DEEP LEARNING SOLUTIONS FOR AUTONOMOUS VEHICLES: DOES INTERMITTENT CONTROL THEORY HAVE ANY PRACTICAL IMPACT ON REINFORCEMENT LEARNING?

A DISSERTATION SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY FOR THE DEGREE OF MASTER OF SCIENCE IN THE FACULTY OF SCIENCE AND ENGINEERING.

October 2020.

Authored by Ante Tomicic

Supervised by Ryan Cunningham

Department of Computing and Mathematics

Table of Contents

List of Figures

List of Tables

Abstract

In 2018., there were 1.35 million deaths from road accidents worldwide and another 50 million injuries in the year 2016. Traffic accidents are the 8th highest cause of death for people of all ages at 2.5% of total deaths, according to NHTSA, 94% of all road accidents are human fault. This project will investigate the potential value Intermittent Control Systems can provide to RL in the field of Autonomous vehicles. ICS can be found in biological systems and have shown to have promising results when applied in the field of mechanics, specifically when balancing an inverted pendulum. Applying Intermittent Control as a feedback control method could in theory improve the rate of learning and execution time in the field of Autonomous Driving Systems. This thesis will firstly be tested on navigating a simple maze problem. Secondly, by using the CARLA simulator, a simple DQN algorithm combined with Intermittent Control to see if there is any significant performance change.

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. It has been undertaken in accordance with the University research ethics standards, by the terms of permit number 24357.

Signed:

Date: 09/10/2020

Acknowledgements

I would like to thank the following people for their help and support during my time at Manchester Metropolitan University and the whole of my education.

Thank you to my excellent supervisor Dr. Ryan Cunningham for all the undeserved time, help, guidance, feedback and most of all patience throughout this project.

Thank you to all the faculty members, classmates and especially terrific lecturers. Thank you for your dedication, your effort and all the hard work. Your great teaching skills have set an example for the kind of a teacher I want to become.

Also, thank you to all my close friends who provided their help and their support. Thank you for your stimulating discussions, for all the long conversations, for shaping the course of my education, for the much needed distractions, all the advice, the adventures and the wicked humour.

Lastly, special thanks to my family. Thank you to my parents for your financial support, for your patience, your love, your care, your wise advice and taking care of me when I was sick. Thank you to my sister for your one of a kind companionship over the years, great conversations and bad jokes.

Abbreviations

AI – Artificial Intelligence

API – Application Programming Interface

DL – Deep Learning

DQN – Deep Q Network

FPS – Frames per Second

IC – Intermittent Control

IDE – Integrated Development Environment

NHTSA – National Highway Traffic Safety Administration

NN – Neural Network

RL – Reinforcement Learning

WHO – World Health Organisation

TD – Temporal Difference Learning

WHO – World Health Organisation

1 Introduction

This chapter will go through the basic premise of the project. Following a short explanation of the project, the first part will lay out the background of the project and then the background of the main underlying problem will be explained in part two called Problem Analysis. Furthermore, third part will go more in depth on aims and objectives of the project and the dissertation itself. The last part of the Introduction will consist of short overviews of each chapter in the dissertation followed by a short summary of the contents of the chapter.

1.1 Background

According to the WHO-s "Global status report on road safety 2018." there were 1.35 million deaths from road accidents worldwide and another 50 million injuries in the year 2016. Besides natural causes traffic accidents are now the leading cause of death for people between 5 and 29 years of age and coming up as the 8th cause of death for people of all ages at 2.5% of total deaths. [1] Also, according to NHTSA-s "Critical reasons for crashes investigated in the national motor vehicle crash causation survey" report 94% of all road accidents are human fault. [2]

That means that 1.269 million people die each year and 48 million people get injured every year due to human error when participating in traffic as a driver, passenger or a pedestrian. Besides accidents, problems such as traffic congestion, waste of time, caused stress, unnecessary energy consumption and CO2 pollution remain unresolved.

A possible solution to all the points mentioned above could be the development of autonomous vehicle systems with tools such as AI, DL and RL. Possible

benefits could be: reduction and mitigation of accidents, transporting the mobility- impaired, reduction of stress, congestion mitigation, road casualty reduction, decreased energy consumption, decreased pollution and increased productivity .[3] However, progress and development in the field of autonomous vehicles still presents a big challenge.

1.2 Problem analysis

The main problem in working with AI and DL for autonomous driving is the amount of computational power and the time required to train DL systems. [4][5][6] If that can be address it could lead to better road safety, fewer traffic incidents, less pollution, etc.

ICS are found deep inside the brain, within the Basal ganglia and it compliments RL within the brain [7]. That same idea was transferred from a biological system to a mechanical system of the inverted pendulum and proved to be successful.[8] Therefore, this project will investigate the potential value ICS can provide to RL. By applying Intermittent Control as a feedback control method which provides a spectrum of possibilities between continuous control systems and discrete control systems, allows for better control decisions, avoidance of limit cycles, etc.

1.3 Aims and Objectives

This project aims to contribute to research in autonomous vehicles, AI and RL, to advance the current state of autonomous driving vehicles. This project will try to successfully apply the theory of intermittent control systems (ICS) to reinforcement learning (RL) techniques. With that said there are three main goals of this project:

- Successfully explain the main ideas and principles of autonomous vehicles and AI such as RL, Q-Learning, NN and DQN along with providing a small example of a successful implementation,
- Modifying the already existing and the most basic RL method called DQN algorithm by adding in IC and applying it to an already known problem. Furthermore, to gather the results and determine the impact IC has on an implementation as simple as navigating a 10 by 10 maze,
- Lastly, the main aim of this project is to test IC combined with DQN in a bit more complex environment such as CARLA. CARLA simulator provides a quality interface which can have real world applications for autonomous vehicles.

1.4 Dissertation Chapter Overview

- The Introduction will go over the basic concepts that lay the foundation to the whole project. That will include the reasons why this project is important, the main problem that it is trying to solve as well as ultimate goals. Lastly it will provide a brief introduction to the structure of the dissertation.
- Literature review will provide a summary and a critical analysis of all the research papers, published articles, references and other used resources. Most relevant resources will be reviewed individually while some will be included in giving an overview on that specific part in the research field.
- Technical details contains a background on all the technologies used in the project. Those technologies are all software open source products such as the programming language, the IDE and the simulator.
- Reinforcement Learning will provide the first step into the world of AI and the first step towards understanding the thesis of this project. It goes over Markov decision processes and the high level details of RL.

- Q-Learning is one of the main principles of RL and of this project. The chapter will explain what Q-Learning is and how it works, what a Q-Table and the idea behind TD learning. It will also pride the first coded example of the maze problem.
- The chapter on Neural Networks will go a bit more in depth on what a NN is, how it is structured, what activation functions are and how a NN is used to learn behaviour.
- Deep Q-Networks chapter provides a brief insight into how Q-Learning and NN work together. It combines the principles which were presented in the last two chapters and presents them as a unified idea. Along with implementing a NN into a DQN algorithm in order to solve the maze problem.
- The chapter Intermittent Control provides an explanation to the key element of the thesis of this project and that is IC in RL systems. Also, it will provide an example of implementation, here the first part of the study is done. IC is applied to the DQN agent to solve the maze.
- DQN and IC applied CARLA, this chapter will briefly go over the specific methodology and the environment in which the agent will train to gather the results for the core experiment of this project. The environment is designed in CARLA simulator.
- Results and evaluation consists of the analysis of project aims and objectives and assessing if they were met. It will also go over the results and provide an analysis which will determine if the project was successful.
- Conclusion and future work will consist of a short summary of the whole dissertation laying down the main problems and the results achieved. Also it will go over some possibilities of future work that can further the research on DL, RL and ICS.

- References, Bibliography and Appendices will provide Harvard styled references to all the materials and resources used in the making of this project and the dissertation.

This chapter went over the critical parts of this dissertation, which includes the background on the state of traffic accident statistics, brief analysis of the main problem, provided aims and objectives of the project as well as giving a short overview of the structure of the report. Expanding on that, the next chapter will present a short review of the scientific literature used in the making of this report.

2 Literature Review

Following the introduction, this chapter will further expand on the main idea of this dissertation and how it fills the gap which exists in scientific research on the topic of Artificial Intelligence, Reinforcement Learning and Autonomous Vehicles.

The thesis of this project is the following, "Deep Learning solutions for autonomous vehicles: Does Intermittent Control Theory have any practical impact on reinforcement learning?". To be more specific, the main question this project answers is if IC can increase performance of RL systems. By performance it is meant learning time or execution time and learning rate. Each year, over a million people are killed in traffic accidents, even more injured,[1][2] not to mention the Co2 pollution, noise pollution, energy waste and all other negative effects that optimised traffic has on the environment[9]. The future of traffic lies with autonomous vehicles, the sooner that can be achieved the less irreversible damage will be done to the society and to the environment.

Research in the field of Reinforcement Learning and autonomous vehicles has made substantial progress over the course of the last decade. Reinforcement Learning is a well document and explained field of research,[10] as well as the field of autonomous vehicles[3]. However, there is still much progress that needs to be made to improve the state of autonomous vehicles. Another key field of research for this thesis is the field of control theory. As seen here [7], [11]; control theory is already applied in biological systems deep inside the brain. It was also applied to a mechanical system [8]. In those systems Intermittent Control is used instead of Continuous control to achieve faster, better and more stable results. There is an obstacle in AI and Deep Learning and that is the

amount of time and computational power required to train AI to successfully drive a vehicle.[10] [4][5][6] There is also an opportunity with Intermittent control to better the performance of the learning process of AI, a gap in the research filed which could allow for a significant increase in performance. The application of Intermittent Control has not yet been researched or applied to any AI or RL problem.

Autonomous vehicles are the future of transport, there is also a significant benefit for the humanity and for the environment in autonomous vehicles. The research so far has made good progress in exploring AI for autonomous vehicles, however, the learning process of AI is slow and the current state has still not progressed enough to make a significant change in the world. But based on the analysis of biological systems deep inside the human brain, a new control system was successfully applied to a mechanical problem. This project is going to explore the same application of Intermittent Control system over the standard Continuous Control system. Continuing on that, the following chapter will provide a first insight into the making of this project by going over the technical details and the tools used in the project.

3 Technical details

Following the introduction and the literature review which served a purpose of setting a foundation for the project, this chapter will be the first chapter to go in depth on the implementation, tools and work that made this project. All the technologies, programming language, tools will be presented here as well as the reasoning behind their selection.

3.1 Programming language

Regarding the programming languages that would be best for this particular implementation there were two options: C++ and Python. C++ can offer the quickest execution time and it conforms better to the Zero Overhead Principle than Python. However, Python proved to be a better choice because it is a language of higher abstraction level which can reduce the amount of code and writing therefore speed up development. Python provides more extensive libraries and materials online as well as detailed documentation, especially for the CARLA simulator. Python is also most commonly used in scientific literature on AI and RL.

3.2 Platform (IDE)

For the Integrated Development Environment (IDE), since Python was chosen as the programming language, Anaconda was chosen since it can provide a wide range of tools that would be useful for the project, those tools being: Jupyter Notebook and Spyder. Anaconda is an open-source distribution of Python mostly used for scientific computation, data science, machine learning and AI.

Jupyter Notebook proved to be the best choice for the first part of the project which was getting to know the libraries and the main principles behind NN and RL. Those libraries are: TensorFlow, Keras, NumPy and Matplotlib; those principles are: Q-Learning, RL environment, reward and agent. It also provides an easy way to handle multiple notebooks, examples, tutorials and learning exercises which would serve as a foundation for familiarisation and for the next step in the implementation.

For development beyond familiarisation with RL, Spyder proved to be really good when dealing with DQN implementation for the main part of the project. Spyder has a built in console and a great plot management tool which allowed easy monitoring and handling of all the errors, results, images and pictures. It also provides easy troubleshooting.

3.3 Simulator

As for the simulation, there were two main choices which were most widely used in scientific literature, those were CARLA and SUMO (Simulation of Urban Mobility). After testing and getting to know the simulators and their APIs, SUMO turned out to be the worse option due to its lack of decision control for the agent and strict rules. SUMO would be better suited for RL regarding traffic control and management, not for RL for autonomous vehicles. CARLA is an open-source simulator for autonomous driving research and it proved to be the better choice due to its superior vehicle control, better collision detection, camera management and easy data retrieval. Those features make the implementation of the environment, in which the agent will train, much easier to code. CARLA also has the feature to pause the simulation while the training of the NN was performed using its synchronous simulation mode. That also allowed for a fixed FPS which was set at 20. One downside to CARLA is the inability to speed up simulation time that ran in real time, which took up most of the run time.

3.4 Libraries

Python libraries used in this project were:

- TensorFlow 2.0
- Keras
- NumPy
- Matplotlib
- Time

TensorFlow is an open-source library mostly used for large numerical computations in the field of machine learning, DL and AI, created by Google Brain team. Along with Keras, which is an open-source NN library, they provide a high level API for dealing and working with NNs.

NumPy is also an open-source Python library used for scientific and statistical research, but it mostly deals with arrays which are curtail when working with a NN. It also proved to be quick and useful when dealing with statistical analysis of the results.

Matplotlib is an open-source library for creating visualisations in Python, it also serves as a mathematical extension of NumPy. For this project, it was mostly used for plotting the maze environment and graphs of reward arrays, loss arrays and the number of steps per episode arrays.

Lastly, Time is a built in Python module which provides functions and methods for dealing with time. Time was used for measuring the execution time of each training cycle.

All these libraries were chosen because they are almost exclusively used in scientific literature; because they are reliable, well documented and easy to use; and also because of their well-deserved popularity. This chapter went over all the tools and technologies used for the implementation of this project.

Main takeaways from this chapter are the programming language, the platform, the simulator and all the libraries that were used in the project, as well as short summaries and justifications for their choice. The following chapter will continue on this one with an introduction to RL, the first and main principle behind the project and the thesis.

4 Reinforcement Learning

Following a breakdown of the tools and technologies used in the development of this project in the previous chapter, this chapter will move forward with the first steps to explain the main principles behind this project, that is RL. Along with RL, other ideas such as Markov decision process, reward policies, Temporal Difference learning and the idea of episodes will be explained.

