City, University of London

MSc in Computer Games Technology

Project Report

2016-2017

Prediction of Future States of Features Extracted from Deep Reinforcement Learning Networks

Joaquin Ollero Garcia

Supervised by: Dr. Christopher Child

September 28, 2017

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed:
Joaquín Ollero García.

ABSTRACT: Deep Learning is a class of Machine Learning algorithms which automatically extract information from raw data by using artificial neural networks in which neurons deeper in the network naturally encode features in the data. Reinforcement Learning algorithms provide a mechanism that rewards or punishes intelligent agents so they can learn that a certain feedback related to a particular situation leads to a good or a bad state in the future. The combination of these techniques, termed Deep Reinforcement Learning Networks, offers the possibility to use a Deep Learning Neural Network to produce an approximate Reinforcement Learning value table that allows extraction of features from neurons in the hidden layers of the network. The aim of the project is to solve a pathfinding problem within an environment by implementing the Reinforcement Learning Q-Learning technique by building a Q-Table, a Q-Network and a Deep Q-Network, using the Machine Learning library TensorFlow. The results show that the extraction of features from the hidden layers of the Deep Q-Network decreases the training time of the agent and proves the existence of encoded information regarding the best action to take from a specific state. The project attempts to make a contribution to research on Machine Learning.

KEYWORDS: Reinforcement Learning, Neural Networks, Deep Learning, Feature Extraction, TensorFlow.

A Emma y a ti, que ya lo sabes.

Acknowledgements

I would like to thank all the people who have supported me in the development of this project. Dr. Christopher Child for being an outstanding supervisor who has guided me with his wisdom and dedication during the whole academic year and specially throughout the summer. Dr. Gregory Slabaugh for being a figure of inspiration as an exceptional course director. My family who has always been there with me in the distance with their constant affection. Natalia, for holding my hand unconditionally and who has taken care of me with her endearment. My "Granada family" in London: Rafa, Gustavo, Álex and Pablo, for their true friendship. Sofía, for being with me from the beginning in this amazing adventure. Federico and Derek, of whom I have learned a lot and for having fought all the battles by being united. Luis and Laura, always. And finally to all those people who have shed a ray of light in my life.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction and Objectives

1.1. Background and Motivation

Deep Learning is a class of Machine Learning algorithms based on learning data representations, instead of learning specific tasks. By using an architecture such as a Deep Neural Network (Hinton and Salakhutdinov, 2006), it is possible to extract information from neurons placed in the hidden layers of the network that automatically encode valuable features from the raw data used as inputs.
Reinforcement Learning algorithms, as opposed to supervised and unsupervised learning, present a mechanism to train an artificial agent in the learning process of how to solve a specific problem, by providing positive or negative feedback. Proceeding with this principle, an agent that is currently in a state and receives a certain feedback will know if that relation leads to a good or bad state in the future. Overall, a Reinforcement Learning algorithm maps every state with every possible action that can be taken within a given environment to a specific value. This value will inform the agent how good or bad taking each action is in relation to the next state that the agent will experience straight afterwards. Q-Learning is a popular Reinforcement Learning technique, used to produce optimal selections of actions for any Markov decision process (Bellman, 1957).
The combination of these two techniques, termed Deep Reinforcement Learning Networks (Mnih et al., 2015), offers the possibility to use a Deep Learning Neural Network in order to produce an approximate Reinforcement Learning value table for the training of the agent. This will allow the extraction of features encoded in neurons in the hidden layers of the network.

Reinforcement Learning using Deep Learning Neural Networks is currently receiving intense media attention. There is growing interest in this area of artificial intelligence since (Mnih et al., 2015) introduced Deep Q-Networks and the research was published in the multidisciplinary scientific journal Nature (Figure 1). The presented work proved that the proposed model achieved a higher level of expertise than that of a professional human game player in a set of 49 time-honored Atari 2600 games (Bellemare et al., 2012). This revelation is one of the key motivations behind this project, since a groundbreaking achievement as such marks a high-point in the field and opens the path for related research. There are plenty of areas worth exploring as it still

remains to be determined whether the implementation of Deep Q-Networks can be extended to contemporary games and more complex environments in general. Moreover, feature extraction from Deep Q-Networks is a state of the art approach, because it has not been previously combined with Reinforcement Learning.

Figure 1: Nature. Vol. 518, No. 7540. 26th February 2015

The field of Machine Learning is experiencing an enormous evolution in terms of revolutionary applications and groundbreaking research that is leading the area to tremendous popularity. Along with this, open source tools that allow scientists to experiment with algorithms and structures are becoming more accessible and straightforward to learn and use. One of the Machine Learning libraries that is experiencing the biggest growth is TensorFlow (Abadi et al., 2016), an open-source software library for Machine Intelligence developed by Google. Analysing the activity of the TensorFlow community, which

is expanding and getting engaged in countless projects of all kinds, is another of the main motivations behind this project.

The purpose of the project is to acquire a model of an envinronment using features extracted from different levels of a Deep Reinforcement Learning Network using TensorFlow. The system will model features extracted from an initial level of a Deep Neural Network, using them as inputs of a reduced Deep Neural Network (in terms of number of layers) to prove that the information of the inputs is automatically encoded and preserved and can be used to increase the speed of learning of the agent. Performing feature extraction from a later level of a Deep Neural Network will allow the system to classify the codifications and verify if they can be used to predict future states of the features.

1.2. Beneficences

It is intended that the attempted academic research will make a contribution to the community of the computer science area of artificial intelligence and more precisely to Machine Learning researchers. Ultimately, the main purpose of this project is to propose a new approach to Deep Q-Networks and to expand an existing and proved innovative model. The project targets the previously mentioned community and specialized researches as its beneficiaries.

1.3. Methods Overview

A structured set of methods will be followed to undertake the project (see Figure 2). First, an environment that will sustain the algorithms that the agent will use needs to be defined and should be both reliable and flexible. The cornerstone of the research is the Reinforcement Learning algorithm that will teach the agent how to learn over time the specifics of the environment: the states, actions and reward values. In this sense, the Q-Learning technique will be first implemented using a Q-Table, then a Q-Network and finally a Deep Q-Network, which will contain the fundamental structure that will allow the feature extraction to occur. Consequently, it will be proved if it is possible to extract these components and use them to improve the originally proposed model by predicting the future states of the features.

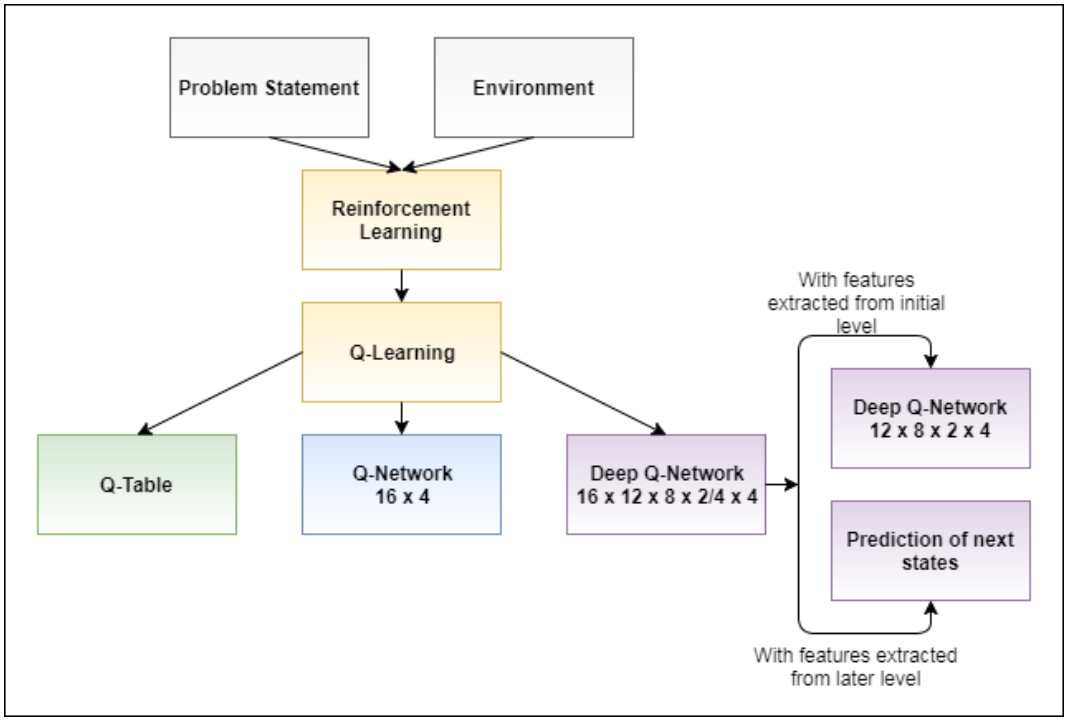


Figure 2: Methods overview

1.4. Expected Results

By attempting the project, a set of specific results is expected (see Figure 3). This includes a comprehensive definition of an environment for an intelligent agent and a successful training to solve a Reinforcement Learning problem using different structures: Q-Table, a Q-Network, a Deep Q-Network and a Deep Q-Network using Feature Extraction. Numerical data and plots about specifics training times, accumulated rewards over time, first successful learning episodes and optimal pathfinding testing should make crucial observations about the behaviour of the agent following the aforementioned algorithm in the Q-Table and the various neural networks. Moreover, the most important result of the project is to conclude if the encoded information present in the hidden layers of the Deep Q-Network can be used to improve the learning process of the agent. It is expected that the results will answer the originally proposed research question, by shedding light on the intrincate inner structure of a Deep Q-Network.

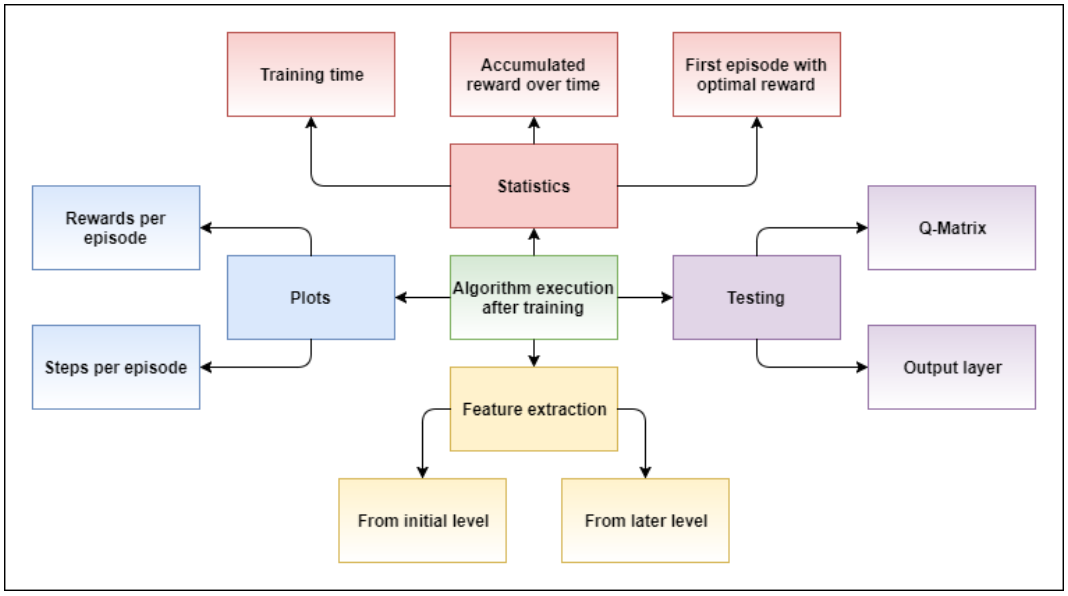


Figure 3: Expected results

1.5. Objectives

The capital objective of this thesis is the training of an intelligent agent with a Reinforcement Learning algorithm via the implementation of a Deep Q-Network within a previously designed environment. Emphasis have been given to the profound study of this structure by extracting features from its hidden layers in order to use them in an environment modelling algorithm by reducing the state space and by predicting the future states of these features. More precisely, the objectives can be summarized as follows:

1. Lay the foundation of the problem by defining the model of an environment for the intelligent agent to solve.
2. Implement the reinforcement learning Q-Learning technique for the training of the agent to provide optimal solutions to the proposed problem using a Q-Table, a Q-Network, a Deep Q-Network and a Deep Q-Network using Feature Extraction.
3. Study of the topology of the Deep Q-Network along with the encoded features in every hidden layer. More specifically, the features extracted from the initial level of the Deep Q-Network will be used to model another Deep Q-Network in order to improve the learning process of the agent based on several metrics. By extracting features from the later level of the network, the focus is on the use of the features to prove if they contain encoded information about the best action to take from each state.

4. Generate a set of results and plots in terms of training times, first successful learning episodes and accumulated rewards over time to provide comparative results between the different structures and to determine if the two previously stated assumptions are true or false.
5. Produce a project report that comprises the whole development process and comprehensively presents the overall research within the period of time stated by the University policy.

By the completion of these objectives, it is expected to give an answer to the originally proposed Research Question: "Can features extracted from the later level of a Deep Reinforcement Learning Network be used to predict the future states of the features in an environment modelling algorithm?"

This video summarizes the project by showing the training process of the agent and the most important results.

1.6. Work Plan

The development process followed to meet the objectives can be visualized in the work plan included in Appendix B, which highlights the main phases of the project development.

1.7. Structure of the Report

The report is structured into the following chapters:

- *Chapter 1. Introduction and Objectives.* The introductory section to the report establishes the background and motivation of the research while presenting an overview of the methods applied, the expected results to be obtained and the specific objetives to achieve.
- *Chapter 2. Context.* A literature review that covers topics related to Machine Learning, Reinforcement Learning, Neural Networks, Deep Learning and more precisely, Deep Reinforcement Learning Networks along with its applications, focusing on feature extraction.
- *Chapter 3. Methods.* A detailed exposition of the methods that comprise the development process. Starting with a elaborated description of the Reinforcement Learning algorithm used and followed by the different structures that use it: a Q-Table, a Q-Network, a Deep Q-Network and finally a Deep Q-Network using Feature Extraction.

- *Chapter 4. Results.* A collection of the outcomes obtained by the application of the described methods. Results are mainly stated in terms of testing the behaviour of the agent, training time, accumulated reward over time, first episode where the reward was optimal and a series of plots that highlight the rewards and number of steps over the training process.
- *Chapter 5. Discussion.* In relation with the proposed methods and results, a thorough analysis of the model is introduced, while evaluating hypothetical alternatives that could result in compelling additions to the acquired knowledge.
- *Chapter 6. Evaluation, Reflections and Conclusions.* To finish the main body of the report, overall considerations are evaluated while making reflections about them. Implications of the work are accounted and a set of future tasks that would improve and contribute to the research are discussed.
- *References.* A list of all the referenced work mentioned along the report.
- *Appendices.*
    - *Appendix A: Project Proposal.* The original project proposal submitted as the Coursework Task 2 for the Research Methods and Professional Issues (RMPI) module.
    - *Appendix B: Work Plan.* The work plan followed during the development process is included in this appendix as a Gantt diagram.
    - *Appendix C: Source Code.* This appendix collects all the source code that has been produced for the development of the project.

2. Context

This chapter attempts to explain the current state of Deep Reinforcement Learning Networks, both in theory and in practice. First, an introduction to Machine Learning, Deep Learning, Reinforcement Learning and neural networks is made with the intention of settling important concepts related to the state of the field studied in this project. An analysis of Deep Reinforcement Learning Networks as proposed by (Mnih et al., 2015) and various extensions made to this work are introduced next. To finish the chapter, different applications of this type of Deep Neural Networks are presented as well, while making emphasis on the topic of feature extraction.

2.1. Machine Learning

Machine Learning is a field of computer science that, as Arthur Samuel states, gives "computers the ability to learn without being explicitly programmed" (Samuel, 2000) by constructing algorithms that can learn from and make predictions on data using a set of inputs to build a model. This technique is used to solve problems that involve algorithms that are difficult to design and to explicitly implement while having good performance.

It is possible to distinguish between different types of Machine Learning. Supervised Learning (Hastie et al., 2009) trains an algorithm to ultimately select the best description of the input data. Unsupervised Learning is mainly used in pattern recognition and descriptive modelling as it works with unlabeled data. Supervised and Unsupervised Learning are usually offline or batch algorithms, as a static dataset is given at the beginning and an output that would solve the problem is expected at the end. Reinforcement Learning (Sutton and Barto, 1998), the one that is going to be used in this research, makes an agent to continuosly learn by gathering information from an environment using online learning, meaning that the entire input is not available from the start, so the agent can start the process over again once it has taken an action in order to keep discovering the environment. More details regarding this type of Machine Learning are discussed later in this chapter.

Deep Learning is a subset of Machine Learning that imitates how the human brain processes data and how it creates patterns for use in decision making. Unlike shallow learning, Deep Learning Neural Networks consist of successive stacked hidden layers, which would be composed of different linear and non-

linear transformations, between the input and output layers. A special type of neural network, Autoencoders, can be used in order to build a Deep Neural Network (Hinton and Salakhutdinov, 2006). An Autoencoder is a neural network with the same number of inputs and outputs and a specific number of hidden layers in between, in which each hidden layers has a smaller number of neurons than the input and output layers have. The objective of this type of neural network is to have the input (X) reconstructed in the output ($\hat{X}$) (See Figure 4), so the input is first encoded in the neurons of the hidden layers and then decoded towards the output. The information of the inputs it is codified automatically in the neurons of the hidden layers. This fact will be used in this research to extract features that are automaticall encoded in the hidden layers of the Deep Q-Network.

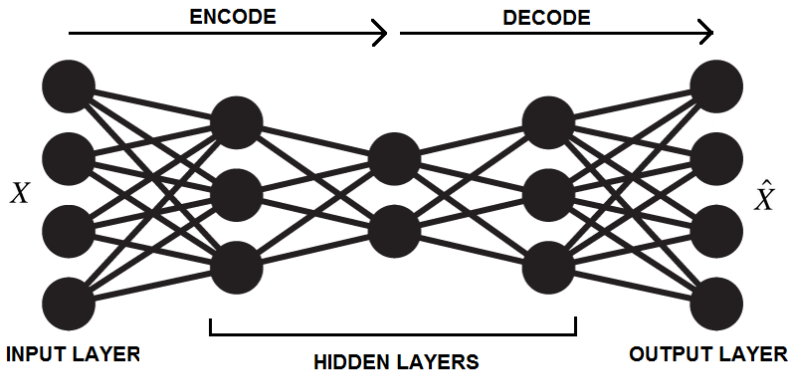


Figure 4: An Autoencoder with 3 hidden layers

2.2. Reinforcement Learning

As described in (Sutton and Barto, 1998), a Reinforcement Learning model is composed by various elements: an Agent, an Environment, States, Actions and Rewards, composing an architecture as the next one:

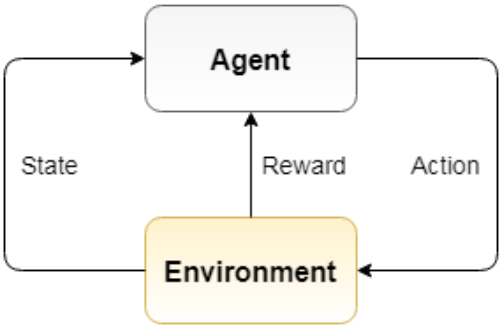


Figure 5: Reinforcement Learning model

An intelligent agent is an entity that acts with autonomy by observing an environment through sensors and acting using actuators in order to achieve a certain goal. The environment is a scenario that the agent is supposed to explore, which is composed by a set of states. The agent interacts with the environment by taking actions to progress through it with the objective of achieving a certain goal that would provide a positive reward to the agent. In this sense, negative rewards can be provided to the agent as well, so it would be able to learn that taking certain actions from certain states lead to a good situation in the future or viceversa.

In order to properly discover the environment and to have a successful learning process, the agent should inspect the environment by taking random actions (exploration) and once it has discovered it, actions that has maximum rewards associated should be taken (exploitation). A balance must be maintained between these two ways of taking actions. In this sense, policies such as the ϵ-greedy policy (explained in Chapter 3) and the decaying ϵ-greedy policy (discussed in Chapter 5) are interesting strategies to be implemented so the agent is able to learn the specifics of the environment.

There is a wide variety of Reinforcement Learning techniques. The Q-Learning technique (Watkins and Dayan, 1992) describes a table mapping each combination of state and action in an environment to a reward. This technique is the one used in this project and it is implemented using a Q-Table, a Q-Network and a Deep Q-Network. The Monte Carlo methods (Robert, 2004) include a technique called the Monte-Carlo tree search that is used to find the best actions to take in an environment. An agent using Temporal Difference (Sutton and Barto, 1998) learns by discovering the environment following a specific policy and by calculating the best current estimated value according

to the values learned before.

2.3. Neural Networks

Artifial neural networks were first imagined in 1943, when McCulloch and Pitts introduced the first neuron model (McCulloch and Pitts, 1943). Five years later, Alan Turing, in his "Intelligent Machinery" work (Turing, 2004), proposed a computing architecture based on the idea that it would be possible to have simple units, neurons, which would make simple calculations based on connections with other neurons and that it would be possible to tune these connections. Such an architecture would be unorganized at the beginning, but after training it would be able to organize itself in order to a complete a certain task.

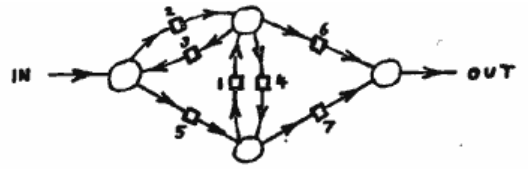


Figure 6: An unorganized machinery from (Turing, 2004), page 11

A single layer forward neural network, knows as a perceptron, was first introduced by (Rosenblatt, 1958). This architecture is basically an algorithm for supersived learning that decides whether an input belongs to a specific class or not.

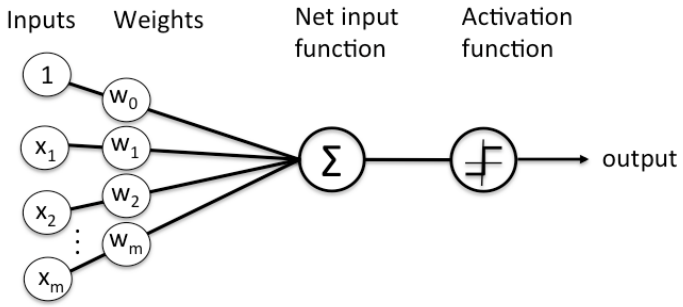


Figure 7: Schematic of a perceptron based on (Rosenblatt, 1958)

This helps illustrating how a neural network works. Such an architecture is composed by a set of inputs connected to a set of outputs by weighted connections. The inputs are multiplied by the weights and summed to produce a single result. Then, an activation function is applied to this result, to be

ultimately transmitted to the outputs. To train a neural networks is to basically find the most adequate values for the weights that, in combination with the inputs and the activation function, would solve a specific problem. The curse of dimensionality (Bellman, 2013) is a situation related to find the mentioned values for the weights. In order to find the ideal value for a weight that would minimize a certain cost, a lot of values should be tested, and as more weights (dimensions) are added to the problem, the number of evaluations increase dramatically, making the problem unsolvable in terms of computation time. The solution for this is, for example, to implement a method called gradient descent, which calculates the minimum of a function. A Gradient Descent Optimizer is used in the implementations of the Q-Network and the Deep Q-Networks of this research. Such a method works jointly with a backpropagation algorithm, to calculate the error contribution of each neuron after the data is being processed.

2.4. Deep Reinforcement Learning Networks

The renaissance of Reinforcement Learning is being contemplated nowadays (Krakovsky, 2016), mainly due to the emerge of Deep Q-Networks (Mnih et al., 2015). The Deep-Q Network framework (Mnih et al., 2015) was motivated due to the limitations of Reinforcement Learning agents when solving real-world complex problems, because they must obtain efficient representations from the inputs and use these to relate past experiences to the next situations the agent will be. The developed framework, a combination of Reinforcement Learning with Deep Neural Networks, was able to effectively learn policies directly from the inputs. More precisely, the agent was tested on 49 classic Atari 2600 games (Bellemare et al., 2012), receiving only the pixels of the image and the game score as inputs, performing with a level comparable to the one of a proffesional human player. In conclusion, the research introduced the first intelligent agent that was able to learn how to solve a set of different tasks with broundbreaking results.

A Deep Convolutional Neural Network was built to approximate an optimal action/value function (See Figure 8). As in with other Reinforcement Learning problems, the objetive of the agent was to select actions to would maximize the cumulative future reward that the agent would receive. Using the same algorithm, network architecure and hyperparameters and by only changing the inputs, pixels and game score, it was demonstrated that the agent was able to robustly learn successful policies for the whole set of different games.

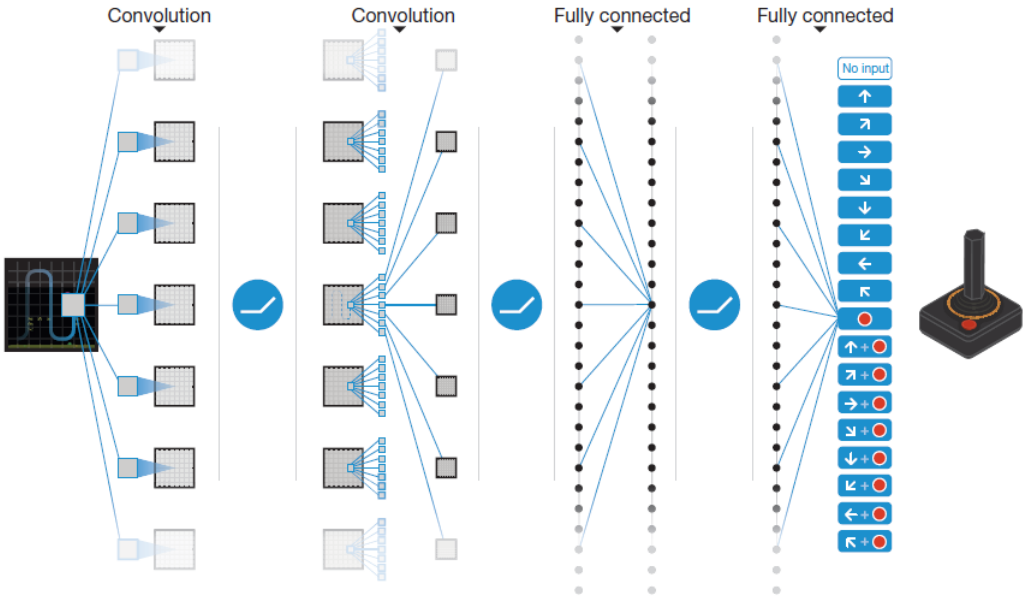


Figure 8: The Deep Q-Network architecture used in (Mnih et al., 2015)

Several extensions have been proposed to the work presented in (Mnih et al., 2015). The adaptation of the Deep Q-Network framework using the Double Q-Learning algorithm has reduced observed overestimations (Van Hasselt et al., 2016). By prioritizing experience replay, a technique that lets the agent review important transitions more frequently, the learning process of the agent can be improved in terms of efficiency (Schaul et al., 2016). In contrast to architectures such as Convolutional Neural Networks or Autoencoders, a new structure, termed Dueling Network Architecture, is introduced to prove the generalization of learning accross actions without performing any change to the Reinforcement Learning algorithm (Wang et al., 2015). All these works have produced agents that perform better in the Atari 2600 games domain in comparison with the original Deep Q-Network. Finally, the original author of Deep Q-Networks (Mnih et al., 2015), introduced four different asynchronous methods for Reinforcement Learning: one-step Q-Learning, one-step Sarsa, n-step Q-Learning and advantage actor-critic, being this last one the algorithm that performed the best in general (Mnih et al., 2016).

2.5. Applications

In an early stage of combining Reinforcement Learning with neural networks, TD-Gammon (Tesauro, 1994) showed that an agent could learn how to play the board game Backgammon by playing against itself and learning from

the results it was obtaining while playing. With no knowledge at the beginning of the learning process, apart from a description of the board state, an implementation based on the Temporal Difference (TD) Reinforcement Learning algorithm (Sutton, 1988) made the agent to play at an intermediate level.

AlphaGo, a computer program that plays the game of Go, defeated the European Go champion by 5 games to 0 on a full-sized 19x19 board, becoming the first computer program in defeating a human professional player in this game (Silver et al., 2016). This classic game has been considered as one of the most challenging games for Artificial Intelligence due to its huge search space. In a 19x19 board there are more possible legal positions (10^{170}) than atoms exist in the entire observable universe (10^{79}) (Ade et al., 2016). Deep Neural Networks trained with a combination of Supervised Learning and Reinforcement Learning made the computer program to accomplish the mentioned achievement.

