional layers used and the neurons present on each of them are described in the following table.

Map	Map07		
ConvNet used	Number and distribution of neurons		
2 Conv Layers (equals)	16 x2	32 x2	64 x2
3 Conv Layers (equals)	16 x3	32 x3	64 x3

Table 0.4. Distribution of the convolutional layers used to train the model on Map07.

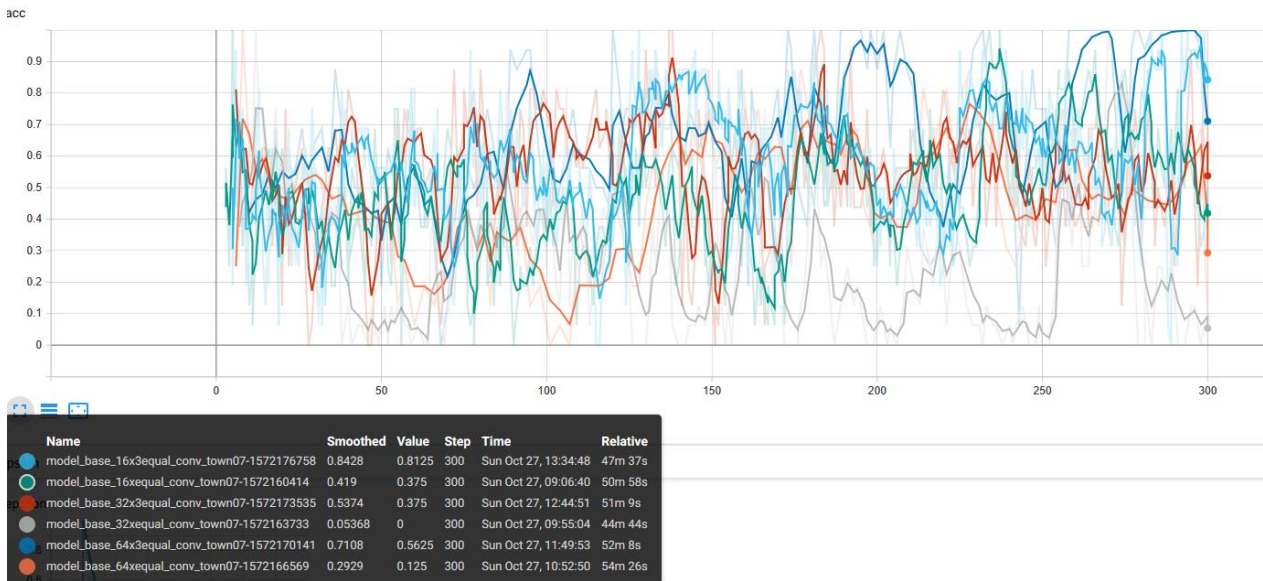


Figure 0.13. Accuracy traces for all the tested models according to Table 0.4. on Map07. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey – 32x2equal_conv), with 64 neurons (on light orange – 64x2equal_conv) and the models set with three identical convolutional layers with 16 neurons (on light blue - 16x3equal_conv), 32 neurons (on red – 32x3equal_conv), 64 neurons (on dark blue – 64x3equal_conv).

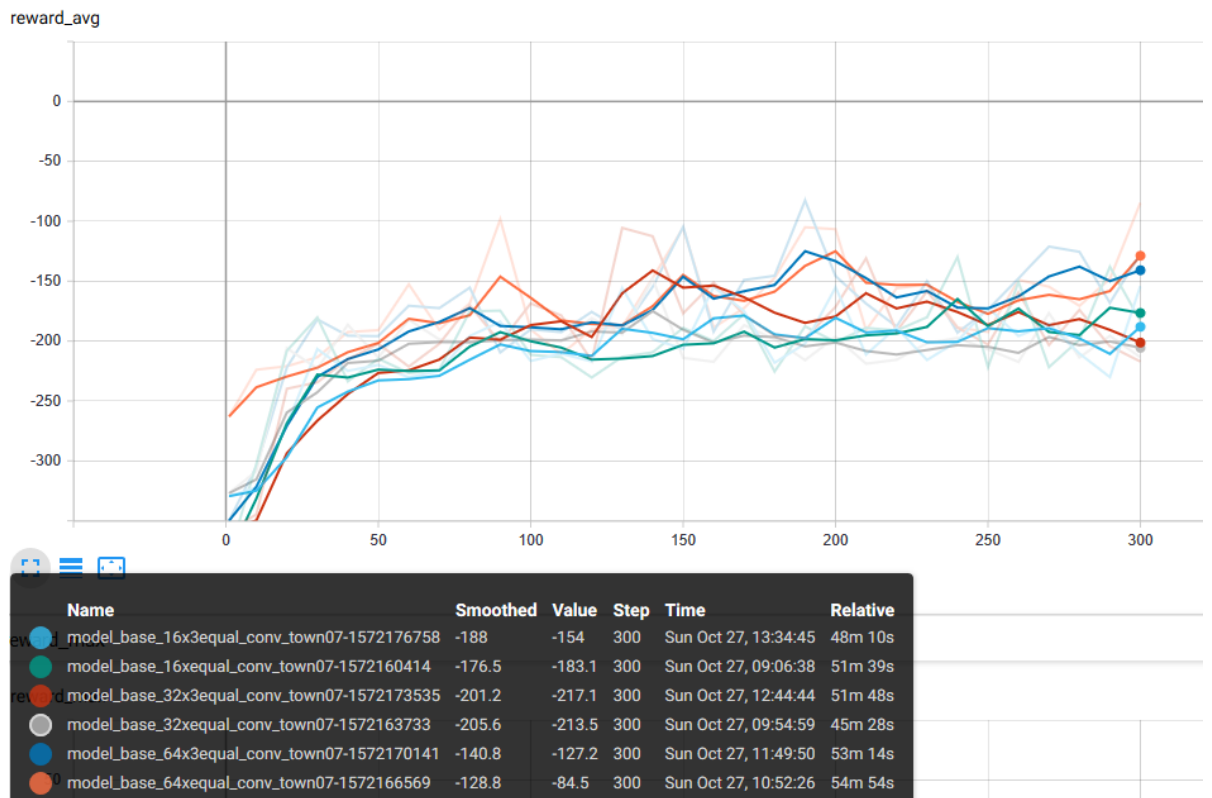


Figure 0.14. Average reward traces for all the tested models according to Table 0.4. on Map07. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey – 32x2equal_conv), with 64 neurons (on light orange – 64x2equal_conv) and the models set with three identical convolutional layers with 16 neurons (on light blue - 16x3equal_conv), 32 neurons (on red – 32x3equal_conv), 64 neurons (on dark blue – 64x3equal_conv).

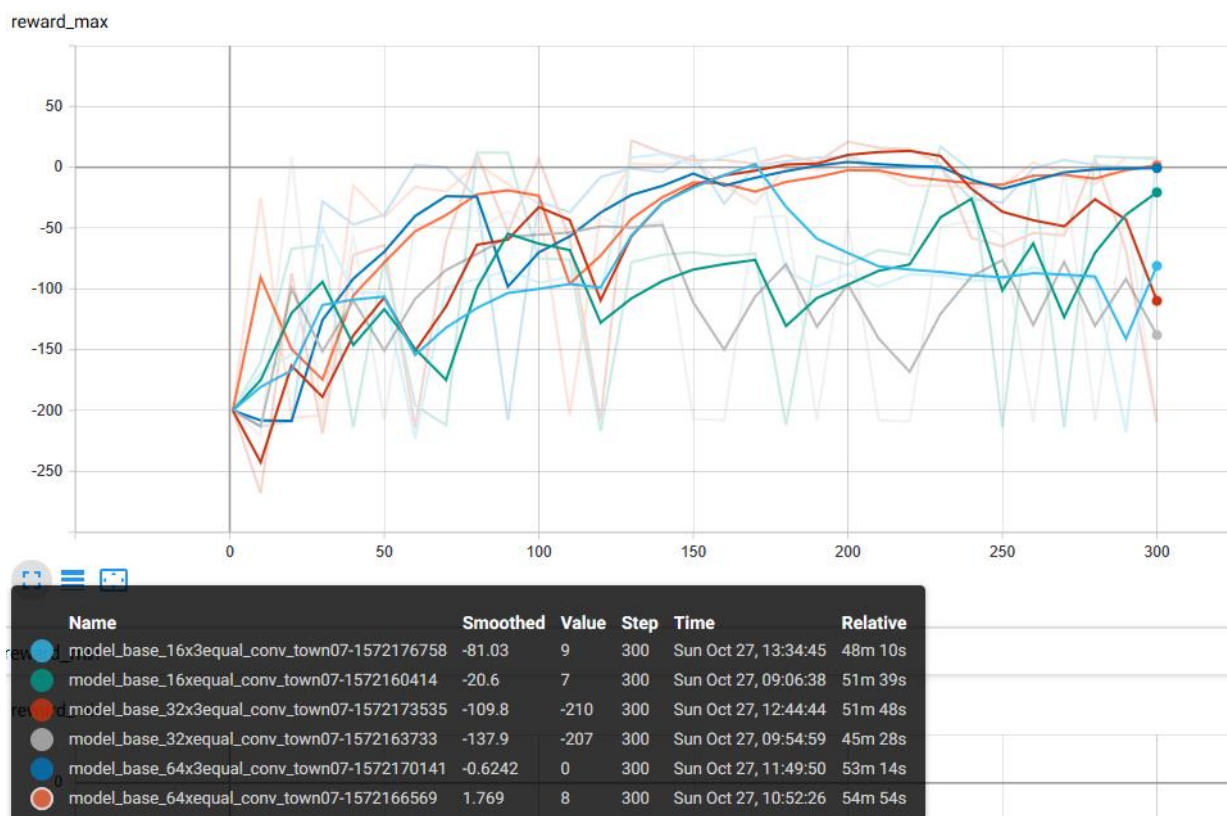


Figure 0.15. Maximum reward traces for all the tested models according to Table 0.4. on Map07. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey – 32x2equal_conv), with 64 neurons (on light orange – 64x2equal_conv) and the models set with three identical convolutional layers with 16 neurons (on light blue - 16x3equal_conv), 32 neurons (on red – 32x3equal_conv), 64 neurons (on dark blue – 64x3equal_conv).