4.1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) is one of three basic approaches to Machine Learning (ML), along with Supervised Learning and Unsupervised Learning. RL is a process of learning where an agent, or an actor, that is learning is trying to navigate a given environment. The agent performs an action, that action results in a change of state of the agent and/or the environment and for that action an agent is given a reward based on the outcome. With that reward, the agent tries to learn and to maximise the reward. In the case of autonomous vehicles, the agent is the car and the environment would be the world in which it is driving. If the agent reaches a given goal, for example driving from point A to point B, it is given a positive reward to further facilitate that behaviour. If the agent crashes or fails to reach the goal, it is given a negative reward as to stop the set of action that led to that outcome from happening again.

The process of RL can be described using an idea from dynamic systems theory called Markov decision process. Figure 4.1.1 shows the interactions between the agent and the environment in the Markov decision process. The agent is palced into the environment and has a state S_t that is attached to it. The agent choses an action A_t and performs it, that action affects the environment, the environment then gives a reward R_{t+1} to the agent and changes the state of the agent S_{t+1}. That course of events is known as Markov decision process. [10]

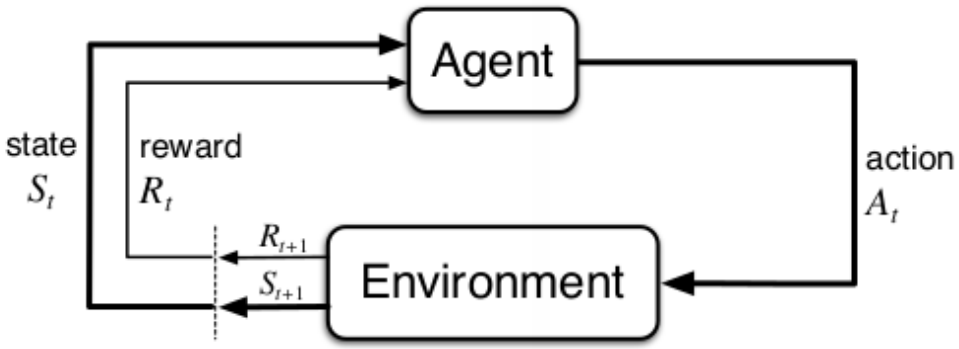

Figure 4. 1. Markov decision process [10]

Markov decision process is then repeated over and over again until the goal has been reached, until the agent got into a terminal state or the environment decided that the process is over. After each process the agent is supposed to learn which actions resulted in which rewards, the environment is reset and the Markov decision process starts again. The main idea behind RL is that eventually, after a number of Markov decision processes, the agent will learn which states are desirable and which states aren't desirable based on the reward it gets from the environment. Therefore, the agent will be able to navigate the environment successfully and get the highest reward possible. [10]

4.2 States, actions, rewards, and goals

4.2.1 States

The state is a feature that is tied to an agent which is situated in a certain environment. It is also an interface between the agent and the environment in so that it provides a connection between them. [10] When an action is performed the environment will change the state of the agent in a way that is appropriate for that environment. For example, in a maze which is represented by an array size of (n_x, n_y) = [10, 10], meaning it has 10*10=100 "squares" on which the agent can "step on". That means there are a 100 states for the agent to find itself in. The state of an agent would then be defined as a position in the array, for

example (x, y) = [4, 7]. If the agent then selects an action to go down in the maze then the environment will move the agent one state or "square" down. The state then would be (x, y) = [5, 7].

States can vary depending on the environment, it could be a location with one [x], two [x, y], three [x, y, z] or k [n_1, n_2, …., n_k] dimensions. It could be also represented as a distance or as an image from a camera or a radar or a lidar in the case of autonomous vehicles, or in most high-end cases a combination of all the cases mentioned above.

4.2.2 Actions

Actions, in the case of RL, are the decisions that the agent is allowed to make in a given environment to navigate the environment. Those actions are exclusively determined by the environment itself. For example, in the case of a maze those actions would be: move up, move right, move down, move left, and could even be to stay in place. In the case of autonomous vehicles, those actions could be: the amount of throttle, amount of brake, gear, steering angle, handbrake, lights etc. In more complex implementations each of those action would be represented as a continuous variable while in simpler ones those actions can be represented by a small number of scalar or boolean variables which would encompass the scope of the action.

4.2.3 Rewards

After each time step, each action taken, each change of state the agent is given a reward from the environment. If the agent can reach the goal or a checkpoint the reward will be positive, if the agent finds itself in a terminal state then it gets a negative reward. Usually when the agent reaches the most desired state, or in the case of autonomous vehicles, the "finish line" the reward is the biggest,

if it reaches a terminal state or it “crashes" the reward is the smallest. The reward itself is only a simple number, higher than 0 if the action is desirable, and lower than 0 if the action is not desirable. That reward is later used in calculating the quality values or Q-values of each state that the agent finds itself in, which is the basis for Q-Learning and later on DQN.

4.2.4 Goals

Goals in RL are a simple term, the goal of an agent is to maximise the total reward it receives during the course of the Markov decision process. That means maximising not the immediate reward, but the cumulative reward in the long run. That principle of a goal as maximizing the cumulative reward is one of the most distinctive features of RL which sets it apart from other approaches to Machine Learning. [10]

4.3 Episodes

Episode is best explained as a single iteration of Markov decision process. Each episode could be descried as plays of a game or trips through a maze. Every episode ends in a terminal state which would be reaching the gaol or causing a crash in the case of autonomous vehicles or hitting a wall in the case of a maze. That is followed by the resetting of the environment and returning the agent to the starting state, therefore each episode is independent for the previous one. The process of RL relies on the continuous repetition of that process, which means that RL consists of a number of episodes in which the agent is trying to learn a certain behaviour in the environment. As soon as the agent learns it, not perfectly but rather to the degree that is acceptable, the RL process is over. [10]

4.4 Hyperparameters

In RL there are 5 curtail parameters that determine how successful the learning process will be, those are:

- number of episodes
- epsilon
- epsilon decay
- discount factor
- learning rate

Number of episodes is an integer number which determines how many episodes the agent will be able to train. Each episode the environment is reset and the agent returned to its starting state and starts the Markov decision process over.

Epsilon is a parameter that holds a value between 0 and 1. It determines the relationship between exploration and exploitation. In order for the agent to be able to successfully navigate the environment it needs to be able to make random decision thereby exploring the environment. That is called exploration, while exploitation uses the knowledge which agent gathered while exploring, to make informed decision on which action to take. That gathered knowledge is commonly stored in a Q-Table or in a NN, which will be explained in the following chapters. If epsilon is always 1 the agent will always be exploring and making random decision so it won't be able to exploit the knowledge it has gathered. If the epsilon is always 0 the agent will always make decisions based on what it has previously learned, however that will significantly slow down the learning process and may even result in unsuccessful training. Epsilon is most commonly set to 0.99 before the first episode, and gets decreased by epsilon decay after each episode.

Epsilon decay is also a parameter with a value between 0 and 1. The main purpose of epsilon decay is to decrease epsilon after each episode. Therefore, epsilon decay dictates the relationship between exploration and exploitation. At the start of the learning process epsilon is high and that gives the agent the opportunity to explore the environment. As the episodes pass by epsilon decreases and that gives the agent the ability to make informed decisions based on the knowledge it gathered while exploring.

Discount factor and learning rate will be explained in the next chapter along with temporal difference because they go together and are therefore easier to explain and understand.

This chapter gave a brief introduction to basic concepts of RL. It went over the main ideas behind RL, which is Markov decision process and its subsequent components. Those components being the states, actions, rewards and goals. It also provided a brief insight in what hyperparameters are and what purpose they serve. Following that, section named episodes is supposed to give a big picture approach and put all those pieces together to give an overall understanding of RL works. The next chapter will feature first code snippets and real implementation figures. It will also try to explain what is Q-Learning and why it is critical to understand it before we can move on to Neural Networks, DQNs and the main body of the project.

5 Q-Learning

The previous chapter on RL is supposed to serve as a foundation for understanding how the main process of RL and thereby Q-Learning works. Following that, this chapter will explain what Q-Learning is, what a Q-Table is, how the agent learns to decide what the best action to take in a given state is. It will also go over what hyperparameters are and why they are important in the process of learning. Lastly, we will go over one examples which will lay the foundation of code on which we will build upon in later chapters.

5.1 Introduction to Q-Learning

Q-Learning is an algorithm used in RL that relies on a Q-Table which stores the values of all actions in every possible state. It also relies on TD as a way of updating the values in the Q-Table while the process of learning is taking place.

Firstly we will dive into exactly what a Q-Table is, then explain what TD is and then bring it together.

5.2 Q-Table

The Q-Table is a multidimensional array which contains a value for each combination of action and state. This is best explained using an example, the following example is an implementation of a Q-Table for a maze. Figure 5.1 shows a maze that will be used for the explanation. The maze is the size of (n_x, n_y) = [10, 10], which means the maze has 100 states all together. Figure 5.2 shows how the Q-Table for this maze would look like when there are 4 actions the agent can take. Moving up, moving right, moving down and moving left.

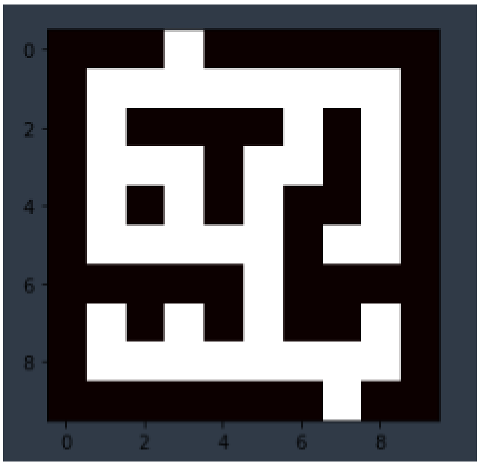

Figure 5. 1 Preview of a 10 by 10 maze

```
[[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]

 [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]]
```

Figure 5. 2 Q-Table of a maze size 10 by 10 with 4 actions available

In the case of this maze, the Q-Table is a 3 dimensional array; width of 10 that represents each column in the maze, height of 10 which represents each row in the maze and depth of 4 for each action.

During the course of the learning process, the agent will chose an action and transition from state to state, as the agent is navigating the maze the Q-Table will be updated and will store the values for corresponding state-action pairs every time the agent takes an action and thereby changes its state. Those values are called Q-Values and are unique for each state-action pair.

5.3 Temporal difference learning

Temporal difference learning (TD) is a central RL idea which allows the agent to learn directly from experiences gained through interacting with the environment instead of having a model of the environment's dynamics. TD is simply represented as a mathematical formula which uses the Bellman Equation to update the Q-Table. We are using one of the more simple TD methods called the tabular TD, it is a special case of more general stochastic approximation methods used to estimate the state-action value function of a finite Markov decision process. Before going into the TD formula and the Bellman Equation we need to break down the remaining two hyperparameters, the discount factor and the learning rate which are used in TD. [10]

5.3.1 Discount factor

Discount factor is a parameter between 0 and 1, it determines how much the agent should value distant future rewards versus the reward in the immediate future. It slightly reduces the value of all subsequent distant future rewards and allows the agent to get a bigger reward the closer it gets to the end goal.

5.3.2 Learning rate

Learning rate is also a parameter between which determines how much the newly gained information will rewrite the old information. To be more specific,

it determines what percentage of the calculated TD will be added to the old Q-Value when updating the Q-Table with a new Q-Value.

5.3.3 Bellman Equation

The Bellman Equation is the main equation used when updating the Q-Table as shown in equation (5.1),

$$V(S_t) = V(S_t) + \alpha * TD \tag{5.1}$$

where:

- S_t is the old state in which the agent was in and is being updated,
- $V(S_t)$ is the Q-Value which is contained in the Q-Table and will be updated,
- α is the learning rate,
- TD is the Temporal Difference value calculated beforehand.

As shown in the equation above, the new Q-Value which will be stored into the Q-Table is a sum of the old Q-Value and the calculated TD decreased by the learning rate.

5.3.4 Temporal Difference Equation

Finally, the last step before tying up the Q-Learning and implementing it into an algorithm is the equation for TD which is presented by the equation (5.2).

$$TD = R_{t+1} + \gamma * max(V(S_{t+1})) - V(S_t) \tag{5.2}$$

Where:

- S_t is the state in which the agent was before it made the new step and is the state which will be updated,

- S_{t+1} is the new state in which the agent has found itself in,
- TD is the Temporal Difference,
- R_{t+1} is the reward given to the agent in the new state,
- γ is the discount factor,
- $\max(V(S_{t+1}))$ is the maximum Q-Value, or better said the maximum reward, which can be gained from the new state S_{t+1},
- $V(S_t)$ is the Q-Value of the previously executed action A_t and the old state S_t.

5.4 Q-Learning algorithm

Q-Learning is essentially an algorithm and here we will break down how it works. The pseudo code for the algorithm is show in the Figure 5.3.

- The environment is reset and the agent given its starting state S,
- The agent chooses an action A determined by the π policy, meaning with regards to the epsilon hyperparameter,
- The agent performs the chosen action A, receives the reward R and is now in a new state S',
- Bellman equation is implemented and TD is calculated and the Q-Table updated,
- The new state S' becomes the current state S,
- The loop is repeated until terminal state is reached.
- Episode is finished, and the whole process repeats until all episodes are completed.

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 5. 3 Q-Learning algorithm with TD [10]

5.5 Solving the maze problem with Q-Learning

This section will briefly go over the implementation of the Q-Learning algorithm with the example of the maze problem. The maze that will be solved is represented by Figure 5.1, the corresponding Q-Table is represented by Figure 5.2. Hyperparameters that were used in the example are shown in the figure 5.4 and the code snippet for the Q-Learning algorithm is shown in the Figure 5.5. The results of the learning will be provided in the Results subsection.

5.5.1 Rewards

The rewards are provided to the agent in the following way:

- If the agent reaches the goal it is given a reward of 100,
- If the agent steps on the black square, meaning the wall, it gets a negative reward of -100.
- Lastly, for each step on the white square the rewards is -1, that is because the agent is motivated to maximize the reward and therefore make the least amount of steps resulting in the shortest path.