DeepMind researchers have recently produced groundbreaking results in different publications. For example, a research has taught digital creatures to learn how to navigate accross complex environments (Heess et al., 2017). The inputs are the terrain and a set of movement types with any precomputed motion database and no handcrafted rewards. The output is a set of motions that maximizes the reward, which is to make forward progress. Therefore, the further the agent gets, the higher the reward is. Ultimately, the agent learns to run, jump, crouch and climb in complex terrains with a Deep Reinforcement Learning algorithm, as shown in the next figure:

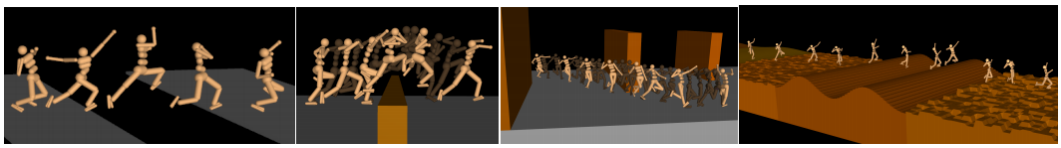


Figure 9: Sequences of the Humanoid performing in different terrains as shown in (Heess et al., 2017)

StarCraft II is a highly technical real-time strategy video game released in 2010. Blizzard Enterntainment, the company that developed the video game, and DeepMind have published a joint paper that presents a Reinforcement Learning environment (Figure 10) based on this game in order to offer a challenging environment for testing Deep Reinforcement Learning algorithms and

architectures (Vinyals et al., 2017). An intelligent agent in this game would have imperfect information due to a partially observed map and long-termed strategies should be undertaken because a bad decision in an initial stage of the game would result in losing the game at the end. To play the game, Deep Reinforcement Learning is used to observe the environment and to choose the best action that would maximize the game score. This reward is provided in two different stages of the game, one while the agent is playing and the other one at the end of the game. A reward is provided to the agent depending on if it wins, tie or loses the game, and throughout the game, a weighted sum of collected resources, upgrades, units and buildings is provided as well to inform the agent if it is progressing adequately.

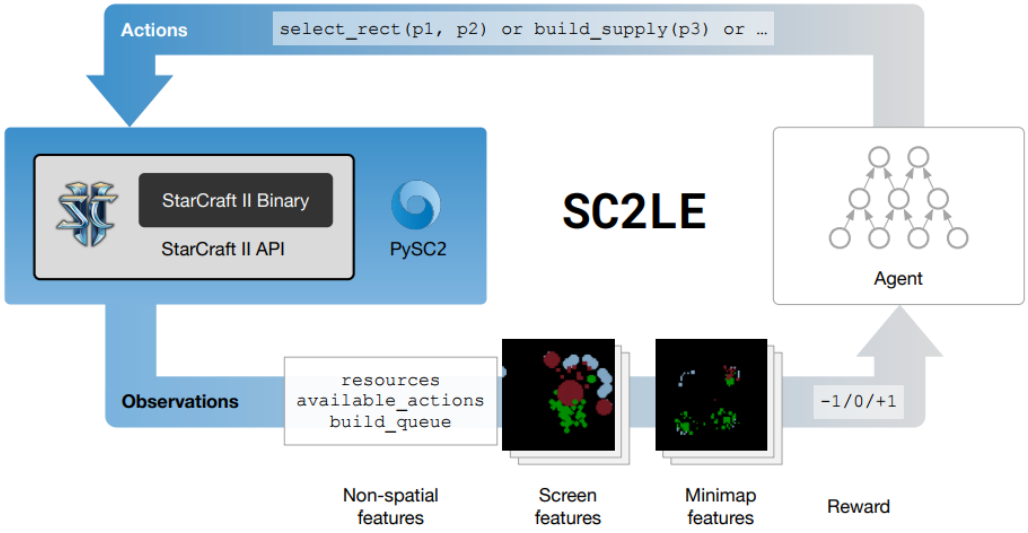


Figure 10: The StarCraft II Learning Environment as shown in (Vinyals et al., 2017)

Another work poses the task for an agent to push boxes onto red target squares to successfully complete a level (Weber et al., 2017). The challenge for the agent is that some actions might lead to irreversible mistakes resulting in the level being impossible of being completed. For example, if the agent stacks a box into a corner, there would be no way of moving that box from that specific state of the environment. The order the agent follows to push the boxes is critical, therefore it must proceed with long-term planning so all the boxes end in the target placeholders. Procedurally generated levels were created to test the agent (See Figure 11). The agent uses imagination, which is a routine to choose not only one action, but entire plans consisting of several steps to ultimately select the one that has best expected reward.

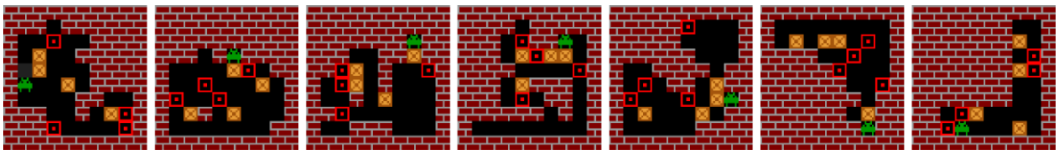


Figure 11: Different Sokoban levels (Weber et al., 2017)

Deep Q-Networks have been applied to other areas, such as the experimentation with simulated robotics tasks or spoken dialogue systems by using Deep Reinforcement Learning Networks to generate dialogues to model future rewards for better coherence and ease of answering. Machine translation, text sequence prediction, neural architecture design or personalized web services are other application scenarios. Less technical areas include healthcare, finance and music generation (Li, 2017).

2.6. Feature Extraction

"Inceptionism" (Mordvintsev et al., 2015*b*) and "DeepDream" (Mordvintsev et al., 2015*a*) are two related projects designed to understand how neural networks work and to ultimately examine what each hidden layer has being able to learn. These works proved that after training a Deep Neural Network with a high number of related images and adjusting the network parameters, each layer progressively contains higher-level features of the image, until reaching the output layer, that ultimately makes a decision on what the image shows. It was concluded that the first hidden layers of a Deep Neural Network trained under these circumnstances, contained low-order features, such as edges or cornes, which are straightforward patterns. Then, the intermediate hidden layers contained information about simple shapes, such as doors or leaves. Finally, the last hidden layers put together this information to form complete figures such as buildings or trees. By feeding an image to a Deep Neural Network as the one described and let the system decide what it sees has created impactating pictures. For example, the next figure is composed by features extracted from the first hidden layers.

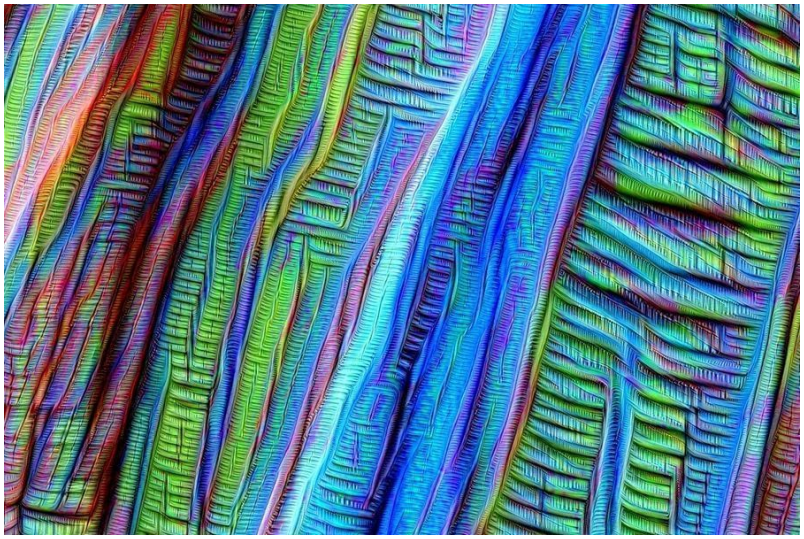


Figure 12: An image composed by low-order features

The next one is composed by features extracted from the last hidden layers, where high-order features reside. Because all the images that were shown to the neural network contained a saxophone being played by a human, the system decided that the human face would part of the musical instrument.

Figure 13: Saxophone Dreams, an image composed by high-order features

Similar techniques have been applied to build applications that by training a Recurrent Neural Network with some kind of data, it will learn to produce similar data related to it. For example, by feeding text to this type of neural network, it will end up producing text that is related to it (Kiros et al., 2015). Music is another example, so if a lot of, for example, classic music compositions are fed to a Recurrent Neural Network, this one will be able to produce new music (Johnson, 2015).

3. Methods

3.1. Overview

This chapter describes the methods that have been applied for the undertaken activities that were necessary to complete the project. First, the problem statement is presented, making emphasis in specifying the problem to solve and in the definition of the environment along with the motivation about why this decision was taken. Later, the cornerstone of the project is introduced: the Q-Learning algorithm. The artificial agent will make use of this algorithm to learn how to solve the proposed environment by using the following structures: a Q-Table that proves the performance of the algorithm, a Q-Network that opens the door of solving the task by using a neural network, a Deep Q-Network which is the fundamental structure of the project. Using a Deep Neural Network trained with the Q-Learning algorithm will allow features to be extracted from different hidden layers of this structure and demonstrate if it is possible to use this information to improve the performance of the agent.

3.2. Implementation

The algorithms and structures have been implemented using the programming language Python 3.5 along with the following external packages: itertools, math, os, time, matplotlib.pyplot (Hunter, 2007), numpy (Walt et al., 2011) and colorama. The Q-Network and the different Deep Q-Networks have been built using TensorFlow 1.2 (Abadi et al., 2016), an open-source software library for Machine Intelligence. The project has been developed using the Python API of TensorFlow on Windows with CPU support.
TensorFlow computations can be represented as data flow graphs where each graph is built as a set of operations. These units of computation take tensors as inputs and output other tensors. Tensors are the main data structure for TensorFlow which can be represented as multidimensional arrays of numbers and are passed between operations in the computation graph. Variables maintain state accross executions of the graph. The weights and biases of the Deep Q-Networks implemented are defined as variables. Placeholders are values that are fed with inputs when TensorFlow runs a computation. The inputs and outputs of the neural networks implemented are defined as placeholders so, for example, the input layer can be fed with different inputs. TensorBoard is a suite of visualization tools that is used to visualize TensorFlow graphs. In

general, TensorFlow models should be reusable.
Summarizing, the fundamental steps to implement a model using TensorFlow are listed below:

1. Define placeholders for inputs and outputs.
2. Define weights and biases as variables.
3. Define the inference model.
4. Define the loss function.
5. Define the optimizer.
6. Initialize variables.
7. Train the model.

The Q matrix of the Q-Table and the weights corresponding to the different neural networks implemented are initialized randomly to normalize the variance of the output of each neuron to 1 by scaling its weight vector by the square root of the number of inputs:
Weights $\sim \bigcup[\frac{-1}{\sqrt{numStates}}, \frac{1}{\sqrt{numStates}}]$, where $\bigcup[-a, a]$ is the uniform distribution in the interval $(-a, a)$ and numStates is the number of inputs of the algorithm (Glorot and Bengio, 2010).
The biases are initially set to 0, because the symmetry between hidden units of the same layer is broken by initializing the weights randomly in the indicated range.

3.3. Problem Statement

The agent must learn how to solve a pathfinding problem, which is to find the shortest route between two points. The agent will start in an initial state and its objective will be to learn over time how to arrive at a specific goal state with the addition of learning how to avoid the a set of bad states (holes) that are present in the environment.

'S'	'F'	'F'	'F'
'F'	'H'	'F'	'H'
'F'	'F'	'F'	'H'
'H'	'F'	'F'	'G'

Table 1: Original Open AI Gym Frozen Lake Environment

In order to represent the pathfinding problem, it is fundamental to define an environment which will be composed of a set of states, actions and definitions of rewards in relation to the states. The OpenAI Gym Frozen Lake Environment (Brockman et al., 2016). provides inspiration with its representation. This offered environment is composed of 4x4 matrix and 4 possible types of states (defined by characters) for every element in the matrix: 'S', start state, 'F', frozen floor, 'H', hole and 'G', goal state (Table 1). The actions to take from every state are the logical cardinal points of a 2 dimensional system: north/up, south/down, right/east and left/west. An agent introduced in this environment will always start in 'S' and will have to learn how to arrive to 'G', which will provide a positive feedback, by dodging the 'H' states, which will provide negative rewards.

Instead of using the hardcoded and original Frozen Lake environment from the Open AI organization, the ideas from it were taken and translated to a personal and customized representation. The description of the environment that will be used throughout the implementation is a Frozen Lake replica, a 4x4 matrix composed by the letters 'A' to 'P' in alphabetical order. Along with it, to complete the meaning of the environment, the initial, goal and holes states are designated. More precisely, and to act as the Frozen Lake environment does, it is stated that 'A' is the initial state, 'P' is the goal state, and the holes are 'F', 'H', 'L' and 'M'.

'A' [1]	'B'	'C'	'D'
'E'	'F' [2]	'G'	'H' [2]
'I'	'J'	'K'	'L' [2]
'M' [2]	'N'	'O'	'P' [3]

[1] Yellow: Initial state.

[2] Red: Holes.

[3] Green: Goal State.

Table 2: Environment: Frozen Lake replica

Given this environment, two fundamental 16x4 matrices are automatically generated: the R and Q matrices. Both will contain a row for every state and a column for each possible action to take, making them 16x4 matrices. The Q matrix initializes every value with random numbers in the range $[\frac{-1}{\sqrt{numStates}}, \frac{1}{\sqrt{numStates}}]$, and will ultimately represent the memory of

what the agent has learned through many experiences . The R is computed as follows:

Algorithm 1 Matrix R generation

```
environment ←
[('A','B','C','D'),
('E','F','G','H'),
('I','J','K','L'),
('M','N','O','P')]
numStates ← 16
numActions ← 4
R ← empty[numStates][numActions]

for i = 0; i < environmentCols < i++ do
    for i = 0; i < environmentRows < i++ do
        if i − 1 ≥ 0 then
            R[i][j] ← environment[i − 1][j]
        else
            R[i][j] ← environment[i][j]
        if i + 1 < environment.shape[0] then
            R[i][j] ← environment[i + 1][j]
        else
            R[i][j] ← environment[i][j]
        if j + 1 < environment.shape[1] then
            R[i][j] ← environment[i][j + 1]
        else
            R[i][j] ← environment[i][j]
        if j − 1 ≥ 0 then
            R[i][j] ← environment[i][j − 1]
        else
            R[i][j] ← environment[i][j]
```

Therefore, the R matrix contains for every state, the next state resulting in taking all of the available actions. In this sense, the first row of the matrix, corresponding to the state 'A', will lead to itself (like it has not moved or crashed into a wall) if the "up" action is taken, to 'E' with the "down" action, to 'B' with the "right" action and to 'A' again if the agent takes the action "left". A similar procedure is followed with every other state, resulting in the

aforementioned R matrix, which will allow the agent to move with meaning accross the environment.

	↑	↓	→	←
From 'A'	'A'	'E'	'B'	'A'
From 'B'	'B'	'F'	'C'	'A'
From 'C'	'C'	'G'	'D'	'B'
From 'D'	'D'	'H'	'D'	'C'
From 'E'	'A'	'I'	'F'	'E'
From 'F'	'B'	'J'	'G'	'E'
From 'G'	'C'	'K'	'H'	'F'
From 'H'	'D'	'L'	'H'	'G'
From 'I'	'E'	'M'	'J'	'I'
From 'J'	'F'	'N'	'K'	'I'
From 'K'	'G'	'O'	'L'	'J'
From 'L'	'H'	'P'	'L'	'K'
From 'M'	'I'	'M'	'N'	'M'
From 'N'	'J'	'N'	'O'	'M'
From 'O'	'K'	'O'	'P'	'N'
From 'P'	'L'	'P'	'P'	'O'

Table 3: Matrix R generated from Environment (Table 2)

3.4. Q-Learning

The Q-Learning algorithm works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and consequently following the optimal policy thereafter. The pseudocode of the implementation the Q-Learning algorithm is presented next. Variations of this method have been implemented to solve the same proposed problem using different structures: a Q-Table, a Q-Network, a Deep Q-Network and a Deep Q-Network using Feature Extraction.

Algorithm 2 Q-Learning

```
1: while episodes < numEpisodes do
2:     state ← initialState
3:     stepsPerEpisodes ← 0
4:     accumulatedReward ← 0
5:     goalReached ← False
6:
7:     while stepsPerEpisode < 99 && goalReached == False do
8:         action ← actionWithMaxQValue from state
9:         if random(0, 1) < ε then
10:            action ← actionRandom from state
11:
12:        nextState ← state taking action
13:        qValues ← qValues from nextState
14:        maxQValue ← max(qValues)
15:
16:        if nextState == goalState then
17:            reward ← +1
18:            goalReached ← True
19:        if nextState == hole then
20:            reward ← −1
21:        accumulatedReward ← accumulatedReward + reward
22:
23:        oldQValue ← qValue from state taking action
24:        newQValue ← oldQValue + α ⋆ (reward + (γ ⋆ maxQValue)) −
   oldQValue
25:        oldQValue ← newQValue
26:
27:        state ← nextState
28:        stepsPerEpisode ← stepsPerEpisode + 1
29:
30:    episodes ← episodes + 1
```

Essentially, the algorithm works with the basic information of the environment (the initial, goal and holes states) and the R and Q matrices previously computed.

The agent will have a precise number of episodes to learn how to find the shortest path from the initial to the goal state, and in each of these episodes it will have a number of steps to move through the environment in its search of the goal and to continuously set the values for the Q matrix.

It starts by picking one action from the state it is in at the moment. This action will be either the best one as calculated until that moment or a random one. This decision is taken based on the ϵ-greedy policy. By setting ϵ to a specific value, the agent will have probability ϵ to choose an action at random in order to explore, otherwise (with probability $1 - \epsilon$) to choose a greedy action, which translates to exploit the action with the maximum value. This policy keeps a balance between exploration (taking random actions) and exploitation (taking best actions at that moment) in the discovery process of the environment. Further details regarding this policy will be discussed in Chapter 5.

Once the action has been selected, thanks to the R matrix, the next state that the agent will be in can be known. By knowing the next state, the Q values associated to that are known as well, so it is possible to obtain the maximum Q value that will be used later in the fundamental Q-Learning formula. Because it is known in which next state the agent will be in the next time step, it is possible to assign a reward, positive or negative, depending on the nature of that state. For the goal state, a positive reward of $+1$ is given, and for any of the holes, a negative reward of -1 is given with the objective to inform the agent that these states are good or bad. For any other state, no reward is provided. All the computed rewards in an episode are stored for statistic purposes so in the future the summatory of them can be divided by the total number of episodes to produce the accumulated reward over time.

Next, the Q-Learning formula is computed, by setting the new entry value in the matrix Q. This is calculated from the old corresponding entry in matrix Q added by the multiplication of the learning rate and the addition of the rewards and the multiplication of a discount parameter, γ, and the maximum Q value for all actions in the next state.

Finally, if the computed next state is not the goal state, the exploration of the environment needs to continue, so the next state is assigned as the current state. The algorithm is closed by incrementing the number of steps taken in this episode and the number of episodes elapsed until this moment.

The algorithm will ultimately replace all the random initialized values of

the Q matrix, that will represent the minimum or shortest path from any state to the goal state. In order to test this, from each state, the maximum value corresponding to an action will be taken, leading the agent to a next state. This procedure is repeated until the agent reaches the goal state, so it is possible to know every state required in order to reach the goal. After a training process as such, the Q-Learning algorithm attempts to provide optimal routes from any state to the goal state, known as a policy.

3.5. Q-Table

The Q-Table structure faithfully implements the Q-Learning technique (Algorithm 2) described previously. The resulting Q matrix proves that the agent learns over time how to reach the goal state in a set of minimum steps.

The required inputs for this structure to execute the aforementioned algorithm are as follows:

- Environment matrix to set the R and Q matrices.
- Initial state, goal state and holes states.
- Alpha, α, (learning rate) and gamma, γ, (discount factor) values.
- Number of episodes to train the agent.

And the outputs after the training process:

- 16x4 Q matrix.
- Training time.
- Accumulated reward over time.
- First episode with reward +1.
- Rewards per episode plot.
- Steps per episode plot.

The implementation of the Q-Table is straightforward but it is not a scalable structure. While it is suitable for a small environment as the one solved, it is not feasible to be applied to a real-world environment which will be hypothetically composed by hundreds of states and possible actions. In order to solve this issue, and because it is necessary some way to take a description of our state, and produce Q-values for actions without a table, it is possible to use a neural network acting as a function approximator, by taking any number of possible states that can be represented as a vector and learn to map them to Q-values.

3.6. Q-Network

To solve the proposed problem with a more flexible structure such as a neural network, a one-layer network (Figure 14) will be used. This network takes every state encoded in a unique 1x16 vector (Table 4), and produces a vector of 4 Q-values, one for each action, as an output. Such a neural network acts kind of like a glorified table, with the weights connecting the input neurons with the output neurons serving as the old cells of the Q matrix. Such a structure is easily expandable with more layers and different activation functions. This would have been nearly impossible to achieve with a simple Q-Table.

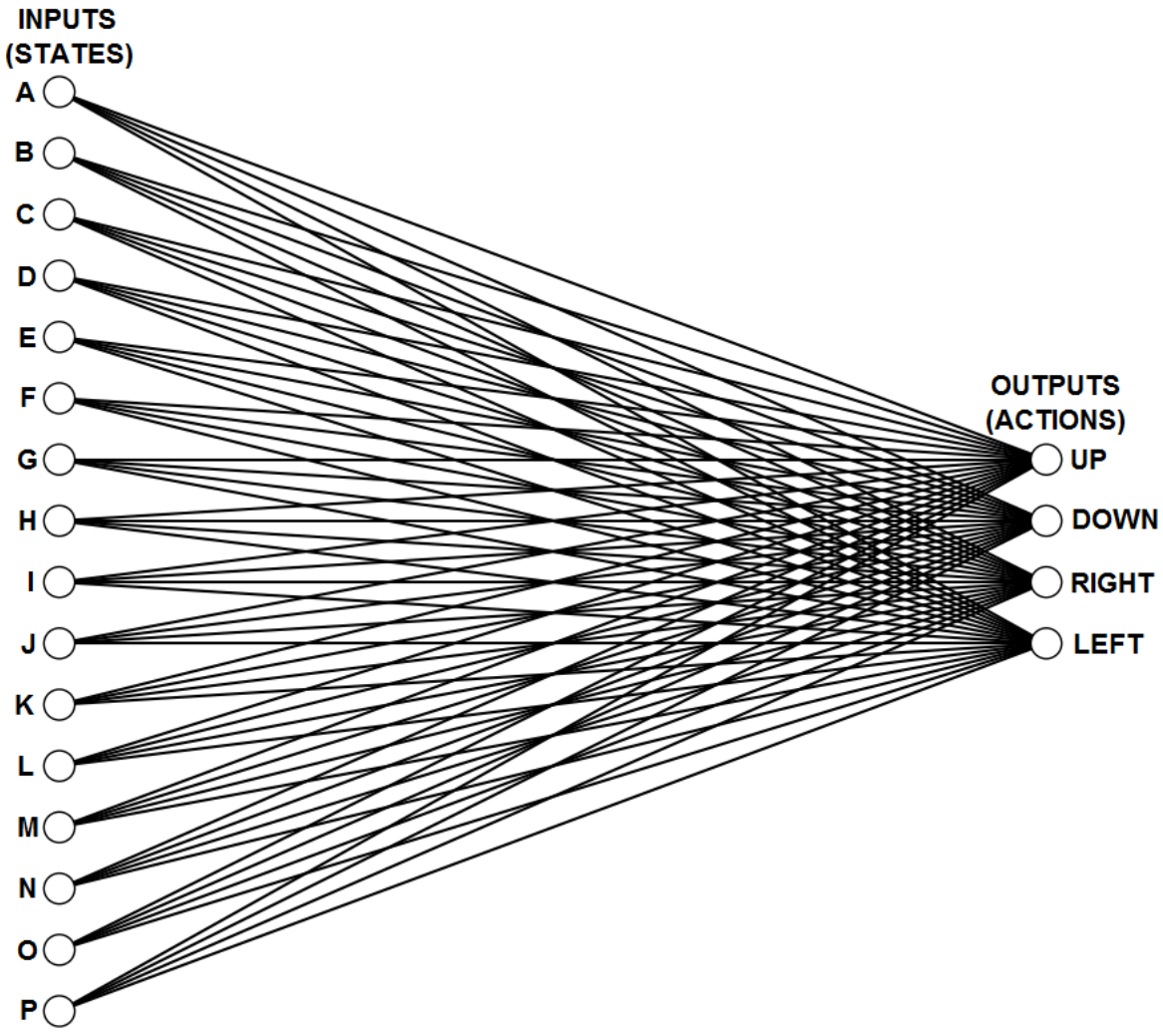


Figure 14: 16x4 Q-Network, with the weights acting like entries in the Q-Table

State	Input: 1x16 vector
A	[**1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
B	[0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
C	[0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
D	[0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
E	[0, 0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
F	[0, 0, 0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
G	[0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0, 0]
H	[0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0, 0]
I	[0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 0, 0, 0, 0]
J	[0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 0, 0, 0]
K	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 0, 0]
L	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0, 0]
M	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0, 0]
N	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0, 0]
O	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**, 0]
P	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, **1**]

Table 4: Q-Network inputs for each state, in the form of 1x16 vectors

Next, the Tensorboard generated graph for the 16x4 Q-Network is presented. This graph presents in a visual way how the neural network is structured. The input 'X' and the output 'Y' are placeholders which will be fed with data when the model is trained. The weights are defined as a TensorFlow variable. The rest of the graph shows the loss function and the gradient descent optimizer that are used:

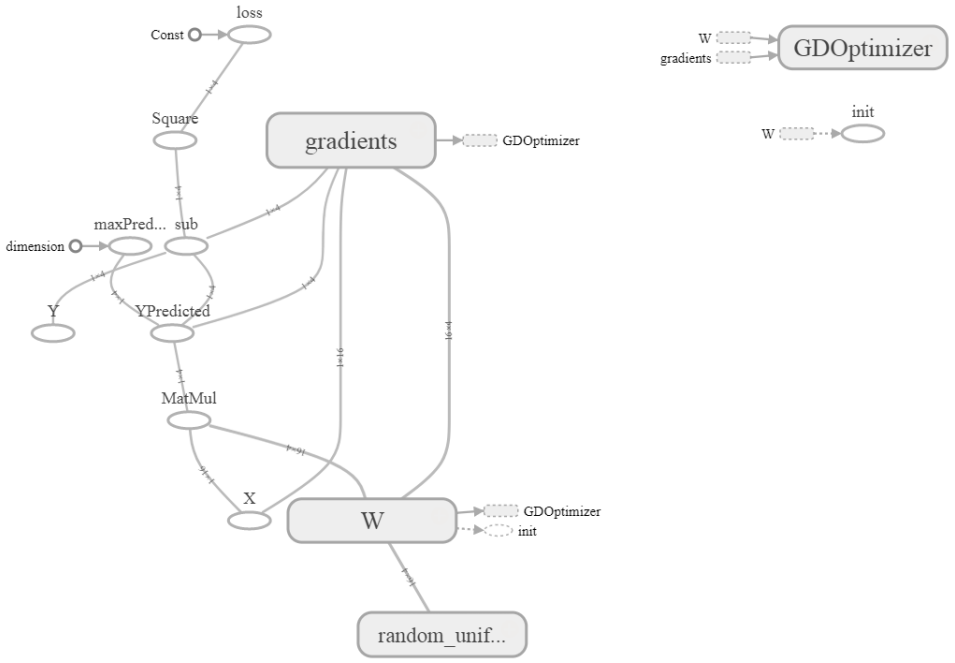


Figure 15: Q-Network graph generated in TensorBoard

The method of updating will be modified to be adapted to this structure, by using backpropagation and a loss function. Instead of directly updating the Q matrix, the loss function is defined as the sum-of-squares loss (with the Tensorflow function "reduce_sum"), where the difference between the output and the predicted output is computed. In this case, the target Q value for the chosen action is the equivalent to the new Q-value computed in the algorithm 2. Finally, the agent is trained with a gradient descent optimizer in order to minimize the loss.