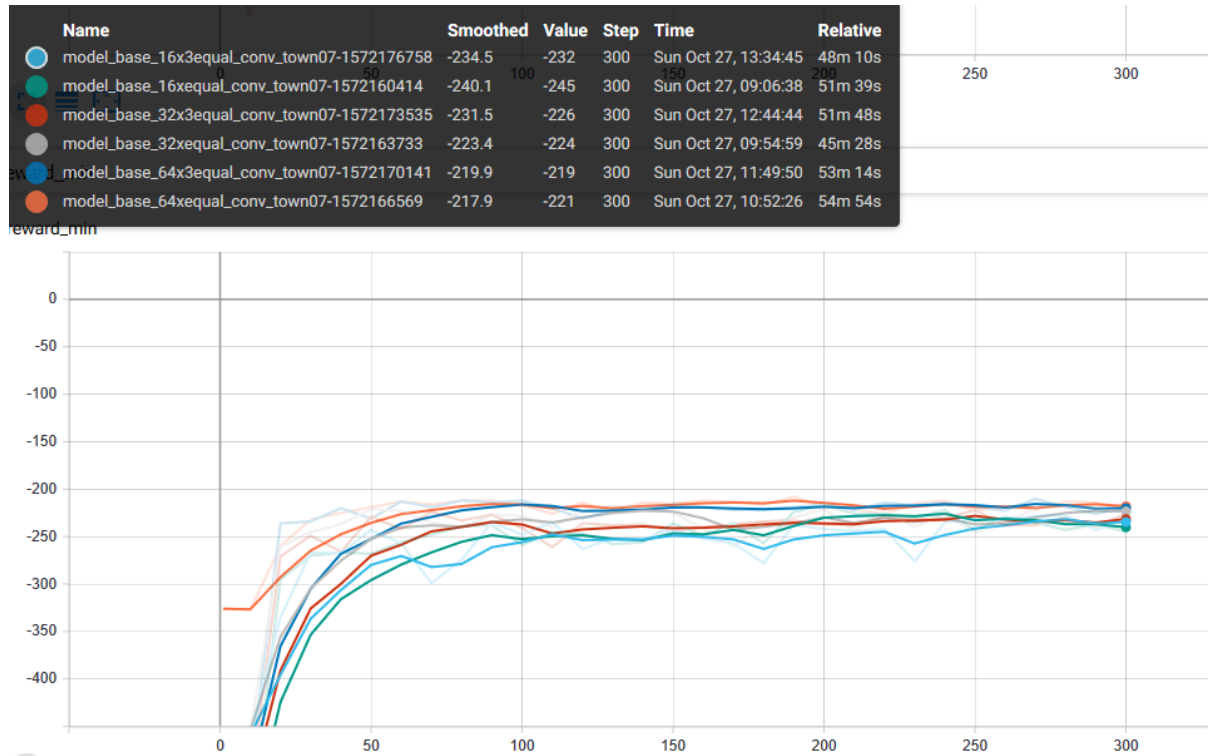


Figure 0.16. Minimum reward traces for all the tested models according to Table 0.4. on Map07. All the traces represented contain the models set with two identical convolutional layers with 16 neurons (on green – 16x2equal_conv), with 32 neurons (on grey – 32x2equal_conv), with 64 neurons (on light orange – 64x2equal_conv) and the models set with three identical convolutional layers with 16 neurons (on light blue - 16x3equal_conv), 32 neurons (on red – 32x3equal_conv), 64 neurons (on dark blue – 64x3equal_conv).

A1.2 Comparative between maps

A second comparative is done between the same models on the three different maps in order to compare the performance of the same model on different situations.

A1.2.1 2 Conv Layers of 16 neurons each

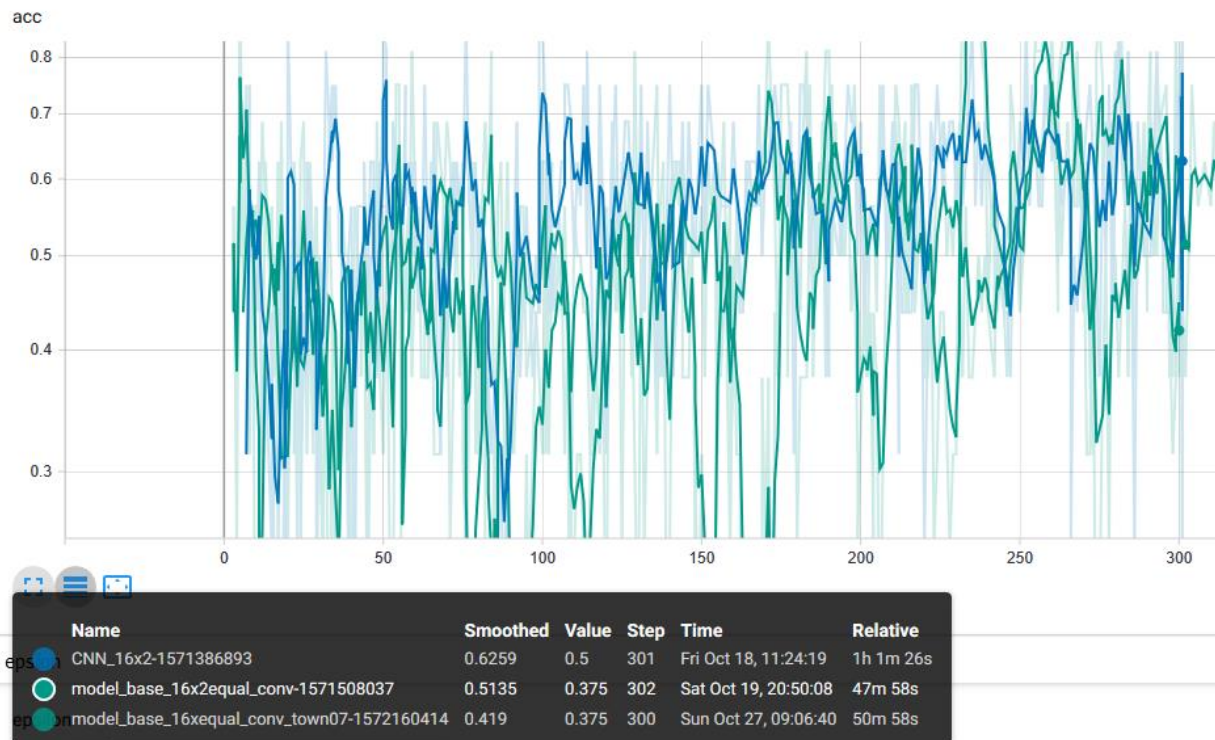


Figure 0.17. Accuracy traces for all the tested models set with two identical convolutional layers with 16 neurons on Map02 (trace on green), on Map04 (trace on blue) and on Map07 (trace on green).

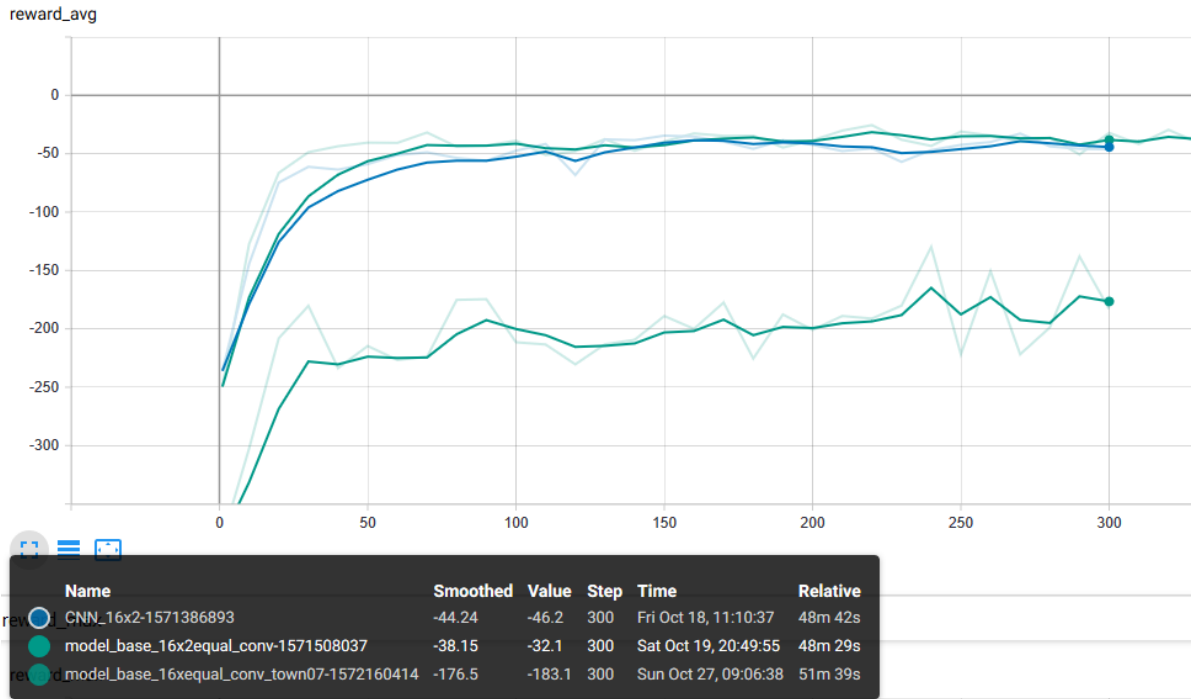


Figure 0.18. Average reward traces for all the tested models set with two identical convolutional layers with 16 neurons on Map02 (trace on green), on Map04 (trace on blue) and on Map07 (trace on green).