5.5.2 Implementation

```
1  #define training parameters
2  discount_factor = 0.9
3  learning_rate = 0.9
4  number_of_episodes = 2000
5  epsilon = 0.99
6  epsilon_decay = 0.999
7
8  min_epsilon = 0.01
```

Figure 5. 4 Hyperparameters used in the maze example

Explanation, line by line:

- Line 24: loop for each episode is announced,
- Lines 25 and 26: the environment is reset and the agent given its starting state, described by row and column,
- Line 28: loop for a single Markov decision process is announced,
- Line 29: the agent chooses an action,
- Line 30: the current state becomes the old state and is saved,
- Line 31: the agent performs the chosen action,
- Line 32: the agent gets the reward,
- Line 34: the Q-Value of the old state and the action which was choses is retrieved,
- Line 35: TD is calculated,
- Line 36: the Q-Value of the old state is updated using the Bellman equation,
- Line 37: the new Q-Value is stored into the Q-Table,
- The while loop repeats until terminal state is reached,

- Lines 40 and 41: decrease the epsilon up until the minimum value determined by the min_epsilon,
- The for loop repeats until all episodes are finished.

```
24 for episode in range(number_of_episodes):
25     row_index = 0
26     column_index = 3
27 
28     while not is_terminal_state(row_index, column_index):
29         action_index = get_next_action(row_index, column_index, epsilon)
30         old_row_index, old_column_index = row_index, column_index
31         row_index, column_index = get_next_location(old_row_index, old_column_index, action_index)
32         reward = rewards[row_index, column_index]
33 
34         old_q_value = q_values[old_row_index, old_column_index, action_index]
35         temporal_difference = reward + (discount_factor * np.max(q_values[row_index, column_index])) - old_q_value
36         new_q_value = old_q_value + (learning_rate * temporal_difference)
37         q_values[old_row_index, old_column_index, action_index] = new_q_value
38 
39 
40     if epsilon > min_epsilon:
41         epsilon *= epsilon_decay
```

Figure 5. 5 Q-Learning algorithm in the example of a maze

5.5.3 Results

The results and the performance can be monitored by a number of different metrics, in this example the main metrics will be the following:

- Firstly, the shortest path, indicating the success of the Q-Learning algorithm, Figure 5.6,
- Secondly, the graph containing rewards per episode, Figure 5.7,
- Thirdly, the graph showing the mean Q-Value of each episode, Figure 5.8,
- Lastly, the graph showing the average number of steps for each episode, Figure 5.9.

Also one more graph showing the minimum, maximum and the average reward will be provided in the Figure 5.10.

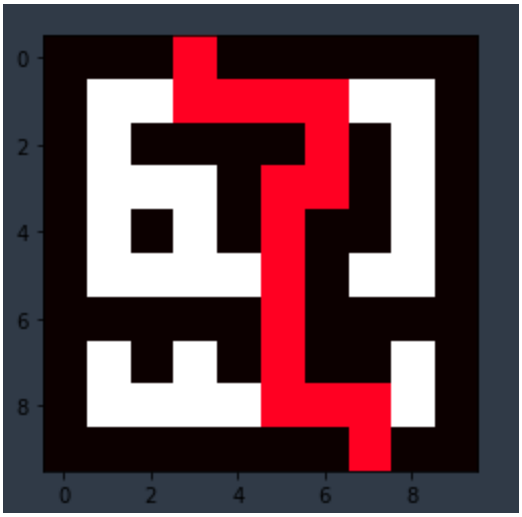

Figure 5. 6 The shortest path of Q-Learning for the maze example

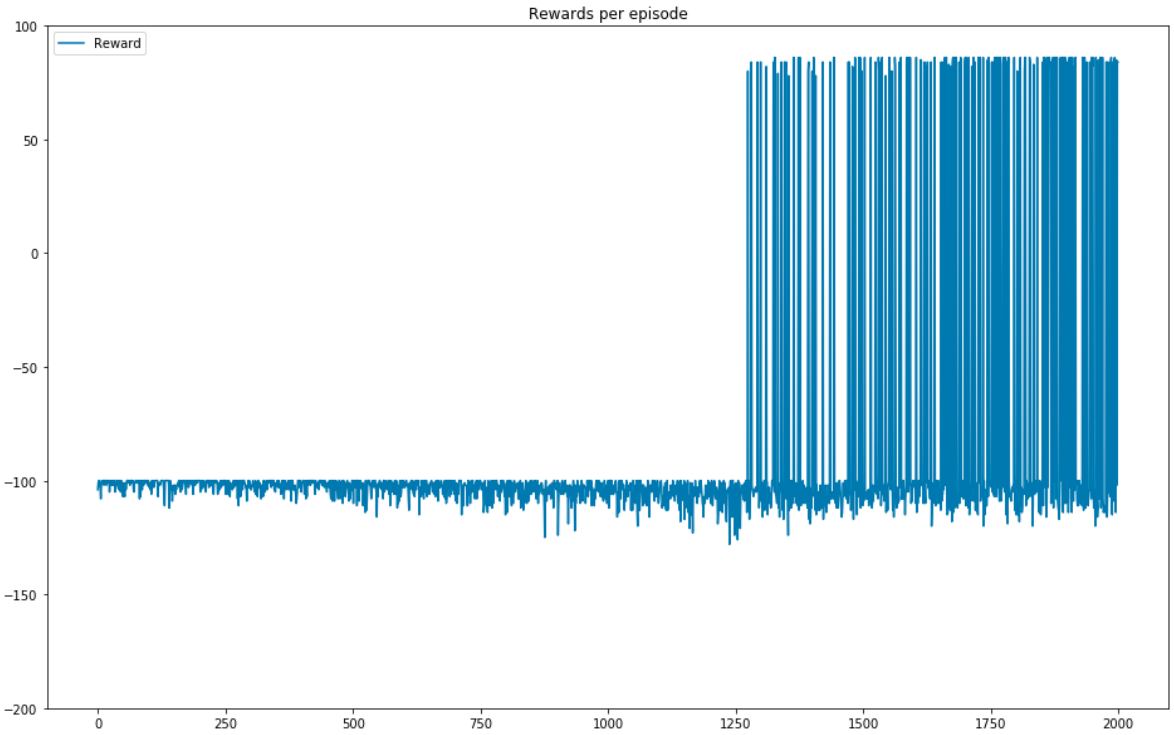

Figure 5. 7 Rewards per episode of Q-Learning for the maze example

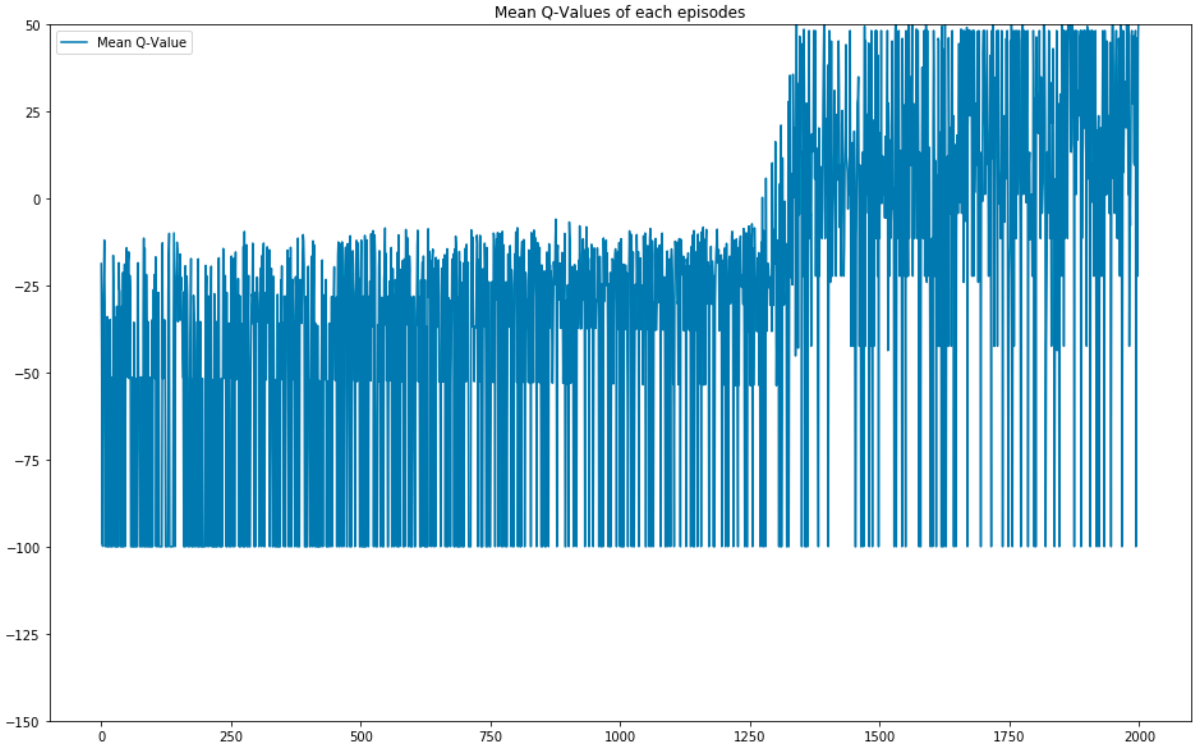

Figure 5. 8 Mean Q-Values per episode of Q-Learning for the maze example

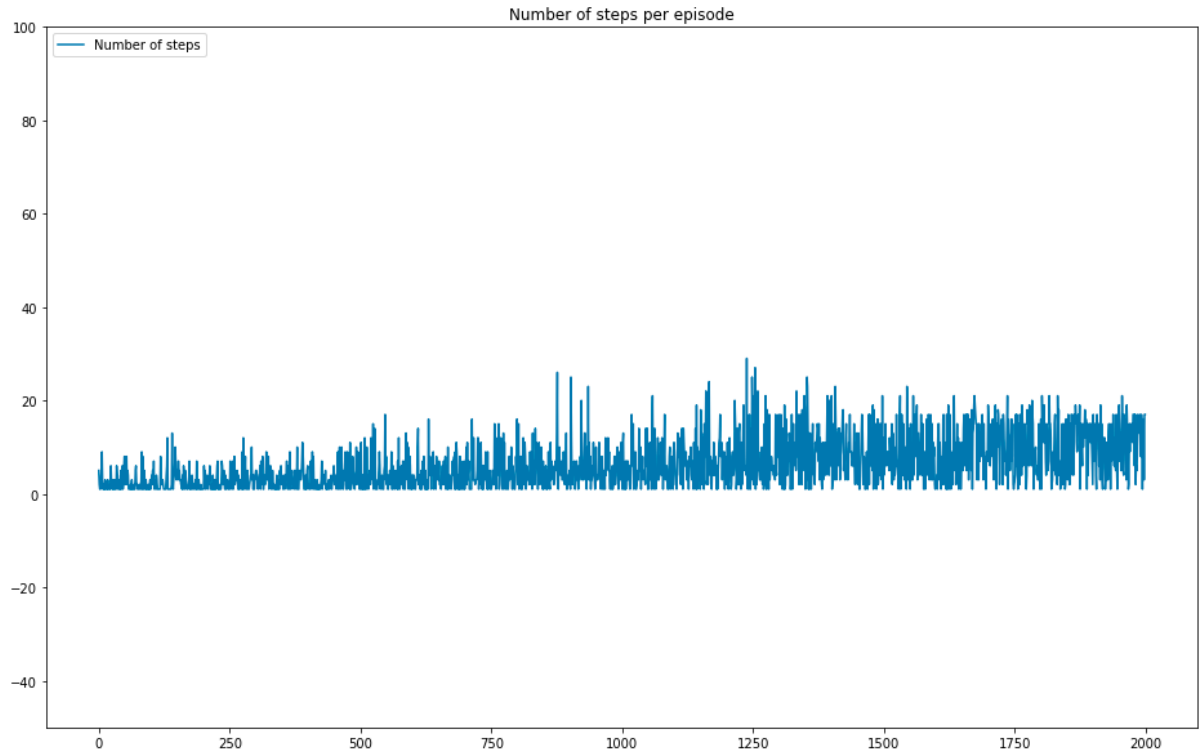

Figure 5. 9 Number of steps per episode of Q-Learning for the maze example

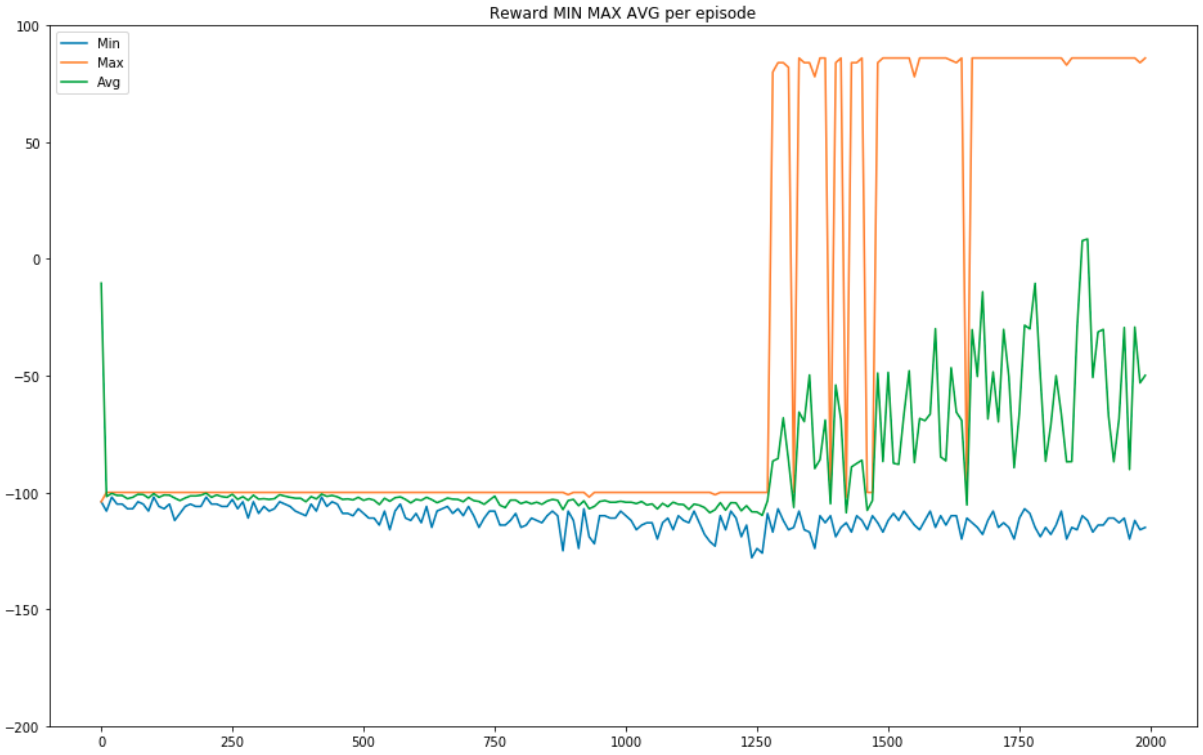

Figure 5. 10 Minimum, maximum and average reward for each episode of Q-Learning for the maze example

Finally, the Q-Table for this example is shown in figure 5.11. Instead of only zeroes, now it contains a Q-value for each state-action pair, indicating the quality of any action in every state.