The inner working of the neural network can be visualized as only one neuron being activated in the input layer, the one corresponding to a specific state, while the rest of input neurons are set to 0 (Figure 16). This will cause only the weights connected to that activated neuron to actually propagate the information of the training process. The inputs and the weights are multiplied and then the hyperbolic tangent activation function (Figure 17) is applied to that resulting value to finally set the value of the output neurons. This process goes along during the whole training, where the agent is moving from state to state by the activation of the corresponding neurons in the input layer and updating the Q-values in the ouput layer.

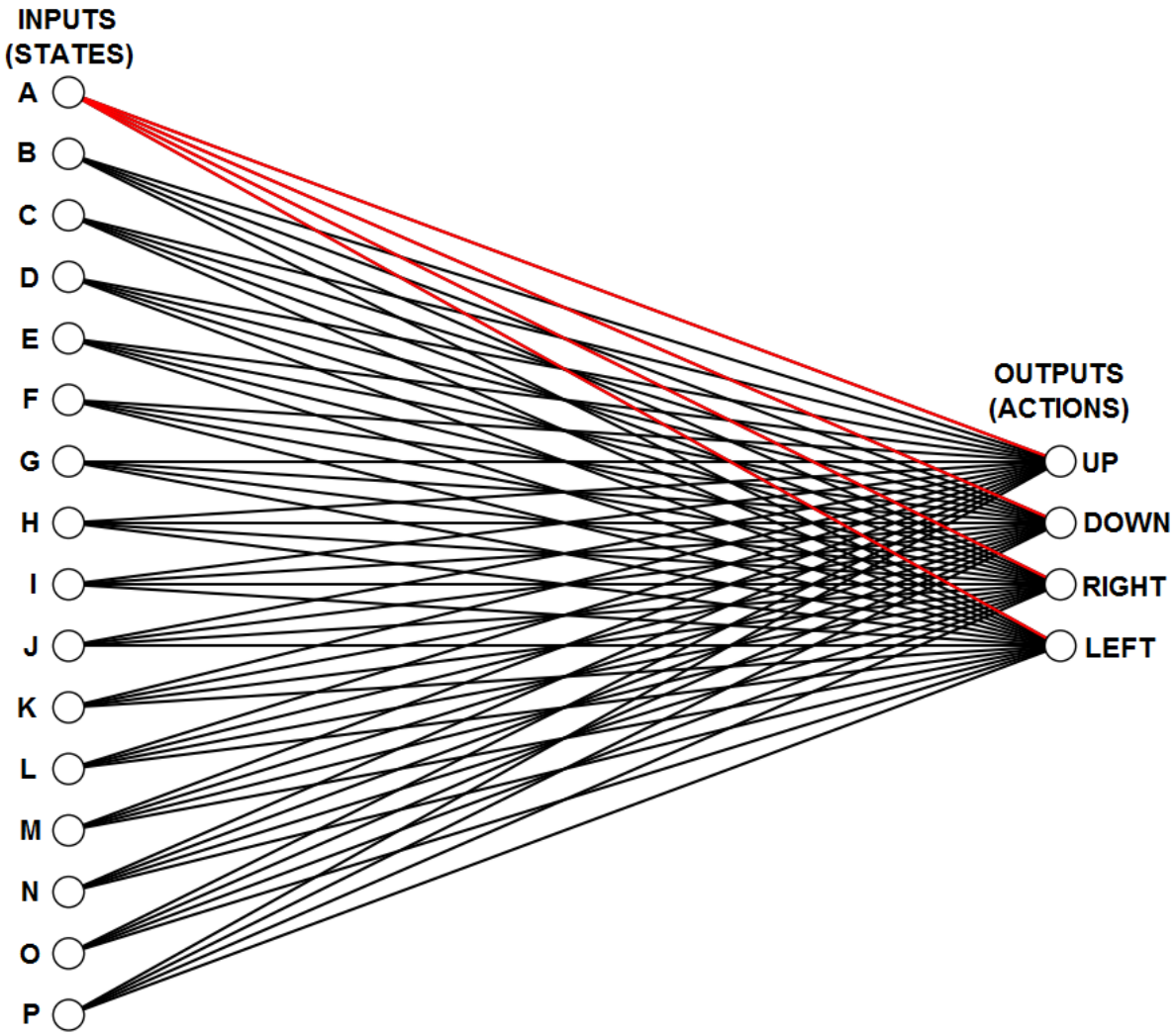


Figure 16: 16x4 Q-Network with state 'A' activated

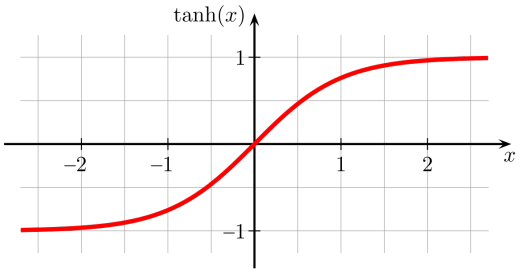


Figure 17: Hyperbolic tangent activation function

Summarizing, the inputs for the algorithm to work with the defined neural network are:

- Environment matrix to set the R matrix.

- Initial state, goal state and holes states.
- Alpha, α, (learning rate) and gamma, γ, (discount factor) values.
- Number of episodes to train the agent.
- Neural network:
  - Inputs: 16 (as a Tensorflow placeholder).
  - Outputs: 4 (as a Tensorflow placeholder).
  - Hidden layers: 0.
  - Weights: 16x4, connecting input neurons with output neurons (as a TensorFlow variable), randomly initialized in the range $[\frac{-1}{\sqrt{numStates}}, \frac{1}{\sqrt{numStates}}]$.

The obtained outputs after the execution of the algorithm in the 16x4 neural network are the following:

- 16x4 Q matrix (trained weights between the input layer and the output layer).
- Activations of the neurons of the output layer from each state.
- Training time.
- Accumulated reward over time.
- First episode with reward +1.
- Rewards per episode plot.
- Steps per episode plot.

With these outputs, there are two ways to test if the artificial agent finds the shortest path from every state to the goal. First, the weights of the neural network produce a 16x4 table equivalent to the Q matrix obtained using the Q-Table structure. Secondly, the activations of the neurons of the output layer from each state returns a similar table. Using either of these two tables, it is possible to choose the maximum value from each state, corresponding to one specific action, that will lead the agent to move from state to next state accross the environment to reach the goal state in an optimal way.

3.7. Deep Q-Network

The Q-Network that implements the Q-Learning algorithm serves as a base to build the Deep Q-Network. In this sense, the input (Table 4) and output

layers of the Deep Q-Network are similar to the ones implemented for the Q-Network. In order to have a Deep Neural Network, it is necessary to include hidden layers between the input and output layers. In relation to the number of inputs and outputs and to have a representative number of hidden layers, a first hidden layer with 12 neurons was added, connected to a second hidden layer composed by 8 neurons connected to the last hidden layer composed only by 2 neurons (Figure 18). The activation function of every neuron in the hidden and output layers continues to be the hyperbolic tangent function (Figure 17).

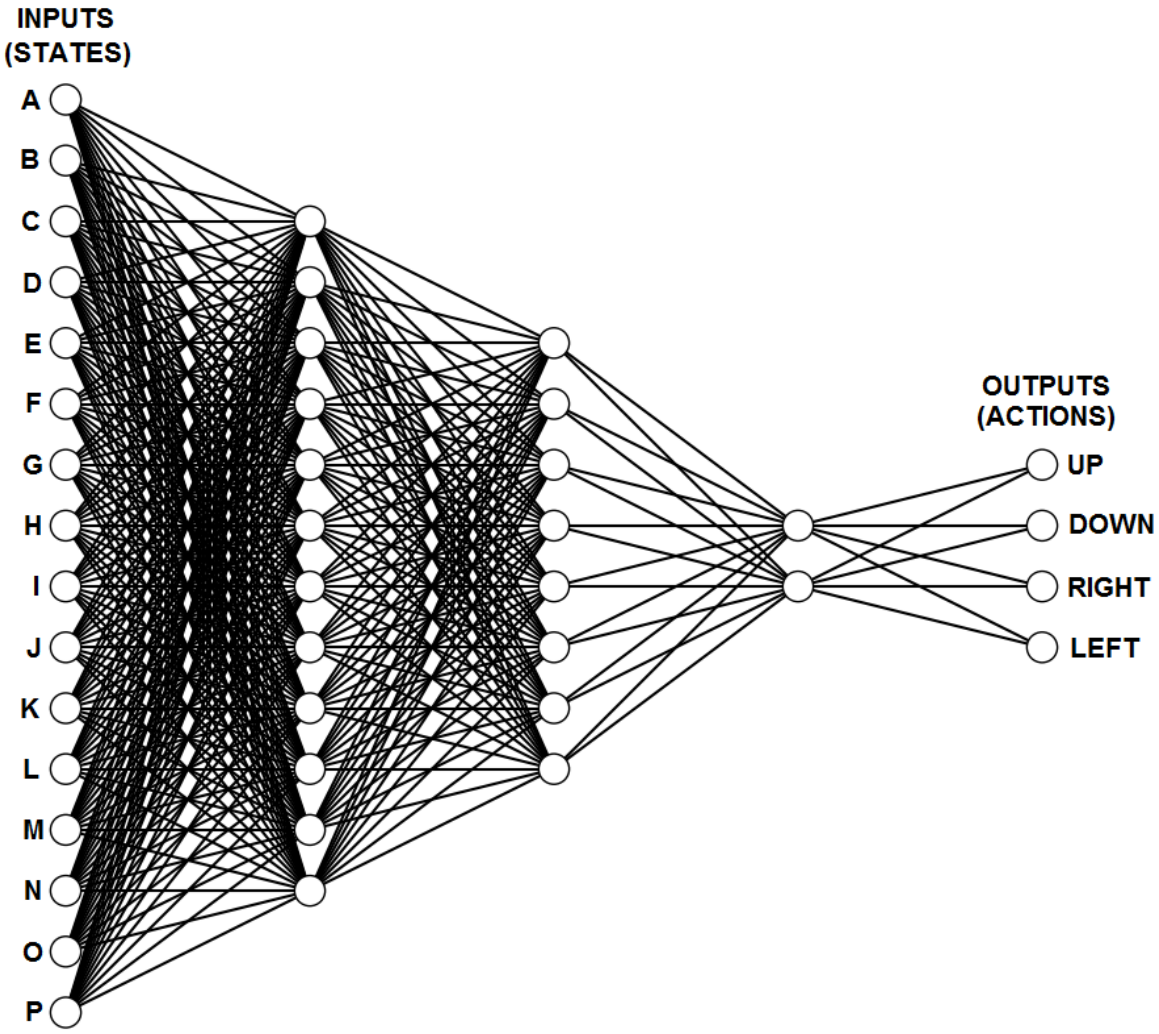


Figure 18: 16x12x8x2x4 Deep Q-Network

Next, an excerpt of the Tensorboard generated graph for this Deep Q-Network is presented. This excerpt of the graph shows the first part of the graph. The input placeholder is labeled as 'X', which is multiplied by the

weights and this result is added to the biases. These computations will ultimately compose the definition of the first hidden layer:

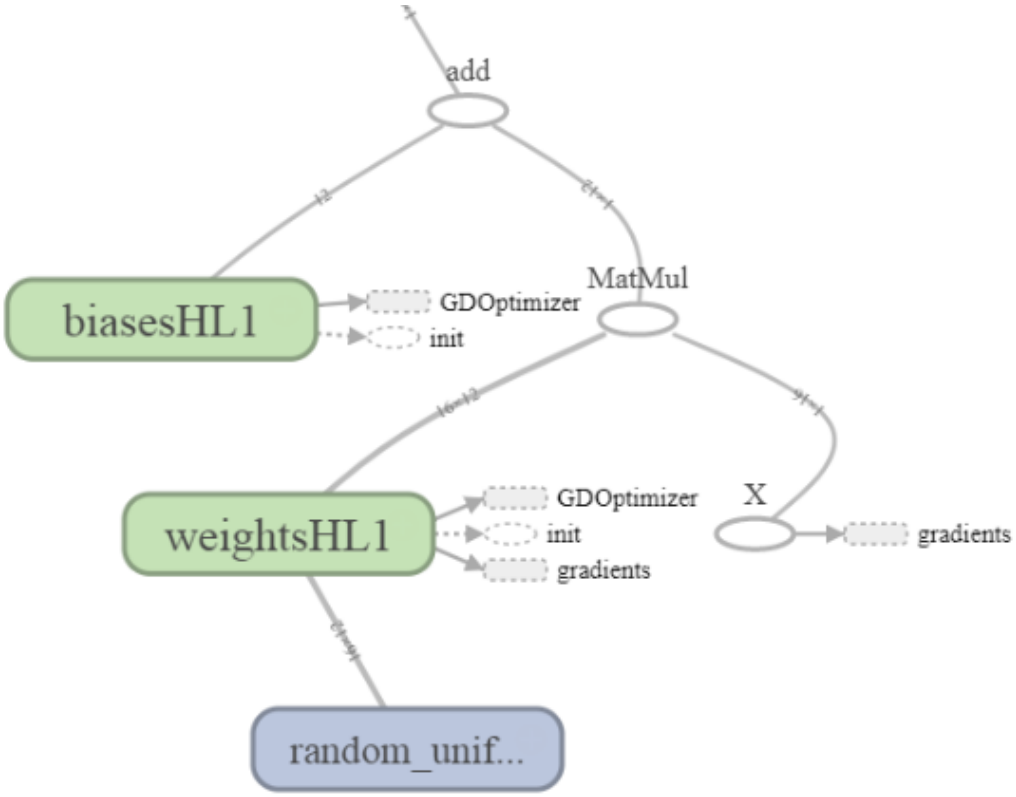


Figure 19: Excerpt of the Deep Q-Network graph generated in TensorBoard

The rest of the structure that composes the Deep Q-Network follows a similar pattern, defining the other hidden layers until the output layer. The graph ends with subgraph that is similar to the one obtained for the Q-Network (Figure 15).

The execution of the algorithm in this structure works in a similar way to the Q-Network, with the difference that with a Deep Neural Network the information is propagated through the weights of the whole network: from the inputs to the first hidden layer, then to the second hidden layer, then to the last hidden layer to finally arrive to the output layer.

This structure allows to study what is exactly happening in the hidden layers of the Deep Q-Network. After the agent is trained, it is possible to check the activations of the neurons of each hidden layer from every state. In this sense, features are going to be extracted via the activations of the neurons of

specific hidden layers. The activations of the neurons are related to the activation function used, in this case, the hyperbolic tangent function. A neuron activation will be in the range provided by the activation function, providing a wide range of "intensity" in the activation. For the stated purposes, it is only essential to check if a specific neuron is activated or not so the "level" of activation is not required. The threshold of activation of a neuron using the tanh activation function is 0.0, because it is a zero-centered function, so if an activation value is above the threshold, the neuron will be considered to be activated and set with a value of 1. On the contrary, if an activation value is below the threshold, the neuron will be considered as not activated and set with a value of 0.

The aforementioned extracted features are the collection of neurons both activated and not activated from a specific hidden layer and state. Extracted features from the first hidden layer are going to be used to build a new deep neural network with a number of inputs equal to the existing number of neurons of the first hidden layer of this network. The features that are extracted from the last hidden layer will prove that the neurons that compose this layer contain encoded information regarding the best action to take from each state. In conclusion, the system will end up with a collection of features from the first hidden layer and from the last hidden layer for each input state.

Summary of inputs for the algorithm using a Deep Q-Network:

- Environment matrix to set the R matrix.
- Initial state, goal state and holes states.
- Alpha, α, (learning rate) and gamma, γ, (discount factor) values.
- Number of episodes to train the agent.
- Deep Neural Network:
    - Inputs: 16 (as a Tensorflow placeholder).
    - Outputs: 4 (as a Tensorflow placeholder).
    - Hidden layers: 3 (12, 8 and 2 neurons).
    - Weights (as TensorFlow variables): 16x12, connecting input neurons to the neurons of the first hidden layer; 12x8, connecting neurons of the first hidden layer to the neurons of the second hidden layer; 8x2, connecting neurons of the second hidden layer to the

neurons of the last hidden layer and 2x4, connecting neurons of the last hidden layer to the output neurons. The weights are initialized with random values in the range $[\frac{-1}{\sqrt{numStates}}, \frac{1}{\sqrt{numStates}}]$.

- Biases (as TensorFlow variables): 12 for the first hidden layer, 8 for the second hidden layer, 2 for the last hidden layer and 4 for the output layer. The biases are initialized with values equal to 0.

Collection of outputs after training the agent with Q-Learning in a Deep Q-Network:

- 16x4 Q matrix (by performing ordered matrix multiplications between all the weights between every layer of the network: 16x12 $\rightarrow$ 12x8 $\rightarrow$ 12x2 $\rightarrow$ 2x4 = 16x4).
- Activations of the neurons of the first hidden layer from each state.
- Activations of the neurons of the last hidden layer from each state.
- Activations of the neurons of the output layer from each state.
- Training time.
- Accumulated reward over time.
- First episode with reward +1.
- Rewards per episode plot.
- Steps per episode plot.

3.8. Deep Q-Network using Feature Extraction

A new Deep Neural Network is built to prove if the features extracted from the first hidden layer of the previous Deep Neural Network contain a codification of the inputs and these can be used to improve the performance of the agent. Thus, this Deep Neural Network can be considered as a reduced version of the previous one. It is composed by an input layer with 12 neurons (which was the size of the first hidden layer of the previous network), connected to a first hidden layer of 8 neurons, which is connect to a second hidden of 2 neurons to end up with a connection to the output layer with 4 neurons as it has been established during the whole process. This new Deep Neural Network contains one layer less than the previous one (Figure 20).

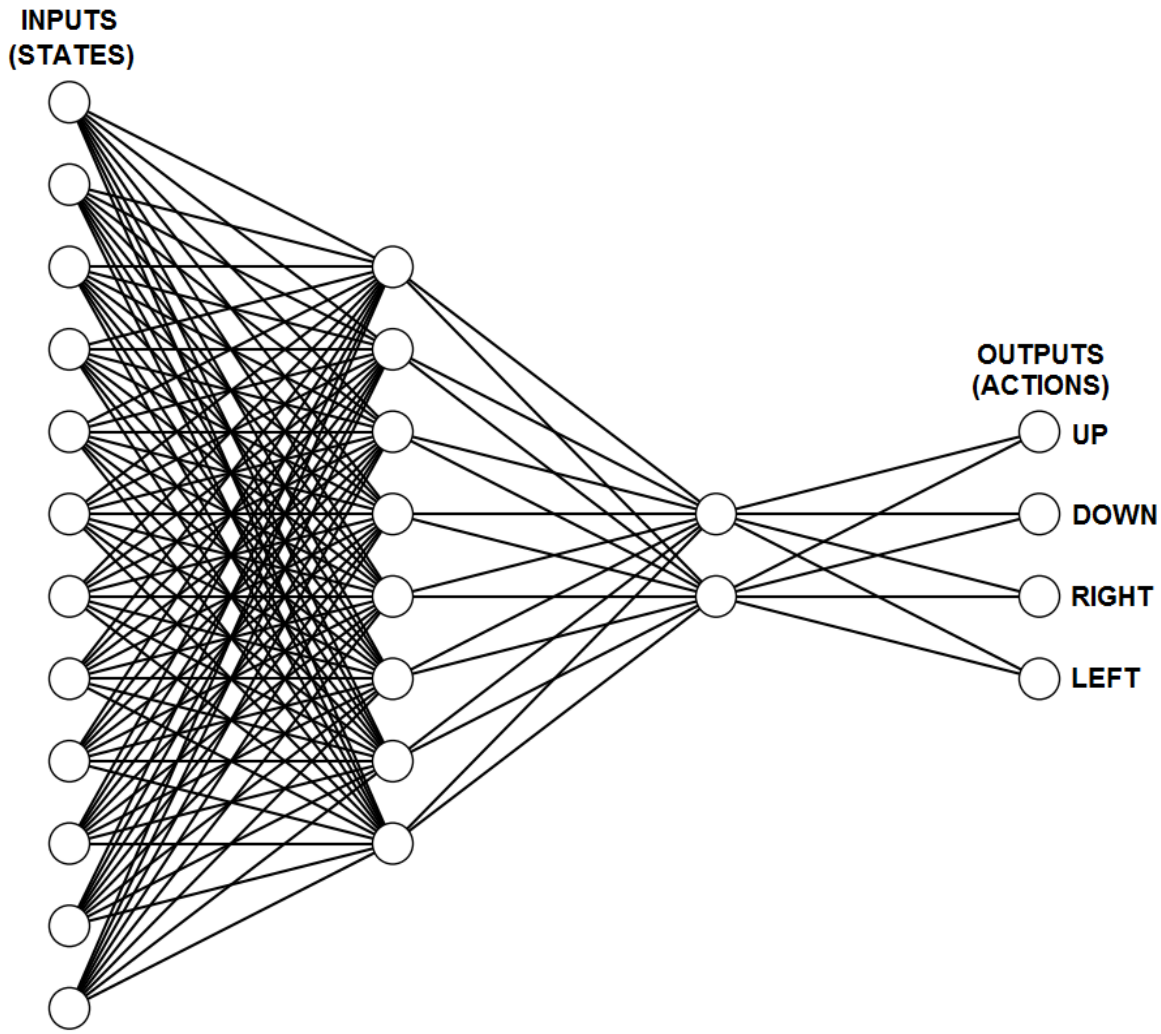


Figure 20: 12x8x2x4 Deep Q-Network that uses features extracted from the previous Deep Q-Network (Figure 18) as inputs (Table 5)

Because in this Deep Neural Network the input layer is composed by 12 neurons and the environment is consists of 16 states, it is not possible to have an identity vector with only one value set to 1, because it would not be possible to cover all the states. Therefore, in this structure, the relation of every neuron corresponding to one state is no longer valid. Instead of this, after the training process of the previous Deep Q-Network, every state is going to have a specific and unique codification in a form of a 1x12 vector. As seen in Chapter 4, a previously obtained codification for the states could be the following (Table 19).

State	Input: 1x12 vector
A	[0, 0, **1**, **1**, 0, **1**, 0, 0, 0, 0, 0, **1**]
B	[**1**, **1**, **1**, **1**, 0, 0, 0, **1**, **1**, **1**, 0, **1**]
C	[0, 0, 0, **1**, 0, **1**, 0, 0, 0, **1**, 0, 0]
D	[**1**, **1**, 0, **1**, **1**, 0, 0, **1**, **1**, **1**, 0, **1**]
E	[0, 0, **1**, 0, **1**, **1**, 0, **1**, **1**, **1**, **1**, 0]
F	[0, 0, **1**, 0, 0, **1**, 0, 0, 0, 0, 0, **1**]
G	[0, 0, 0, **1**, **1**, **1**, **1**, 0, 0, **1**, **1**, **1**]
H	[**1**, **1**, 0, **1**, **1**, **1**, **1**, 0, **1**, **1**, **1**, **1**]
I	[**1**, 0, **1**, 0, 0, 0, 0, 0, **1**, 0, 0, **1**]
J	[0, **1**, **1**, 0, 0, 0, 0, 0, 0, 0, 0, **1**]
K	[0, 0, **1**, 0, **1**, **1**, **1**, 0, 0, 0, **1**, **1**]
L	[0, 0, **1**, 0, 0, 0, **1**, 0, 0, 0, **1**, 0]
M	[**1**, 0, 0, 0, 0, 0, 0, **1**, **1**, 0, **1**, 0]
N	[0, **1**, **1**, 0, 0, **1**, 0, **1**, 0, 0, 0, 0]
O	[**1**, **1**, **1**, **1**, 0, 0, 0, 0, 0, **1**, 0, 0]
P	[0, **1**, **1**, 0, **1**, **1**, **1**, 0, 0, **1**, 0, **1**]

Table 5: An example of features extracted and used as inputs for the new Deep Q-Network

In conclusion, the size of the input layer of this Deep Neural Network is smaller than the number of states. The objective is to prove if it is possible to use a codification of the space of states in order to improve the performance of the agent. Its purpose is to solve the proposed problem while providing optimal solutions in terms of finding the path between every state and the goal state while dodging holes states. This reduced Deep Q-Network works in a similar way as the previous Deep Q-Network did. The only difference is how the inputs are encoded, in this occasion, in a 1x12 vector for each state. Therefore, this structure runs the real environment so the agent can explore it and exploit it accordingly in order to solve the proposed problem while improving its learning process in comparison with the original Deep Q-Network.

Inputs for the Deep Q-Network using Feature Extraction to be trained with Q-Learning:

- Environment matrix to set the R matrix.
- Initial state, goal state and holes states.

- Alpha, α, (learning rate) and gamma, γ, (discount factor) values.
- Number of episodes to train the agent.
- (Reduced) Deep Neural Network:
  - Inputs: 12 (as a Tensorflow placeholder).
  - Outputs: 4 (as a Tensorflow placeholder).
  - Hidden layers: 2 (8 and 2 neurons).
  - Weights (as TensorFlow variables): 12x8, connecting inputs neurons to the neurons of the first hidden layer; 8x2, connecting neurons of the first hidden layer to the neurons of the last hidden layer and 2x4, connecting neurons of the last hidden layer to the output neurons. The weights are initialized with random values in the range $[\frac{-1}{\sqrt{numStates}}, \frac{1}{\sqrt{numStates}}]$.
  - Biases (as TensorFlow variables): 8 for the first hidden layer, 2 for the last hidden layer and 4 for the output layer. The biases are initialized with values equal to 0.

Outputs from the algorithm:

- Activations of the neurons of the last hidden layer from each state.
- Activations of the neurons of the output layer from each state.
- Training time.
- Accumulated reward over time.
- First episode with reward +1.
- Rewards per episode plot.
- Steps per episode plot.

3.9. Experiments

Experiments and test results are obtained after executing the programs whose source codes are included in Appendix C. The programs contain the definition of the environment, all the necessary parameters for the algorithms to work as well as the different neural networks implemented.

The programs are executed separately and their results are collected in order to be compared and consequently obtain conclusions. The plots are

automatically generated in separate windows so they can be saved as independent files. The results can be visualized in a standard console or in the output window of the IDE used (PyCharm).

Two excerpts of the output after executing SourceCode/4FeatureExtraction2.py are shown next. The first one shows the beginning of the execution of the program, which contains the environment definition, the learning process of the agent and part of the Q matrix obtained when the training has finished:

```
Environment:
 [['A' 'B' 'C' 'D']
 ['E' 'F' 'G' 'H']
 ['I' 'J' 'K' 'L']
 ['M' 'N' 'O' 'P']]
Initial state: A
Goal state: P
List of holes: ['F' 'H' 'L' 'M']

Training:
Episode: 0
Episode: 250
Episode: 500
Episode: 750
Episode: 1000
Episode: 1250
Episode: 1500
Episode: 1750

Testing with Q Matrix
Final Q matrix (weights of the NN):
 [[ 0.11270706  1.08008623  0.8618331   0.31807625]
 [-0.12777585 -2.03694272  0.37852359  0.18320942]
 [ 0.10443239  1.23663199  0.40505299  0.13686296]
 [-0.22047545 -2.79394102 -0.54948497 -0.16632441]
 [ 0.00769221  0.78145456 -1.12205148 -0.4520154 ]
 [ 0.15907921  1.14292741  1.85304499  0.70433486]
 [-0.06654987  1.02806032 -3.28831887 -1.30282485]
 [-0.24927329 -2.16552973 -2.27866888 -0.85294449]
 [-0.09398424 -2.06306577  1.22081649  0.5128144 ]
```

Figure 21: Excerpt of the output that shows an initial stage of the execution of SourceCode/4FeatureExtraction2.py

The second one shows a later stage of the program, where features from the last hidden layer of a Deep Q-Network have been extracted and used to prove

that the neurons that compose this hidden layer contain information about the best action to take from each state:

```
Activations of neurons of the last hidden layer:
From A : [[-0.14063105 -0.9149282 ]]
From B : [[-0.61730015 -0.06342645]]
From C : [[ 0.09992401 -0.92369145]]
From D : [[-0.45017555  0.3306694 ]]
From E : [[ 0.54442418 -0.87041849]]
From F : [[-0.51119542 -0.92407042]]
From G : [[ 0.86381656 -0.8406288 ]]
From H : [[ 0.41988307  0.09407438]]
From I : [[-0.79078138 -0.08819551]]
From J : [[-0.44108674 -0.87319785]]
From K : [[ 0.7021237  -0.93112528]]
From L : [[ 0.2602469  -0.95867485]]
From M : [[ 0.15797597  0.18723182]]
From N : [[-0.09801647 -0.85488081]]
From O : [[-0.91985303 -0.91604149]]
From P : [[-0.00773647 -0.49829587]]
[[0, 0], [0, 0], [1, 0], [0, 1], [1, 0], [0, 0], [1, 0], [1, 1], [0, 0], [0, 0], [1, 0], [1, 0], [1, 1], [0, 0], [0, 0], [0, 0]]

Unique activations of neurons of the last hidden layer:
[[0, 0], [1, 0], [0, 1], [1, 1]]

Theory demonstrated with this combination of actions:
Group of states 0 taking this action: Right
Group of states 1 taking this action: Down
Group of states 2 taking this action: Left
Group of states 3 taking this action: Up
```

Figure 22: Excerpt of the output that shows a later stage of the execution of SourceCode/4FeatureExtraction2.py

4. Results

4.1. Overview

The outputs produced by the training of the artificial intelligent are presented in this chapter. The agent learns how to solve a pathfinding problem, which consists in finding the shortest path between two specific states within an environment. To make it more compelling, an additional characteristic is introduced into the environment, which is to have some states considered as holes, that hypothetically would cause damage the agent. The Q-Learning algorithm (Algorithm 2) has been implemented in the structures described in Chapter 3, consequently producing the next results:

- Q matrix (except for the Deep Q-Network using Feature Extraction).
- Activations of the neurons of the output layer (except for the Q-Table)
- Training time.
- Accumulated reward over time.
- First episode with reward equal to +1.
- Plot: Rewards per episode.
- Plot: Steps per episode.
- Activations of the neurons of the first hidden layer (only for the Deep Q-Network).
- Activations of the neurons of the last hidden layer (only for the Deep Q-Network).