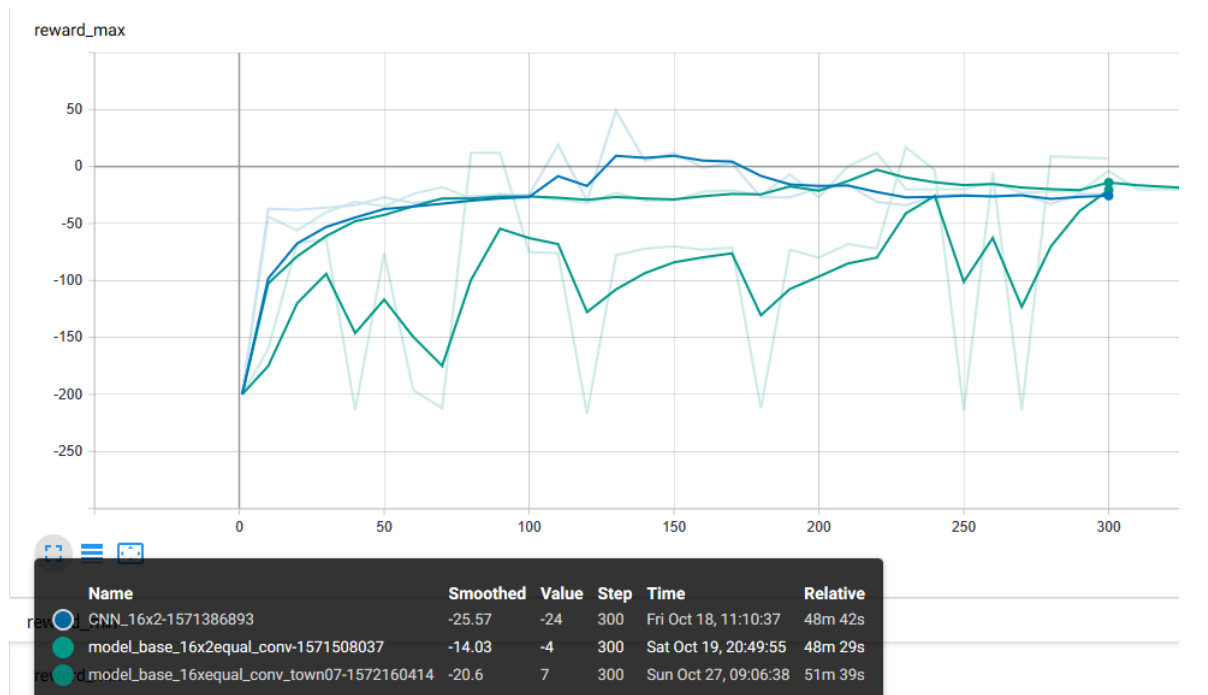


Figure 0.19. Maximum reward traces for all the tested models set with two identical convolutional layers with 16 neurons on Map02 (trace on green), on Map04 (trace on blue) and on Map07 (trace on green).

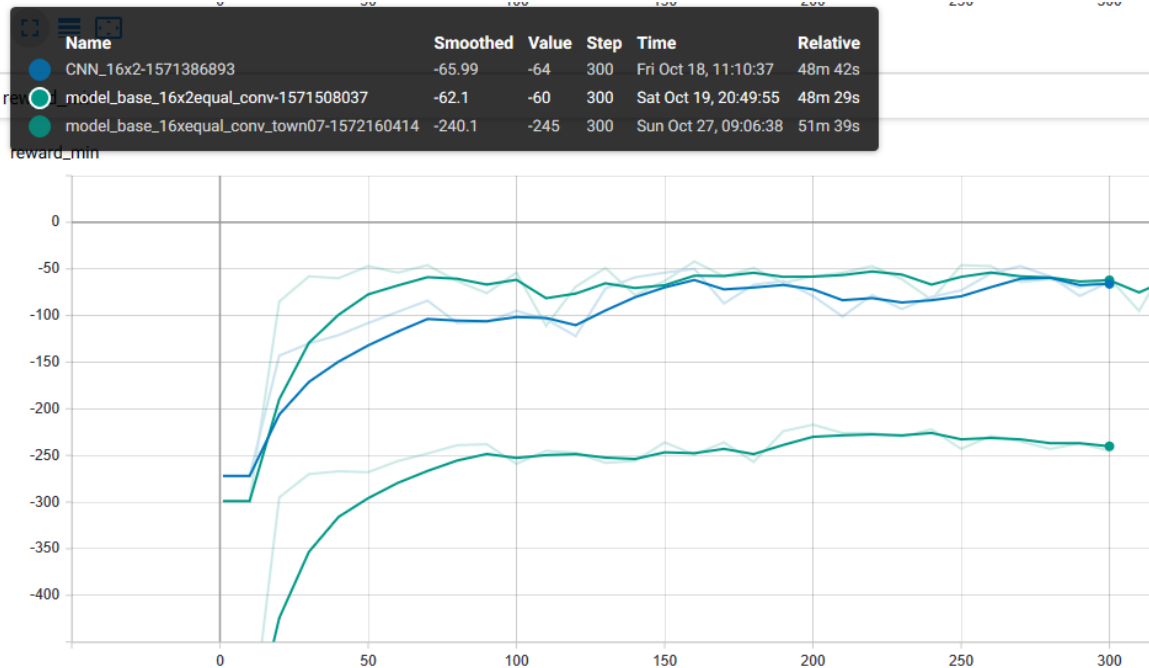


Figure 0.20. Minimum reward traces for all the tested models set with two identical convolutional layers with 16 neurons on Map02 (trace on green), on Map04 (trace on blue) and on Map07 (trace on grey).

A1.2.2 2 Conv Layers of 32 neurons each

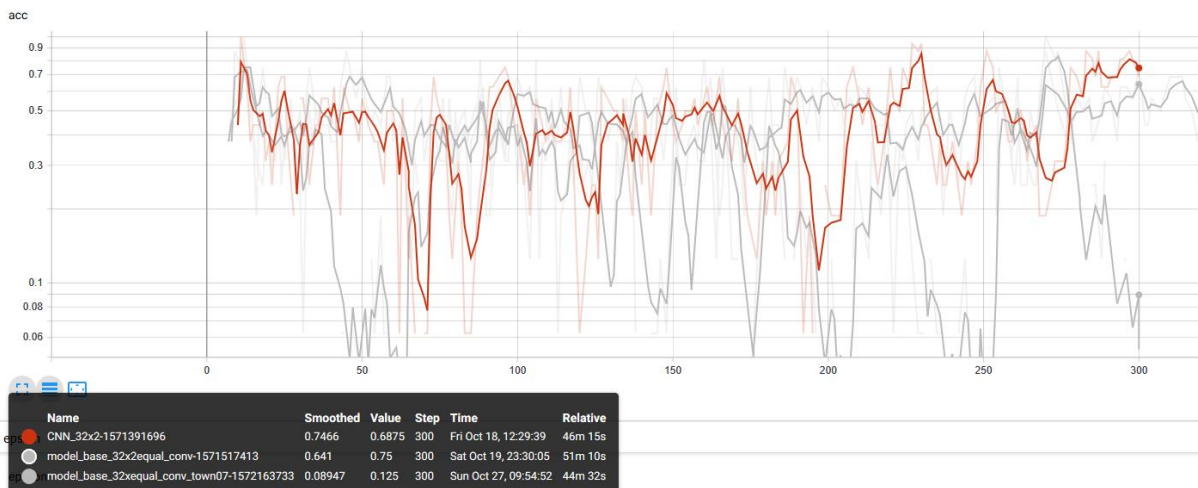


Figure 0.21. Accuracy traces for all the tested models set with two identical convolutional layers with 32 neurons on Map02 (trace on grey), on Map04 (trace on orange) and on Map07 (trace on grey).

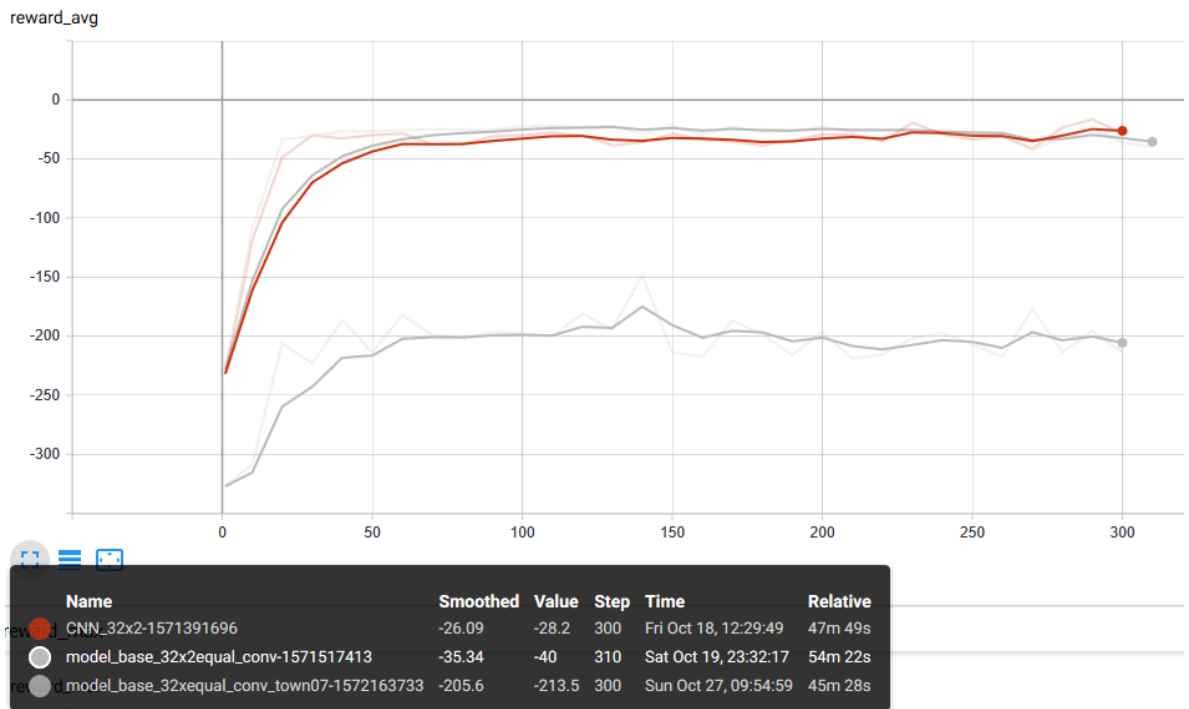


Figure 0.22. Average reward traces for all the tested models set with two identical convolutional layers with 32 neurons on Map02 (trace on grey), on Map04 (trace on orange) and on Map07 (trace on grey).