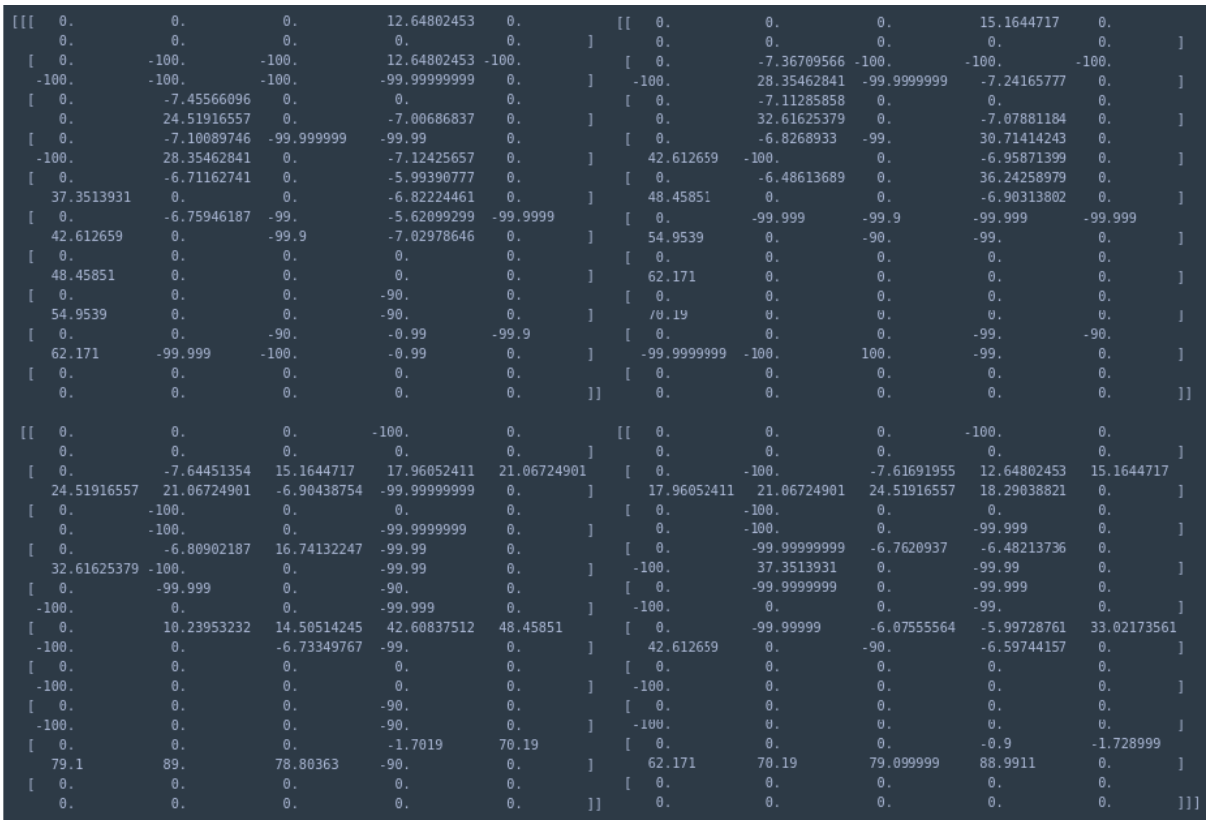

Figure 5. 11 Resulting Q-Table for the maze example

Code for the example above is in the Appendix 1, a Jupyter Notebook. The execution time is 1.23 seconds, that shows that Q-Learning as a RL method is really fast, however a big downfall of Q-Learning is that it gets exponentially slower the bigger the Q-Table. That means that Q-Learning is good for small sets of state-action pairs but proves to be slow and insufficient when trying to train an agent in a complex environment.

This chapter went over what Q-Learning is, how it works and what are the mayor components behind it as well as provided a working example along with code snippets and results. Moving forward we will be talking about NNs, what role they have in dealing with the downfalls of Q-Learning and also provide a solution to the same maze. This time the function of the Q-Table will be replaced by the NN.

6 Neural Networks

The previous chapter explained touched on the first step towards understanding the main thesis of this dissertation. That is Q-learning, now followed by an explanation of NNs which will be a crucial component of the project going forward. In this chapter we will go over the structure of a NN, briefly give an explanation of how it works on the most basic level.

In the example from the previous chapter we used Q-Table to store Q-Values to determine the quality of any action in every state. It was mentioned that Q-Learning performs worse and worse the more state-action pairs there are. That means that for autonomous vehicles which can have a lot of actions and a lot more states, Q-Learning will not be efficient. Therefore, NN will serve as function approximator, a closed system that appears rather simple when looking at it from the highest level of abstraction.

At the highest level of abstraction, a NN can be described as a black box, the agent provides an input, input being the state. The NN calculates Q-Values for every action regarding that single state and then returns them as an output.

6.1 Structure of a Neural Network

A NN is best described as a complex data structure. The main component, a single unit of the data structure is a node, nodes come together to form layers and the layers together form the NN. There is a slight difference between the layers, the first layer in the NN is called the Input layer because, the last layer is called the Output layer. Every layer in between is called a Hidden layer, and the main feature of a NN is that every node in single layer is connected to every node in the previous layer and the following layer. The meaning behind

connected nodes will be explained in the section Weights and Biases. Figure 6.1 shows a NN with 8 nodes in the input layer, 3 hidden layers with each containing 9 nodes and an output layer with 4 nodes.

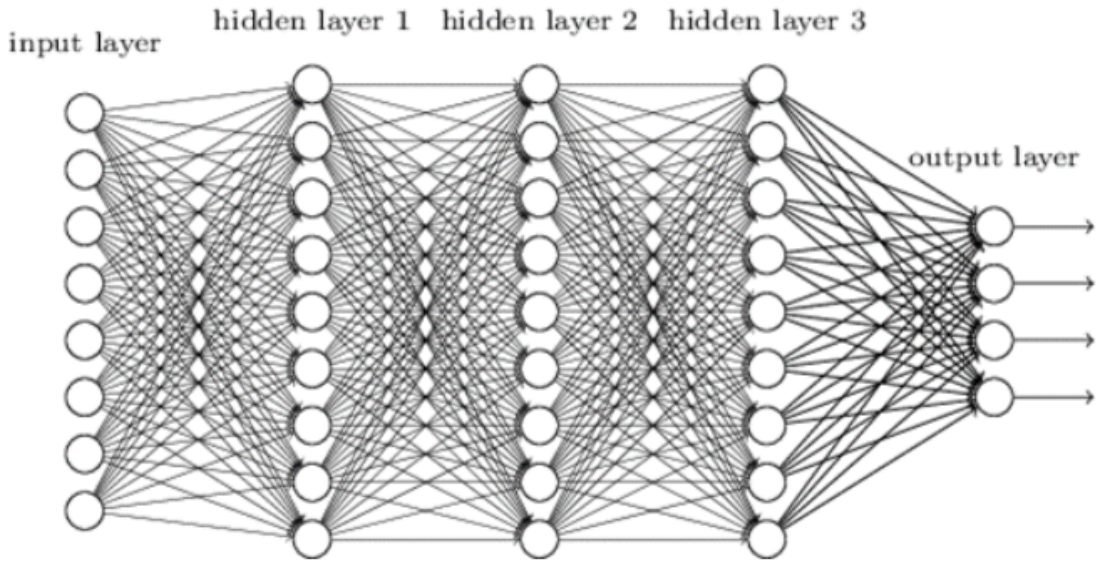

Figure 6. 1 A visualised example of a NN

6.1.1 Input layer

The input layer in a NN is always used to pass a state to a NN. That state can vary depending on the context. For example, in the case of the maze from the previous chapter, the state of the agent was described by a row and a column in which the agent was located. That would mean that a NN for that maze will have 2 nodes in the input layer, meaning 2 inputs. Those inputs would range from 0 to 9, including 9, since there were 10 rows and 10 columns. In that example, a NN doesn't provide any benefit over a Q-Table, in fact the NN is much slower. However, in the case of autonomous vehicles there are a number of different parameters which can be included as a state for a vehicle. For example, the inputs could be following: steering angel, amount of gas, revolutions per minute, amount of brake, amount of handbrake, gear, engine temperature, oil level, water level, location, position on the road, distance to

other vehicles, obstacle recognition, pedestrian recognition, lane keeping, radar; lidar and camera information regarding the surroundings, weather information… There are a lot more inputs in autonomous vehicles and many of them are not an integer value, but rather big arrays and continuous non discrete values. When all of those are combined they form close to infinite number of states.

Nodes in the input layer only contain values for each dimension of the state and aren't interconnected with the first hidden layer. The role of the input layer is to pass the state to the first hidden layer, that's where the real computation happens.

6.1.2 Outputs

The output layer in a NN is always used to return the Q-Value of all available actions. The output layer, in the maze example from the previous chapter would contain 4 nodes because there are 4 actions that the agent can choose from. In the example of autonomous vehicles those actions could be: turn the wheel, brake, shift gear, handbrake, windshield wipers, change throttle application, etc. Outputs can be finite or infinite and discrete or continuous values, in the case of the maze they are values anywhere between 0 and 1.

When making a Q-Table of a complex system as mentioned before, the amount of state-action pairs can be infinite, and therefore big multidimensional arrays are slow. That is why a NN is needed, it can predict the Q-Values for all state-action pairs no matter how many of them there are.

6.1.3 Hidden layers

The last and the main component of a NN are hidden layers. A NN must have one input layer and one output layer, but the number of hidden layers is not

specifically defined. A different amount of hidden layers and nodes within them can be considered for each specific application. If a NN has no hidden layers no computation will be performed and the inputs will be passed to the outputs directly, therefore the NN can't store any gained knowledge. In the example of the maze there are 5 hidden layers, each with 10 nodes.

6.2 Nodes, weighs and biases

Nodes are the building blocks of a NN. The execution time of making predictions and updating the NN depends on the number of nodes there are. The bigger the NN the more time it takes to process information and is therefore less efficient. The smaller the NN can be to perform the same function the better, since the decrease in size decreases the number of nodes. However, larger NNs come with the ability to learn more complex tasks and behaviours. For each application the NN should be optimal, meaning the smallest it can while still having the ability to learn the required behaviour.

Every node has weights and biases. Weights are numbers which are assigned to each input value that nodes in the hidden layer receive. The biases are also a simple number assigned to each node, meaning each node has only one bias. Here is why that is important.

Each node in the first hidden layer is connected to each node from the input layer and each node in the first layer gets the value of every input node. That means that each node gets all the values that describe a given state S. Furthermore, every node multiplies the every weight with every input it gets, the node then performs a sum operation on all those multiplications. Following that the node adds a bias to that sum, which could be negative or positive and then passes is through an activation function. That end value will then be passed on to the second hidden layer. Then the second layer does the same, so

does the third and so on, until the output layer. Equation (6.1) shows how the output of a node is calculated.

$$v = \alpha(w_1 * a_1 + w_2 * a_2 + w_3 * a_3 + ... + w_n * a_n + b) \quad (6.1)$$

Where:

- v is the output value of the node,
- α is the activation function,
- i=1,2,3,…,n is the number of nodes in the previous layer,
- w is the weight for each,
- a is the output value of each node from the previous layer,
- b is the bias for the node.

6.3 Activation functions

Activation functions are simple functions which, depending on an input provide an output. They can be easily understood only by showing their function graphs. We will go over some activation functions that were used in this project. Note that there is no best activation function, some are better than others in particular cases and vice versa. The choice of activation function depends only on the specific case in which it is applied, each activation function has its own advantages and disadvantages.

X axis represents the input and the y axis represents the output of the function.

6.3.1 Linear

Figure 6.2 shows the Linear activation function, equation (6.2) shows the function itself.