Finally, to specifically answer the original research question and meet the established objectives, an emphasis is made in relation with the features extracted from the Deep Q-Network. The features are presented to analyze their meaning and studied to understand how can they be used to ultimately check if they can improve the performance of the agent.

Every structure is analyzed separately and an overall comparison collecting all the results together is made at the end of the chapter. The same parameters are going to be used in every structure:

Parameter	Value
Maximum number of episodes	2000
Maximum number of steps per episode	99
Learning rate (α)	0.05
Discount factor (γ)	0.99
ϵ value	0.1
Reward for reaching the goal state	+1
Reward for reaching any hole	−1
Activation function for hidden and output neurons	Hyperbolic tangent (tanh)
Loss function	$\sum((Y - Ypredicted)^2)$
Optimizer	Gradient descent

Table 6: Parameters for Algorithm 2 and configuration of the neural networks

4.2. Q-Table

The output that will allow the testing of the agent is the Q matrix that is computed during the training of the agent. This matrix contains values for every state in relation to how good is to take each action. In order to test the agent to check if it finds the shortest path between any state and the goal state, the maximum value for every state must be retrieved, which will lead to a specific next state. This procedure continues until the agent reaches the goal state, constructing the whole path that it has taken.

The 16x4 Q matrix obtained after training the agent with a Q-Table structure and the previously described parameters (Table 6) is presented next. The maximum values for every action are highlighted in green:

	Up	Down	Right	Left
A	8.61270399e-010	1.73994020e-008	1.73994020e-008	8.61270399e-010
B	1.73994020e-008	-4.99996485e-002	3.51503071e-007	8.61270399e-010
C	3.51503071e-007	7.10107214e-006	2.94453588e-307	1.73994020e-008
D	7.21484904e-310	-4.99999991e-002	7.21484904e-310	3.51503071e-007
E	8.61270399e-010	3.51503071e-007	-4.99996485e-002	1.73994020e-008
F	1.73994020e-008	7.10107214e-006	7.10107214e-006	1.73994020e-008
G	3.51503071e-007	1.43456003e-004	-4.99999991e-002	-4.99996485e-002
H	1.73994020e-008	-4.71018989e-002	-2.22382222e-001	7.10107214e-006
I	1.73994020e-008	-4.99928989e-002	7.10107214e-006	3.51503071e-007
J	-4.99996485e-002	1.43456003e-004	1.43456003e-004	3.51503071e-007
K	7.10107214e-006	2.89810107e-003	-4.71018989e-002	7.10107214e-006
L	-4.99999991e-002	5.85474963e-002	-6.55822456e-002	1.43456003e-004
M	3.51503071e-007	-4.99928989e-002	1.43456003e-004	-4.99928989e-002
N	7.10107214e-006	1.43456003e-004	2.89810107e-003	-4.99928989e-002
O	1.43456003e-004	2.89810107e-003	5.85474963e-002	1.43456003e-004
P	-3.58111597e-002	1.72676693e-001	-2.19744389e-001	-1.83903685e-001

Table 7: Q-Table Q Matrix

Using the Q matrix, and starting from 'A', it is possible to see that the maximum value is related to taking the action 'Down', which will lead the agent to be in the state 'E'. It is necessary to proceed with this protocol until the agent reaches the goal state.

It is interesting to see, for example, that being in 'B' and taking the action 'Down' has a very negative value associated, because this action would lead the agent to be in a hole, something it is intended to avoid.

The policy generated for the agent from every state using the Q matrix is presented below:

From	Optimum path
A	['E', 'I', 'J', 'K', 'O', 'P']
B	['C', 'G', 'K', 'O', 'P']
C	['G', 'K', 'O', 'P']
D	['C', 'G', 'K', 'O', 'P']
E	['I', 'J', 'K', 'O', 'P']
F	['G', 'K', 'O', 'P']
G	['K', 'O', 'P']
H	['G', 'K', 'O', 'P']
I	['J', 'K', 'O', 'P']
J	['K', 'O', 'P']
K	['O', 'P']
L	['P']
M	['N', 'O', 'P']
N	['O', 'P']
O	['P']
P	['P']

Table 8: Q-Table testing: optimum paths following the Q matrix

This table shows the path from every state to the goal state. It is possible to observe that every path is optimal and it does not traverse through any hole, so the agent has learned how to find the shortest path from every state to goal while learning which states are the ones to avoid (marked as holes). Because the Q-Learning algorithm does not end an episode if the agent goes to a hole, these states have values that can lead the agent to the goal state properly. Even in the goal state, 'P', the best action to take is 'Down', meaning to not move from this state, because it is the goal and provides positive rewards to the agent. Taking the action 'Right' when the agent is in 'P' should provide a similar positive reward for the agent, but the reward by taking this action is negative (Table 7). This is caused because the action 'Right', whose value is initialized randomly, has not been sufficiently explored, because the agent has exploited the action 'Down', which provided positive rewards.

The agent has been able to solve this problem in a very short amount of time, even if the maximum number of episodes for the training process was 2000. In addition to the training time, two other numerical results are

obtained. First, the accumulated reward calculated over time, which is the summatory of all the rewards received in every episode, divided by the maximum number of episodes. Lastly, the first episode (being each episode a set of 99 steps at maximum) in which the agent received a reward equal to +1. These three results are presented next:

Training time	0 minutes, 1.65 seconds
Accumulated reward over time	86.55 %
First episode with reward equal to +1	12

Table 9: Q-Table statistics

The accuracy or accumulated reward over time can be considered high and could be even higher if no random actions were taken at all after the agent has learned the best path to the goal state. The agent finds the goal the first time in episode 12, which will start setting all the positive values in relation with the actions that take the agent to the goal state. These two statistics can be checked in the following plot, in which it is possible to see that at the very beginning, the agent has various negative rewards because it is still learning the best path. After episode 12 and until the end, the reward in every episode is mostly +1, but sometimes it is possible to observe that it is equal to 0 or −1. Even if the agent already knows the path to the goal in these episodes, random actions can lead it to holes in sporadic occasions, reducing the resulting reward in an episode.

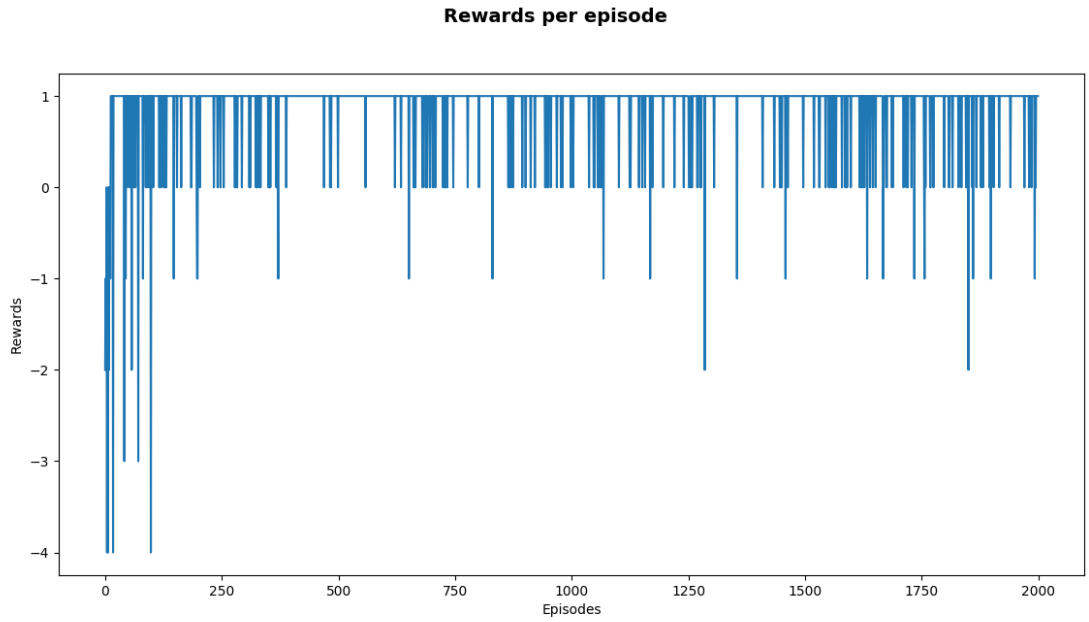


Figure 23: Q-Table: Rewards per episode

The next plot shows how the agent takes a lot of steps in the first episodes because it still does not know the best path to reach to the goal state, but when it learns it, almost the rest of times, it takes optimal paths.

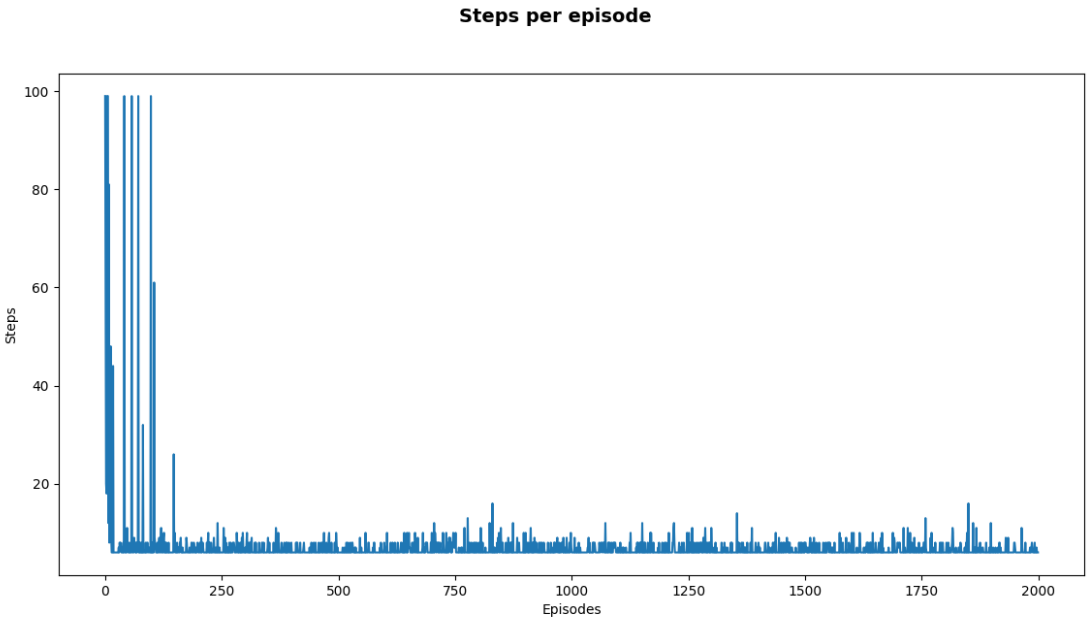


Figure 24: Q-Table: Steps per episode

4.3. Q-Network

Solving the proposed problem with a Q-Network produces very similar result to the ones obtained with a Q-Table. This occurs because the neural network is actually some kind of a table with the same dimensions, using the weights connecting the input to the output neurons as the cells of a Q matrix. In this occasion it is possible to test the agent by using two different tables.

A 16x4 Q matrix, which are exactly the values corresponding to the weights:

	Up	**Down**	**Right**	**Left**
A	0.87580311	0.52094007	1.44828343	1.01644969
B	0.97737134	-0.20034854	1.52596784	0.88082212
C	0.98716962	1.62527239	0.7693491	0.99946612
D	0.00736772	-0.38933271	0.19350925	1.02084148
E	0.01478865	0.6930114	-0.24566148	-0.12084872
F	-0.00694567	0.25637355	1.2308085	0.01657992
G	0.99313748	1.76283491	-0.2185134	-0.1948283
H	-0.18298721	-0.1195878	-0.05103004	1.14237034
I	0.00703762	-0.32230383	0.89658362	-0.22783643
J	-0.15105955	-0.05780063	1.28761721	0.04853328
K	1.10667384	1.98221397	-0.24305129	0.94758511
L	-0.22833858	0.04533289	-0.18693197	1.10799718
M	-0.19145203	-0.06614321	0.33902135	-0.06280524
N	0.18414959	0.2304166	1.09517455	-0.10044809
O	1.1239028	1.13279629	2.63423586	0.80130965
P	-0.15552884	-0.09863257	-0.249165	0.09514409

Table 10: Q-Network Q Matrix

And using the activations of the neurons of the output layer from each state. These are in the range $[-1, 1]$ due to the activation function that it is used in the output layer.

	Up	Down	Right	Left
A	0.7043106	0.47842523	0.89535296	0.76841652
B	0.75192577	-0.19771025	0.90973204	0.70683098
C	0.75615293	0.92538542	0.64655083	0.76136988
D	0.00736759	-0.37078482	0.19112951	0.77020913
E	0.01478757	0.59991306	-0.24083607	-0.12026382
F	-0.00694556	0.25090045	0.84281361	0.0165784
G	0.75869709	0.94281876	-0.21510068	-0.19240004
H	-0.18097177	-0.11902094	-0.05098579	0.81521076
I	0.0070375	-0.31158856	0.71463031	-0.22397432
J	-0.14992093	-0.05773634	0.8585012	0.0484952
K	0.80288357	0.96274918	-0.23837572	0.73868787
L	-0.22445121	0.04530185	-0.18478461	0.80335331
M	-0.18914665	-0.06604692	0.32660341	-0.06272279
N	0.18209586	0.22642362	0.79875904	-0.1001116
O	0.80892217	0.81197417	0.98974997	0.66476834
P	-0.15428682	-0.09831396	-0.24413356	0.09485803

Table 11: Q-Network activations of the neurons of the output layer

It is possible to observe that the Q matrix and the activations of the neurons of the output layer of the Q-Network are very similar, therefore both methods are effective to test the agent.

'L' and 'P' are not producing optimal results because the holes states and the goal state are not always properly trained. In the case of the holes, these are not trained in depth because as long they have been explored to a certain extent, when the agent is exploiting the environment, which it does the 90 % of times, it will never take an action that will lead to a hole, because these will have bad rewards associated. Regarding the goal state, when the agent finally reaches this state, it will only have 1 opportunity to take an action in order to explore actions from it. There is a high probability that these opportunities will let the agent to explore which are the right actions to remain in the goal state, but this might not always occur. Having stated this, it has been observed that in other executions of the training process these two states, along with the rest of them (including the other holes), produce optimal results, but due to the discussed reasons, it is not guaranteed that this will always happen like this.

Following both tables the same testing of the agent is obtained:

From	Optimum path
A	['B', 'C', 'G', 'K', 'O', 'P']
B	['C', 'G', 'K', 'O', 'P']
C	['G', 'K', 'O', 'P']
D	['C', 'G', 'K', 'O', 'P']
E	['I', 'J', 'K', 'O', 'P']
F	['G', 'K', 'O', 'P']
G	['K', 'O', 'P']
H	['G', 'K', 'O', 'P']
I	['J', 'K', 'O', 'P']
J	['K', 'O', 'P']
K	['O', 'P']
L	['K', 'O', 'P']
M	['N', 'O', 'P']
N	['O', 'P']
O	['P']
P	['O', 'P']

Table 12: Q-Network testing: optimum paths following both the Q matrix and activations of the neurons of the output layer

It is considered important to remark that the training is more accurate for the states that are not holes or the goal state, and even more for the states that are part of the optimal path for the initial state. This can be observed in other executions of the training process when using a neural network and a Deep Neural Network to solve the proposed task.

The statistics obtained by the Q-Network are very similar to the Q-Table, although the algorithm takes more time to be completed, due to the intrinsic and more complex nature of a neural network. Regarding the accumulated reward over time and the first successful episode, these are very high and occurring in an early episode respectively:

Training time	0 minutes, 22.89 seconds
Accumulated reward over time	86.55 %
First episode with reward equal to +1	2

Table 13: Q-Network statistics

The plot collecting the rewards per episode clearly shows that the agent is capable of finding the goal state almost at the beginning of the process and from there, exploiting the best stores values in the weights, continuing to complete successful episodes.

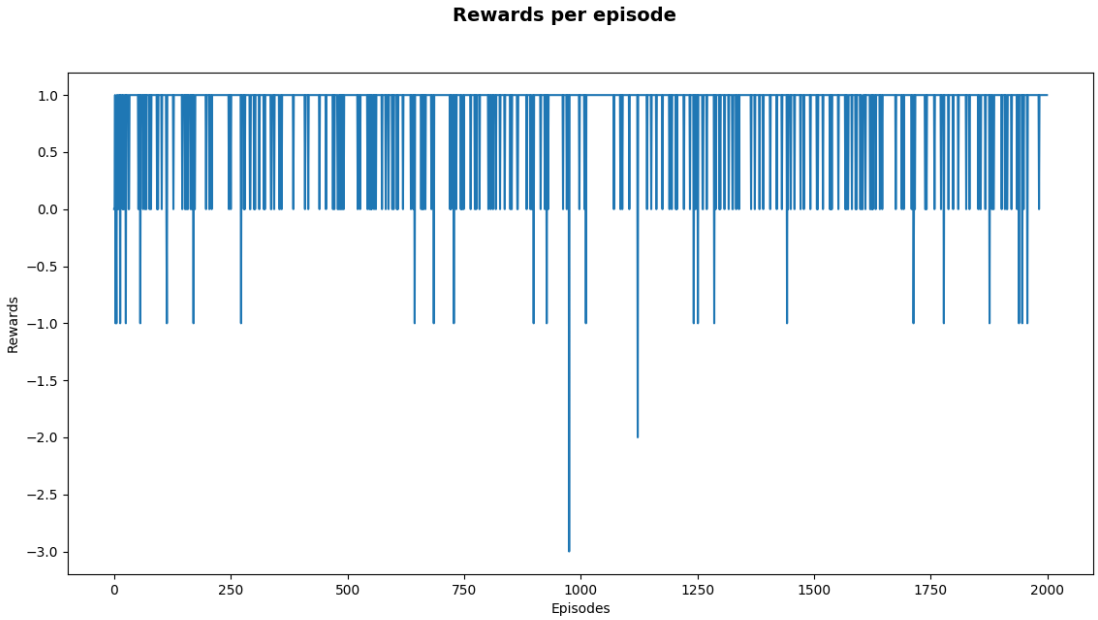


Figure 25: Q-Network: Rewards per episode

The steps per episode show a more progressive learning than the one obtained using the Q-Table. This occurs at the beginning of the process, until it finally stabilizes at 6 steps per episode on average, which is the optimal amount of steps required to go from the initial to the goal state.

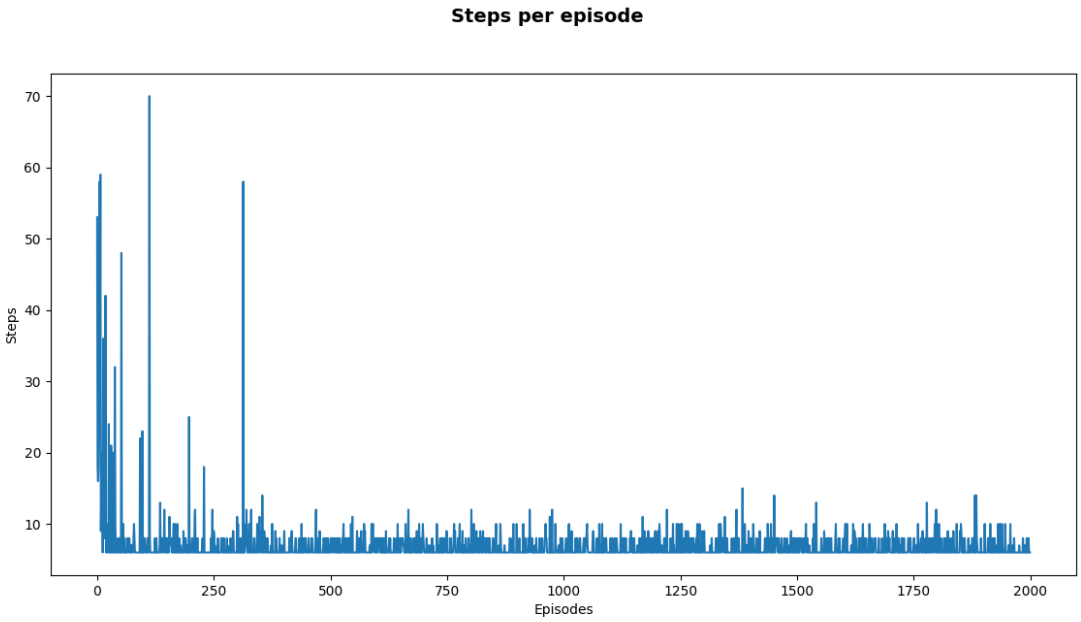


Figure 26: Q-Network: Steps per episode

4.4. Deep Q-Network

The Deep Q-Network is the fundamental structure that it is used to meet the objectives of the research. It allows the agent to be trained properly as it has been achieved using the Q-Table and a simple 16x4 Q-Network but it will provide an opportunity to obtain more information about how the whole process is done. This information will be encoded in the neurons of the hidden layers of the Deep Neural Network, and it will be used to obtain crucial conclusions.

First, it is always necessary to test the agent to check if the learning process has been successful or not. The 16x4 matrix obtained by multiplying the weights of the Deep Neural Network in order and the derived paths from every state are presented next:

	Up	Down	Right	Left
A	2.10795179e-01	1.46476805e+00	1.49352956e+00	6.68452144e-01
B	-7.18600675e-02	-2.63468862e+00	1.20140886e+00	3.54345083e-01
C	1.95102572e-01	1.69919431e+00	1.10720181e+00	5.25039434e-01
D	-1.72432631e-01	-2.33754063e+00	-3.09033394e-01	-2.36149073e-01
E	-1.61554456e-01	6.13288164e-01	-2.53521156e+00	-9.85610604e-01
F	1.95810899e-01	1.20293355e+00	1.51370025e+00	6.63937092e-01
G	-3.20679605e-01	1.27640152e+00	-5.07960129e+00	-1.97250009e+00
H	-2.75257498e-01	-1.67953134e+00	-2.13704371e+00	-9.36444461e-01
I	-5.02483323e-02	-2.68394804e+00	1.51429272e+00	4.77255464e-01
J	2.06698075e-01	9.82188940e-01	1.82827151e+00	7.79276490e-01
K	-8.55929554e-02	1.50377202e+00	-2.28750968e+00	-8.43607247e-01
L	-4.71949205e-03	1.13172185e+00	-9.66291845e-01	-3.32480907e-01
M	-7.76585713e-02	-1.57713103e+00	2.80876279e-01	3.66196632e-02
N	4.16768901e-02	1.21204719e-01	4.30187970e-01	1.78076759e-01
O	3.79172057e-01	-3.77094746e-01	5.09922409e+00	2.02360439e+00
P	-8.87625292e-03	6.35033131e-01	-6.21001720e-01	-2.18111724e-01

Table 14: Deep Q-Network Q Matrix

From	Optimum path
A	['B', 'C', 'G', 'K', 'O', 'P']
B	['C', 'G', 'K', 'O', 'P']
C	['G', 'K', 'O', 'P']
D	['D']
E	['I', 'J', 'K', 'O', 'P']
F	['G', 'K', 'O', 'P']
G	['K', 'O', 'P']
H	['D', 'D']
I	['J', 'K', 'O', 'P']
J	['K', 'O', 'P']
K	['O', 'P']
L	['P']
M	['N', 'O', 'P']
N	['O', 'P']
O	['P']
P	['P']

Table 15: Deep Q-Network testing: optimum paths following the Q matrix

By checking the testing results, it is possible to observe that either 'D' or 'H' (because 'H' leads to 'D'), do not find the path to reach to 'P'. By examining the environment, it can be noticed that 'D' is a very particular state within the environment. It is placed on a corner that leads to nowhere and it is not by any chance in the path from the initial to the goal state. Because every episode always starts in the initial state, if the agent, at the beginning of the process, has explored the environment by taking random actions, and after that has exploited the best values for the next states to take, 'D' is probably not going to be sufficiently explored neither exploited. This does not mean that states with the same attributes as 'D' are never going to find an optimal path to the goal state, because ideally, all the environment should be explored by the agent, but it clearly states that a specific part of an environment may not be enough explored. This could be addressed by implementing a decaying ϵ-greedy policy as it is discussed in Chapter 5.

It is possible to see that using the activations of the neurons of the output layer as a testing method, 'D' is not properly explored either because the optimal action for 'D' to take would always be 'Left', to go 'C' and then from

there lead itself to the goal:

	Up	Down	Right	Left
A	0.87359774	0.94727916	0.9410547	0.89397633
B	0.84240854	-0.10408835	0.94417679	0.87657779
C	0.87083179	0.96243829	0.89804375	0.87059712
D	0.8231684	-0.29529324	0.81449246	0.79939753
E	0.83412647	0.9482829	-0.11398733	0.60210496
F	0.87490898	0.93328196	0.95748514	0.90562695
G	0.80476195	0.91941011	-0.78719276	0.29608893
H	0.79713869	-0.01463005	-0.05146434	0.56059307
I	0.84701461	-0.02572082	0.95755035	0.8897624
J	0.87742352	0.9040696	0.97589904	0.92302197
K	0.84976214	0.97735375	0.22490261	0.69321179
L	0.85846114	0.97233886	0.61500812	0.77973211
M	0.85076255	0.26229379	0.95170254	0.88717526
N	0.86962312	0.87720579	0.95548117	0.90106493
O	0.88632089	0.68991834	0.99686003	0.96317834
P	0.8601625	0.95337701	0.7666707	0.81779569

Table 16: Deep Q-Network activations of the neurons of the output layer

From	Optimum path
A	['E', 'I', 'J', 'K', 'O', 'P']
B	['C', 'G', 'K', 'O', 'P']
C	['G', 'K', 'O', 'P']
D	['D']
E	['I', 'J', 'K', 'O', 'P']
F	['G', 'K', 'O', 'P']
G	['K', 'O', 'P']
H	['D', 'D']
I	['J', 'K', 'O', 'P']
J	['K', 'O', 'P']
K	['O', 'P']
L	['P']
M	['N', 'O', 'P']
N	['O', 'P']
O	['P']
P	['P']

Table 17: Deep Q-Network testing: optimum paths following the activations of the neurons of the output layer

Another interesting detail can be highlighted from this testing result is that by using the Q matrix, 'A' had 'Right' as the best action to take, therefore moving to 'B', but with the activations of the neurons of the output layer, the agent thinks that taking 'Down' is better. Analyzing the environment proves that either 'B' and 'E' will produce same optimal paths for the initial state, and furthermore, as it is possible to see, the values for 'Down' and 'Right' for 'A' are almost identical, very high and only differ by a small amount, clearly indicating that both actions are really good options for the agent. In comparison, 'Up' and 'Left' have values below 0.90, far away from the values of 'Down' and 'Right'. Ultimately, it is possible to observe that testing either with the Q matrix and the activations of the neurons of the output layer are effective methods, although the final results differ at some points. This happens because the activations of the neurons of the output layer have the activation function involved, when the Q matrix is merely the consecutive multiplication of the weights between each layer of the network.

What it can be considered more surprising is that the statistics received

after the training process are way worse than the ones obtained with the previous structures. The algorithm has lasted more than 7 minutes to be completed, with a very negative accumulated reward and completing successfully an episode for the first time in a very late stage of the training process. This can be better understood by taking a look at the usual plots of rewards and steps per episode provided next.