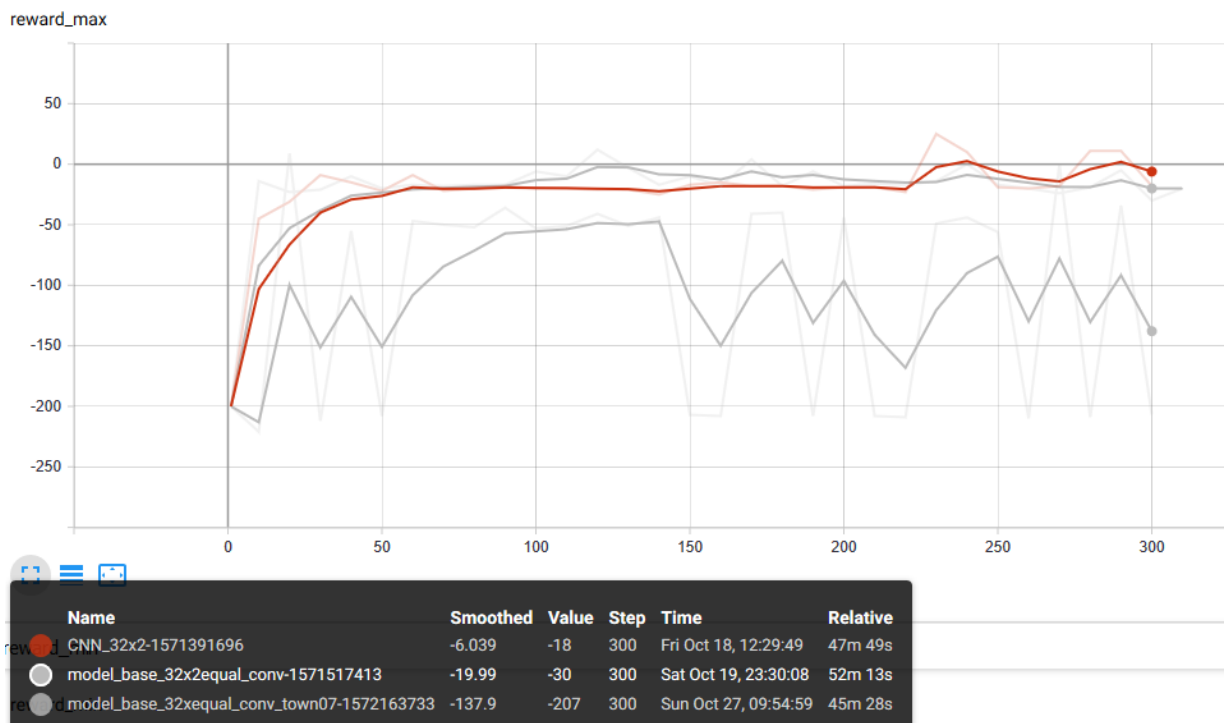


Figure 0.23. Maximum reward traces for all the tested models set with two identical convolutional layers with 32 neurons on Map02 (trace on grey), on Map04 (trace on orange) and on Map07 (trace on grey).

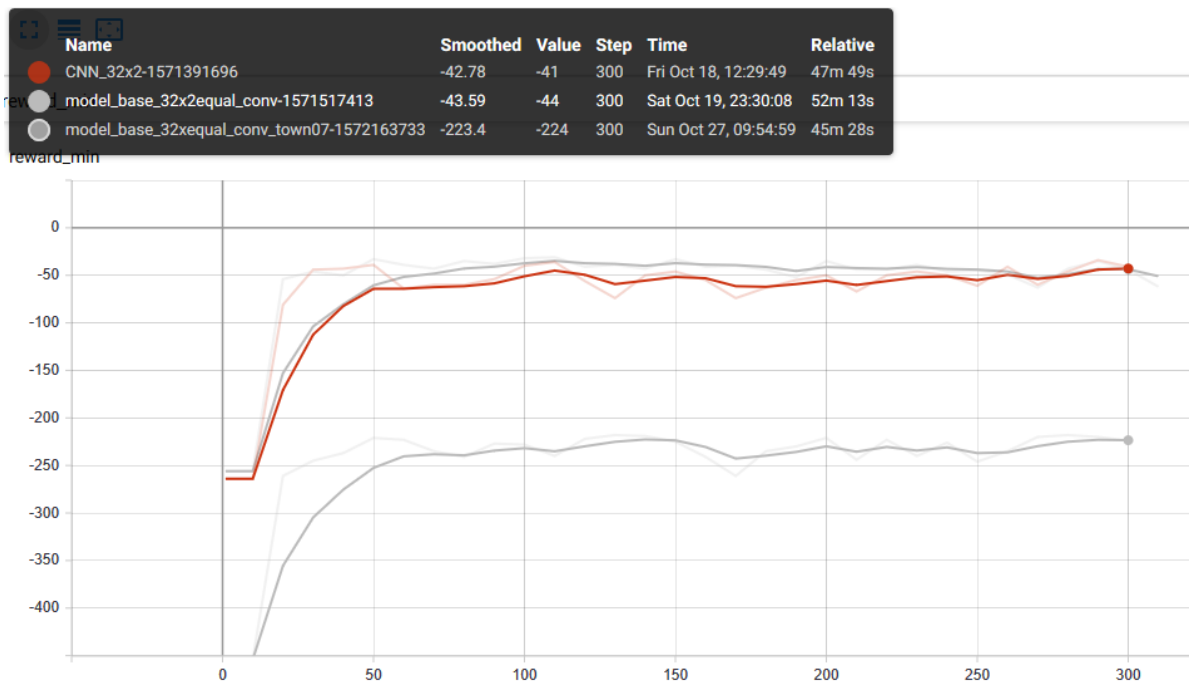


Figure 0.24. Minimum reward traces for all the tested models set with two identical convolutional layers with 32 neurons on Map02 (trace on grey), on Map04 (trace on orange) and on Map07 (trace on grey).

A1.2.3 2 Conv Layers of 64 neurons each

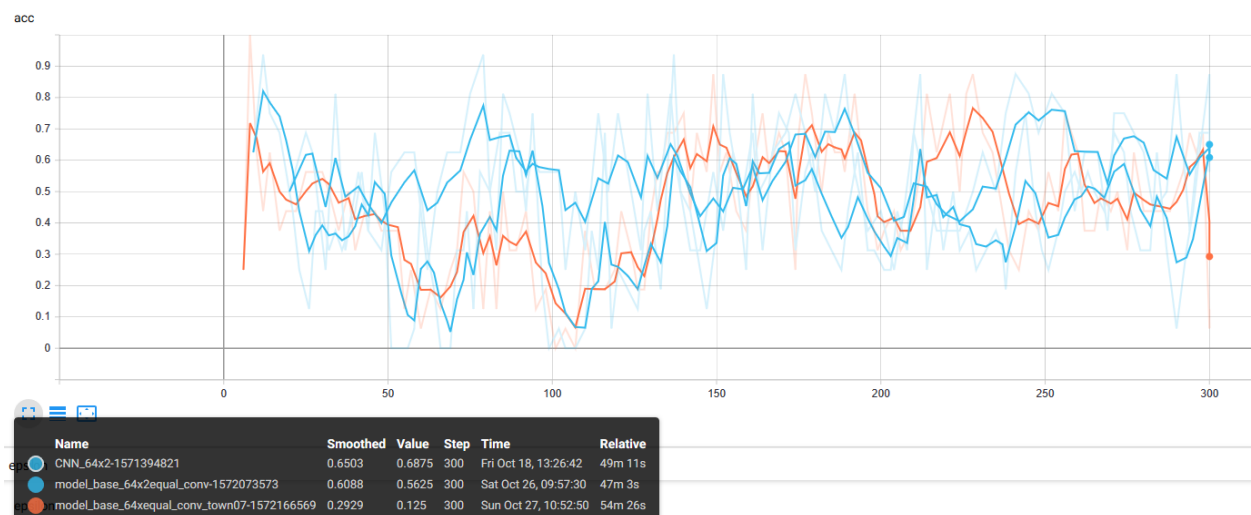


Figure 0.25. Accuracy traces for all the tested models set with two identical convolutional layers with 64 neurons on Map02 (trace on blue), on Map04 (trace on green) and on Map07 (trace on orange).

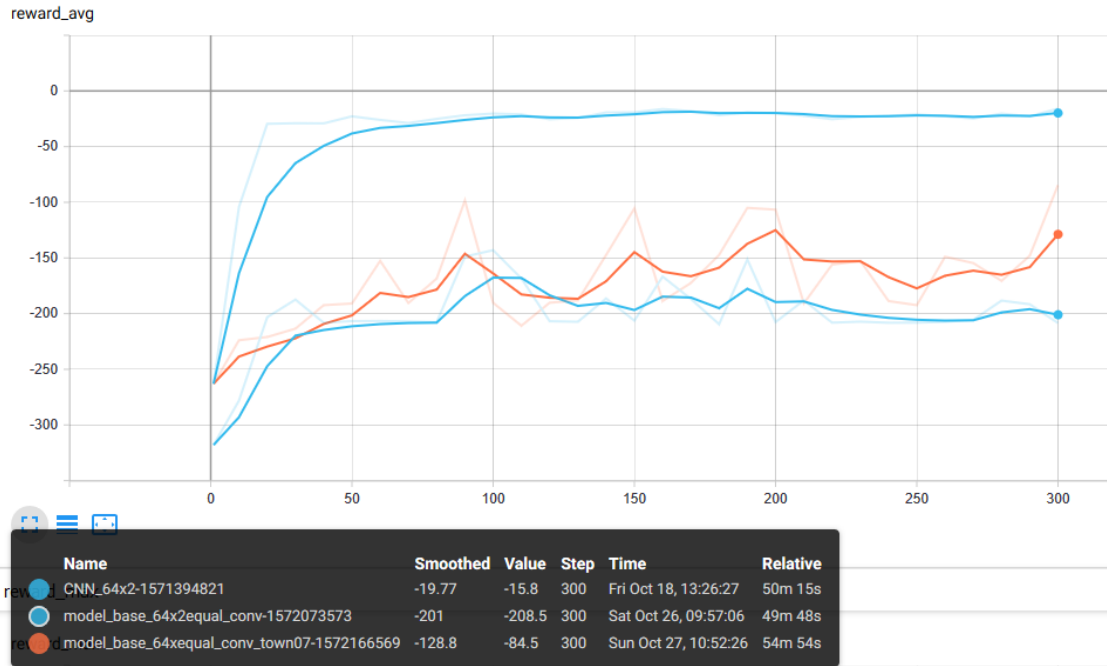


Figure 0.26. Average reward traces for all the tested models set with two identical convolutional layers with 64 neurons on Map02 (trace on blue), on Map04 (trace on green) and on Map07 (trace on orange).