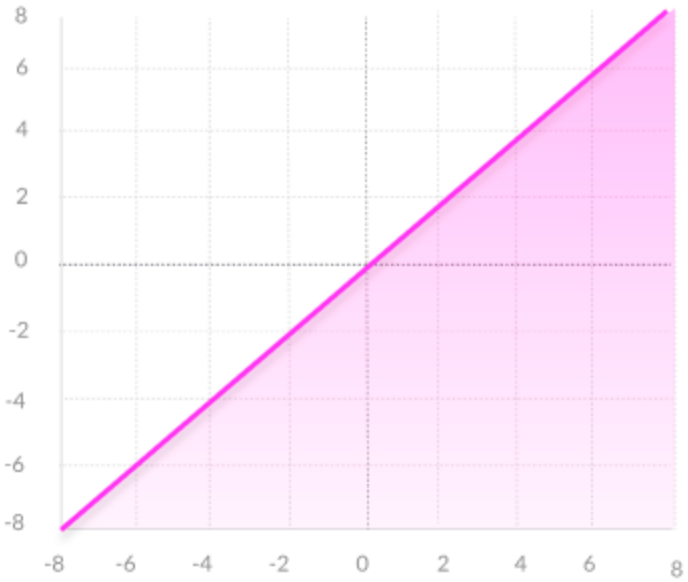

Figure 6. 2 Linear activation function

$$f(x) = x \tag{6.2}$$

6.3.2 Rectified Linear Unit

Figure 6.3 shows the Rectified Linear Unit (ReLU) activation function, equation (6.3) shows the function itself.

$$f(x) = max(0, x) \tag{6.3}$$

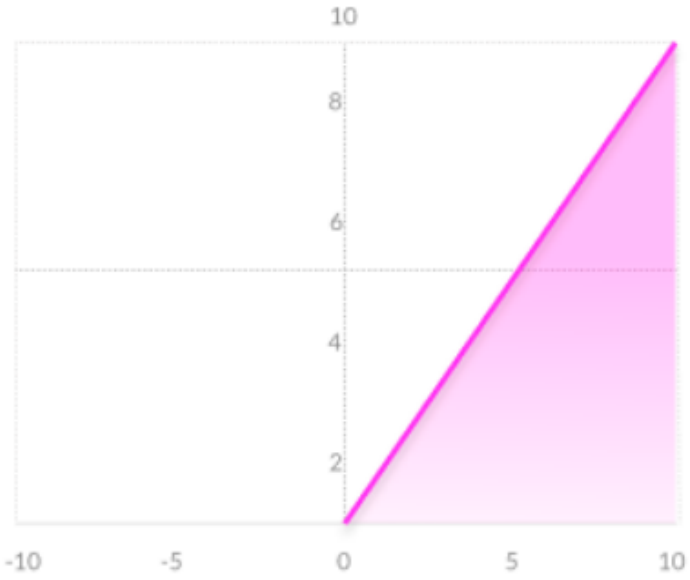

Figure 6. 3 Rectified Linear Unit activation function

6.3.3 Swish

Figure 6.4 shows the Swish activation function.

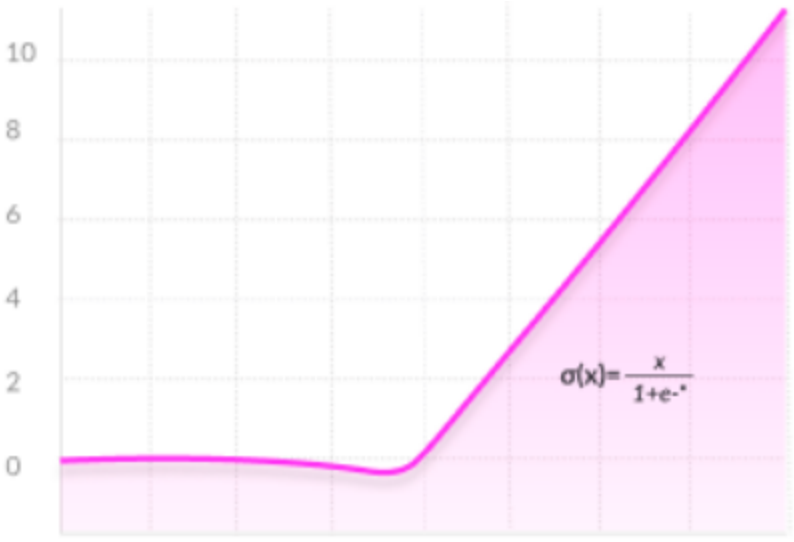

Figure 6. 4 Swish activation function

$$f(x) = \frac{x}{1 + e^{-x}} \tag{6.4}$$

6.4 How a Neural Network works?

NN perform two main functions, that is predicting output values for a given input and updating the weights and biases to better fit the desired output values.

6.4.1 Predicting

The main purpose of the NN is to predict the output values based on some input, meaning it provides the quality values for actions based on the state that the agent is in. The key to understanding how NN make predictions is in the fact that each layers presents an input to the following layer. Meaning, input layers directly gives the values to the first hidden layer, the first hidden layer performs calculations and provides the second hidden layer with his outputs and so on, until the output layer. Each layer consist of nodes, every node receives an output value from every node in the previous layer as input. Then the node multiplies every input with corresponding weights, sums all those multiplications, adds the bias to the sum and passes it through the activation function. That end value is the output of the node and is then passed on to every single node in the next layer. Eventually, the output layer will get the inputs from the last hidden layer, perform the same operations and get the quality values of each action for the given state. That process of prediction is also called forward propagation, it is called that because the flow of information and calculations goes from the first layer to the last layer.

6.4.2 Updating

Updating the NN consists of two main components: loss function and backpropagation. Loss function is the function that calculates the difference between the predicted value and the value the NN was supposed to be predicted. That function can be: mean squared error, binary crossentropy, sparse categorical crossentropy, etc.

The loss function used in this project was only mean squared error. In the example of the maze the loss is calculated in the following way. There are 4 outputs, and in this example the only decision that should be made is to go down. So the target output is [1, 0, 0, 0], but the NN outputs [0.15, 0.0, 0.0, 1]. Then the total loss is calculated as the sum of square differences of each predicted action and each target action. A shown in the equation (6.5).

$$Loss = (0.15 - 1)^2 + (0 - 0)^2 + (0 - 0)^2 + (1 - 0)^2 \tag{6.5}$$

Loss function is the foundation of backpropagation. Backpropagation is the process of updating the weights and biases of the whole NN. It flows in reverse from forward propagation, meaning from the outputs to the last hidden layer then to the second last hidden layer and so on until the first hidden layer. The process goes as follows:

- The output layers prediction values are changed to target values,
- Then every node in the last hidden layer can make a change to their weights and biases to match their output value to the target values of the output layer,
- When that is completed, the second to last hidden layer now has the target values in the last hidden layer,
- The second hidden layer also changes its weights and biases to match the targets in the last hidden layer,
- The process propagates to the input layer where it stops.

The process of updating the NN is called backpropagation because the flow of information and calculations goes from the last layer to the first one.

Figure 6.5 shows the code snippet for implementing a NN with:

- Line 1: Input layer with 2 inputs,
- Lines 2,3,4, and 5: 6 hidden layers, each with 10 nodes and ReLU activation function,
- Line 6: Output layers with 4 outputs, and Linear activation function,
- Line 7: Creates the NN.

```
1 inputs = tf.keras.layers.Input(shape=(2))
2 skip = tf.keras.layers.Dense(10,activation='relu')(inputs)
3 for i in range(5):
4     layer = tf.keras.layers.Dense(10,activation='relu')(skip)
5     skip = tf.keras.layers.Add()([skip,layer])
6 outputs = tf.keras.layers.Dense(4,activation='linear')(skip)
7 model = tf.keras.Model(inputs,outputs)
```

Figure 6. 5 Implementation of a NN using TensorFlow and Keras

6.4.3 Stochastic Gradient Descent

A NN uses a method called Stochastic Gradient Descent, which is an iterative method that optimizes an objective function, it is most commonly used in high-dimensional optimization problems. It can reduce computational requirements, iterate fast in exchange for a lower convergence rate. In the case of NN it is used to reduce the loss and find the optimal weights and biases for the given application. Since everything in the universe can be described as complex polynomial function with multiple variables, a NN using Stochastic Gradient Descent serves as a function approximator and tries to fit the NN to that polynomial function.

This chapter gave a brief overview on what NN are, what their components are, how they work and provided the reader with a code example along with a short explanation. The main takeaway from this chapter should be generalized view on how a NN works but more importantly what it is supposed to do. In the next chapter we will go deeper into what is DQN and solve the example of the maze using a NN in the DQN.

7 Deep Q-Networks

This previous chapter explained the most important and high level concepts of NN. By now the reader is expected to have grasped the basic principles of RL, Q-learning and NN because all those concepts are important in moving forward with understanding the project. In this chapter we will try to briefly explain what a DQN is, the role a NN has in it, solve the example of the maze using DQN and briefly go over the results.

7.1 Introduction to Deep Q-Networks

Deep Q-Networks or Deep Q-Learning is a representation of successfully combined Deep NN and RL. The main principles behind DQN have their roots in Q-Learning, DQN is also based on Markov decision processes. However, there is a key difference, DQN doesn't use a Q-Table for storing Q-Values of state-action pairs. Instead, it uses a NN to approximate the action values for a given state.

The best way to understand how DQN works is to see it in an example, there isn't much difference between DQN and Q-Learning.

7.2 Solving the maze problem using a DQN

This section will briefly go over the implementation of a DQN agent in the example of the maze problem. One key change made from the Q-learning code is that the agent wasn't given the permission to step on black squares. That was done because the NN is much slower at learning then the Q-Learning algorithm and that was a good way of speeding up learning. Also the training will introduce a test mode, where the agent will test if the NN has learned to solve the maze, if the test is successful training will be complete.

NN used in this example is represented by the figure 6.5. Figure 7.1 shows the hyperparameters used in the example, as well as the optimizer that was used, and another parameter called MAX_STEPS. Since the agent isn't allowed to step on a wall it can get caught in an infinite loop between to states. MAX_STEPS variable is an insurance so that that doesn't happen.

```
18    learning_rate = 0.001
19    opt = tf.optimizers.Adam(lr=learning_rate)
20    discount_factor = 0.95
21    epsilon = 0.99
22    epsilon_decay = 0.99
23    number_of_episodes = 500
24    MAX_STEPS = 500
```

Figure 7. 1 Hyperparameters for solving the maze problem with a DQN

Figure 7.2 provides a snippet of code containing the algorithm for DQN, it is quite similar to Q-Learning apart from some basic differences. We will break all that down:

- Lines 28 – 32: Start the for loop which represents each episode, resets the state of the agent to the starting position, and resets the environment flags steps to 0 and end to false,
- Line 33: Besides reaching the terminal state, due to new environment features, the number of steps has also been added to limit the play time as well as a flag for reaching the goal,
- Lines 34 – 38: The same as Q-Learning, the agent selects an action, saves the old state-action pair, performs the action, gets the new state, reward and end flag Boolean value,
- Lines 40 and 41: Instead of retrieving the Q-Value from the Q-Table, we use the NN to get the Q-Values for the newly acquired state and extract the maximum Q-Value,
- Lines 43 – 46: Bellman equation is implemented and the value is normalized,

- Lines 48 and 49: Predictions from the previous state are edited by changing the Q-Value of the action that was already taken,
- Lines 51 and 53: The NN is updated to fit the new prediction targets for the previous state,
- The while loop is repeated until the terminal state is reached, maximum number of steps is exceeded or when the agent reaches the goal,
- Epsilon is edited and therefore the episode finished,
- The process repeats until all episodes are finished.

```
28  while episode in range(number_of_episodes):
29          start = np.array([[0,3]])
30          state = start.copy()
31          step = 0
32          end = False
33          while not is_terminal_state(state) and step < MAX_STEPS and not end:
34              action, y_predictions = get_next_action(state, epsilon)
35              old_action = action.copy()
36              old_state = state.copy()
37
38              state, reward, end = get_next_state(old_state, old_action)
39
40              y_predictions = model(state / (N-1))
41              max_prediction = get_max_prediction(y_predictions, state)
42
43              value = reward + discount_factor * max_prediction
44              V.append(value.squeeze())
45              value = value - np.mean(V)
46              value = value / (np.std(V) + 0.0000001)
47
48              y_target = old_y_predictions.copy()
49              y_target[0][action] = value
50
51              loss, grads = loss_func(model,old_state/(N-1), y_target)
52              opt.apply_gradients(zip(grads, model.trainable_variables))
53
54          if epsilon > 0.01:
55              epsilon *= epsilon_decay
```

Figure 7. 2 Implementation of DQN in the example of the maze

7.2.1 Results

Figure 7.3 shows the reward array for solving the maze using a DQN. Execution time was 186.25 seconds with 327.345 loss and it took the agent 127 episodes to complete training.

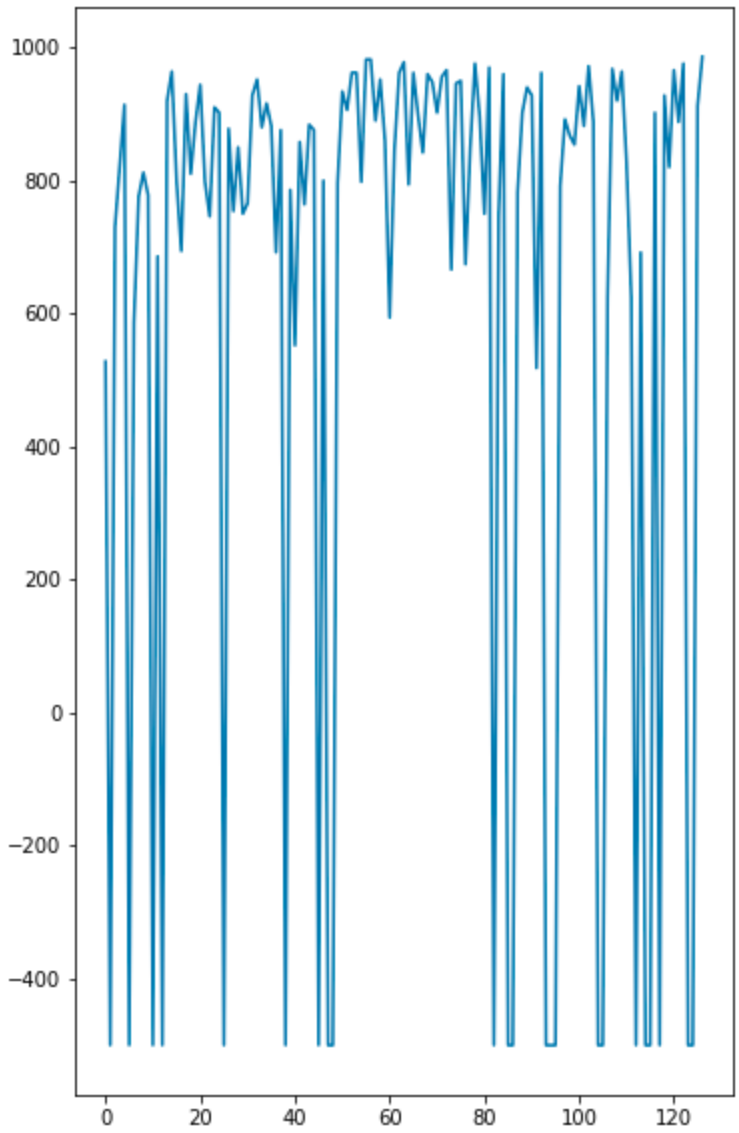

Figure 7. 3 Rewards per episode for solving the maze example using a DQN

Compared to Q-Learning, DQN is much faster to get the reward regarding the episode number, however its execution time is drastically higher. That difference can be up to two or even three orders of magnitude, that is how much DQN is slower. But DQN will overtake Q-Learning as soon as the number of state-action pairs drastically increases.