Training time	7 minutes, 35.50 seconds
Accumulated reward over time	-82.25 %
First episode with reward equal to +1	1203

Table 18: Deep Q-Network statistics

Due to the complex structure of the Deep Neural Network, in terms of a high number of neurons, hidden layers and consequently weights connecting these layers, an agent can take a long time to solve a problem as the one proposed. Basically until approximately the episode 1500, the agent has been exploring the environment and exploiting it by taking action it considered to be adequates, but failing continuously in finding its way to the goal state. This is actually the learning process happening for the agent, because during this whole time, all the weights have been carefully tuned with the Reinforcement Learning algorithm to lead the agent to finally be able to solve the problem and continuously doing it afterwards thanks to the learning process. It is important to remark that in other executions of the Deep Q-Network, the agent can learn hot to solve in less time, but it is always around or more than 5 minutes. This would mean that the episode in which it first received a reward equal to +1 would be before than receiving it around episode 1500, but never before than episode 750.

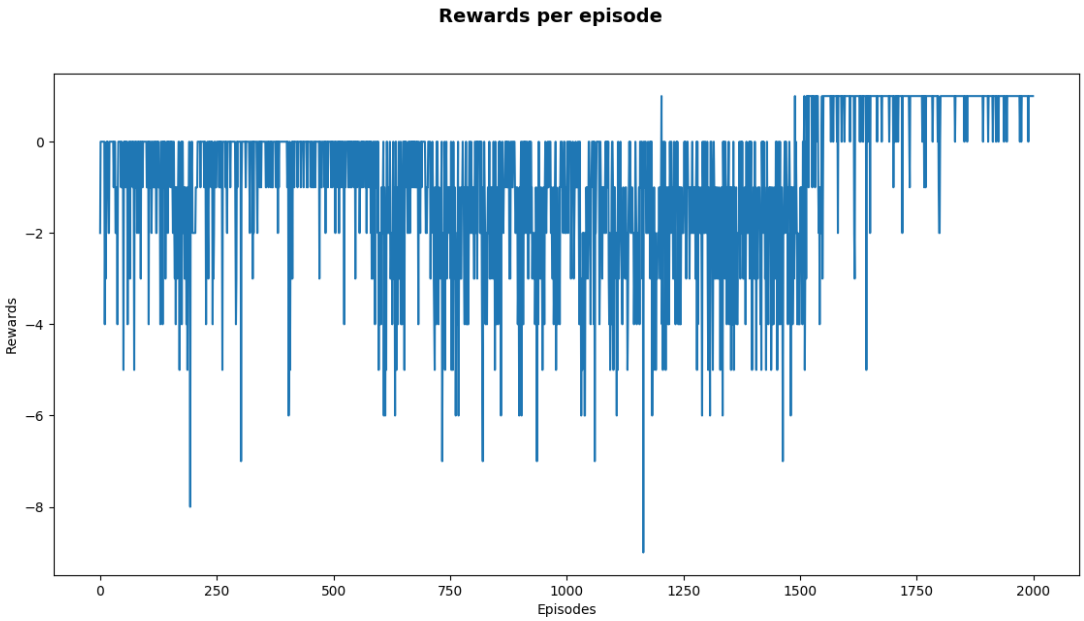


Figure 27: Deep Q-Network: Rewards per episode

The steps taken by the agent in every episode reveal why the agent took so long to complete the training process. Essentially, for the first 1500 episodes, he has reached the maximum number of steps allowed to find the goal state. This means that the agent has executed 1500 episodes $\cdot$ 99 steps $\approx$ 150.000 iterations of the algorithm in order to learn how to solve the problem. After this, and once it knows the optimal paths, the average number of steps is around 6, as it was obtained using both the Q-Table (Figure 24) and the Q-Network (Figure 26).

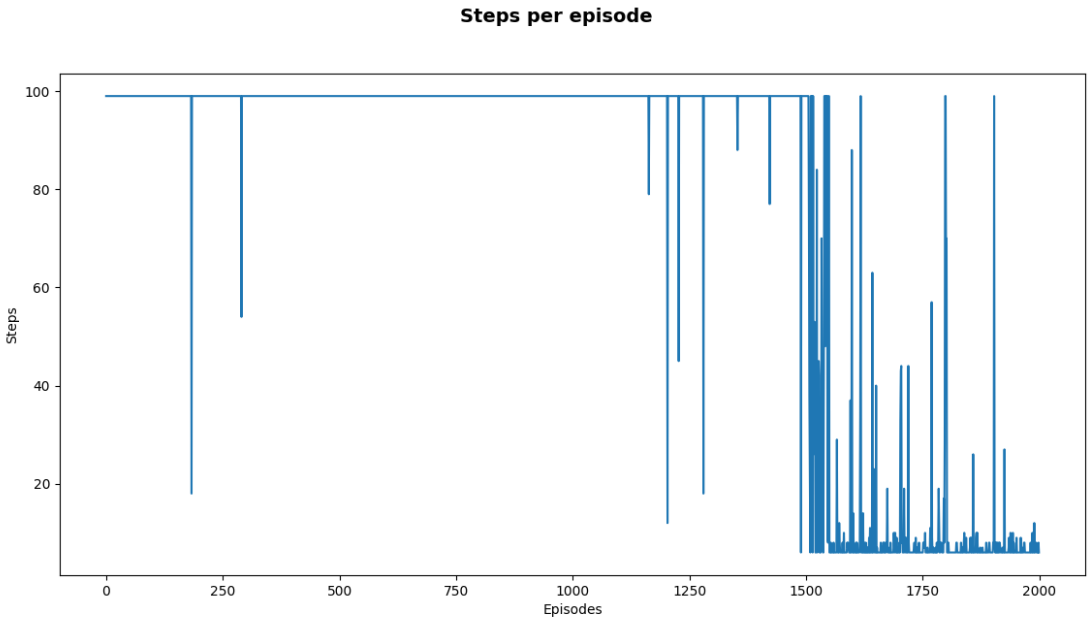


Figure 28: Deep Q-Network: Steps per episode

Until this point, the data that has been analyzed has been the same as the ones interpreted for the Q-Table and Q-Network. And even more, the results obtained until now have shown that the Deep Q-Network is the worst structure between these three to solve the proposed problem.

The use of a Deep Neural Network to solve a Reinforcement Learning problem allows to extract the information encoded in the hidden layers of the network. These results are presented next.

Activations of the neurons of the first hidden layer from each state:

State	First hidden layer activations	Activations 0 or 1
A	[-0.18, -0.06, 0.28, 0.05, -0.009, 0.01, -0.34, -0.40, -0.46, -0.28, -0.27, 0.14]	[0, 0, **1**, **1**, 0, **1**, 0, 0, 0, 0, 0, **1**]
B	[0.41, 0.22, 0.04, 0.23, -0.05, -0.33, -0.18, 0.18, 0.27, 0.10, -0.34, 0.07]	[**1**, **1**, **1**, **1**, 0, 0, 0, **1**, **1**, **1**, 0, **1**]
C	[-0.24, -0.05, -0.01, 0.07, -0.23, 0.17, -0.27, -0.35, -0.56, 0.11, -0.26, -0.14]	[0, 0, 0, **1**, 0, **1**, 0, 0, 0, **1**, 0, 0]
D	[0.32, 0.08, -0.09, 0.25, 0.13, -0.24, -0.03, 0.10, 0.29, 0.13, -0.22, 0.09]	[**1**, **1**, 0, **1**, **1**, 0, 0, **1**, **1**, **1**, 0, **1**]
E	[-0.46, -0.24, 0.08, -0.02, 0.08, 0.10, -0.07, 0.11, 0.04, 0.16, 0.13, -0.05]	[0, 0, **1**, 0, **1**, **1**, 0, **1**, **1**, **1**, **1**, 0]
F	[-0.07, -0.02, 0.36, -0.30, -0.13, 0.08, -0.23, -0.36, -0.27, -0.19, -0.30, 0.006]	[0, 0, **1**, 0, 0, **1**, 0, 0, 0, 0, 0, **1**]
G	[-0.53, -0.07, -0.28, 0.10, 0.33, 0.35, 0.26, -0.22, -0.002, 0.02, 0.39, 0.10]	[0, 0, 0, **1**, **1**, **1**, **1**, 0, 0, **1**, **1**, **1**]
H	[0.29, 0.06, -0.15, 0.20, 0.16, 0.12, 0.12, -0.12, 0.31, 0.11, 0.02, 0.18]	[**1**, **1**, 0, **1**, **1**, **1**, **1**, 0, **1**, **1**, **1**, **1**]
I	[0.53, -0.12, 0.09, -0.08, -0.19, -0.47, -0.30, -0.04, 0.28, -0.11, -0.10, 0.21]	[**1**, 0, **1**, 0, 0, 0, 0, 0, **1**, 0, 0, **1**]
J	[-0.09, 0.18, 0.27, -0.26, -0.32, -0.04, -0.22, -0.39, -0.25, -0.25, -0.26, 0.09]	[0, **1**, **1**, 0, 0, 0, 0, 0, 0, 0, 0, **1**]
K	[-0.22, -0.15, 0.28, -0.07, 0.27, 0.41, 0.19, -0.08, -0.25, -0.12, 0.07, 0.01]	[0, 0, **1**, 0, **1**, **1**, **1**, 0, 0, 0, **1**, **1**]
L	[-0.33, -0.07, 0.11, -0.32, -0.16, -0.10, 0.14, -0.02, -0.24, -0.23, 0.17, -0.07]	[0, 0, **1**, 0, 0, 0, **1**, 0, 0, 0, **1**, 0]
M	[0.34, -0.008, -0.18, -0.07, -0.23, -0.14, -0.23, 0.12, 0.04, -0.20, 0.02, -0.01,]	[**1**, 0, 0, 0, 0, 0, 0, **1**, **1**, 0, **1**, 0]
N	[-0.14, 0.18, 0.23, -0.05, -0.25, 0.11, -0.31, 0.09, -0.0003, -0.06, -0.17, -0.15]	[0, **1**, **1**, 0, 0, **1**, 0, **1**, 0, 0, 0, 0]
O	[0.31, 0.09, 0.31, 0.08, -0.25, -0.48, -0.30, -0.09, -0.50, 0.04, -0.65, -0.22]	[**1**, **1**, **1**, **1**, 0, 0, 0, 0, 0, **1**, 0, 0]
P	[-0.16, 0.10, 0.24, -0.02, 0.004, 0.11, 0.02, -0.14, -0.16, 0.05, -0.09, 0.18]	[0, **1**, **1**, 0, **1**, **1**, **1**, 0, 0, **1**, 0, **1**]

Table 19: Deep Q-Network activations of the neurons of the first hidden layer

The first hidden layer is composed of 12 neurons that have a value in the range $[-1, 1]$. As explained in Chapter 3, it is only fundamental to know if the neurons are either activated or not, leaving aside the intensity of the activation. Every neuron of the first hidden layer has been set to 0 or 1 for every input state if the raw value is below or above a certain threshold. It is possible to say that every input state is encoded as a unique 1x12 vector composed by neurons activated or not after a traning process. These codifications will be used in the next section, when a new deep neural network is created that will use these codifications as its inputs, instead of the identity 1x16 vectors composed by only one value equal to 1 used in the employed structures until now.

The last hidden layer of the Deep Q-Network is composed only by 2 neurons. This decision was taken based on the assumption that these neurons will have encoded information about the best action to take from every input state. Because the agent can choose between 4 actions to take in total, this number can be codified with 2 neurons and these possible combinations: $[0, 0]$, $[0, 1]$, $[1, 0]$ and $[1, 1]$. Next, the activations of the neurons of the last hidden layer from each state is presented:

State	Last hidden layer activations	Activations 0 or 1
A	[-0.93882567, 0.11430452]	[0, 1]
B	[-0.105243, -0.92879552]	[0, 0]
C	[-0.93258178, 0.30550086]	[0, 1]
D	[0.16701196, -0.82030517]	[1, 0]
E	[-0.40334278, 0.7550137]	[0, 1]
F	[-0.93392402, -0.00858557]	[0, 0]
G	[-0.02692717, 0.95495647]	[0, 1]
H	[0.3848446, -0.25672483]	[1, 0]
I	[-0.18066072, -0.93407029]	[0, 0]
J	[-0.93416369, -0.20946378]	[0, 0]
K	[-0.68820113, 0.86539537]	[0, 1]
L	[-0.78508657, 0.64397079]	[0, 1]
M	[-0.29127112, -0.75220829]	[0, 0]
N	[-0.78612602, -0.17365076]	[0, 0]
O	[-0.94624203, -0.90892178]	[0, 0]
P	[-0.75348556, 0.3992658]	[0, 1]

Table 20: Deep Q-Network activations of the neurons of the last hidden layer

By checking the activations of these neurons, it is straightforward to realize that some of the codifications are shared by some states. Because it is believed that the states that share the same codifications are meant to take the same action, but at this point it it still unknown, it is possible to try to find the corresponding action for each state that ultimately will lead the agent to the goal state. There are 4 actions to choose from and 4 possible codifications, so it is possible to check if any ordered combination of the actions match the codifications by leading the agent to the goal state from every state. In this sense, these are all the possible combination of actions that the codifications can take: ['Up', 'Down', 'Right', 'Left'], ['Up', 'Down', 'Left', 'Right'], ['Up', 'Right', 'Down', 'Left'], ['Up', 'Right',' Left', 'Down'], ['Up', 'Left', 'Down', 'Right'], ['Up', 'Left', 'Right', 'Down'], ['Down', 'Up', 'Right', 'Left'], ['Down', 'Up', 'Left', 'Right'], ['Down', 'Right', 'Up', 'Left'], ['Down', 'Right', 'Left', 'Up'], ['Down', 'Left', 'Up', 'Right'], ['Down', 'Left', 'Right', 'Up'], ['Right', 'Up', 'Down', 'Left'], ['Right', 'Up', 'Left', 'Down'], ['Right', 'Down', 'Up', 'Left'], ['Right', 'Down', 'Left', 'Up'], ['Right', 'Left',

'Up', 'Down'], ['Right', 'Left', 'Down', 'Up'], ['Left', 'Up', 'Down', 'Right'], ['Left', 'Up', 'Right', 'Down'], ['Left', 'Down', 'Up', 'Right'], ['Left', 'Down', 'Right', 'Up'], ['Left', 'Right', 'Up', 'Down'] and ['Left', 'Right', 'Down', 'Up'].

It is necessary to try every combination with the states that are codified with [0, 1], [0, 0], [1, 0] and [1, 1]. For example, all states with a codification of [0, 1], will take the action 'Up' from ['Up', 'Down', 'Right', 'Left']. If with a specific combination of actions, all the states that share the same codification taking a corresponding action, they reach the goal state, then it has been proven that information that is encoded in the last hidden layer of a Deep Neural Network is related to the action that the agent will take from every state, being able to predict the best next state without even needed to reach the output layer and check those activations.

The results obtained by checking all the combinations revealed that all the states the shared the codification [0, 1] are meant to take the action 'Down', those ones with [0, 0] to take 'Right' and those ones with [1, 0] to go 'Left'. 'Up' does not appear because in this environment is not required at all to take that action at any point in order to find an optimal path that leads to the goal state. The next table summarizes these results:

Unique groups of activations	**Action**
[0, 1]	Down
[0, 0]	Right
[1, 0]	Left
[1, 1]	Up

Table 21: Deep Q-Network unique groups of activations of the neurons of the last hidden layer and the corresponding action

With this result it is possible to state that the features extracted from a later level of a Deep Q-Network contain information about the best action to take from a specific state, thus the next state that the agent will be in. Furthermore, it is not necessary at all to use the Q matrix or the activations of the output layer to test an agent, because this extraction of features defines a complete set of optimal next states for every state that the agent can be in, with the objective to reach the goal state and solving the proposed problem.

4.5. Deep Q-Network using Feature Extraction

Constructing a new Deep Neural Network that uses as inputs the features extracted from the first hidden layer of the previous Deep Neural Network will allow to prove if the training process of the agent can be improved in the specified terms.

First of all, it is not possible to have as Q matrix as it was before, because now the input layer is composed by 12 neurons, so ultimately, a 12x4 matrix would be obtained, but it would be useless as there are 16 states in total. Therefore, in order to test the agent, the activations of the neurons of the output layer are going to be used:

	Up	**Down**	**Right**	**Left**
A	0.94443101	0.97912008	0.99368155	0.92667651
B	9.03678000e-01	4.57584771e-04	9.94524002e-01	9.76494968e-01
C	0.94159228	0.98004764	0.97993034	0.88317198
D	0.88661242	-0.22115648	0.94469732	0.94581735
E	0.92618692	0.97686177	0.02600441	0.44298393
F	0.94171697	0.9759441	0.98818433	0.90914929
G	0.92720199	0.98109102	-0.09127071	0.38243988
H	0.89815503	0.72517383	-0.02272882	0.64848745
I	0.90601736	0.08676265	0.99493325	0.97626138
J	0.93859595	0.92913657	0.99796128	0.96530694
K	0.93005377	0.98263019	0.21595478	0.47787264
L	0.93347073	0.98159975	0.67634726	0.64977902
M	0.88241732	0.09196892	0.51343864	0.83833593
N	0.94301939	0.97589481	0.99266517	0.92480111
O	0.94230437	0.94062954	0.9991138	0.97413385
P	0.93521249	0.98016733	0.8334766	0.73474693

Table 22: Deep Q-Network using Feature Extraction activations of the neurons of the output layer

Using the activations of the neurons of the output layer allow to check if the agent has learned to find the shortest path between each state and the goal state using this reduced version of deep neural network:

From	Optimum path
A	['B', 'C', 'G', 'K', 'O', 'P']
B	['C', 'G', 'K', 'O', 'P']
C	['G', 'K', 'O', 'P']
D	['C', 'G', 'K', 'O', 'P']
E	['I', 'J', 'K', 'O', 'P']
F	['G', 'K', 'O', 'P']
G	['K', 'O', 'P']
H	['D', 'C', 'G', 'K', 'O', 'P']
I	['J', 'K', 'O', 'P']
J	['K', 'O', 'P']
K	['O', 'P']
L	['P']
M	['I', 'J', 'K', 'O', 'P']
N	['O', 'P']
O	['P']
P	['P']

Table 23: Deep Q-Network using Feature Extraction testing

A set of results is directly obtained by observing the testing of the agent. First, using this Deep Neural Network, the agent has learned how to go from 'H' and 'D' to 'P', when using the previous DNN, it did not solve correctly this part of the environment. Secondly, from 'M' it decides to go 'Up' to 'I' instead of directly going to 'N' which would be the optimum path. This is caused because 'M' is a hole and these states are not properly trained to find the optimum path, plus due to exploration causes the agent did not realize about the shortest path from 'N' because it was not sufficiently explored. The rest of paths show excellence performance.

What is more surprising are the obtained statistics after the training process:

Training time	0 minutes and 54.51 seconds
Accumulated reward over time	75 %
First episode with reward equal to +1	91

Table 24: Deep Q-Network using Feature Extraction statistics

The agent has been able to solve the problem in less than 1 minute, with an accumulated reward which results to be very high and finding the goal in a very early stage of the learning process. This can be viewed in the next plots:

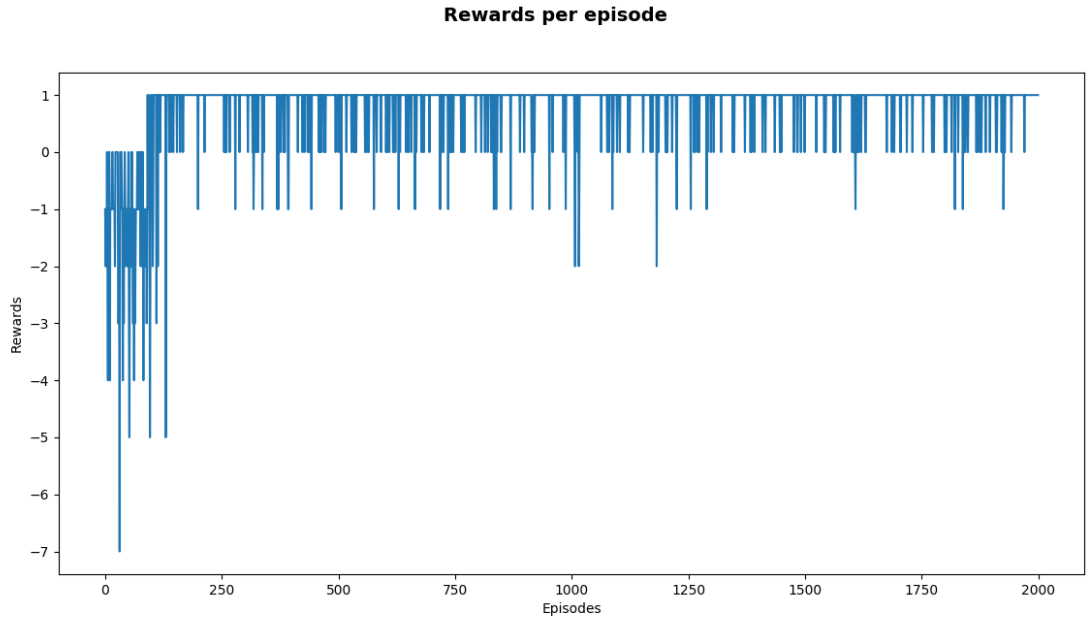


Figure 29: Deep Q-Network using Feature Extraction: Rewards per episode

It took approximately 100 episodes for the agent to learn how to solve the environment and then it continued to reinforce the paths it calculated.

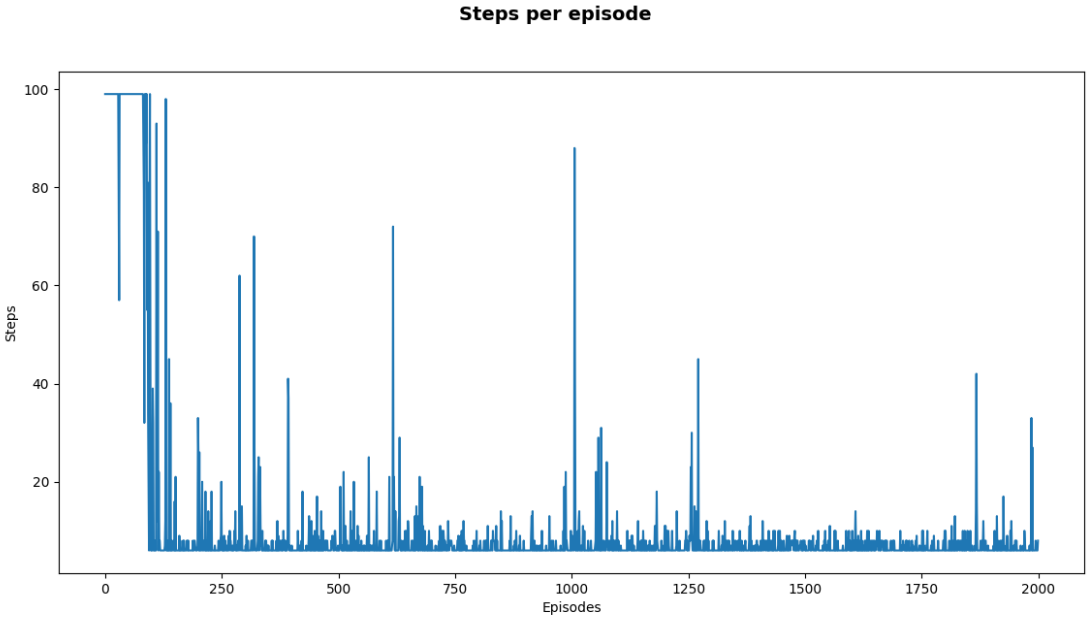


Figure 30: Deep Q-Network using Feature Extraction: Steps per episode

This plot shows how, at the beginning, a series of episodes were completed in the maximum number of steps, while the agent was trying the learn the environment, and when it did it, the minimum required number of steps were taken for the rest of the training process.

This result show that features extracted from the first hidden layer of a previously trained Deep Neural Network can be used in a different and reduced Deep Neural Network in order to improve the performance of an agent in a specific Reinforcement Learning process.

4.6. Comparison

The next table collects of the statistics previously obtained for the different treated structures:

	Training time	**Accumulated reward over time**	**First episode with reward +1**
Q-Table	0 minutes, 1.65 seconds	86.55 %	12
Q-Network	0 minutes, 22.89 seconds	86.55 %	2
Deep Q-Network	7 minutes, 35.50 seconds	-82.25 %	1203
Deep Q-Network using Feature Extraction	0 minutes and 54.51 seconds	75 %	91

Table 25: Comparison between all programs

Comparing the Deep Q-Network with its reduced version that uses features extracted from its first hidden layer a huge difference can be seen. The agent has been able to solve the problem in less than 1 minute, when with the original Deep Q-Network, it lasted more than 7 minutes. Moreover, the accumulated reward results to be very high and close to the ones obtained when using the Q-Table and the Q-Network.
It is important to remark that the structure of the Deep Q-Network using Feature Extraction is much more complex than the ones of the Q-Table and Q-Network, and the results obtained are similar, proving the effectiveness of a Deep Neural Network solving a Reinforcement Learning problem thanks to the extraction of features after a previously accomplished learning process. Moreover, the main reason in using a Deep Reinforcement Learning Network is that this structure can generalise better (and to unseen states) and produce a compressed representation when compared to a table with 2^n entries (n is the number of boolean state variables). Finally, it is expected that the features can be used to train the neural network on different goals in a similar environment, as discussed in Chapter 6.

5. Discussion

This chapter collects different aspects worth mentioning in relation with the methods applied and results obtained, described in Chapter 3 and Chapter 4 respectively.

5.1. Environment Definition

The matrix R (Table 3 generated with Algorithm 1) defines moves from some states that could be considered as invalid in the sense that the agent is not moving from the state that it is at that moment but an action has been taken anyway. This could be visualized as the environment having walls and the agent crashing to the walls without moving from the state it is present at that precise time. For example, if the agent is on state 'A' and takes either action 'Up' or 'Left' it will still remain in 'A'. Therefore, these actions should have associated null rewards. This would make the environment more realistic and if these invalid actions would not be considered, it would help the agent in the learning process. Therefore, the agent would only move to next states that are different from the state it is currently on. An alternative matrix R that would represent this could be the following:

	↑	↓	→	←
From 'A'	'–'	'E'	'B'	'–'
From 'B'	'–'	'F'	'C'	'A'
From 'C'	'–'	'G'	'D'	'B'
From 'D'	'–'	'H'	'–'	'C'
From 'E'	'A'	'I'	'F'	'–'
From 'F'	'B'	'J'	'G'	'E'
From 'G'	'C'	'K'	'H'	'F'
From 'H'	'D'	'L'	'–'	'G'
From 'I'	'E'	'M'	'J'	'–'
From 'J'	'F'	'N'	'K'	'I'
From 'K'	'G'	'O'	'L'	'J'
From 'L'	'H'	'P'	'–'	'K'
From 'M'	'I'	'–'	'N'	'–'
From 'N'	'J'	'–'	'O'	'M'
From 'O'	'K'	'–'	'P'	'N'
From 'P'	'L'	'–'	'–'	'O'

Table 26: Matrix R with actions that only lead to other different states

By using this new matrix R, a new condition should be included in the algorithm step where the agent chooses the next action to explore, that would indicate that the agent could only select an action if it is different to the invalid character '–'. Because the amount of actions the agent can choose from is reduced, in comparison with the original matrix R (Table 3), and depending on the state that the agent is at a precise time, the performance of the agent would improve.

5.2. ϵ-greedy Policy

The ϵ-greedy policy that it is followed in the implementation of the Q-Learning technique (Algorithm 2) keeps a balance between exploration (taking random actions) and exploitation (taking actions with maximum values). Experiments were made changing the ϵ value with the Q-Table structure in order to see which one was the value that produced the best training time and the accumulated reward over time:

ϵ value	Training time	Accumulated reward over time	Optimal solution
0.05	1.25 seconds	0.9205 %	No
0.1	1.84 seconds	0.8815 %	Yes
0.2	1.10 seconds	0.7225 %	Yes
0.3	1.62 seconds	0.559 %	Yes
0.4	1.86 seconds	0.2675 %	Yes
0.5	3.03 seconds	−0.155 %	Yes
0.6	2.64 seconds	−0.666 %	Yes
0.7	4.48 seconds	−1.5345 %	Yes
0.8	5.35 seconds	−3.2955 %	Yes
0.9	8.65 seconds	−6.2335 %	Yes

Table 27: ϵ-greedy: Balance between exploration and exploitation using Q-Table

An ϵ value of 0.1 shows the best balance between training time and accumulated reward over time while providing an optimal solution to the problem.