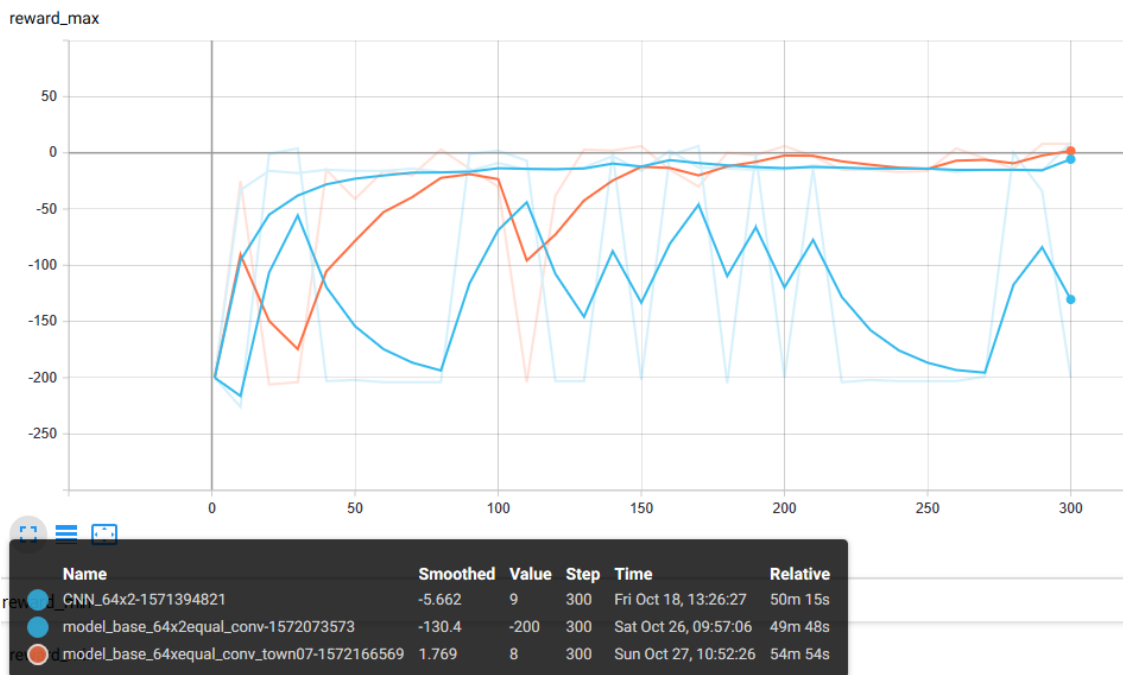


Figure 0.27. Minimum reward traces for all the tested models set with two identical convolutional layers with 64 neurons on Map02 (trace on blue), on Map04 (trace on green) and on Map07 (trace on orange).

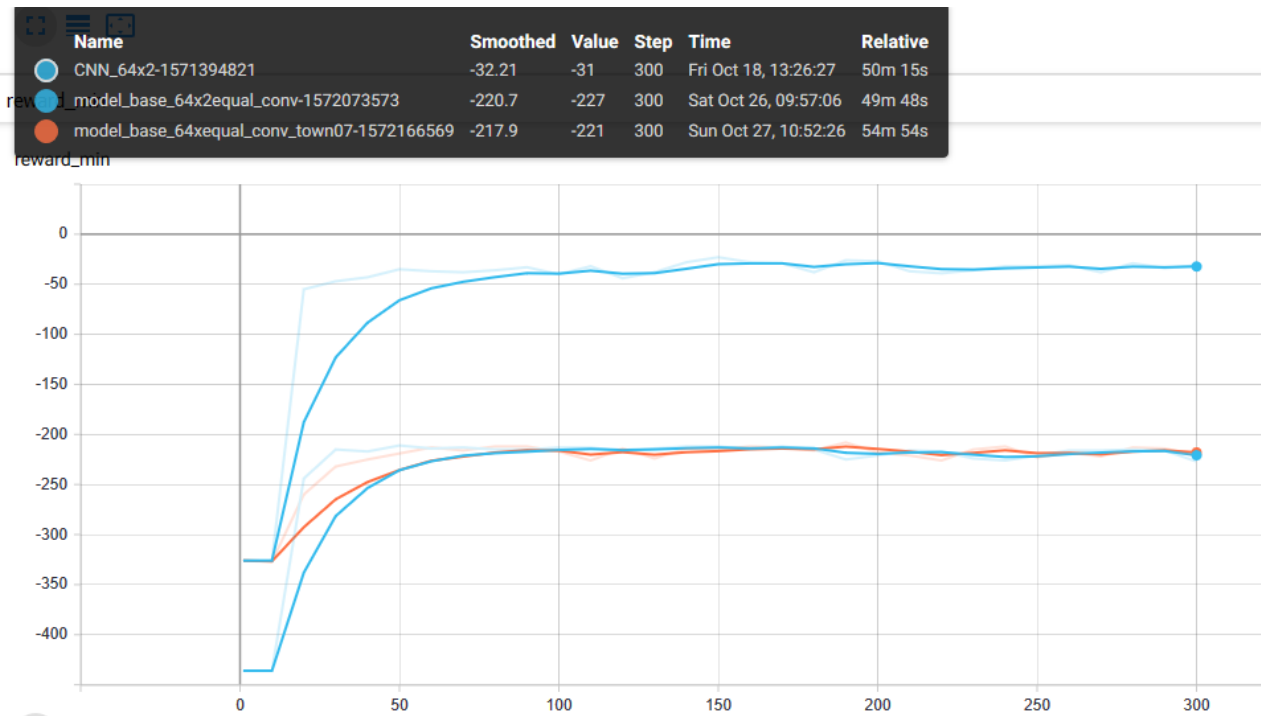


Figure 0.28. Maximum reward traces for all the tested models set with two identical convolutional layers with 64 neurons on Map02 (trace on blue), on Map04 (trace on green) and on Map07 (trace on orange).

A1.2.4 3 Conv Layers of 16 neurons each

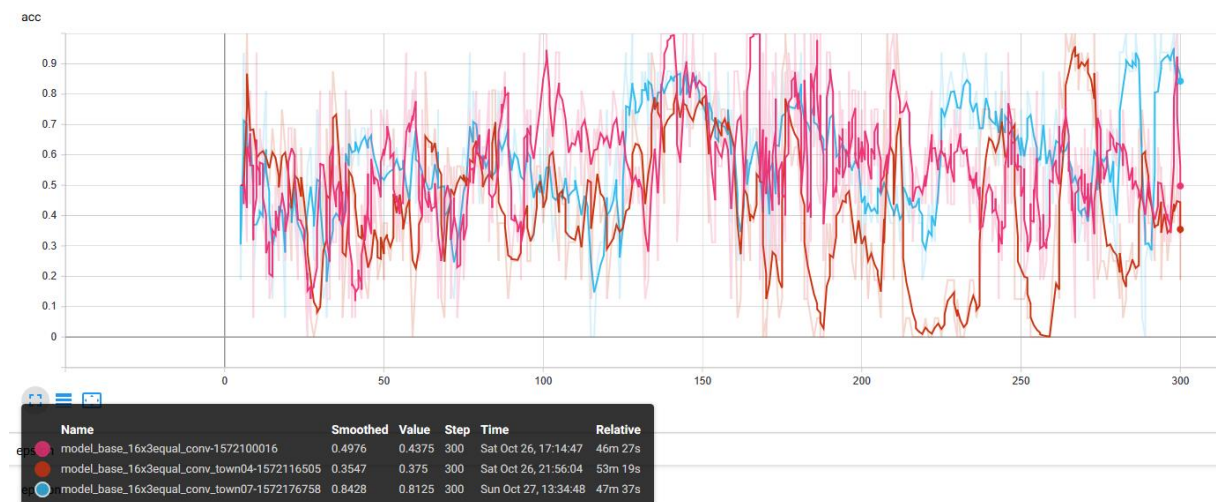


Figure 0.29. Accuracy traces for all the tested models set with three identical convolutional layers with 16 neurons on Map02 (trace on pink), on Map04 (trace on red) and on Map07 (trace on blue).

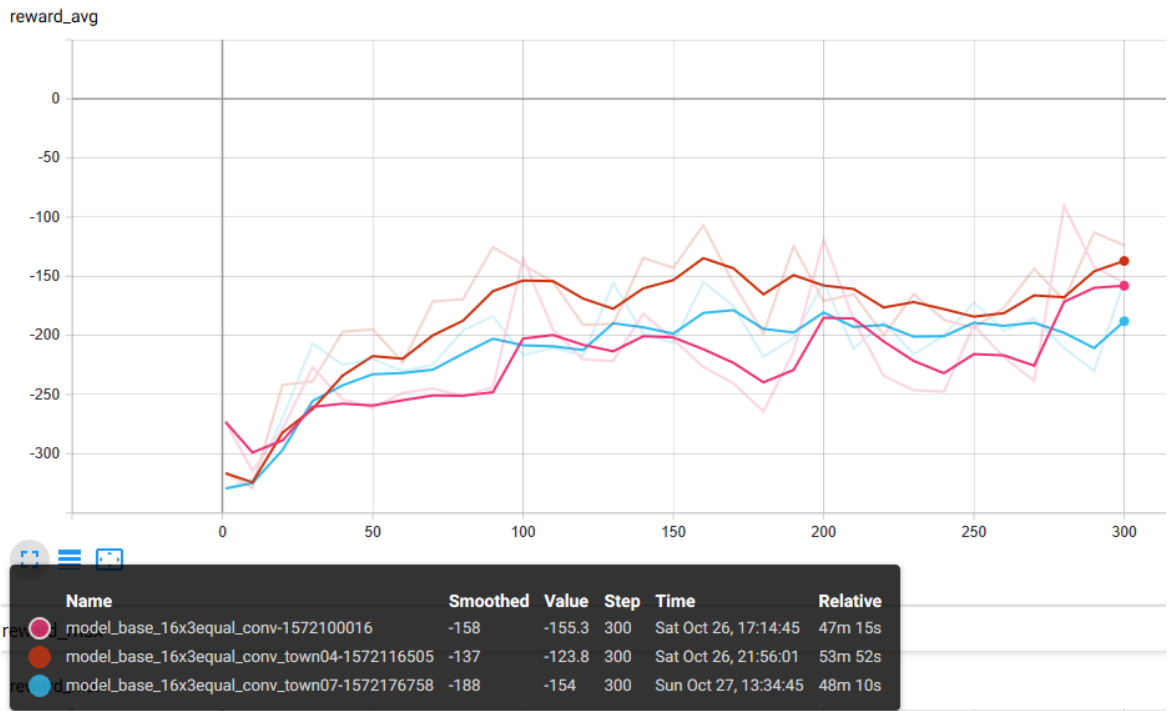


Figure 0.30. Average reward traces for all the tested models set with three identical convolutional layers with 16 neurons on Map02 (trace on pink), on Map04 (trace on red) and on Map07 (trace on blue).