This chapter has laid the first pillar on which this dissertation lies, that is DQN. It gave a brief explanation of what DQN is and provided an insight in a working example of the same problem we've dealt with before, that is solving the maze. It is of importance to grasp the concepts we went through up until now, since the following chapter will combine DQN and Intermittent Control to research their interaction and deliver the first findings on their combination.

8 Intermittent Control

The previous chapter, along with those before worked up to explain the first main part of this project which is DQN. A RL principle that uses a NN to predict Q-Values, just like Q-Learning, in a Markov decision process. This chapter will firstly provide a brief introduction into IC and then will provide short snippets of code of the implementation. Also it will provide a short insight into the differences between DQN without IC and DQN combined with IC.

Intermittent Control (IC) is a method/technique for controlling feedback, it comes from the field of Control Theory. Systems with IC can be found in human control systems and can have applications in system control in the field of engineering, AI and RL. In Control theory there are two main ways of control: continuous control and discrete. IC divides the gap between continuous and discrete control. [11]

In the context of RL, continuous control can be recognized when training a NN to learn a certain behaviour. In so that the NN has to be updated every time the state changes, that can result in unnecessary computation since the change in state is so small. Furthermore, the slowest part of the DQN and RL is generally always the process of updating the NN, as seen from previous examples.

IC can help with lowering the computational demand by not updating the NN every step but rather every n steps, where $n \epsilon N$, $n > 1$. Therefore, the learning process can be significantly sped up. Complex autonomous driving systems based on RL take weeks and months to train, if IC can achieve any improvement in training time that would be a crucial development in the field.

8.1 Implementing Intermittent Control

To implement IC, we will only add a couple of lines of code to the already existing code for the DQN. The only difference is that the NN will be updated when the control counter reaches the value of the refractoriness parameter. Figure 8.1 shows the code snippet with DQN and IC implemented together.

```
    while episode in range(number_of_episodes):
        start = np.array([[0,3]])
        state = start.copy()
        step = 0
        end = False
        control_counter = 0

        while not is_terminal_state(state) and step < MAX_STEPS and not end:
            action, y_predictions = get_next_action(state, epsilon)

            if control_counter % refactor_interval == 0:
                ic_old_y_predictions = y_predictions.numpy().copy()
                ic_old_action = action.copy()
                ic_old_state = state.copy()


            old_action = action.copy()
            old_state = state.copy()

            state, reward, end = get_next_state(old_state, old_action)

            if control_counter % refactor_interval == 0 or end == True:
                y_predictions = model(state / (N-1))
                max_prediction = get_max_prediction(y_predictions, state)

                value = reward + discount_factor * max_prediction
                V.append(value.squeeze())
                value = value - np.mean(V)
                value = value / (np.std(V) + 0.0000001)

                y_target = ic_old_y_predictions.copy()
                y_target[0][ic_old_action] = value

                loss, grads = loss_func(model,ic_old_state/(N-1), y_target)
                opt.apply_gradients(zip(grads, model.trainable_variables))
                control_counter = 0

            control_counter += 1

        if epsilon > 0.01:
            epsilon *= epsilon_decay
```

Figure 8. 1 Solving the maze example using DQN and IC

The differences between DQN and DQN with IC are the following:

- Line 15: There is a variable called Control Counter which determines after how many steps the NN updates,
- Line 20 – 23: The state, action and NN predictions are saved when the Control Counter meets the Refactor interval,
- The while loop will go on for as many iteration as the Refactor interval specifies without updating the NN,
- Line 31-44: The Control Counter will allow the NN to update the targets predicted in the state determined by the Control Counter,
- Line 45: Control Counter is reset,
- Line 47: After each iteration of the while loop, each Markov decision process, Control Counter is incremented.

8.2 Solving the maze problem using a DQN with IC

The NN used in this implementation is described by Figure 8.2. The NN has 4 hidden layers with 75 nodes using ReLU activation function, 2 input nodes and 4 output nodes using Linear activation function.

```
inputs = tf.keras.layers.Input(shape=(2))
skip = tf.keras.layers.Dense(75, activation='relu')(inputs)
for i in range(4):
    layer = tf.keras.layers.Dense(75,activation='relu')(skip)
    skip = tf.keras.layers.Add()([skip,layer])
outputs = tf.keras.layers.Dense(4,activation='linear')(skip)
model = tf.keras.Model(inputs,outputs)
```

Figure 8. 2 NN used in DQN and IC for the maze problem

Hypermeters used in this implementation are shown with Figure 8.3.

```
learning_rate = 0.001
opt = tf.optimizers.Adam(lr=learning_rate)
discount_factor = 0.95
epsilon = 0.99
epsilon_decay = 0.99
number_of_episodes = 500
MAX_STEPS = 500
refactor_interval = 6
```

Figure 8. 3 Hyperparameters used in DQN and IC for the maze problem example

8.2.1 Results

In order to ensure optimal results a study including the model, hyperparameters, activation functions and the refractoriness factor were examined. This section will only provide a brief summary of the best results achieved, however the whole study can be found in the chapter called Results and evaluation.

Figure 8.4 shows the reward per episode array for DQN with IC when soling the maze problem. Execution time was 10.52 seconds with 62.795 loss and it took the agent 13 episodes to complete training.

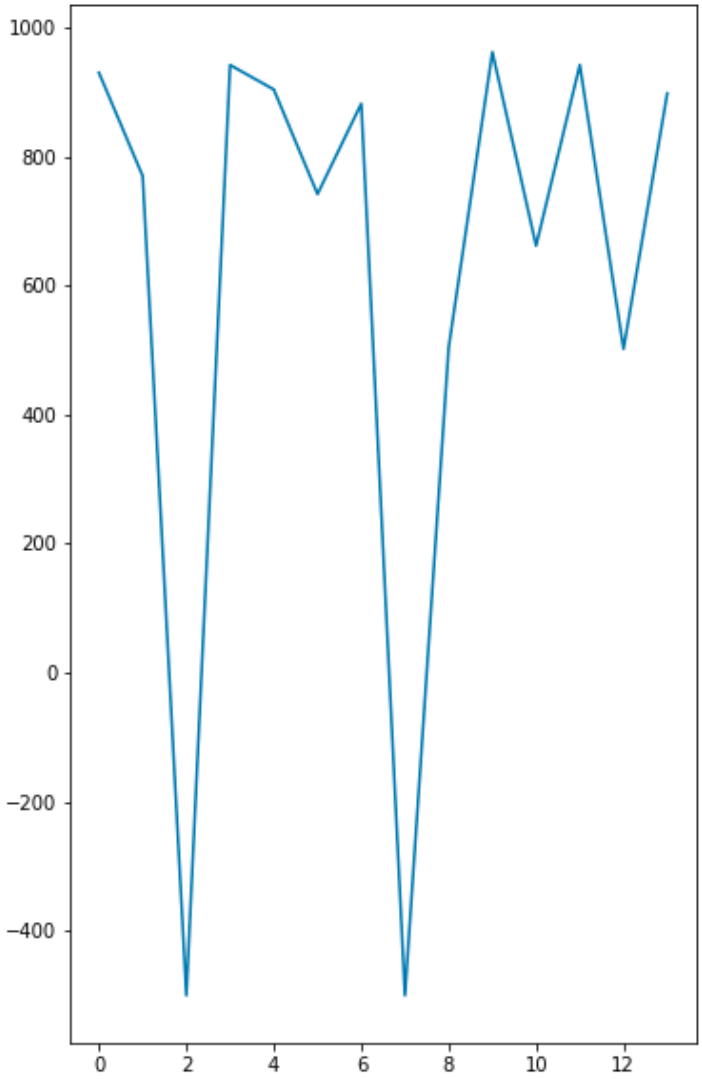

Figure 8. 4 Rewards per episode for solving the maze example using DQN and IC

This chapter provided a brief explanation on IC as well as providing the implementation for the maze problem along with the best results it achieved. Also, it was supposed to put together RL and Control Theory by combining DQN and IC. In this example, as we can take away from the results, when providing a Refractoriness Factor of 6, an improvement of an order of magnitude can be achieved in regards to training time. The next chapter will explain the biggest part of this project and that is the implementation of DQN and IC in the CARLA simulator. That will be the final part of this project and therefore the most important in regards to evaluation the thesis and the gathered results.

9 DQN and IC applied in CARLA

All the previous chapter, including the last one, provided a step by step guide on how this project was made. The last chapter successfully implemented IC with DQN to achieve an improvement in training results. It is of vital importance to understand the basics of RL, Q-Learning, NN, DQN and IC to be able to understand the main thesis of this dissertation. In this chapter we will go over the environment in which the agent trained and the go over the methodology of how we gathered the results.

9.1 The environment

The simulator which was used to gather the main results of the study and test the thesis was CARLA simulator with fixed rate of FPS at 20 and synchronous mode active. That means that the simulation would run one frame when it was told to by the client in which the code for DQN and IC was running.

The environment is shown in Figure 9.1. Blue mark specifies where the agent starts and the red mark specifies the goal, if the agent reaches the red line it will get the biggest available reward. The objective of the agent is to be able to make the left hand turn and proceed straight until it reaches the finish line. It takes the agent, on average, 14 seconds to get to the finish line. The agent had a camera attached to it so it recorded the last episode of every successful training epoch. The agent also had a collision sensor attached to it, if the agent crashed into a wall or into a light pole, the environment and the agent would immediately be reset.

Figure 9. 1 CARLA training environment

9.1.1 Rewards

There were 3 rewards for the agent:

- If the agent crashes, the reward is -1000 minus the distance to the middle of the finish line,
- If the agent reaches the finish line the reward is 1000,
- If the agent didn't crash and didn't finish the reward was 0.

The decision to remove the -1 negative reward for each step was made because the agent started to crash as fast as possible and reduce the number of steps taken. Instead the distance to the finish line was implemented.

9.1.2 NN, inputs and outputs

The NN used in the CARLA simulator for testing the thesis and gathering the results is shown in the Figure 9.2. The NN had 2 inputs, amount of throttle and the steering angle, also it had 4 outputs which were discrete and limited. Two outputs were for 15% and 35% left hand steering, the other two outputs were for 75% and 100% throttle application.

```
inputs = tf.keras.layers.Input(shape=(2))
skip = tf.keras.layers.Dense(20, activation='relu')(inputs)
for i in range(4):
    layer = tf.keras.layers.Dense(20,activation='relu')(skip)
    skip = tf.keras.layers.Add()([skip, layer])
outputs = tf.keras.layers.Dense(4,activation='linear')(skip)
model = tf.keras.Model(inputs, outputs)
```

Figure 9. 2 NN used in CARLA

9.2 Methodology

The thesis was tested in the following manner.

There were 7 Refractoriness interval values tested for: 1, 2, 5, 10, 20, 50 and 100.

For each Refractoriness interval 10 training sessions, or epochs were conducted.

Each epoch consisted of maximum 150 episodes, after the first time the agent crossed the finish line, test mode was activated. That meant that after each episode the agent would be tested with epsilon being 0.

If the agent was able to cross the finish line in test mode, the training session would be over, camera would film the path it took and saved it to the disk.

The main idea behind this study was to see if the agent can learn to make a double apex turn and learn to balance between the steering angles and the throttle application while increasing the Refractoriness interval.

Footage of the agent managing to navigate the environment and pass the test is available with Appendix 5.

This chapter went over the basic setup of the environment, the NN and the methodology oh the way the main thesis was tested. The main part of the project is explained in this chapter. The following chapter will go over the results of the test in CARLA as well as the DQN combined with IC on the example of the maze. That will be followed up by a brief evaluation and analysis of the results.

10 Results and evaluation

The previous chapter touched on the implementation of the main thesis of this project, how it the environment was set up, how the results were gathered overall methodology. Following that, this chapter will give an overview of the results gathered and a brief evaluation.

10.1 Maze problem with DQN and IC results

In order to gain optimal results we first had to determine what was the best model and the hyperparameters for this example. This section will be divided into four parts:

- Model experimentation
- Activation function experimentation
- Hyperparameter experimentation
- IC control factor experimentation

For each experiment 10 tests were ran to determine the mean value of the following parameters:

- Execution time,
- Episode the training finished,
- Reward,
- Loss.

The hyperparameters that were used in NN experimentation were as follows:

- Discount factor = 0.95,
- Epsilon = 0.99,
- Epsilon decay = 0.99,

- Max steps = 500,
- Learning rate = 0.001,
- IC control factor = 10.

10.1.1 Model structure experimentation

The first part to obtaining optimal results was experimenting with the structure of the NN, table 10.1 shows those results. There were 20 different NN models, the first change was the amount of hidden layers which were: 1, 2, 3 and 4. Along with the amount of nodes in each layer, which was: 10, 25, 50, 75 and 100.

Activation function that was used in hidden layers was ReLU and in the output layer was Linear.