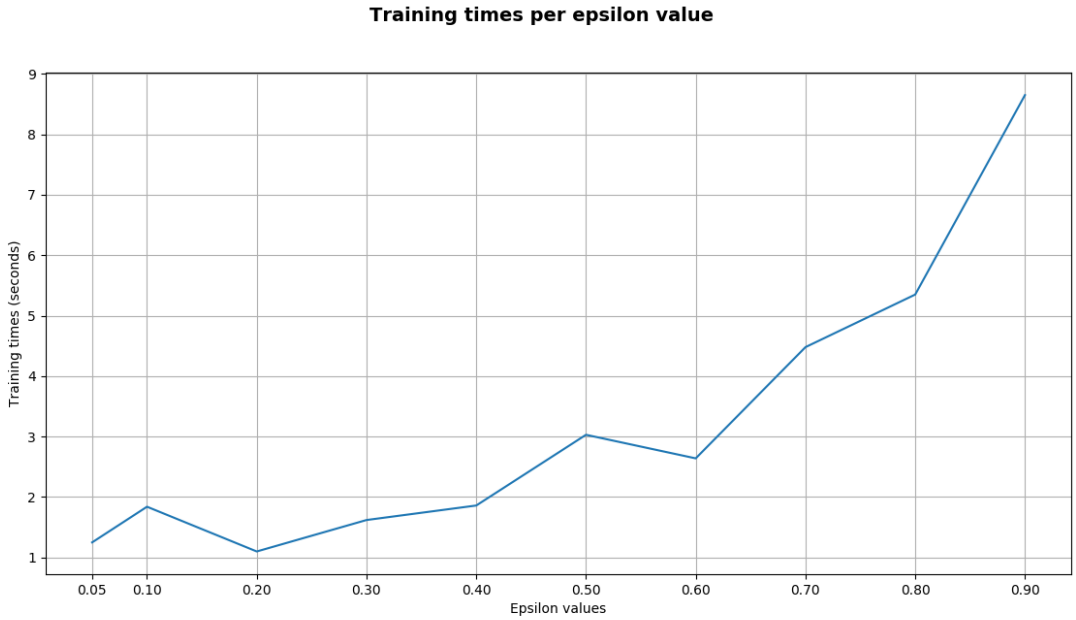


Figure 31: ϵ values in relation with training times

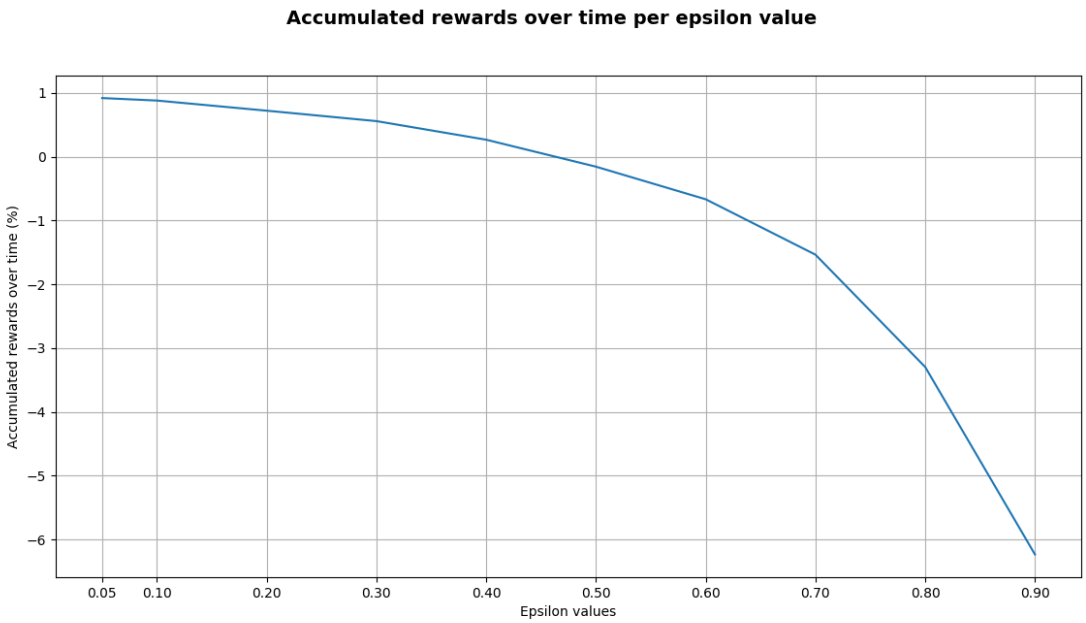


Figure 32: ϵ values in relation with accumulated rewards over time

It is possible to conclude that because the environment is only composed by 16 states, the agent needs to explore it an amount of times much smaller than exploit it, which would be desired to do continuously once it has explored the whole environment. In this sense, the decaying ϵ-greedy policy could be implemented. It starts by setting the ϵ value very high, to give the agent the opportunity to explore the environment thoroughly, and every time it reaches the goal state, the ϵ value should be decreased to reduce the probability of useless exploring at this point. At the end, the ϵ value should be very close to 0, because by then, the agent must have learned all the specifics of the environment and only needs to exploit the actions that lead to optimal next states.

5.3. Initialization of the Q matrix and Weights

The previous decaying ϵ-greedy policy could work jointly with a different initialization of the Q matrix and the weights of the Deep Neural Network. Currently, these are initialized randomly following a uniform distribution in the interval:
Weights $\sim \bigcup[\frac{-1}{\sqrt{numStates}}, \frac{1}{\sqrt{numStates}}]$.
Therefore, there is a probability of setting some values that are supposed to be negative (the ones that lead to holes) to positive random values. The problem

is that if the agent does not explore properly these values in order to set them accordingly to the environment definition, these values will remain as they were initialiazed. An alternative for this would be to initialize the Q matrix or the weights with values equal to 0 (or even to -1). This would let the agent explore thoroughly at the beginning of the training process to discover the environment and then it should exploit the best values it has found. Proceeding like this, there would be no risk in having at the end some values initialized randomly and not explored sufficiently.

5.4. Activation Functions and Thresholds

The activation function used for the neurons of the Q-Network and the different Deep Q-Networks is the hyperbolic tangent (Tanh) activation function (Figure 17). Other activation functions such as the sigmoid or a combination of ReLU (Rectifier Linear Unit) for the hidden layers and Softmax (normalized exponential) for the output layer could have been used in the implementation the neural networks as well (Figure 33).

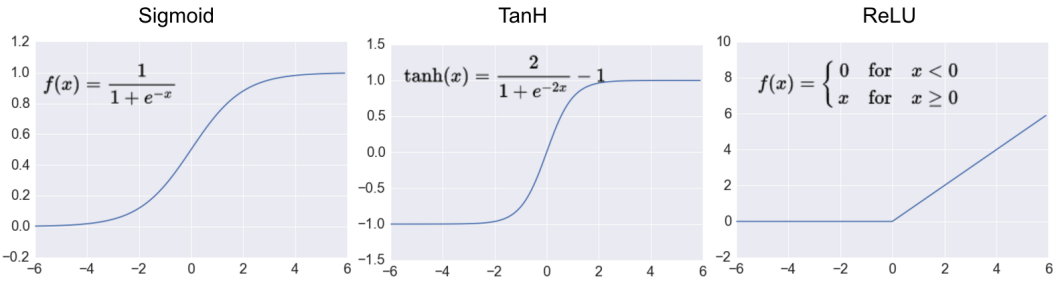


Figure 33: Activation functions: Sigmoid, Tanh and ReLU

The thresholds, to consider if a neuron is either activated or not, for these activation functions would be the next ones:

	Sigmoid	**Tanh**	**ReLU**
Neuron not activated (0)	If x [(1)] < 0.5	If x < 0	If x < 0
Neuron activated (1)	if x > 0.5	If x > 0	If x > 0

[1] x: raw activation value of a neuron.

Table 28: Thresholds for the activation functions: Sigmoid, Tanh and ReLU

In general, the thresholds are very accurate, but they will not be as accurate as if the activation function would be a step function:

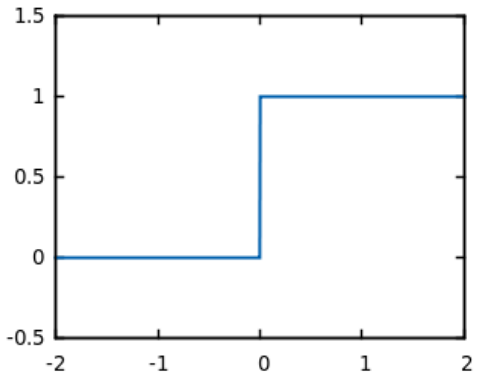


Figure 34: Step activation function

It has been observed that sometimes, the threshold for Tanh, even if it is supposed to be exactly 0, has been displaced in the range of approximately $[-0.05, 0.05]$. This is due to the curved nature of the function itself, but consequently, it is fundamental to carefully check if a displacement has occurred or not. Testing with the other activation functions mentioned should be done, although the training with the hyperbolic tangent function produces optimal results. In conclusion, the issue of having a slightly displacement of the threshold can result in less accurate results, when, for example, checking if the neurons of the last hidden layer contain information about the best action to take from a specific state, as it is critical to group each neuron that has a same codification.

5.5. Topology of the Deep Q-Network

A different topology of the Deep Q-Network containing four neurons in the last hidden layer instead of two (Figure 18) has been tested to check if the last hidden layer would still contain encoded information about the best action to take from a specific state.

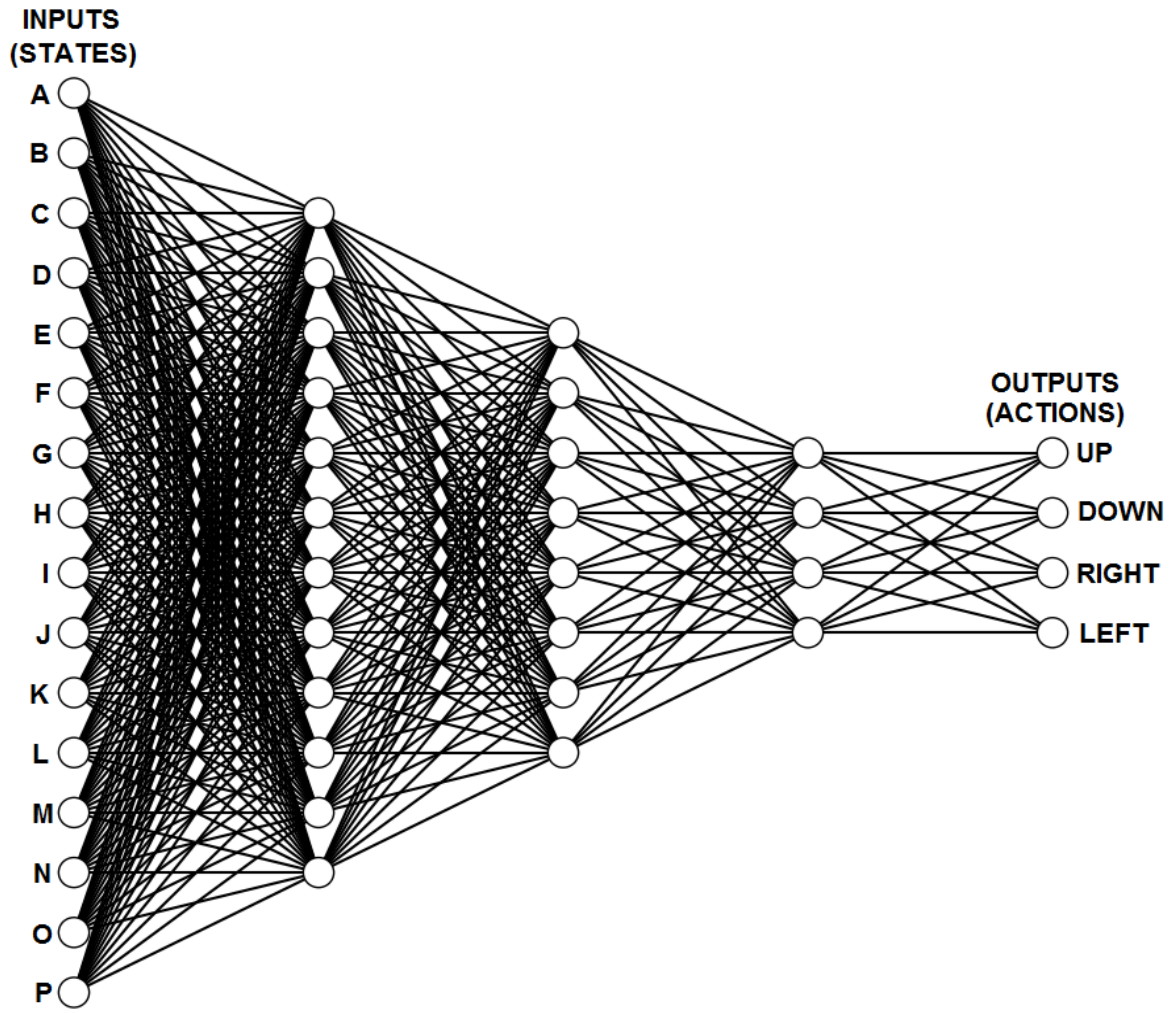


Figure 35: 16x12x8x4x4 Deep Q-Network

After running the algorithm, the agent learned how to find optimal paths from every state to the goal state. In order to extract the features from the last hidden layer of this Deep Q-Network, the activations of the neurons of this hidden layer were obtained, producing the next result:

State	Last hidden layer activations	Activations 0 or 1
A	[-0.5117892, 0.32149127, 0.17292397, 0.46579027]	[0, 1, 1, 1]
B	[-0.20224683, 0.43432829, -0.55792177, 0.00182688]	[0, 1, 0, 1]
C	[-0.54673713, 0.38875079, 0.28867286, 0.58479261]	[0, 1, 1, 1]
D	[-0.10601068, 0.32831088, -0.45802996, -0.15127732]	[0, 1, 0, 0]
E	[-0.39403787, 0.06600142, 0.57761848, 0.23270541]	[0, 1, 1, 1]
F	[-0.52947843, 0.37895349, 0.24179627, 0.54479551]	[0, 1, 1, 1]
G	[-0.42446965, 0.03250928, 0.7018159, 0.23400939]	[0, 1, 1, 1]
H	[-0.1697319, 0.26360354, -0.18424474, -0.04780662]	[0, 1, 0, 0]
I	[-0.23060238, 0.45361552, -0.57975578, 0.04738365]	[0, 1, 0, 1]
J	[-0.49761561, 0.25381467, 0.45064175, 0.55190182]	[0, 1, 1, 1]
K	[-0.47010425, 0.01200398, 0.79494476, 0.39597672]	[0, 1, 1, 1]
L	[-0.52261657, 0.04508076, 0.83458948, 0.56787795]	[0, 1, 1, 1]
M	[-0.47349238, 0.44817498, -0.13251278, 0.43631816]	[0, 1, 0, 1]
N	[-0.55085778, 0.49119166, -0.0456657, 0.58332509]	[0, 1, 0, 1]
O	[-0.78049135, 0.8809275, -0.90793747, 0.92261648]	[0, 1, 0, 1]
P	[-0.34979451, 0.32905281, -0.10018327, 0.28916463]	[0, 1, 0, 1]

Table 29: Activations of the neurons of the last hidden layer with a Deep Q-Network with four neurons in this layer

Even though the number of actions available (go Up, Down, Right or Left) can be codified with only 2 neurons, in order to prove that the last hidden layer contain encoded information regarding the best action to take from every state (Table 21), a similar result should be obtained by using 4 neurons. Actions that have the same codification should be related to a unique best action to take. After running the algorithm introduced in Chapter 4 that tests all the possible combinations of actions that would take the agent from each state to the goal state, it was finally obtained that states sharing the same following codifications make the agent take the best action from every state.

Unique groups of activations	Action
[0, 1, 1, 1]	Down
[0, 1, 0, 1]	Right
[0, 1, 0, 0]	Left
[0, 1, 1, 0]	Up

Table 30: Unique groups of activations of the neurons of the last hidden layer and the corresponding action with a Deep Q-Network with four neurons in this layer

Therefore, it has been proven that the topology of a Deep Q-Network is independent from the information encoded in the last hidden layer. Although this result has been demonstrated, more experiments should be performed in order to find an optimal topology for a Deep Q-Network that solves the stated problem.

5.6. Feature Extraction discussion

Before studying the characteristics of the hidden layers of a Deep Q-Network, it was thought that the last hidden layer would contain the features to be extracted and use them in another Deep Neural Network to improve the performance of the agent. It has been concluded that this assumption was not correct. The last hidden layer contains features that makes the agent to decide which action is going to take from every state. Furthermore, the first hidden layer contains the codification of the states that can be used separately to improve the learning process of the agent. The second hidden layer is a mix between these two layers. It is proven then that the information of the states is lost along the neural network, as the information regarding the states is transformed into the decision about what actions to take that will lead to the best next states.

The features of the neurons of the second hidden layer are not extracted on purpose because, as it was seen, they do not represent pure information about either the states or the actions, but a processed mix of these as the Reinforcement Learning algorithm executes.

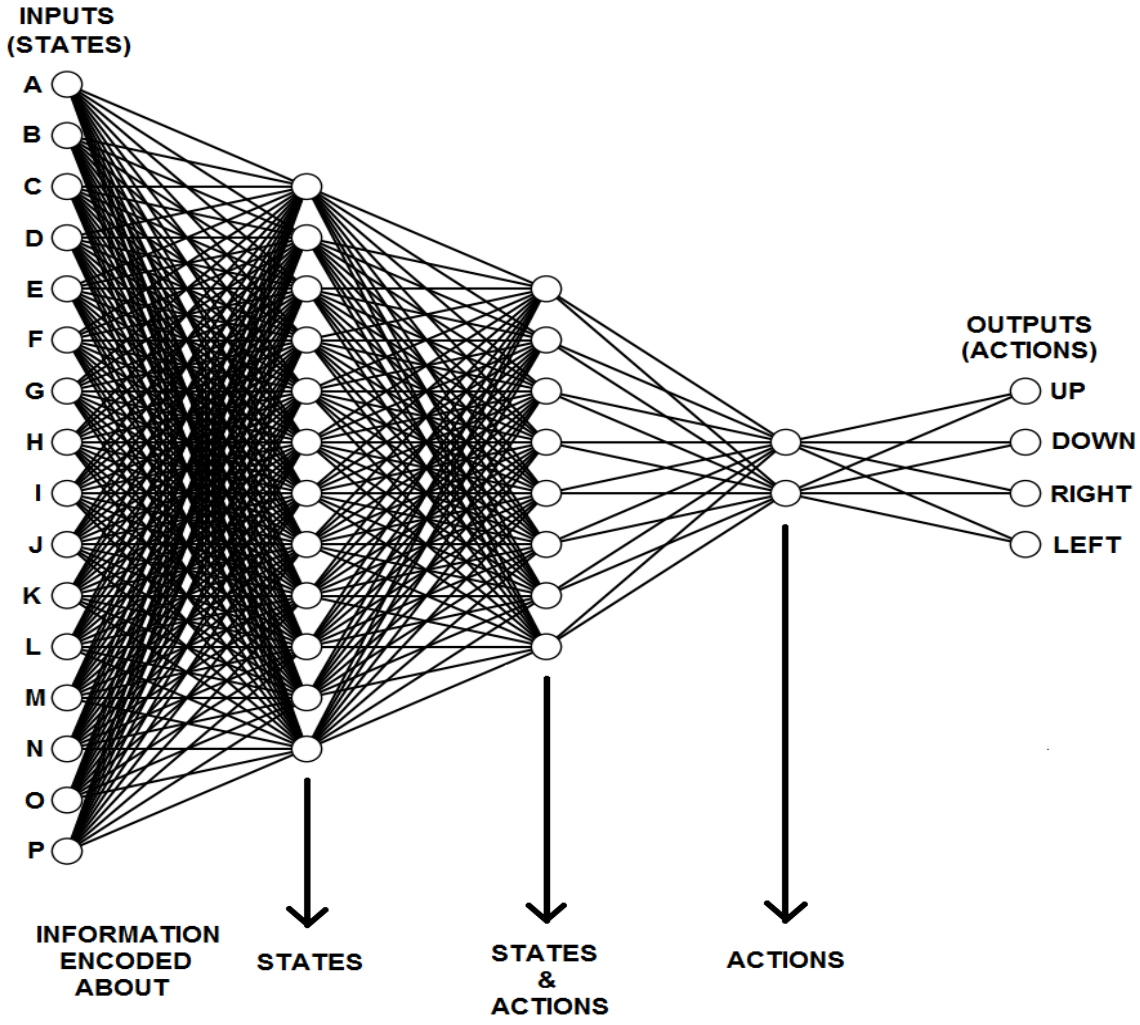


Figure 36: Information encoded in a Deep Q-Network

Apart from that, following the success of the extraction of features from the first hidden layer of a Deep Q-Network, it could be thought that the same process could continue over and over. If from the reduced Deep Q-Network features of the first hidden layer are extracted once again to use them in another Deep Q-Network to keep improving the performance of the agent, at the end, optimally, a 4x4 Q-Network without any hidden layer would be obtained:

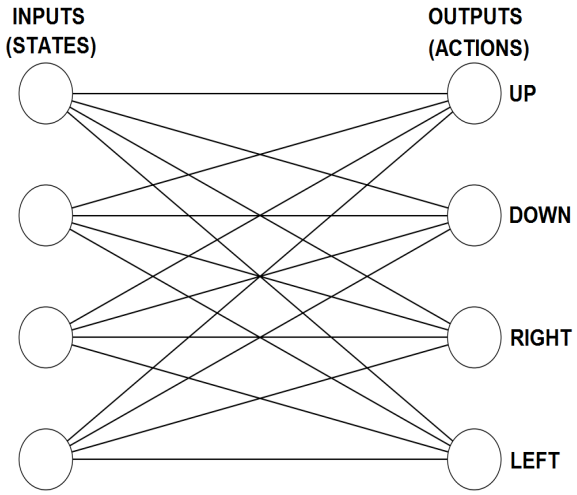


Figure 37: 4x4 Q-Network

This minimal Q-Network would have the following codifications for the input states:

State	Codification
A	[0, 0, 0, 0]
B	[0, 0, 0, 1]
C	[0, 0, 1, 0]
D	[0, 0, 1, 1]
E	[0, 1, 0, 0]
F	[0, 1, 0, 1]
G	[0, 1, 1, 0]
H	[0, 1, 1, 1]
I	[1, 0, 0, 0]
J	[1, 0, 0, 1]
K	[1, 0, 1, 0]
L	[1, 0, 1, 1]
M	[1, 1, 0, 0]
N	[1, 1, 0, 1]
O	[1, 1, 1, 0]
P	[1, 1, 1, 1]

Table 31: 4x4 Q-Network codifications of inputs

It has been tested if this structure would directly solved the proposed envi-

ronment, even with a higher number of episodes (10000 instead of 2000), but the agent has not able to learn the specifics of the environment. This happens because it is not possible to solve the stated problem with such a reduced topology. Even if the input and output layers are well defined, with such a reduced topology, it is strictly necessary to have some hidden layers in between to produce logic operations such as "AND", "OR" or "XOR". It is important to remark that a 1-layer perceptron cannot solve a simple "XOR" problem since "XOR" outputs are not linearly separable (Yanling et al., 2002). Therefore, a 4x4 Q-Network cannot solve a pathfinding problem using Q-Learning because the agent cannot relate the input states with those codifications to the actions in such a small neural network. In conclusion, the reduction of a Deep Q-Network has a limit, and if overpassed, the algorithm does not work anymore. It remains to be seen where is exactly this limit and which one could be the most optimal topology to be used in order to solve the problem.

5.7. Relation between Features Extracted from Convolutional Neural Networks and Deep Q-Networks

A Convolutional Neural Network and a Deep Q-Network differ in some of the characteristic that conform their compositions. First, the nature of their inputs is different: raw pixel data from an image and states of an environment. Secondly, the type of learning, supervised and with reinforcement respectively, determines the categories of problems to solve, a classification problem or, for example, a pathfinding problem, by using one neural network or the other. Despite these differences, their structures or topologies can be very similar, presenting an input layer connected to the output layer through various hidden layers in between. Therefore, a connection can be established in relation to the meaning of the features extracted from the hidden layers of both neural networks.

From the analysis performed on the extraction of features from the implemented Deep Q-Networks it is possible to conclude that the first hidden layers of a Deep Q-Network will contain encoded information about the states while the last hidden layers contain encoded information about the best actions to take. In this sense, a comparison can be made in relation with the extraction of features from a Convolution Neural Network such as the one of the research introduced in Chapter 2.

Features Extracted From	Convolutional Neural Network	Deep Q-Network
First Hidden Layers	Edges, corners	States
Last Hidden Layers	Windows, eyes	Actions

Table 32: Feature Extraction comparison between Convolutional Neural Networks and Deep Q-Networks

In relation with the nature of the features extracted from a Convolutional Neural Network, the first hidden layers contain low order features, while the last hidden layers contain high order features, resulting in more complex patterns. Therefore, the states encoded in the first hidden layers of a Deep Q-Network could be considered as low order features, while the best actions to take from a specific states, which are the data encoded in the last hidden layers of a Deep Q-Network, result to be high order features. Even it would be possible to obtain information from a specific neuron, because in a Convolutional Neural Network, each neuron is specifically dedicated in treating the information of a specific shape or pattern of the input data. Ultimately, how the information is encoded in the hidden neurons of a Deep Neural Network is part of a progressive process that increases in complexity as the data moves forward from the input layer to the output layer of the network.

6. Evaluation, Reflections and Conclusions

6.1. General Conclusions and Contribution

This work has presented an analysis of Deep Reinforcement Learning Networks focused primarily on the features that are automatically encoded in the hidden layers of this type of neural networks. To achieve this, a structured set of methods has been followed. First, an environment to be discovered by an intelligent agent has been defined. Then, the Reinforcement Learning Q-Learning technique has been implemented using various structures: a Q-Table, a Q-Network and a Deep Q-Network. The Q-Learning algorithm and the structures have been implemented using Python and the Machine Learning library TensorFlow. Experiments have been made in order to test if the agent is able to solve a established pathfinding problem within the defined environment and to ultimately compare the effectiveness of the different mentioned structures. Fundamentally, the research has produced two determinant results:

- Features extracted from the initial stage of a Deep Q-Network can be used in a different and reduced Deep Q-Network in order to effectively improve the performance and the learning process of an intelligent agent.
- Features extracted from the later stage of a Deep Q-Network contain information regarding the best action to take from a specific state.

In relation to the proposed methods proposed and to the obtained results, different aspects have been discussed in order to cover alternatives and interesting details that arised in the development process. Furthermore, a list of ideas that could improve the research in the future has been presented. Finally, general conclusions wrap up the research, which attempts to make a contribution to research on Machine Learning.

6.2. Implications

Machine Learning research studies that are focussed on studying Deep Reinforcement Learning Networks might be interested in the obtained results of this research. By including the discoveries made in this research, certain aspects of the implementation of Deep Q-Network could be productively improved, as the learning process of an intelligent agent has been improved in the presented research. The stated findings might inspire new research studies in the sense of how to keep optimizing the use of Deep Q-Networks and to discover

new findings related to the intrincate structure of this type of neural networks. The enormous potential that Deep Q-Networks have has been attempted to communicate with this research, making emphasis in that there are still areas subject to improvement, as this part of the Machine Learning field is recent and holds great potential in terms of new research that could be undertaken. The research has been presented in this self-contained document, but a paper to be submitted to a flagship conference in Artificial Intelligence could be proposed. This potential article would summarize the information here contained and would expand the work by implementing some of the aspects described in the Future work section.

6.3. Further work

A list of tasks that would improve the research in various ways is presented next. These tasks are related to the methods described in Chapter 3 and to the results presented in Chapter 4:

- Implementation of the decaying ϵ-greedy policy as described in Chapter 5. This would improve the way the agent discovers the environment. The agent would explore (choosing random actions) the environment thorougly at the beginning of the learning process. Every time it would reach the goal state, the ϵ-value would be decreased, so there would be a higher probability of exploiting (choosing actions with maximum value) the best actions it would have found.

- Definition of a new metric that would measure how long does it takes to learn the optimal policy. This policy would be the one that, for example, would take the agent from 'A' to 'P' in exactly 6 steps by following actions with maximum values. Until now, the metric "first episode with reward +1" gives an indication about this, but it is affected by random actions, therefore it can be inaccurate.