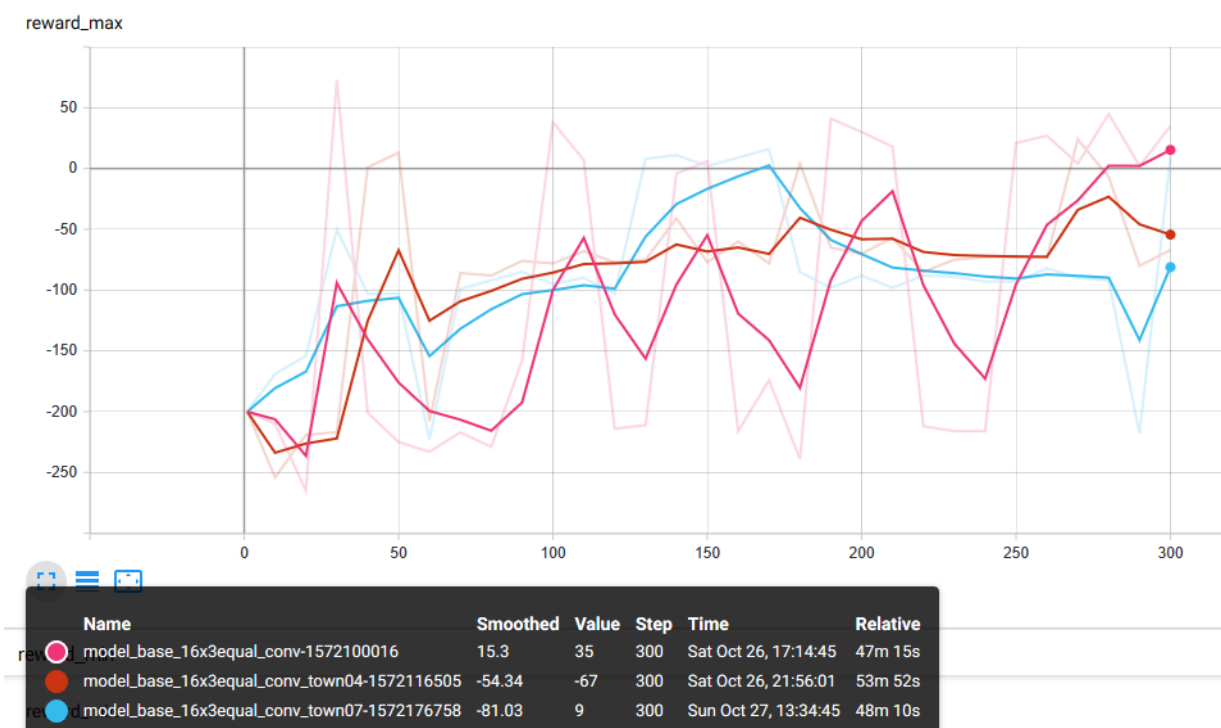


Figure 0.31. Maximum reward traces for all the tested models set with three identical convolutional layers with 16 neurons on Map02 (trace on pink), on Map04 (trace on red) and on Map07 (trace on blue).

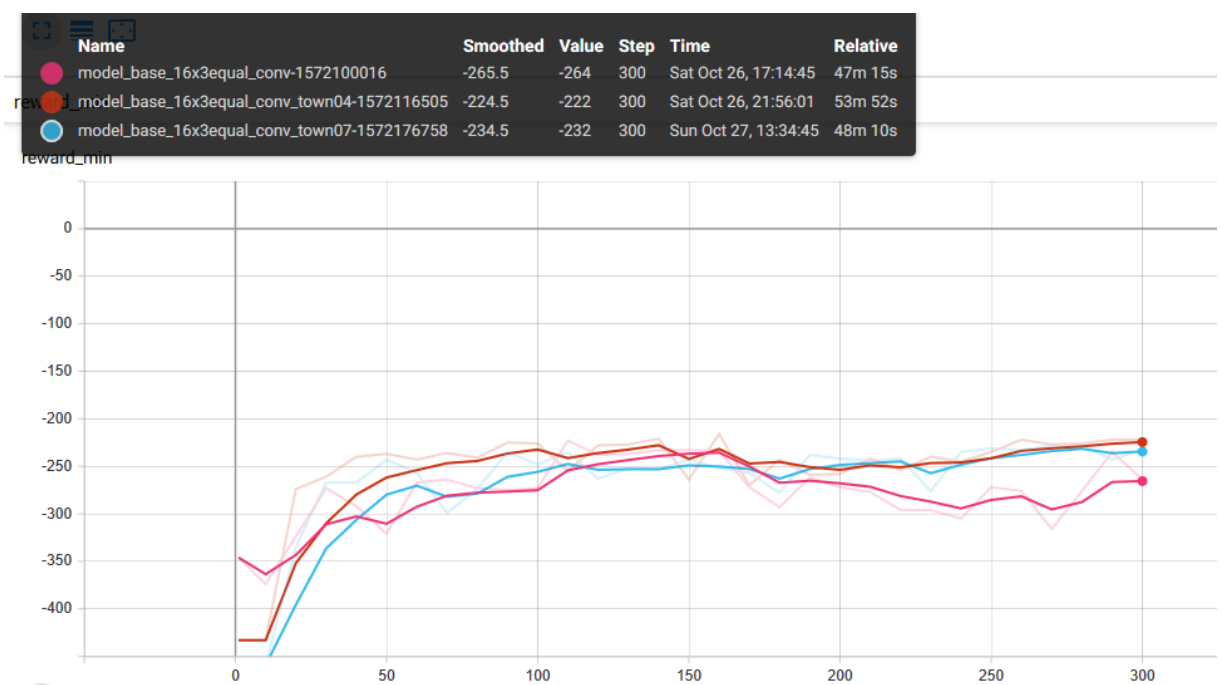


Figure 0.32. Minimum reward traces for all the tested models set with three identical convolutional layers with 16 neurons on Map02 (trace on pink), on Map04 (trace on red) and on Map07 (trace on blue).

A1.2.5 3 Conv Layers of 32 neurons each

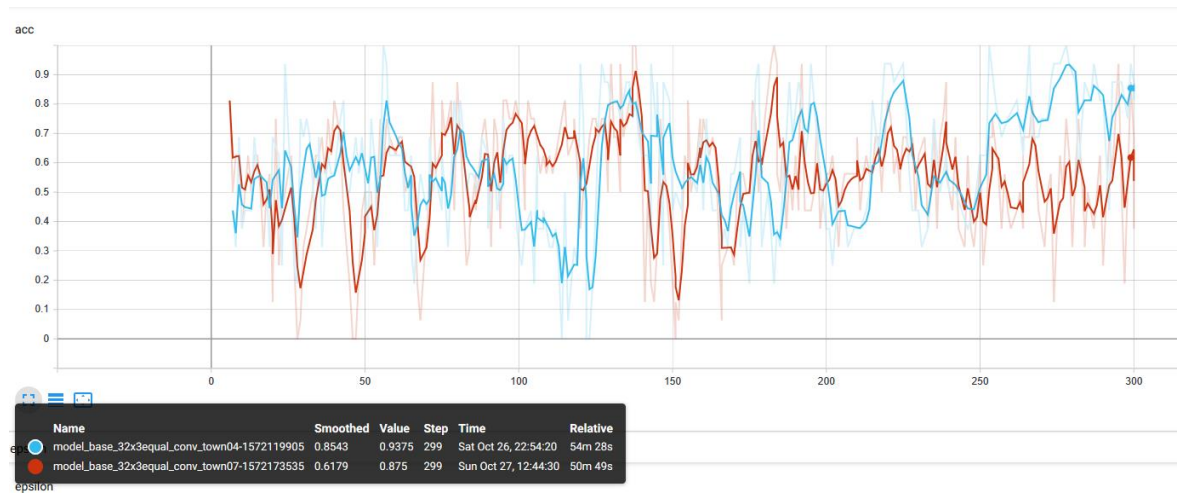


Figure 0.33. Accuracy traces for all the tested models set with three identical convolutional layers with 32 neurons on Map04 (trace on light blue) and on Map07 (trace on red).

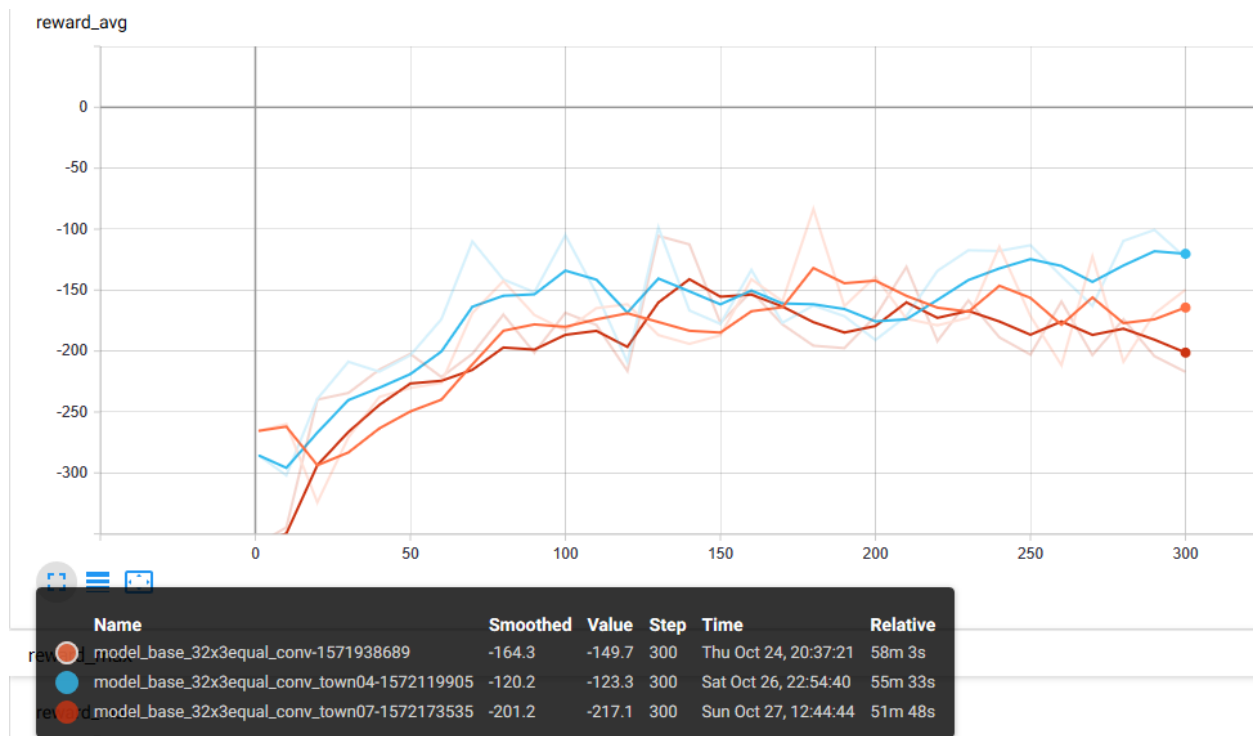


Figure 0.34. Average reward traces for all the tested models set with three identical convolutional layers with 32 neurons on Map02 (trace on orange), on Map04 (trace on light blue) and on Map07 (trace on orange).