Layers:	Nodes:	Execution time (mean):	Episode training finished (mean):	Reward (mean):	Loss (mean):
1	10	168.97	250.90	982.50	44.70
	25	21.97	47.90	983.40	33.74
	50	17.75	36.40	983.00	49.82
	75	11.37	30.10	982.80	37.01
	100	11.71	24.70	982.80	48.78
2	10	145.69	208.10	982.40	37.16
	25	25.66	47.30	983.80	47.77
	50	23.43	41.50	983.20	40.85
	75	14.93	30.40	982.80	14.93
	100	12.82	25.4	983.00	25.27

3	10	84.97	103.90	983.10	28.40
	25	25.54	37.70	983.20	30.92
	50	12.26	18.70	982.60	30.73
	75	15.04	23.00	982.80	39.25
	100	12.35	17.90	982.60	33.76
4	10	158.46	160.60	982.80	25.46
	25	78.44	80.50	982.90	22.92
	50	19.86	28.40	983.00	41.74
	75	**10.90**	**16.30**	**983.20**	**27.05**
	100	18.48	30.10	982.40	17.26

Table 10. 1 Model study

The model with 4 layers and 75 nodes in each layer yielded the best results as seen in the table above.

10.1.2 Activation function experimentation

The second part to obtaining optimal results was experimenting with the activation function of the NN, table 10.2 shows those results. There were 5 different activation functions: ReLU, ELU, SeLU, Swish and Linear. Regarding the output layer, only Linear activation function was used.

Activation function:	Execution time (mean):	Episode training finished (mean):	Reward (mean):	Loss (mean):
ELU	13.04	20.40	983.00	34.66
ReLU	10.90	16.30	983.20	27.05

SeLU	56.25	67.10	983.10	39.66
Linear	**8.17**	**15.00**	**983.00**	**40.71**
Swish	88.26	80.90	982.50	42.98

Table 10. 2 Activation function study

Linear activation function yielded the best results as seen in the table above.

10.1.3 Hyperparameter study

From the previous section we found out that the best NN has 4 layers and 75 nodes in each layer with Linear activation function. For hyperparameter study we will explore the following hyperparameters:

- Discount factor,
- Learning rate,
- Epsilon decay,
- Max steps

The results from experimentation with the aforementioned hyperparameters are shown in tables: table 10.3, table 10.4, table 10.5 and table 10.6.

Discount factor:	Execution time (mean):	Episode training finished (mean):	Reward (mean):	Loss (mean):
0.40	12.31	21.90	983.00	25.47
0.60	21.35	37.00	983.20	28.35
0.80	24.27	39.70	983.60	50.11
0.90	10.76	16.50	983.00	49.34

0.95	**10.90**	**16.30**	**983.20**	**27.05**
0.99	19.56	34.60	982.60	44.97
1	11.70	19.20	983.60	40.32

Table 10. 3 Discount factor study

Discount factor of 0.95 yielded the best results as seen in the table above.

Learning rate:	Execution time (mean):	Episode training finished (mean):	Reward (mean):	Loss (mean):
0.1	20.80	27.10	984.00	83.25
0.01	29.13	43.10	983.80	61.13
0.001	**10.90**	**16.30**	**983.20**	**27.05**
0.0001	24.70	40.60	983.20	35.44
0.00001	315.42	358.60	980.40	48.51

Table 10. 4 Learning rate study

Linear factor of 0.001 yielded the best results as seen in the table above.

Epsilon decay	Execution time (mean):	Episode training finished (mean):	Reward (mean):	Loss (mean):
0.9	242.80	247.50	981.80	203.65
0.99	**10.52**	**13.60**	**983.00**	**62.79**
0.995	12.87	21.50	982.80	33.25
0.999	20.68	32.30	982.80	37.68
0.9995	14.35	21.50	983.40	27.38

Table 10. 5 Epsilon decay study

Epsilon rate of 0.99 yielded the best results as seen in the table above.

Max steps	Execution time (mean):	Episode training finished (mean):	Reward (mean):	Loss (mean):
20	23.09	400.90	980.00	0.58
50	36.20	326.40	980.80	15.03
100	33.91	170.50	981.90	26.86
200	30.86	83.70	982.90	44.06
500	**12.31**	**16.60**	**983.40**	**23.88**
1000	13.73	18.10	983.00	40.38

Table 10. 6 Max steps study

Max steps of 500 yielded the best results as seen in the table above.

10.1.4 IC Control factor study

Now that all the hyperparameters have been set to optimal and we've found the optimal model of the NN for this specific problem, the main thesis of this project can be tested on this example. Table 10.7 shows those results.

IC Control factor:	Execution time (mean):	Episode training finished (mean):	Reward (mean):	Loss (mean):
1	170.88	66.20	983.80	331.42
2	78.99	49.20	983.80	121.93
4	17.83	17.70	982.80	113.04
6	**10.52**	**13.60**	**983.00**	**62.79**
10	12.31	21.20	983.40	23.88
12	13.96	17.40	982.60	39.22
15	40.59	56.20	982.80	104.68

Table 10. 7 IC Control factor study

IC Control factor of 6 yielded the best results as seen in the table above.

The analysis of the results will be handled in the Results evaluations section of this chapter.

10.2 DQN and IC in CARLA results

The main results of the thesis will be laid out here and then commented on later in the next section called Results evaluations. All the hyperparameters and the NN model that were used in this set of tests are the same as the ones used in

the optimal solution for the Maze problem with DQN and IC. For each experiment 10 tests were ran to determine the mean value of the following parameters:

- Number of fits (the number of times the NN had to be updated),
- Execution time,
- Episode the training finished,
- Loss.

Along with the number of failed and successful tests.

IC Control factor	Number of fits (mean):	Execution time (mean):	Episode learning finished (mean):	Loss (mean):	Failed test:	Successful tests:
1	9430.37	913.90	54.40	497.05	1	9
2	5681.91	1202.11	73.20	250.34	2	8
5	1822.54	829.81	51.80	107.87	1	9
10	1353.64	1323.40	76.40	76.75	2	8
20	**698.37**	**1137.66**	**69.10**	**22.42**	**1**	**9**
50	308.01	1588.25	82.10	7.83	3	7
100	177.36	1724.152	98.00	7.01	2	8

Table 10. 8 IC Control factor study in CARLA

The following figures will provide a visualisation for the main results of the project: figure 10.1 shows the mean number of fits, figure 10.2 shows the mean

execution time, figure 10.3 shows the mean episode learning finished, figure 10.4 shows the mean loss and figure 10.5 shows the test success rate.