- Test the agent in different environments. A straightforward way to create new environments is by just changing the initial state, the holes and the goal state. An example of a new and more complex environment could be the following:

'A' [1]	'B' [2]	'C' [3]	'D'
'E'	'F' [2]	'G' [2]	'H'
'I'	'J'	'K' [2]	'L'
'M'	'N'	'O'	'P'

[1] Yellow: Initial state.

[2] Red: Holes.

[3] Green: Goal State.

Table 33: A more complex environment

This environment is richer in the sense that the agent is obligated to take all the available actions to reach the goal state. For example, 'Down' from 'A', 'Right' from 'N', 'Up' from 'P' and 'Left' from 'D'. The agent should be able to learn how to solve any given environment, but it would be interesting to see if the stated results are still successful in different configurations of the environment. Another interesting idea would be to increment the number of states to, for example, 25 (resulting in a 5x5 environment), or even more.

- After training a first Deep Q-Network and having extracted the features from its first hidden layer, it would be interesting to see if the new Deep Q-Network that uses those extracted features as inputs can learn how to find optimal paths in a different configuration of the environment. For example, the goal state could be another one than 'P' or the holes could be changed.

- Obtain an optimal topology for a Deep Q-Network that would solve the stated problem by consecutively extracting features from the first hidden layers of successive Deep Q-Networks. It has been proven that features extracted from the first hidden layer of a Deep Q-Network can be used as inputs of a different and reduced Deep Q-Network with one hidden layer less than the original one. It remains to be seen if this process can be continuously applied in order to reduce the size of a Deep Q-Network, in terms of hidden layers and neurons per layer, and to keep improving the learning process of the agent.

- It would be ideal to have a model built from a table of [Features, Actions] to [Features, Rewards] then to unseen states. Right now the model cannot work without the original environment, as it is only possible to

extract rewards by relating them to the states. Another option would be to have the reward as one of the inputs of the network, so the reward gets encoded as the inputs do in the first hidden layer of the Deep Q-Network. A different alternative would be to only have 1 neuron in the output layer, representing the reward. In this structure, the actions would be added to the last hidden layer, where the inputs have reached this specific without being affected by the actions and by being codified without external information. These options would ultimately get the agent to solve previously unseen environments

- Create a set of visualization tools. It would be very useful to be able to visually check the learning process of the agent as it discovers the environment. In debugging terms, it would be very helpful to be able to see how the data is propagated in the different neural networks, by being able to see which neurons are being activated during the training of the agent.

- Adapt the project into the Unity Machine Learning Agents SDK (Unity-Technologies, 2017). Unity 3D is a popular game engine, used to develop video games and create simulations for a wide range of platforms, from consoles to mobile devices. By using this SDK, it would be possible to create more visual experiments and to ultimately create a video game that would showcase the results obtained in the research.

6.4. Reflections

Valuable knowledge has been acquired by developing this project. The understanding of Reinforcement Learning algorithms and how these can be implemented using neural networks has provided profound awareness of the importance of this area of Machine Learning. An interesting alternative to implement the neural networks would have been to use Keras (Chollet et al., 2015), a high-level neural networks API capable of running on top of TensorFlow, that allows easy and fast prototyping. Finally, the relation of this Artificial Intelligence area with video games makes a promising future for the games industry. Video games would be more engaging by incorporating more captivating intelligent agents that would improve the game experience overall.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M. et al. (2016), 'Tensorflow: Large-scale machine learning on heterogeneous distributed systems', *arXiv preprint arXiv:1603.04467* .

Ade, P. A., Aghanim, N., Arnaud, M., Ashdown, M., Aumont, J., Baccigalupi, C., Banday, A., Barreiro, R., Bartlett, J., Bartolo, N. et al. (2016), 'Planck 2015 results-xiii. cosmological parameters', *Astronomy & Astrophysics* **594**, A13.

Bellemare, M. G., Naddaf, Y., Veness, J. and Bowling, M. (2012), 'The arcade learning environment: An evaluation platform for general agents', *Journal of Artificial Intelligence Research* .

Bellman, R. (1957), 'A markovian decision process', *Journal of Mathematics and Mechanics* pp. 679–684.

Bellman, R. (2013), *Dynamic programming*, Courier Corporation.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. (2016), 'Openai gym', *arXiv preprint arXiv:1606.01540* .

Chollet, F. et al. (2015), 'Keras', `https://github.com/fchollet/keras`.

Glorot, X. and Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks, *in* 'Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics', pp. 249–256.

Hastie, T., Tibshirani, R. and Friedman, J. (2009), Overview of supervised learning, *in* 'The elements of statistical learning', Springer, pp. 9–41.

Heess, N., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, A., Riedmiller, M. et al. (2017), 'Emergence of locomotion behaviours in rich environments', *arXiv preprint arXiv:1707.02286* .

Hinton, G. E. and Salakhutdinov, R. R. (2006), 'Reducing the dimensionality of data with neural networks', *science* **313**(5786), 504–507.

Hunter, J. D. (2007), 'Matplotlib: A 2d graphics environment', *Computing In Science & Engineering* **9**(3), 90–95.

Johnson, D. (2015), 'Composing music with recurrent neural networks'.

Kiros, R., Zhu, Y., Salakhutdinov, R. R., Zemel, R., Urtasun, R., Torralba, A. and Fidler, S. (2015), Skip-thought vectors, *in* 'Advances in neural information processing systems', pp. 3294–3302.

Krakovsky, M. (2016), 'Reinforcement renaissance', *Communications of the ACM* **59**(8), 12–14.

Li, Y. (2017), 'Deep reinforcement learning: An overview', *arXiv preprint arXiv:1701.07274* .

McCulloch, W. S. and Pitts, W. (1943), 'A logical calculus of the ideas immanent in nervous activity', *The bulletin of mathematical biophysics* **5**(4), 115–133.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. and Kavukcuoglu, K. (2016), Asynchronous methods for deep reinforcement learning, *in* 'International Conference on Machine Learning'.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. et al. (2015), 'Human-level control through deep reinforcement learning', *Nature* **518**(7540), 529–533.

Mordvintsev, A., Olah, C. and Tyka, M. (2015*a*), 'Deepdream-a code example for visualizing neural networks', *Google Res* .

Mordvintsev, A., Olah, C. and Tyka, M. (2015*b*), 'Inceptionism: Going deeper into neural networks', *Google Research Blog. Retrieved June* **20**, 14.

Robert, C. P. (2004), *Monte carlo methods*, Wiley Online Library.

Rosenblatt, F. (1958), 'The perceptron: A probabilistic model for information storage and organization in the brain.', *Psychological review* **65**(6), 386.

Samuel, A. L. (2000), 'Some studies in machine learning using the game of checkers', *IBM Journal of research and development* **44**(1.2), 206–226.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D. (2016), Prioritized experience replay, *in* 'International Conference on Learning Representations (ICLR)'.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. et al. (2016), 'Mastering the game of go with deep neural networks and tree search', *Nature* **529**(7587), 484–489.

Stuhlsatz, A., Lippel, J. and Zielke, T. (2012), 'Feature extraction with deep neural networks by a generalized discriminant analysis', *IEEE transactions on neural networks and learning systems* **23**(4), 596–608.

Sutton, R. S. (1988), 'Learning to predict by the methods of temporal differences', *Machine learning* **3**(1), 9–44.

Sutton, R. S. and Barto, A. G. (1998), *Reinforcement learning: An introduction*, Vol. 1, MIT press Cambridge.

Tesauro, G. (1994), 'Td-gammon, a self-teaching backgammon program, achieves master-level play', *Neural computation* **6**(2), 215–219.

Turing, A. (2004), 'Intelligent machinery (1948)', *B. Jack Copeland* p. 395.

Unity-Technologies (2017), 'Unity machine learning agents', `https://github.com/Unity-Technologies/ml-agents`.

Van Hasselt, H., Guez, A. and Silver, D. (2016), Deep reinforcement learning with double q-learning., *in* 'AAAI', pp. 2094–2100.

Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J. et al. (2017), 'Starcraft ii: A new challenge for reinforcement learning', *arXiv preprint arXiv:1708.04782* .

Walt, S. v. d., Colbert, S. C. and Varoquaux, G. (2011), 'The numpy array: a structure for efficient numerical computation', *Computing in Science & Engineering* **13**(2), 22–30.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. and de Freitas, N. (2015), 'Dueling network architectures for deep reinforcement learning', *arXiv preprint arXiv:1511.06581* .

Watkins, C. J. and Dayan, P. (1992), 'Q-learning', *Machine learning* **8**(3-4), 279–292.

Weber, T., Racanière, S., Reichert, D. P., Buesing, L., Guez, A., Rezende, D. J., Badia, A. P., Vinyals, O., Heess, N., Li, Y. et al. (2017), 'Imagination-augmented agents for deep reinforcement learning', *arXiv preprint arXiv:1707.06203* .

Wu, Y. and Tian, Y. (2017), Training agent for first-person shooter game with actor-critic curriculum learning, *in* 'Submitted to Int Conference on Learning Representations'.

Yanling, Z., Bimin, D. and Zhanrong, W. (2002), Analysis and study of perceptron to solve xor problem, *in* 'Autonomous Decentralized System, 2002. The 2nd International Workshop on', IEEE, pp. 168–173.

Appendix A: Project Proposal

City, University of London
MSc Computer Games Technology

***R**esearch **M**ethods and **P**rofessional **I**ssues*

Coursework 2016: Task 2

Project Proposal

Prediction of Future States of Features Extracted from Deep Reinforcement Learning Networks

Joaquín Ollero García
160030893

Supervisor:
Dr. Christopher Child

Introduction

Deep Learning is a class of machine learning algorithms which automatically extract information from raw data by using artificial neural networks in which neurons deeper in the network naturally encode features in the data. Reinforcement Learning algorithms provide a mechanism that rewards or punishes agents so they can learn that a certain feedback related to a particular

situation leads to a good or a bad state in the future, resulting in a table mapping each state/action combination in an environment to a value. The mix of these techniques, termed Deep Reinforcement Learning Networks, offers the possibility to use a Deep Learning neural network to produce an approximate Reinforcement Learning value table that will allow features from neurons in later layers of the network to be extracted.

The purpose of the project is to acquire a model of an environment using features extracted by the network. The system will model these features, predicting which ones will be present in the next time step. The expected results are to increase the speed of learning and to improve the behaviour in previously unseen environments states by using the predictive capabilities of the model. To test the system a set of video game artificial intelligence testbeds will be used.

Figure 38: Top-level view of the system

Research question

Can features extracted from the later level of a Deep Reinforcement Learning Network be used to predict the future states of the features in an environment modelling algorithm?

Motivation and beneficiaries

Reinforcement Learning using Deep Learning Neural Networks is receiving intense media attention since (Mnih et al., 2015) introduced Deep Q-Networks (DQN) and the research was published in the multidisciplinary scientific journal Nature, leading this area of the artificial intelligence field to be of huge interest. The presented work demonstrated that the developed model achieved a high level of expertise compared to the one of a professional human game player in a set of 49 time-honored Atari 2600 games (Bellemare et al., 2012). This fact represents a motivation to undertake this project, since it remains to be determined whether the implementation of DQN can be extended to contemporary games and more complex environments. Moreover, feature extraction from deep neural networks is a state of the art approach, not previously combined with Reinforcemente Learning.

We find to be another incentive the use of TensorFlow (Abadi et al., 2016),

an open-source software library for Machine Intelligence developed by Google. This machine learning system operates at large scale and in heterogeneous environments, presenting an architecture that provides flexibility to developers by enabling them to experiment with novel optimizations and training algorithms. Multiple Google services have adopted Tensorflow for their implementations, therefore it has become widely used for machine learning research, demonstrating a compelling performance in real-world applications.

It is intended that the documentation that will be produced by attempting this project will make a contribution to the artificial intelligence community and more precisely to machine learning researchers. The main objective of this project is to introduce a new approach to Deep Reinforcement Learning Networks and to expand an existing and proved innovative model.

Objetives

Id.	Objective	Priority	Testable result
1	Literature review of Deep Reinforcement Learning Networks.	High	Inclusion into the State of the Art section of the project report.
2	Implementation of a Deep Reinforcement Learning Network using TensorFlow's C++ API.	High	Prove the functionality of the network using a toy example.
3	Extraction of features from the later neurons of the network.	High	Prediction of future states of the features.
4	Run experiments to test the system against a set of video game artificial intelligence testbeds.	High	Data in terms of speed of learning and improved behaviour of agents.
5	Development of a game demo using Unreal Engine (C++) to showcase the system in a more visual and practical environment.	Medium	Video of the playable demo to be published in a video hosting service.
6	Write the project report.	High	The project report itself to be submitted into the system before the deadline.

Critical context

In contrast with shallow learning, Deep Learning networks consist of one or more hidden layers between the input and output layers (Li, 2017). This allows the algorithm to use multiple processing layers, composed of numerous linear and non-linear transformations. At each one of these hidden layers, the input to each unit is computed as the weighted sum of units from the previous layer, resulting in a representation of the input from the previous layer.

Reinforcement Learning involves an agent interacting with an environment over time learning by trial and error (Li, 2017). At each time step, the agent

receives a state and selects an action from an action space. By following a specific policy, which is the behaviour of the agent, it receives a reward, positive or negative, and transitions to the next state depending on the reward function and the transition probability.

Deep Q-Network and extensions

As (Krakovsky, 2016) indicates, we are nowadays witnessing the renaissance of reinforcement learning, mainly due to the emerge of Deep Q-Networks (Mnih et al., 2015) and DeepMind's AlphaGo (Silver et al., 2016), a computer Go program that beat the european champion of this game and later won four games out of five against the best Go player in the world without handicaps on a full-sized 19x19 board. The contributions from (Mnih et al., 2015) were that it stabilized the training of action value function approximation with deep neural networks, designed an end-to-end Reinforcement Learning approach and trained a flexible network with the same algorithm, network architecture and hyperparameters to achieve a high level of performance on a set of different tasks.

Several extensions have been proposed to Deep Q-Networks. (Van Hasselt et al., 2016) introduced Double DQN, which found better policies than DQN on the Atari games. By proposing to prioritize experience replay, (Schaul et al., 2016) emphasized on the fact that important experience transitions should be replayed more frequently, making the learning process more efficient. The dueling network architecture by (Wang et al., 2015) mixed with Double DQN and prioritized experience replay performed better on the Atari games than the other approaches. Finally, (Mnih et al., 2016) introduced four different asynchronous methods for Reinforcement Learning: Q-learning, SARSA, n-step Q-learning and advantage actor-critic, being this last one the algorithm that performed the best in general.

Applications

Deep Reinforcement Learning Networks have been applied to a wide range of disciplines, from games to other technical and more generic areas.

In the games domain, in years before Deep Q-Networks appeared, (Tesauro, 1994) approached the classic board game Backgammon using neural networks to approximate the value function learned with temporal difference learning. Still in the territory of board games, a more recent proposition is the already commented development of AlphaGo (Silver et al., 2016). Even if it is not

trivial to apply Deep Reinforcement Learning Networks to videogames due to sparse and long term rewards, (Wu and Tian, 2017) applied advantage actor-critic with deep neural networks to train an agent in Doom, in order to predict next actions and the value function.

Other application areas of Deep Q-Networks include the experimentation with simulated robotics tasks or handle physical robots, spoken dialogue systems in which Deep Reinforcement Learning Networks are used to generate dialogues to model future reward for better coherence and ease of answering. Machine translation, text sequence prediction, neural architecture design or personalized web services have been other scenarios where researchers have applied the concepts of DQN. In a set of less technical areas, authors found interesting to investigate, due to the large array of opportunities and challenges, around healthcare, finance and music generation with the mentioned machine learning concepts (Li, 2017).

Approaches

Literature review

The literature review presented in this document is only the start point of a future more profound analysis of Deep Reinforcement Learning Networks, its extensions and applications. A collection of the most representative papers in the field has been collected, mainly using Google Scholar, and in the future, a similar strategy will be followed in order to extract relevant information of this machine learning topic and more precisely how it is applied to games.

Methodology

The, GPU supported, TensorFlow C++ API will be used to implement the Deep Reinforcement Learning Network using Microsoft's integrated development environment Visual Studio. Although TensorFlow's Python API is, at the moment, the most complete and easiest to use, it has been decided to use the C++ API due to the high presence of this programming language in the games industry. This fact is backed up by a planned and future integration of the implementation in a game engine such as Unreal Engine to develop a game demonstration that would visually showcase the characteristics of the implemented model. The decision about choosing Unreal Engine instead of other alternatives, such as Unity, is deliberate and due to the programming language, C++, that is used to write the source code in this game engine,

when Unity uses mainly C#.

First, Deep Learning and Reinforcement theoretic concepts are going to be studied and implemented to be properly understood using simple examples. Then, the Deep Q-Network framework (Mnih et al., 2015) will be adapted to learn an approximation of the value function from states in their original unprocessed form, allowing feature extraction and action with a partial network. The later level of the network will function independently in reward function approximation, as the actions will be selectable as inputs connected to this mentioned layer. It will be possible to extract the features from the deep network using techniques for discriminative feature extraction (Stuhlsatz et al., 2012) and these ones will be modelled in future steps. An additional output of the deep network will predict rewards directly from the discriminative features and action.

The output of this implementation will be an agent that can learn to play unseen video games by modelling the evolution over time of key features of an environment extracted using a deep neural network. The planned methodology only applies to time-series data, where a future state can be predicted, allowing to reduce the number of features that need to be modelled. We will compare the obtained performance with the one achieved in (Mnih et al., 2015) using quantitive methods, making emphasis in the average reward and average Q-value of the agent, using the same settings stablished in the mentioned paper, i.e., number of games, ϵ-greedy policy and initial random conditions. Experiments will be run in order to test the system against video game artificial intelligence testbeds, e.g. a simple gridworld toy example, and one sample from (Bellemare et al., 2012), in which the functionality of the network should demonstrate its competence.

Project report

A project report will be the ultimate product of this research. Preliminary, this will be composed by the following set of sections: Cover page, Abstract and Keywords, Acknowledgments, Contents, List of Figures, List of Tables, List of Listings, Introduction, State of the Art, Design, Implementation, Testing and Conclusions. Finally, two drafts will be produced before the final version is delivered in order to detect parts that would be subject of improvement.

Meetings with supervisor

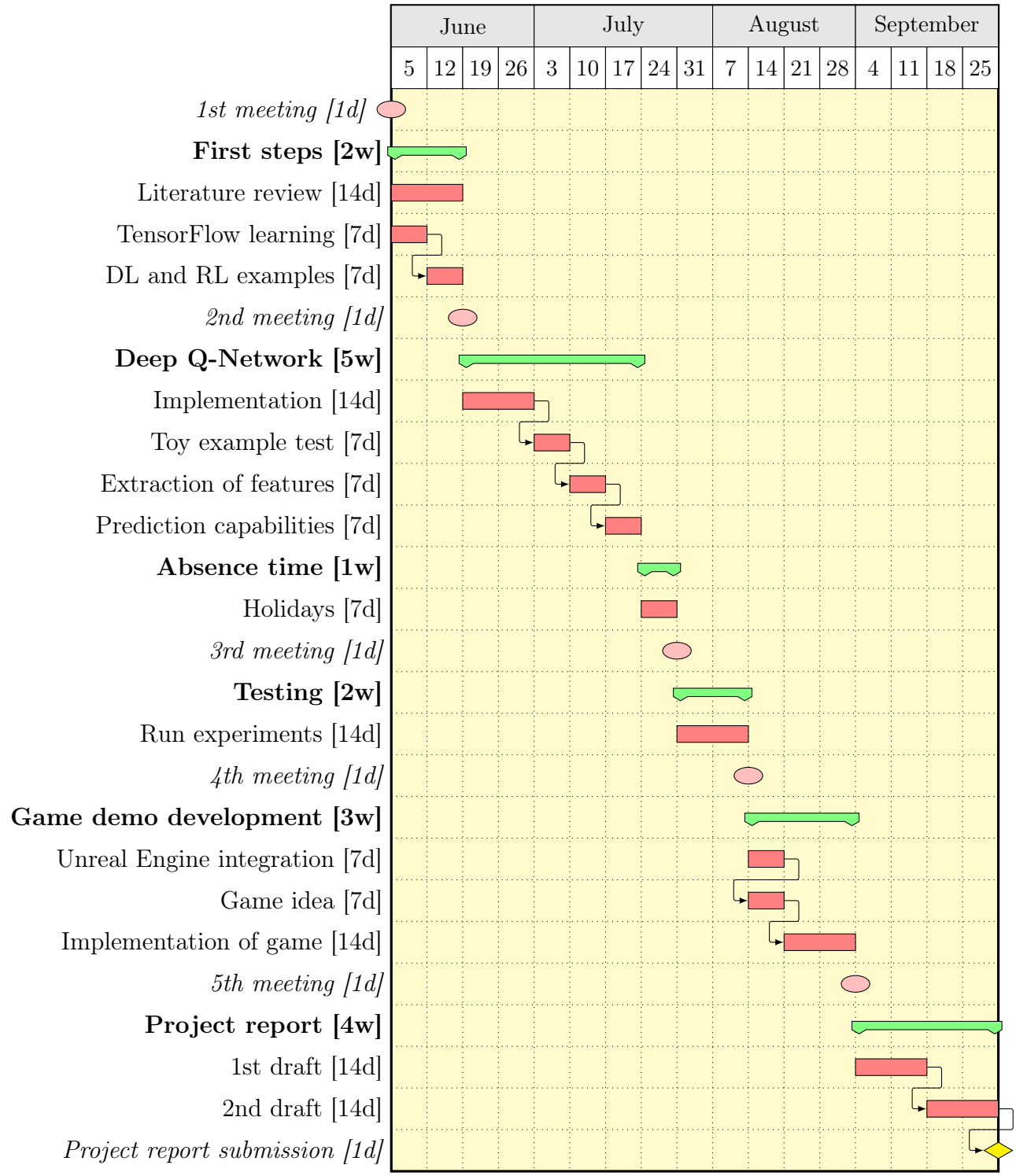
Although they are pending to be formally scheduled, meetings with the su-

pervisor will take place regularly so the project is being followed closely with the objective to meet the main requirements. In addition, these meetings will be useful because they will present unique opportunities to share the ongoing acquirement of knowledge with students undertaking other projects in the area of Deep Reinforcement Learning Networks.

Ethical, legal and professional issues

This project concerns the implementation of a model based on available software that is accessible for all the public which is free, if not used for commercial purposes. Aside, the project does not involve the participation of any human being, animal subject or the gathering of personal and sensitive data. For further details, the mandatory Ethics Review Form for BSc, MSc and MA Projects is presented in the last section of this document.

Work plan


June
July
August
September
5
12
19
26
3
10
17
24
31
7
14
21
28
4
11
18
25
1st meeting [1d]
First steps [2w]
Literature review [14d]
TensorFlow learning [7d]
DL and RL examples [7d]
2nd meeting [1d]
Deep Q-Network [5w]
Implementation [14d]
Toy example test [7d]
Extraction of features [7d]
Prediction capabilities [7d]
Absence time [1w]
Holidays [7d]
3rd meeting [1d]
Testing [2w]
Run experiments [14d]
4th meeting [1d]
Game demo development [3w]
Unreal Engine integration [7d]
Game idea [7d]
Implementation of game [14d]
5th meeting [1d]
Project report [4w]
1st draft [14d]
2nd draft [14d]
Project report submission [1d]

Risks

Id.	Risk description	Likelihood (L) (1 - 3)	Impact (I) (1 - 5)	Value (L x I)	Mitigation strategy
1	The TensorFlow's C++ API documentation is too sparse.	2	3	6	Change to use the TensorFlow's Python API instead.
2	Deep Reinforcement Learning Networks is hard to be implemented using TensorFlow.	2	3	6	Use Theano, a numerical computation library for Python.
3	Deep Reinforcement Learning Networks takes too long to be executed.	2	2	4	Use provided university processing power to run the experiments.
4	Not enough time to develop the game demonstration.	3	2	6	Since it is not the main objetive of the project, not achieving this is not essential.
5	Implemented code or partially written report is lost due to an accidental cause.	2	4	8	Use Git to back up the code and upload report versions to a file hosting service weekly.
6	Project too ambitious for the time allowed to finish it.	2	4	8	Redefine main objectives.

Ethics Review Form: BSc, MSc and MA Projects

Computer Science Research Ethics Committee (CSREC)

Undergraduate and postgraduate students undertaking their final project in the Department of Computer Science are required to consider the ethics of their project work and to ensure that it complies with research ethics guidelines. In some cases, a project will need approval from an ethics committee before it can proceed. Usually, but not always, this will be because the student is involving other people ("participants") in the project.

In order to ensure that appropriate consideration is given to ethical issues, all students must complete this form and attach it to their project proposal document. There are two parts:

Part A: Ethics Checklist. All students must complete this part. The checklist identifies whether the project requires ethical approval and, if so, where to apply for approval.

Part B: Ethics Proportionate Review Form. Students who have answered "no" to questions 1 – 18 and "yes" to question 19 in the ethics checklist must complete this part. The project supervisor has delegated authority to provide approval in this case. The approval may be provisional: the student may need to seek additional approval from the supervisor as the project progresses.