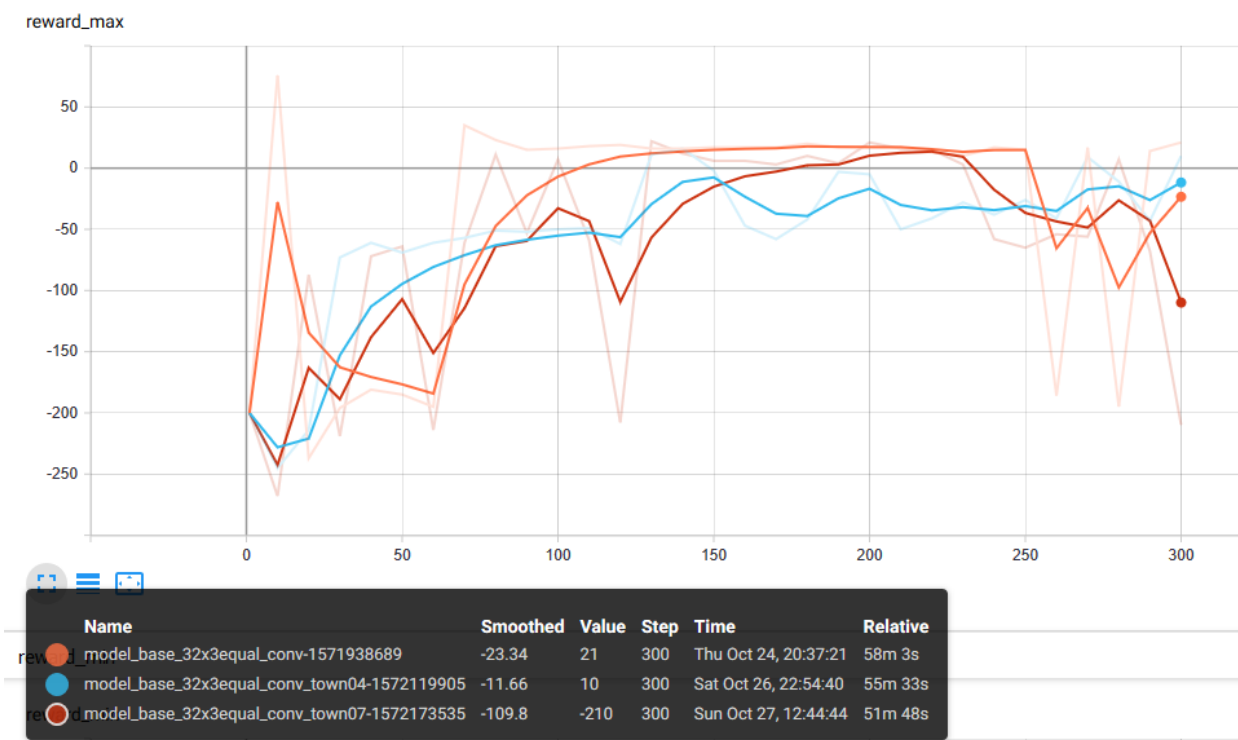


Figure 0.35. Maximum reward traces for all the tested models set with three identical convolutional layers with 32 neurons on Map02 (trace on orange), on Map04 (trace on light blue) and on Map07 (trace on orange).

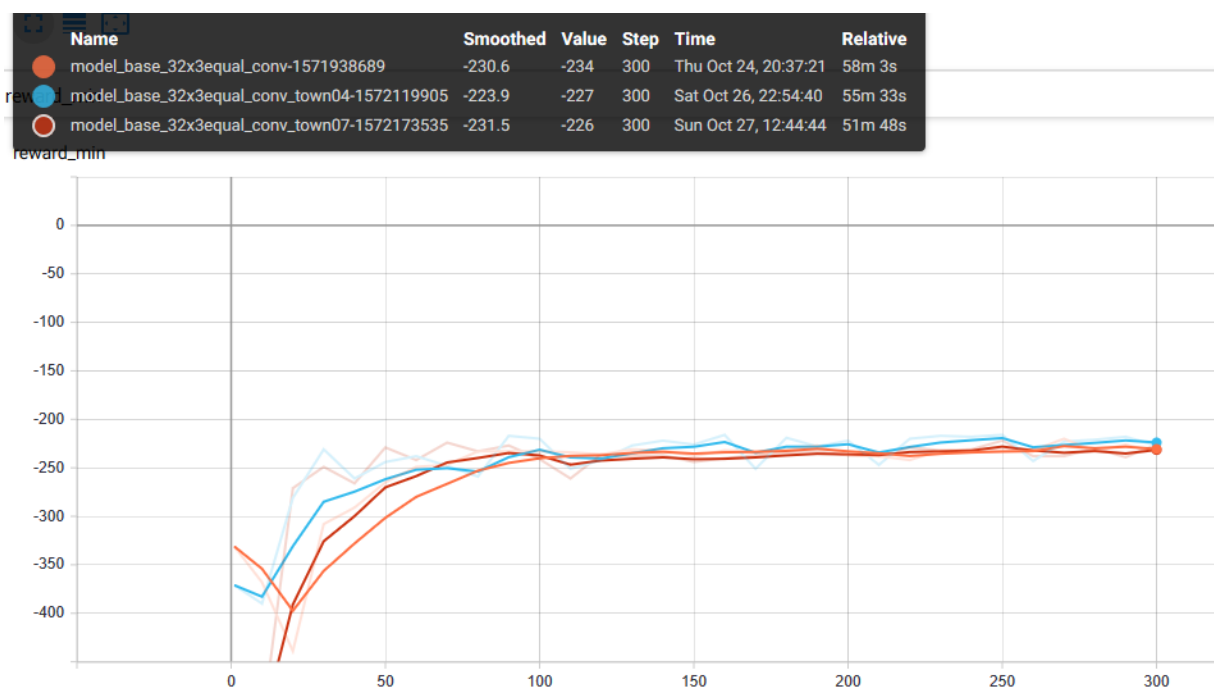


Figure 0.36. Minimum reward traces for all the tested models set with three identical convolutional layers with 32 neurons on Map02 (trace on orange), on Map04 (trace on light blue) and on Map07 (trace on orange).

A1.2.6 3 Conv Layers of 64 neurons each

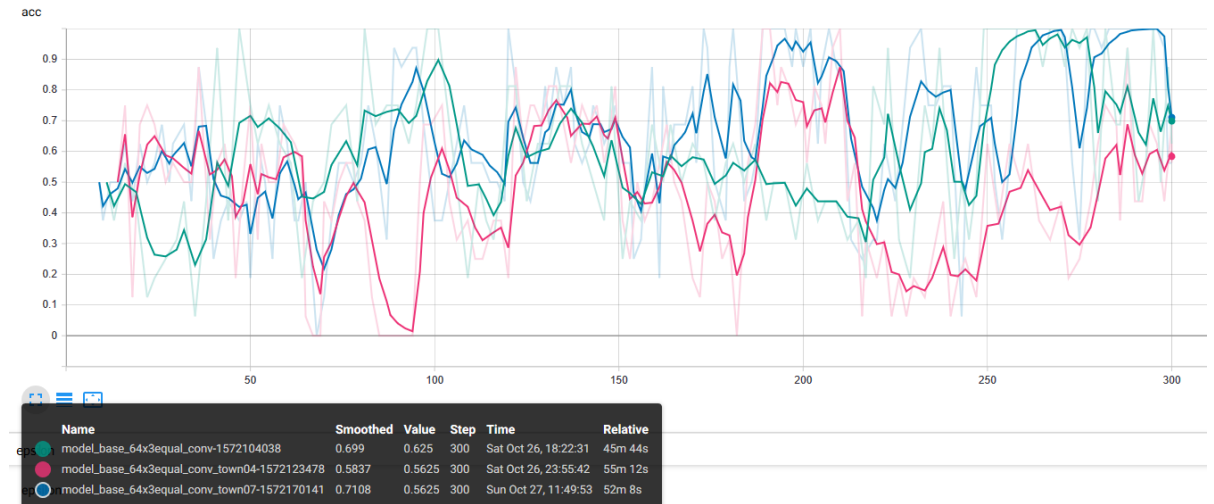


Figure 0.37. Accuracy traces for all the tested models set with three identical convolutional layers with 64 neurons on Map02 (trace on green), on Map04 (trace on pink) and on Map07 (trace on dark blue).