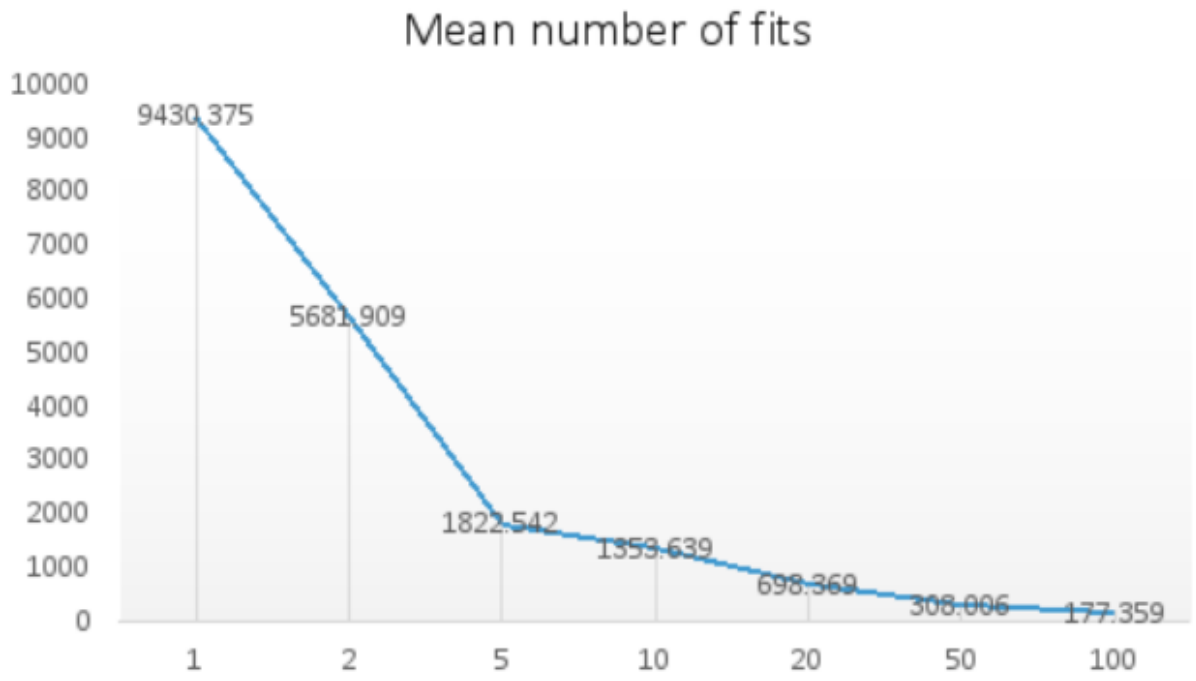

Figure 10. 1 Mean number of fits graph for IC and DQN in CARLA

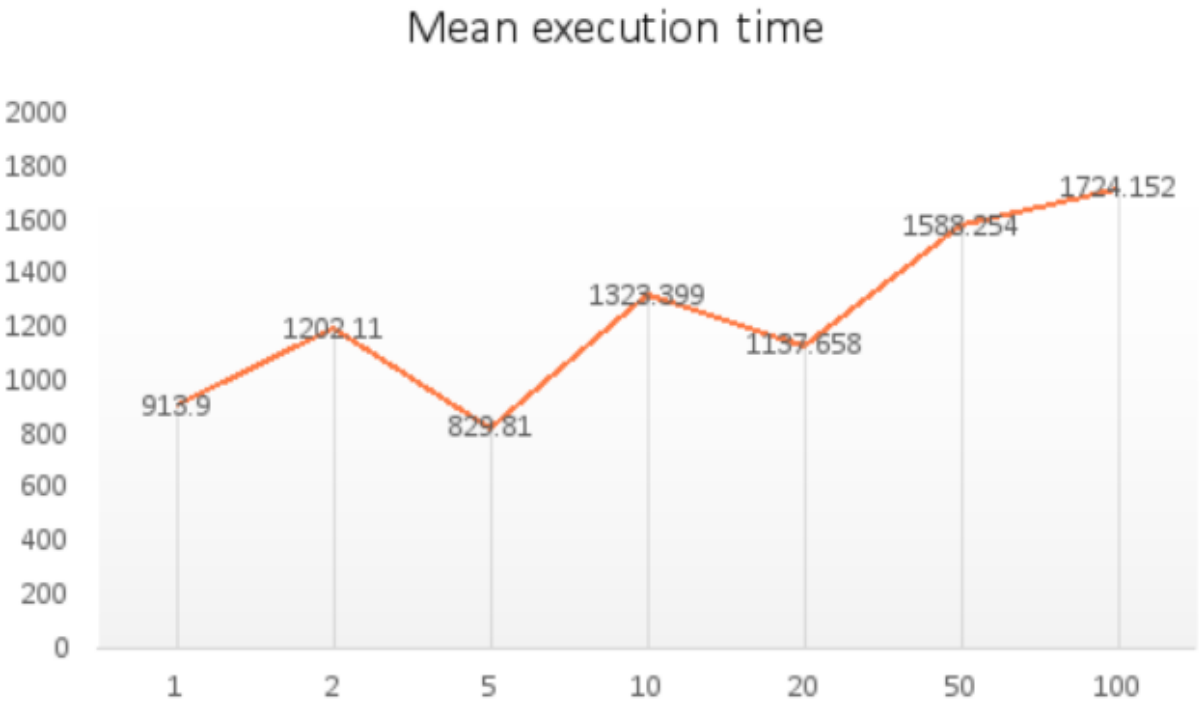

Figure 10. 2 Mean execution time graph for IC and DQN in CARLA

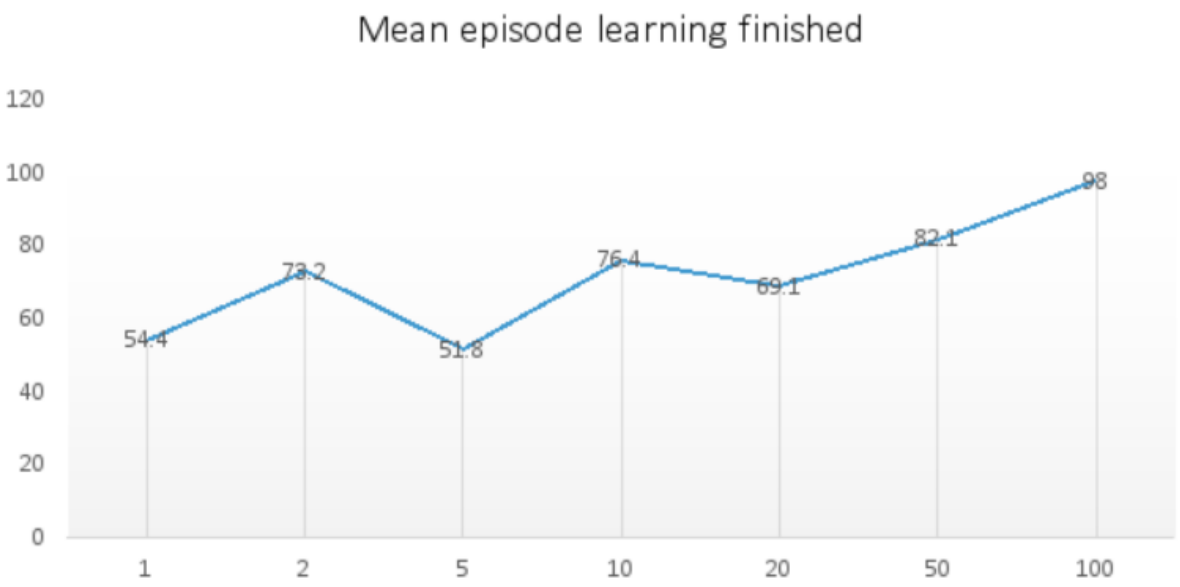

Figure 10. 3 Mean episode learning finished graph for IC and DQN in CARLA

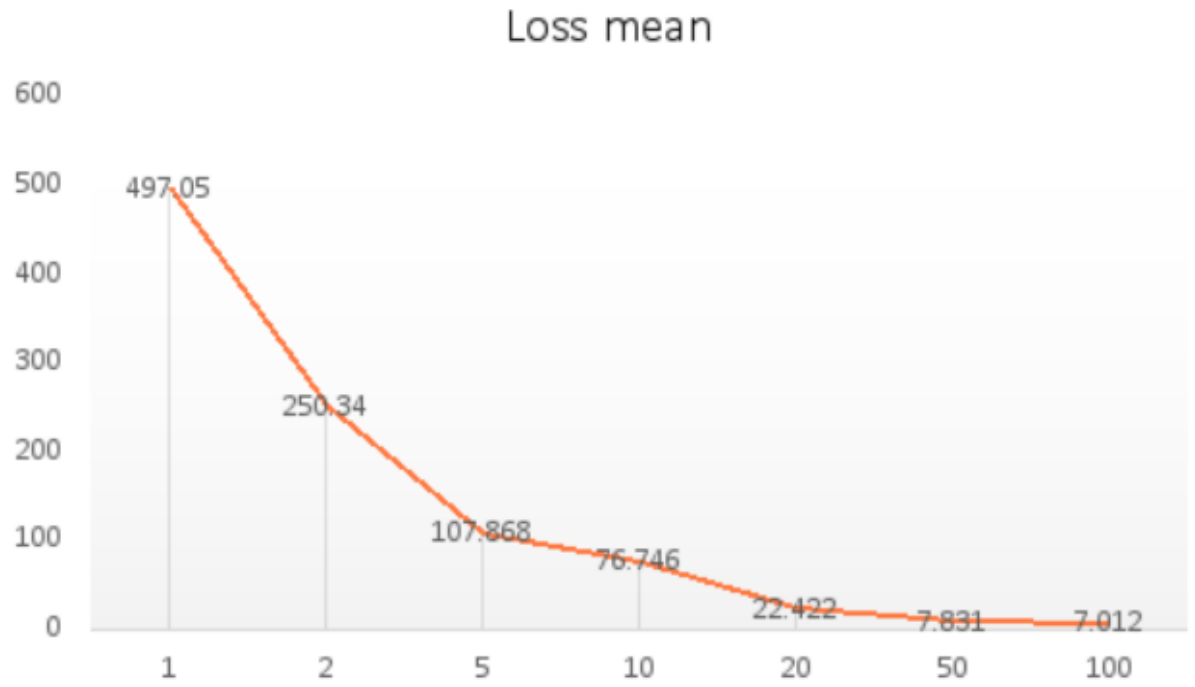

Figure 10. 4 Mean loss graph for IC and DQN in CARLA

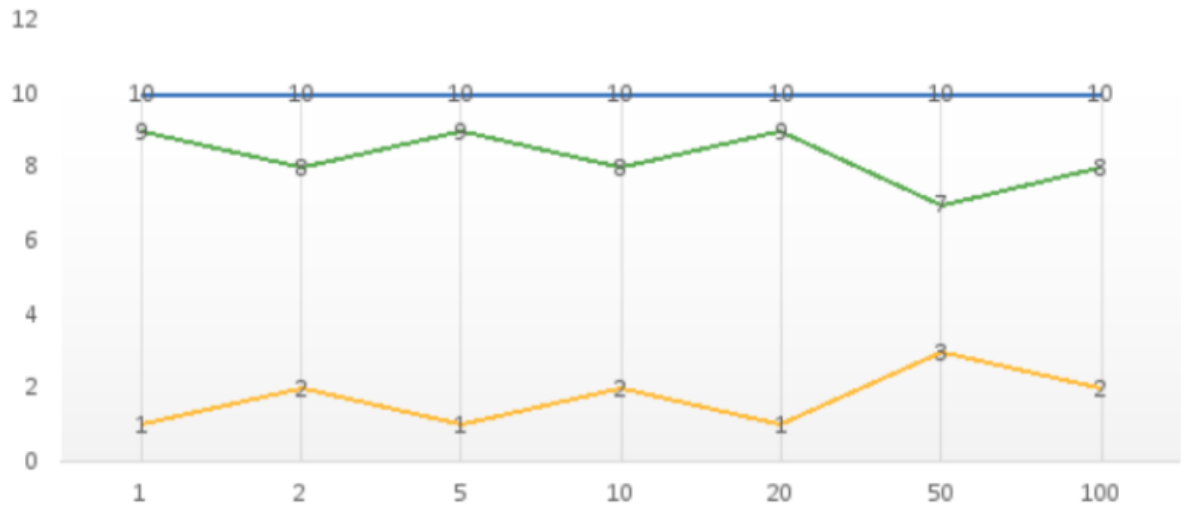

Figure 10. 5 Test success rate graph for IC and DQN in CARLA

10.3 Results evaluation

10.3.1 The maze problem

The study that was performed in the environment of the maze showed that the IC Control factor of 6 performed better than no IC Control factor, meaning factor of 1. When analysing the mean reward and loss, not much information can be extracted. Which means that the gained knowledge in both cases are similar and are able to achieve the same behaviour. However, there is a difference in the rate of learning, the most important performance measures show the following improvement:

- 1624% or 16.24 times faster execution time,
- 487% or 4.87 fewer episodes to finish training.

That would confirm that in the case of the maze, IC will significantly improve performance and training time. Therefore, other real world applications such as: navigation and finding the fastest path by distance or weight using AI and NN can benefit from IC.

10.3.2 CARLA

When applying the DQN with IC in CARLA the results didn't show any execution time gain due to the fact that CARLA simulator ran in real time and had the biggest impact on execution time. Therefore, execution time is not a reliable performance metric in this case because the NN could be updated while the simulation was running. However, another metric that can provide some insight into the effects of IC in this case is the mean number of fits.

Best IC Control factor proved to be 20 because of the following:

- 1350% or 13.50 times fewer fits to the NN which results in faster execution and learning time in systems which are significantly slowed down by the NN,
- The success rate remained the same at 90%,
- 2217% or 22.17 times lower loss.

All that with two small trade-offs:

- 25% or 0.25 times longer execution time,
- 27% or 0.27 times more episodes to finish training.

However, when all that is taken into account, we can conclude that IC results in this case require more episodes to train which in turn prolongs the execution time in these kinds of systems. Meaning systems dependant on the simulators real time execution.

Also, the amount of IC Control factor can't be increased indefinitely. The shortest path in the maze example was 16 step. That means if the IC Control factor is higher than 16, the learning could be hindered. The same applies in the CARLA simulator, as we can see from the results factor of 50 and 100 have shown worse performance, more episodes, twice the execution time, etc.

11 Conclusion and Future Work

The previous went over all the significant results of applying IC along with a brief evaluation of those results. Following that, a brief conclusion, future work and an explanation on the shortcomings from the projects Terms of Reference.

11.1 Conclusion

The main thesis for this project was to test the influence of IC on RL. The main reason for that is the possible improvements that could be made in speeding up and bettering the performance and the learning process of AI in autonomous vehicle systems. The way this was tested was to firstly test it in an example of a maze, where a IC Control factor of 6 proved to be 1624% faster in learning than a system without IC along with 487% fewer episodes it took to finish the learning process. The second implementation where IC was tested for RL was in the CARLA simulator, where the agent tried to learn a double apex left hand turn on a 4 lane road without any additional traffic. The results were positive when analysing the amount of fitting the NN needed and the loss. For a IC Control factor of 20 at 20 FPS, the agent performed 1350% less fitting operations with 2217% less loss. However, a downside to the test was the fact that the CARLA simulator ran in real time and therefore was the bottleneck regarding the execution time. Furthermore, when implementing IC the agent needed 27% more episodes to learn and because of that execution time was prolonged by 25%.

11.2 Future Work

To further explore the effects of IC in RL in the field of autonomous vehicles, best course of further research would be to implement it in a more complex CARLA environment. That could result in less fitting operations performed on the NN, but not the execution time due to the increased number of episodes required to train the agent. The most significant impact to this area of research can be made in a simulator which is not constrained by the execution time of the simulation itself. IC in RL could have a significant impact and could improve the time it takes to train agents by orders of magnitude as is worth further exploring.

11.3 Shortcomings regarding ToR

One of the aims of the project specified in ToR was "Comparison of results with already established reinforcement learning algorithms on Atari games.". This goal wasn't perused nor achieved. The reason is this. After familiarisation with RL, Q-Learning, NN, DQN and IC it was obvious that in order to test if IC works, implementing complex algorithms and testing it in Atari games would be a waste of time and resources since IC can directly be applied in CARLA. Furthermore, after realizing that there isn't much literature on the cutting edge algorithms of RL such as Rainbow method, implementation would be extremely difficult. Not to mention a waste of time since IC can be tested with the simples RL algorithm such as DQN.

Another limitation was the processing power required to run CARLA. Using a single Nvidia GTX 1050 Ti, it proved impossible to train an agent in a much more complex CARLA environment. Due to Covid-19 restrictions, that limitation could not be resolved.

12 References

[1] World Health Organisation. (2018.) *Global status report on road safety 2018.* Geneva: World Health Organisation. (CC BY-NC-SA 3.0 IGO)

[2] S. Singh. (2015.) *Critical reasons for crashes investigated in the national motor vehicle crash causation survey* . Washington DC: NHTSA. (DOT HS 812 11569)

[3] Yurtsever E., et al., (2020.) *A Survey of Autonomous Driving: Common Practices and Emerging Technologies.* Volume 8. New York City: Institute of Electrical and Electronics Engineers (IEEE) [Online] [Accessed on 21st September 2020.] https://arxiv.org/pdf/1906.05113.pdf.

[4] Andrychowicz M., et al., (2017.) *Hindsight Experience Replay* (arXiv:1707.01495v1 [cs.LG]), [Online] [Accessed on 9th October 2020.] https://arxiv.org/abs/1707.01495

[5] Pathak D., et al., (2017.) *Curiosity-driven Exploration by Self-supervised Prediction* (arXiv:1705.05363 [cs.LG]), [Online] [Accessed on 9th October 2020.] https://arxiv.org/abs/1705.05363

[6] Jaderberg M., et al., (2016.) *Reinforcement Learning with Unsupervised Auxiliary Tasks* (arXiv:1611.05397 [cs.LG]) [Online] [Accessed on 9th October 2020.] https://arxiv.org/abs/1611.05397

[7] Doya K., (2000.) *Complementary roles of basal ganglia and cerebellum in learning and motor control* Japan: ScienceDirect [Online] [Accessed on 9th October 2020.] https://www.sciencedirect.com/science/article/abs/pii/S0959438800001537

[8] Zgonnikov A., et al., (2018.) *Evidence Accumulation Account of Human Operators' Decisions in Intermittent Control During Inverted Pendulum Balancing*, Miyazaki, Japan: Institute of Electrical and Electronics Engineers

(IEEE) (10.1109/SMC.2018.00130), [Online] [Accessed on 21st September 2020.] https://ieeexplore.ieee.org/abstract/document/8616126

[9] Condurat M., et al., (2017.), *Environmental Impact of Road Transport Traffic. A Case Study for County of Iaşi Road Network*, Romania: ScienceDirect [Online] [Accessed on 9th October 2020.] https://www.sciencedirect.com/science/article/pii/S1877705817309621

[10] Sutton R. S., Barto A. G., (ed.) (2018) *Reinforcement Learning: An introduction.* 2nd ed., Cambridge, MA: The MIT Press.

[11] Loram I.D., et al., (2012) *Identificaton of intermittent control in man and machine,* . R. Soc. Interface. 92070–2084 [Online] [Accessed on 5th October 2020.] http://doi.org/10.1098/rsif.2012.0142

13 Bibliography

Dabney W., et al., (2020.), *A distributional code for value in dopamine-based reinforcement learning* Nature.com (577, 671–675(2020)) [Online] [Accessed on 9th October 2020.] https://www.nature.com/articles/s41586-019-1924-6?fbclid=IwAR3c1dLc9-puvMAGS2lURZygNEKpfhAyLChl2DDT_pJOf9rG3sWbFOKrZ8A

Mnih V., et al., (2015.), *Human-level control through deep reinforcement learning* Nature.com (518, 529–533(2015)) [Online] [Accessed on 9th October 2020.] https://www.nature.com/articles/nature14236

Mnih V. et al., (2013.) *Playing Atari with Deep Reinforcement Learning* (arXiv:1312.5602 [cs.LG]) [Online] [Accessed on 9th October 2020.] https://arxiv.org/abs/1312.5602

Hessel M., et al., (2017.) *Rainbow: Combining Improvements in Deep Reinforcement Learning* (arXiv:1710.02298 [cs.AI]), [Online] [Accessed on 9th October 2020.] https://arxiv.org/abs/1710.02298

Ljunberg T., et al., (1992.) *Responses of monkey dopamine neurons during learning of behavioral reactions* PubMed.gov (10.1152/jn.1992.67.1.145) [Online] [Accessed on 9th October 2020.] https://pubmed.ncbi.nlm.nih.gov/1552316/

14 Appendices

Appendix 1: *Q-Learning maze*, https://github.com/atomic01/MSc-Project-Appendices/blob/main/notebook%203.%20SOLVING%20THE%20MAZE%20PROBLEM%20WITH%20Q_LEARNING%20using%20numpy.ipynb [Online] [Accessed on 6th October 2020]

Appendix 2: *DQN maze*, https://github.com/atomic01/MSc-Project-Appendices/blob/main/notebook%203.1%20SOLVING%20THE%20MAZE%20PROBLEM%20WITH%20a%20DQN.ipynb [Online] [Accessed on 6th October 2020]

Appendix 3: *DQN with IC maze*, https://github.com/atomic01/MSc-Project-Appendices/blob/main/final_version_maze_DQN_with_IC.py [Online] [Accessed on 6th October 2020]

Appendix 4: *DQN with IC in CARLA*, https://github.com/atomic01/MSc-Project-Appendices/blob/main/final_version_carla_DQN_with_IC.py [Online] [Accessed on 6th October 2020]

Appendix 5: *Footage of DQN with IC in CARLA*, https://github.com/atomic01/MSc-Project-Appendices [Online] [Accessed on 9th October 2020]