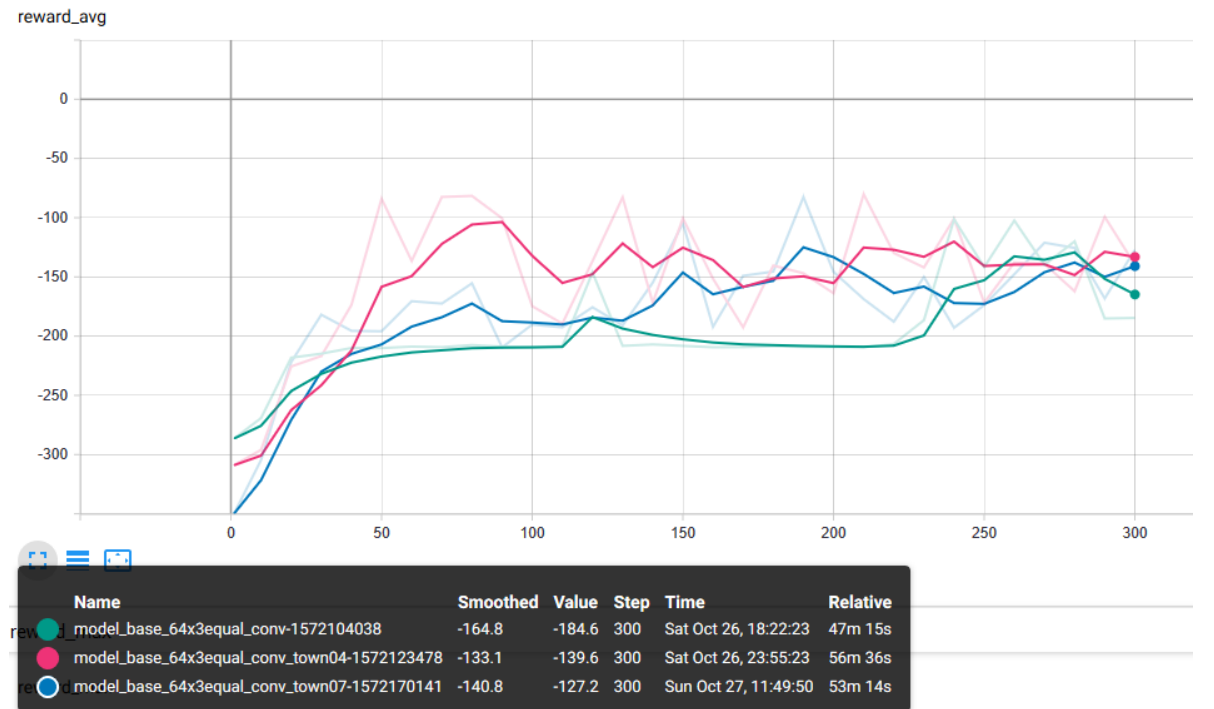


Figure 0.38. Average reward traces for all the tested models set with three identical convolutional layers with 64 neurons on Map02 (trace on green), on Map04 (trace on pink) and on Map07 (trace on dark blue).

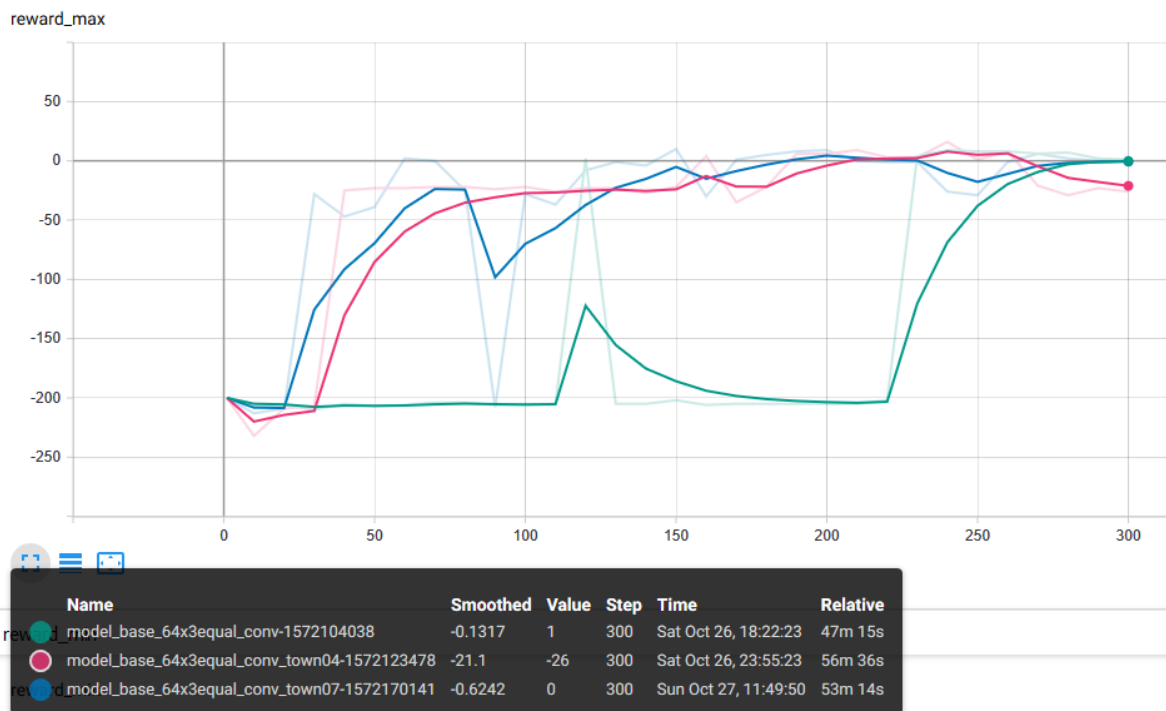


Figure 0.39. Maximum reward traces for all the tested models set with three identical convolutional layers with 64 neurons on Map02 (trace on green), on Map04 (trace on pink) and on Map07 (trace on dark blue).

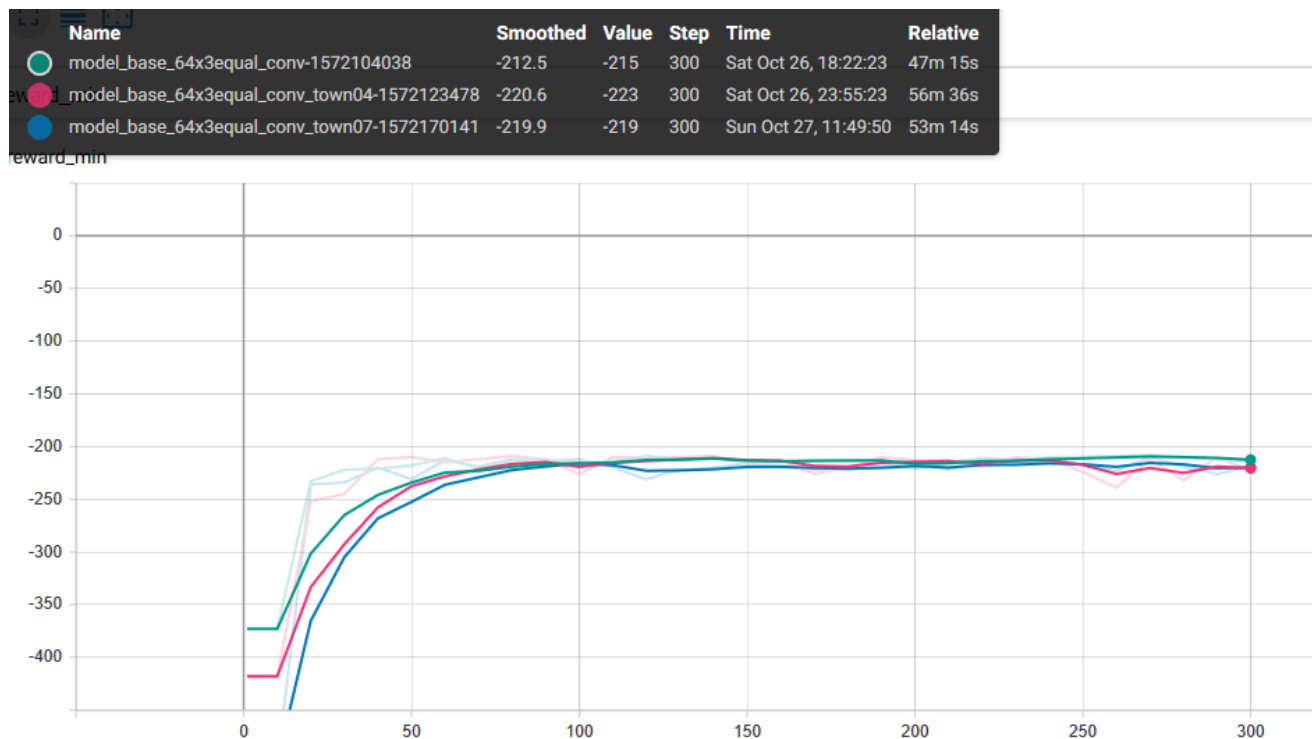


Figure 0.40. Minimum reward traces for all the tested models set with three identical convolutional layers with 64 neurons on Map02 (trace on green), on Map04 (trace on pink) and on Map07 (trace on dark blue).

A1.3 Final conclusion

This final decision is made taking into account the weights given prior to each map (low to map02, medium to map04 and high to map07).

By looking at the section of comparative between models, it seems that the easiest and simple the neural network, the better the performance of the model in terms of average reward. This is a general tendency observed in all three tested maps, although it is quite more evident on Map02 and Map04.

However, comparing and average of 6 models for each map is not a very efficient way of analyzing the data so moving on to the graphs on the comparative between maps it can be seen that on the simple models (2 conv layers) the difference on the performance of the model is quite noticeable. While on the low weight maps they perform quite good, the gap between the traces of them and the trace of high weight map is really large. This means that the simple models can not be scalable and therefore it will be very difficult for them to adapt to more complicated environments.

Moreover, on the models with 3 conv layers the difference on the average reward between different maps is really small, being the traces on all of them similar. Although this means that the efficiency of the model is reduced on 2 of the 3 maps and therefore, by statistics the global average is also reduced (in some cases), the performance on the best rated map (map07) is significantly better on all of them. Analogously, this means that having 3 conv layers can lead to a more stable performance as it is able to extract more facts from the input images on difficult tracks.

Reducing the selection to the models with 3 convolutional layers, a first comparison based on accuracy can be made. According to maximum values, the model with 64 neurons on each layer seems to be the best one. Also, it is the only one to maintain high accuracy values on all models throughout the training episodes. Having a look on reward values, the average rewards can be considered more or less similar on each of the three models. However, the maximum and minimum values are the distinctive fact.

From maximum and minimum values the best model is, again, the model with 64 neurons on each layers as the maximum values for each map converge to 0 (as happens also on the model with 16 neurons, but don't on the model with 32, which is worse) and the minimum values are the best among all other models (model with 16 neurons and model with 32 neurons).

From all of this data, the final decision is to train the main model with a neural network consisting of 3 convolutional layers and 64 neurons on each layer.