A.1 If your answer to any of the following questions (1 – 3) is YES, you must apply to an appropriate external ethics committee for approval.		*Delete as appropriate*
1.	Does your project require approval from the National Research Ethics Service (NRES)? For example, because you are recruiting current NHS patients or staff? If you are unsure, please check at http://www.hra.nhs.uk/research-community/before-you-apply/determine-which-review-body-approvals-are-required/.	**No**
2.	Does your project involve participants who are covered by the Mental Capacity Act? If so, you will need approval from an external ethics committee such as NRES or the Social Care Research Ethics Committee http://www.scie.org.uk/research/ethics-committee/.	**No**
3.	Does your project involve participants who are currently under the auspices of the Criminal Justice System? For example, but not limited to, people on remand, prisoners and those on probation? If so, you will need approval from the ethics approval system of the National Offender Management Service.	**No**

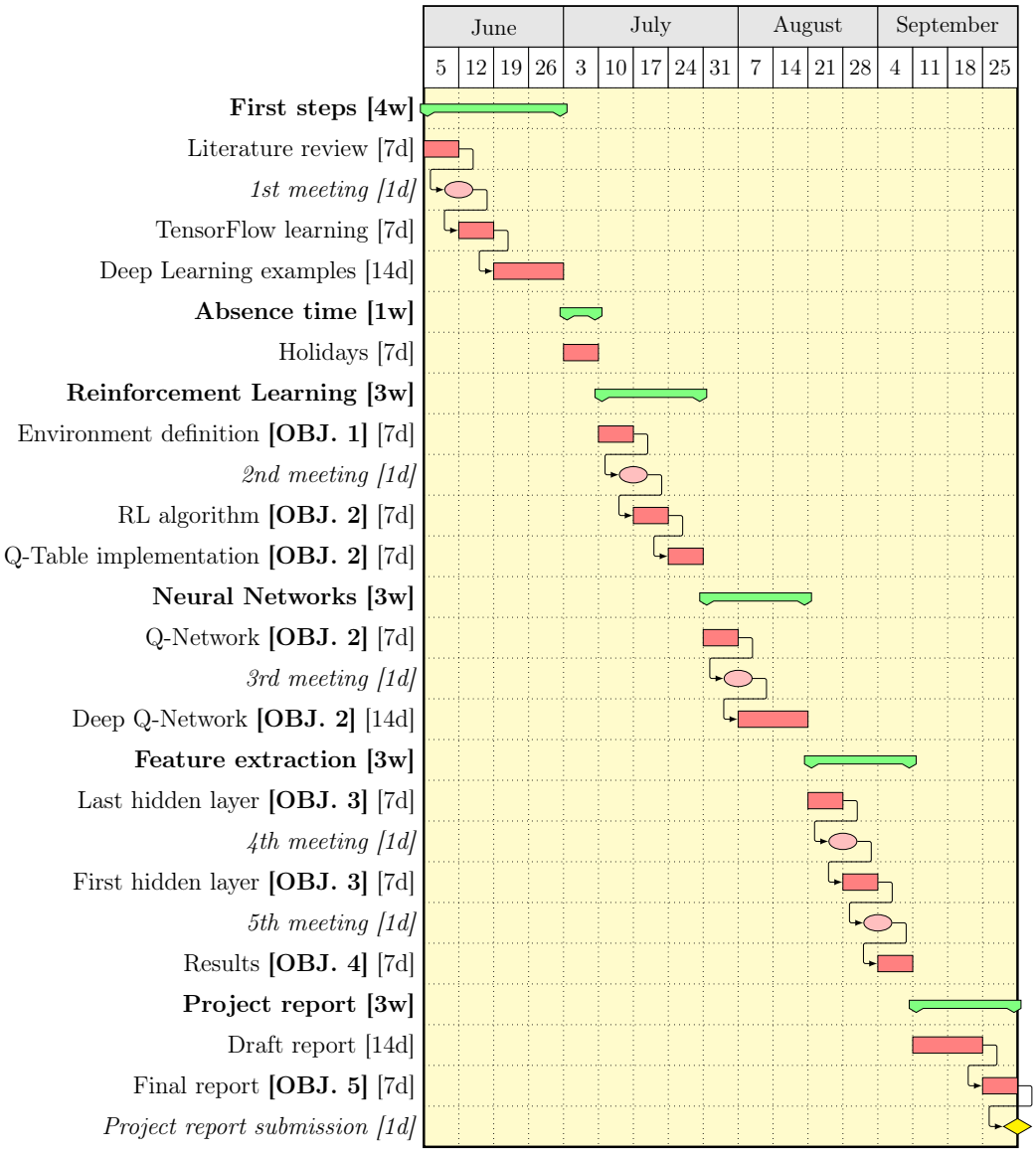
A.2 If your answer to any of the following questions (4 – 11) is YES, you must apply to the City University Senate Research Ethics Committee (SREC) for approval (unless you are applying to an external ethics committee).		*Delete as appropriate*
4.	Does your project involve participants who are unable to give informed consent? For example, but not limited to, people who may have a degree of learning disability or mental health problem, that means they are unable to make an informed decision on their own behalf?	**No**
5.	Is there a risk that your project might lead to disclosures from participants concerning their involvement in illegal activities?	**No**
6.	Is there a risk that obscene and or illegal material may need to be accessed for your project (including online content and other material)?	**No**
7.	Does your project involve participants disclosing information about sensitive subjects? For example, but not limited to, health status, sexual behaviour,	**No**

	political behaviour, domestic violence.	
8.	Does your project involve you travelling to another country outside of the UK, where the Foreign & Commonwealth Office has issued a travel warning? (See http://www.fco.gov.uk/en/)	**No**
9.	Does your project involve physically invasive or intrusive procedures? For example, these may include, but are not limited to, electrical stimulation, heat, cold or bruising.	**No**
10.	Does your project involve animals?	**No**
11.	Does your project involve the administration of drugs, placebos or other substances to study participants?	**No**

A.3 If your answer to any of the following questions (12 – 18) is YES, you must submit a full application to the Computer Science Research Ethics Committee (CSREC) for approval (unless you are applying to an external ethics committee or the Senate Research Ethics Committee). Your application may be referred to the Senate Research Ethics Committee.		*Delete as appropriate*
12.	Does your project involve participants who are under the age of 18?	**No**
13.	Does your project involve adults who are vulnerable because of their social, psychological or medical circumstances (vulnerable adults)? This includes adults with cognitive and / or learning disabilities, adults with physical disabilities and older people.	**No**
14.	Does your project involve participants who are recruited because they are staff or students of City University London? For example, students studying on a specific course or module. (If yes, approval is also required from the Head of Department or Programme Director.)	**No**
15.	Does your project involve intentional deception of participants?	**No**
16.	Does your project involve participants taking part without their informed consent?	**No**
17.	Does your project pose a risk to participants or other individuals greater than that in normal working life?	**No**
18.	Does your project pose a risk to you, the researcher, greater than that in normal working life?	**No**

A.4 If your answer to the following question (19) is YES and your answer to all questions 1 – 18 is NO, you must complete part B of this form.		
19.	Does your project involve human participants or their identifiable personal data? For example, as interviewees, respondents to a survey or participants in testing.	**No**

Appendix B: Work Plan



Appendix C: Source Code

Note:

Inspiration has been taken from the following GitHub repository, which contains a collection of Reinforcement Learning algorithms implemented by Arthur Juliani:
"A set of Deep Reinforcement Learning Agents implemented in Tensorflow"
More precisely, in order to define the Q-Network (and consequently the Deep Q-Networks) and train the agent with the Reinforcement Learning Q-Learning technique by using TensorFlow, this specific tutorial was followed:
"Q-Network.ipynb"
The source code has been used according to the permissions stated under the MIT License presented in the repository:
"The MIT License (MIT). Copyright (c) 2016 Arthur Juliani"

Listing 1: SourceCode/README.txt

```
List of programs included:
- 1QTable.py
- 2QNetwork.py
- 3DeepQNetwork2.py
- 4FeatureExtraction2.py

Programming language:
Python 3.5

External libraries needed:
- itertools
- math
- os
- time
- matplotlib.pyplot
- numpy
- tensorflow
- colorama
```

Listing 2: SourceCode/1QTable.py

```python
from __future__ import absolute_import, division, print_function

import math
import os
import time

import matplotlib.pyplot as plt
import numpy as np
from colorama import init, Fore, Style

if 'PYCHARM_HOSTED' in os.environ:
    convert = False
    strip = False
else:
    convert = None
    strip = None

init(convert=convert, strip=strip)

print(Fore.BLUE + Style.BRIGHT + "1QTable.py" + Style.RESET_ALL)

''' INITIAL PARAMETERS '''
environment = np.array([
    ('A', 'B', 'C', 'D'),
    ('E', 'F', 'G', 'H'),
    ('I', 'J', 'K', 'L'),
    ('M', 'N', 'O', 'P')
])
print(Fore.BLUE + "\nEnvironment:\n" + Style.RESET_ALL, environment)

initialState = ord('A') - 65
print(Fore.YELLOW + "Initial state:", chr(initialState + 65))
goalState = ord('P') - 65
print(Fore.GREEN + "Goal state:", chr(goalState + 65))
listOfHoles = np.array(['F', 'H', 'L', 'M'])
print(Fore.RED + "List of holes:", listOfHoles, Style.RESET_ALL)

e = 0.1

```

```
''' INITIALIZATION '''
R = []
states = []
for i in range(environment.shape[0]):
    for j in range(environment.shape[1]):
        row = []

        # left
        if i - 1 >= 0:
            row.append(environment[i - 1][j])
        else:
            row.append(environment[i][j])
        # right
        if i + 1 < environment.shape[0]:
            row.append(environment[i + 1][j])
        else:
            row.append(environment[i][j])
        # down
        if j + 1 < environment.shape[1]:
            row.append(environment[i][j + 1])
        else:
            row.append(environment[i][j])
        # up
        if j - 1 >= 0:
            row.append(environment[i][j - 1])
        else:
            row.append(environment[i][j])

        R.append(row)
        states.append(environment[i][j])
R = np.array(R)
# print("R:\n", R)
# print("States:\n", states)

numStates = R.shape[0]
numActions = R.shape[1]
# print("numStates:", numStates, ", numActions:", numActions)

```

```
78 Q = np.random.uniform(-1.0 / math.sqrt(numStates), 1.0 /
       math.sqrt(numStates), (numStates, numActions))
79 # print("Q:\n", Q)
80
81 alpha = 0.05 # 1.0
82 gamma = 0.99 # 0.8
83
84 episodes = 0
85 numEpisodes = 2000
86
87 firstEpisodeReward1 = 0
88 reward1Reached = False
89
90 listOfStepsPerEpisode = []
91 accumulatedRewards = []
92
93 startTime = time.time()
94
95 ''' TRAINING '''
96 print(Fore.BLUE + "\nTraining:" + Style.RESET_ALL)
97 while episodes < numEpisodes:
98     state = initialState
99
100    stepsPerEpisode = 0
101    accumulatedReward = 0
102    goalReached = False
103
104    if episodes % 250 == 0:
105        print(Fore.BLUE + "Episode:" + Style.RESET_ALL, episodes)
106
107    while stepsPerEpisode < 99 and goalReached is False:
108        possibleActionsR = []
109        possibleActionsQValue = []
110        possibleActionsQIndex = []
111        for j in range(numActions):
112            possibleActionsR.append((R[state][j], j))
113            possibleActionsQValue.append(Q[state][j])
114            possibleActionsQIndex.append(j)
115        # print("Possible actions R:", possibleActionsR)
```

```
116         # print("Possible actions Q value:", possibleActionsQValue)
117         # print("Possible actions Q index:", possibleActionsQIndex)
118
119         actionWithMaxQValue =
                possibleActionsQIndex[np.array(possibleActionsQValue).argmax(axis=0)]
120         # print("Action with maximum Q value:", actionWithMaxQValue)
121
122         action = [(R[state][actionWithMaxQValue], actionWithMaxQValue)]
123         if np.random.rand(1, 1) < e:
124             action = [possibleActionsR[np.random.randint(0,
                    len(possibleActionsR))]]
125         # print("Action:", action)
126
127         nextState = 0
128         for i in range(len(states)):
129             for a, _ in action:
130                 if states[i] == a:
131                     nextState = i
132         # print("Action number:", actionNumber)
133
134         possibleActionsFromAction = []
135         for j in range(numActions):
136             if R[nextState][j] != '-':
137                 possibleActionsFromAction.append((R[nextState][j], j))
138         # print("Possible actions from action:", action, ":",
                possibleActionsFromAction)
139
140         qValues = []
141         for i in range(len(possibleActionsFromAction)):
142             qValues.append(Q[nextState][i])
143         # print("qValues:", qValues)
144
145         maxQValue = max(qValues)
146         # print("maxQValue:", maxQValue)
147
148         reward = 0
149         if nextState == goalState:
150             reward = 1
151         if action[0][0] in listOfHoles:
```

```
152             reward = -0.25 * len(listOfHoles)
153         accumulatedReward += reward
154 
155         oldQValue = Q[state][action[0][1]]
156         newQValue = oldQValue + alpha * (reward + (gamma * maxQValue))
                - oldQValue
157 
158         Q[state][action[0][1]] = newQValue
159 
160         state = nextState
161         stepsPerEpisode += 1
162 
163         if nextState == goalState:
164             goalReached = True
165 
166     if accumulatedReward == 1.0 and not reward1Reached:
167         firstEpisodeReward1 = episodes
168         reward1Reached = True
169 
170     episodes += 1
171     listOfStepsPerEpisode.append(stepsPerEpisode)
172     accumulatedRewards.append(accumulatedReward)
173 
174 elapsedTime = time.time() - startTime
175 
176 ''' TESTING '''
177 print(Fore.BLUE + "\nFinal Q matrix:\n" + Style.RESET_ALL, Q)
178 
179 paths = []
180 for i in range(numStates):
181     state = i
182     path = []
183 
184     shouldBreak = False
185     steps = 0
186 
187     while state != goalState and not shouldBreak:
188         possibleActions = []
189         for j in range(numActions):
```

```
190             possibleActions.append((Q[state][j], j))
191         # print("Possible actions:", possibleActions)
192 
193         action = [max(possibleActions)]
194         # print("Action:", action)
195 
196         stateChar = R[state][action[0][1]]
197         # print("State:", stateChar)
198         state = ord(stateChar) - 65
199 
200         path.append(stateChar)
201 
202         steps += 1
203         if steps > numStates:
204             shouldBreak = True
205     paths.append(path)
206 
207 print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
208 for i in range(len(paths)):
209     print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
        + 65), ":", paths[i])
210 
211 ''' STATS '''
212 print(Fore.BLUE + "\nStats:")
213 
214 minutes, seconds = divmod(elapsedTime, 60)
215 print(Fore.BLUE + "Training time:" + Style.RESET_ALL, minutes,
    "minutes,", seconds, "seconds")
216 
217 print(Fore.BLUE + "Accumulated reward over time:" + Style.RESET_ALL,
    sum(accumulatedRewards) / numEpisodes, "%")
218 
219 print(Fore.BLUE + "First episode with reward 1.0:" + Style.RESET_ALL,
    firstEpisodeReward1)
220 
221 plt.suptitle('Rewards per episode', fontsize=14, fontweight='bold')
222 plt.plot(accumulatedRewards)
223 plt.xlabel('Episodes')
224 plt.ylabel('Rewards')
```

```
225 plt.show()
226
227 plt.suptitle('Steps per episode', fontsize=14, fontweight='bold')
228 plt.plot(listOfStepsPerEpisode)
229 plt.xlabel('Episodes')
230 plt.ylabel('Steps')
231 plt.show()
```

Listing 3: SourceCode/2QNetwork.py

```
1  from __future__ import absolute_import, division, print_function
2
3  import math
4  import os
5  import time
6
7  import matplotlib.pyplot as plt
8  import numpy as np
9  import tensorflow as tf
10 from colorama import init, Fore, Style
11
12 os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
13
14 if 'PYCHARM_HOSTED' in os.environ:
15     convert = False
16     strip = False
17 else:
18     convert = None
19     strip = None
20
21 init(convert=convert, strip=strip)
22
23 print(Fore.BLUE + Style.BRIGHT + "2QNetwork.py" + Style.RESET_ALL)
24
25 ''' INITIAL PARAMETERS '''
26 environment = np.array([
27     ('A', 'B', 'C', 'D'),
28     ('E', 'F', 'G', 'H'),
```

```
    ('I', 'J', 'K', 'L'),
    ('M', 'N', 'O', 'P')
])
print(Fore.BLUE + "\nEnvironment:\n" + Style.RESET_ALL, environment)

initialState = ord('A') - 65
print(Fore.YELLOW + "Initial state:", chr(initialState + 65))
goalState = ord('P') - 65
print(Fore.GREEN + "Goal state:", chr(goalState + 65))
listOfHoles = np.array(['F', 'H', 'L', 'M'])
print(Fore.RED + "List of holes:", listOfHoles, Style.RESET_ALL)

e = 0.1

''' INITIALIZATION '''
R = []
states = []
for i in range(environment.shape[0]):
    for j in range(environment.shape[1]):
        row = []

        # left
        if i - 1 >= 0:
            row.append(environment[i - 1][j])
        else:
            row.append(environment[i][j])
        # right
        if i + 1 < environment.shape[0]:
            row.append(environment[i + 1][j])
        else:
            row.append(environment[i][j])
        # down
        if j + 1 < environment.shape[1]:
            row.append(environment[i][j + 1])
        else:
            row.append(environment[i][j])
        # up
        if j - 1 >= 0:
            row.append(environment[i][j - 1])
```

```
68          else:
69              row.append(environment[i][j])
70
71          R.append(row)
72
73          states.append(environment[i][j])
74  R = np.array(R)
75  # print("R:\n", R)
76  # print("States:\n", states)
77
78  numStates = R.shape[0]
79  numActions = R.shape[1]
80  # print("numStates:", numStates, ", numActions:", numActions)
81
82  # Implementing the network itself
83  tf.reset_default_graph()
84
85  # Input: 1x16
86  X = tf.placeholder(shape=[1, numStates], dtype=tf.float32, name="X")
87  # Output: 1x4
88  Y = tf.placeholder(shape=[1, numActions], dtype=tf.float32, name="Y")
89
90  # Weights: 16x4
91  W = tf.Variable(tf.random_uniform([numStates, numActions], -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
92                  name="W")
93  # 1: Sigmoid
94  # YPredicted = tf.nn.sigmoid(tf.matmul(X, W), name="YPredicted")
95  # 2: Tanh
96  YPredicted = tf.nn.tanh(tf.matmul(X, W), name="YPredicted")
97  maxPredictedY = tf.argmax(YPredicted, 1, name="maxPredictedY")
98
99  # We obtain the loss by taking the sum of squares difference between
        the output and the predicted output
100 loss = tf.reduce_sum(tf.square(Y - YPredicted), name="loss")
101 trainer = tf.train.GradientDescentOptimizer(learning_rate=0.05,
        name="GDOptimizer").minimize(loss)
102
103 gamma = 0.99
```

```
104
105 episodes = 0
106 numEpisodes = 2000
107
108 firstEpisodeReward1 = 0
109 reward1Reached = False
110
111 listOfStepsPerEpisode = []
112 accumulatedRewards = []
113
114 startTime = time.time()
115
116 ''' TRAINING '''
117 print(Fore.BLUE + "\nTraining:" + Style.RESET_ALL)
118 sess = tf.Session()
119 initVariables = tf.global_variables_initializer()
120 sess.run(initVariables)
121
122 # TensorBoard
123 # writer = tf.summary.FileWriter('./2QNetwork_graphs', sess.graph)
124
125 while episodes < numEpisodes:
126     state = initialState
127
128     stepsPerEpisode = 0
129     accumulatedReward = 0
130
131     if episodes % 250 == 0:
132         print(Fore.BLUE + "Episode:" + Style.RESET_ALL, episodes)
133
134     while stepsPerEpisode < 99:
135         # Choose an action by greedily (with e chance of random
              action) from the Q-network
136         a, qValues = sess.run([maxPredictedY, YPredicted],
              feed_dict={X: np.identity(numStates)[state:state + 1]})
137
138         action = [(R[state][a], a)]
139         if np.random.rand(1, 1) < e:
140             random_action = np.random.randint(0, numActions)
```

```
            action = [(R[state][random_action], random_action)]
        # print("Action:", action)

        # Next state
        nextState = states.index(action[0][0])

        # Reward
        reward = 0
        if nextState == goalState:
            reward = 1
        if action[0][0] in listOfHoles:
            reward = -0.25 * len(listOfHoles)
        accumulatedReward += reward

        # Obtain the next Q values by feeding the next state through
            our network
        nextQValues = sess.run(YPredicted, feed_dict={X:
            np.identity(numStates)[nextState:nextState + 1]})
        # Obtain the max of the next Q values and set our target value
            for chosen action
        maxNextQValues = np.max(nextQValues)
        targetQ = qValues

        # Q-learning formula that directly sets the "new Q value"
        targetQ[0, action[0][1]] = reward + gamma * maxNextQValues

        # Train our network using target and predicted Q values
        _, Q = sess.run([trainer, W], feed_dict={X:
            np.identity(numStates)[state:state + 1], Y: targetQ})

        state = nextState
        stepsPerEpisode += 1

        # If the next state is the goal state, finish this episode
            (goal reached)
        if nextState == goalState:
            break

    if accumulatedReward == 1.0 and not reward1Reached:
```

```
175         firstEpisodeReward1 = episodes
176         reward1Reached = True
177 
178     episodes += 1
179     listOfStepsPerEpisode.append(stepsPerEpisode)
180     accumulatedRewards.append(accumulatedReward)
181 
182 # writer.close()
183 
184 elapsedTime = time.time() - startTime
185 
186 ''' TESTING '''
187 # Testing with Q matrix
188 print(Fore.BLUE + "\nTesting with Q Matrix" + Style.RESET_ALL)
189 print(Fore.BLUE + "Final Q matrix (weights of the NN):\n" +
    Style.RESET_ALL, Q)
190 
191 paths = []
192 for i in range(numStates):
193     state = i
194     path = []
195 
196     shouldBreak = False
197     steps = 0
198 
199     while state != goalState and not shouldBreak:
200         possibleActions = []
201         for j in range(numActions):
202             possibleActions.append((Q[state][j], j))
203         # print("Possible actions:", possibleActions)
204 
205         action = [max(possibleActions)]
206         # print("Action:", action)
207 
208         stateChar = R[state][action[0][1]]
209         # print("State:", stateChar)
210         state = ord(stateChar) - 65
211 
212         path.append(stateChar)
```

```
213
214         steps += 1
215         if steps > numStates:
216             shouldBreak = True
217     paths.append(path)
218
219 print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
220 for i in range(len(paths)):
221     print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
        + 65), ":", paths[i])
222
223 # Testing with activations of neurons of the output layer
224 print(Fore.BLUE + "\nTesting with activations of neurons of the
    output layer" + Style.RESET_ALL)
225 print(Fore.BLUE + "Activations of neurons of the output layer:" +
    Style.RESET_ALL)
226 for i in range(0, 16):
227     print("From", chr(i + 65), ":", sess.run(YPredicted,
        feed_dict={X: np.identity(numStates)[i:i + 1]}))
228
229 activationsOL = []
230 for i in range(numStates):
231     activations = sess.run(YPredicted, feed_dict={X:
        np.identity(numStates)[i:i + 1]})
232     activationsOL.append(activations.tolist())
233 activationsOL = [activationsOL[x][0] for x in range(numStates)]
234
235 fromStateToNextState = []
236 for i in range(len(activationsOL)):
237     best = max(activationsOL[i])
238     fromStateToNextState.append([chr(i + 65),
        R[i][activationsOL[i].index(best)]])
239
240 paths = []
241 for i in range(numStates):
242     shouldBreak = False
243     steps = 0
244
245     nextState = fromStateToNextState[i][1]
```

```
246
247     path = []
248     path.append(nextState)
249
250     while nextState != chr(goalState + 65) and not shouldBreak:
251         for stateAndNextState in fromStateToNextState:
252             if stateAndNextState[0] == nextState and
                    stateAndNextState[0] != chr(goalState + 65):
253                 nextState = stateAndNextState[1]
254                 path.append(nextState)
255
256                 steps += 1
257                 if steps > 15:
258                     shouldBreak = True
259
260     paths.append(path)
261
262 print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
263 for i in range(len(paths)):
264     print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
        + 65), ":", paths[i])
265
266 ''' STATS '''
267 print(Fore.BLUE + "\nStats:" + Style.RESET_ALL)
268
269 minutes, seconds = divmod(elapsedTime, 60)
270 print(Fore.BLUE + "Training time:" + Style.RESET_ALL, minutes,
    "minutes,", seconds, "seconds")
271
272 print(Fore.BLUE + "Accumulated reward over time:" + Style.RESET_ALL,
    sum(accumulatedRewards) / numEpisodes, "%")
273
274 print(Fore.BLUE + "First episode with reward 1.0:" + Style.RESET_ALL,
    firstEpisodeReward1)
275
276 plt.suptitle('Rewards per episode', fontsize=14, fontweight='bold')
277 plt.plot(accumulatedRewards)
278 plt.xlabel('Episodes')
279 plt.ylabel('Rewards')
```

```
280 plt.show()
281
282 plt.suptitle('Steps per episode', fontsize=14, fontweight='bold')
283 plt.plot(listOfStepsPerEpisode)
284 plt.xlabel('Episodes')
285 plt.ylabel('Steps')
286 plt.show()
```

Listing 4: SourceCode/3DeepQNetwork2.py

```
1  from __future__ import absolute_import, division, print_function
2
3  import math
4  import os
5  import time
6
7  import matplotlib.pyplot as plt
8  import numpy as np
9  import tensorflow as tf
10 from colorama import init, Fore, Style
11
12 os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
13
14 if 'PYCHARM_HOSTED' in os.environ:
15     convert = False
16     strip = False
17 else:
18     convert = None
19     strip = None
20
21 init(convert=convert, strip=strip)
22
23 print(Fore.BLUE + Style.BRIGHT + "3DeepQNetwork2.py" +
       Style.RESET_ALL)
24
25 ''' INITIAL PARAMETERS '''
26 environment = np.array([
27     ('A', 'B', 'C', 'D'),
```

```
    ('E', 'F', 'G', 'H'),
    ('I', 'J', 'K', 'L'),
    ('M', 'N', 'O', 'P')
])
print(Fore.BLUE + "\nEnvironment:\n" + Style.RESET_ALL, environment)

initialState = ord('A') - 65
print(Fore.YELLOW + "Initial state:", chr(initialState + 65))
goalState = ord('P') - 65
print(Fore.GREEN + "Goal state:", chr(goalState + 65))
listOfHoles = np.array(['F', 'H', 'L', 'M'])
print(Fore.RED + "List of holes:", listOfHoles, Style.RESET_ALL)

e = 0.1

''' INITIALIZATION '''
R = []
states = []
for i in range(environment.shape[0]):
    for j in range(environment.shape[1]):
        row = []

        # left
        if i - 1 >= 0:
            row.append(environment[i - 1][j])
        else:
            row.append(environment[i][j])
        # right
        if i + 1 < environment.shape[0]:
            row.append(environment[i + 1][j])
        else:
            row.append(environment[i][j])
        # down
        if j + 1 < environment.shape[1]:
            row.append(environment[i][j + 1])
        else:
            row.append(environment[i][j])
        # up
        if j - 1 >= 0:
```

```
67              row.append(environment[i][j - 1])
68          else:
69              row.append(environment[i][j])
70
71          R.append(row)
72
73          states.append(environment[i][j])
74  R = np.array(R)
75  # print("R:\n", R)
76  # print("States:\n", states)
77
78  numStates = R.shape[0]
79  numActions = R.shape[1]
80  # print("numStates:", numStates, ", numActions:", numActions)
81
82  # Implementing the network itself
83  tf.reset_default_graph()
84
85  # Input: 1x16
86  X = tf.placeholder(shape=[1, numStates], dtype=tf.float32, name="X")
87  # Output: 1x4
88  Y = tf.placeholder(shape=[1, numActions], dtype=tf.float32, name="Y")
89
90  # First hidden layer
91  # Weights: 16x12
92  # Biases: 12
93  hiddenLayer1 = 12
94  weightsHL1 = tf.Variable(
95      tf.random_uniform((numStates, hiddenLayer1), -1.0 /
            math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
96      name="weightsHL1")
97  biasesHL1 = tf.Variable(tf.zeros([hiddenLayer1]), name="biasesHL1")
98  h1 = tf.nn.tanh((tf.matmul(X, weightsHL1) + biasesHL1),
        name="hiddenLayer1")
99
100 # Second hidden layer
101 # Weights: 12x8
102 # Biases: 8
103 hiddenLayer2 = 8
```

```
104 weightsHL2 = tf.Variable(
105     tf.random_uniform((hiddenLayer1, hiddenLayer2), -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
106     name="weightsHL2")
107 biasesHL2 = tf.Variable(tf.zeros([hiddenLayer2]), name="biasesHL2")
108 h2 = tf.nn.tanh((tf.matmul(h1, weightsHL2) + biasesHL2),
    name="hiddenLayer2")
109
110 # Third hidden layer
111 # Weights: 8x2
112 # Biases: 2
113 hiddenLayer3 = 2
114 weightsHL3 = tf.Variable(
115     tf.random_uniform((hiddenLayer2, hiddenLayer3), -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
116     name="weightsHL3")
117 biasesHL3 = tf.Variable(tf.zeros([hiddenLayer3]), name="biasesHL3")
118 h3 = tf.nn.tanh((tf.matmul(h2, weightsHL3) + biasesHL3),
    name="hiddenLayer3")
119
120 # Output layer
121 # Weights: 4x4
122 # Biases: 4
123 weightsOL = tf.Variable(
124     tf.random_uniform((hiddenLayer3, numActions), -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
125     name="weightsOL")
126 biasesOL = tf.Variable(tf.zeros([numActions]), name="biasesOL")
127 YPredicted = tf.nn.tanh((tf.matmul(h3, weightsOL) + biasesOL),
    name="YPredicted")
128 maxPredictedY = tf.argmax(YPredicted, 1, name="maxPredictedY")
129
130 loss = tf.reduce_sum(tf.square(Y - YPredicted), name="loss")
131 trainer = tf.train.GradientDescentOptimizer(learning_rate=0.05,
    name="GDOptimizer").minimize(loss)
132
133 # Some parameters
134 gamma = 0.99
135
```

```
episodes = 0
numEpisodes = 2000

firstEpisodeReward1 = 0
reward1Reached = False

listOfStepsPerEpisode = []
accumulatedRewards = []

startTime = time.time()

''' TRAINING '''
print(Fore.BLUE + "\nTraining:" + Style.RESET_ALL)
sess = tf.Session()
initVariables = tf.global_variables_initializer()
sess.run(initVariables)

# TensorBoard
# writer = tf.summary.FileWriter('./3DeepQNetwork_graphs', sess.graph)

while episodes < numEpisodes:
    state = initialState

    stepsPerEpisode = 0
    accumulatedReward = 0

    if episodes % 250 == 0:
        print(Fore.BLUE + "Episode:" + Style.RESET_ALL, episodes)

    while stepsPerEpisode < 99:
        # Choose an action by greedily (with e chance of random
            action) from the Q-network
        a, qValues = sess.run([maxPredictedY, YPredicted],
            feed_dict={X: np.identity(numStates)[state:state + 1]})

        action = [(R[state][a], a)]
        if np.random.rand(1, 1) < e:
            random_action = np.random.randint(0, numActions)
            action = [(R[state][random_action], random_action)]
```

```
173         # print("Action:", action)
174
175         # Next state
176         nextState = states.index(action[0][0])
177
178         # Reward
179         reward = 0
180         if nextState == goalState:
181             reward = 1
182         if action[0][0] in listOfHoles:
183             reward = -0.25 * len(listOfHoles)
184         accumulatedReward += reward
185
186         # Obtain the next Q values by feeding the next state through
                our network
187         nextQValues = sess.run(YPredicted, feed_dict={X:
                np.identity(numStates)[nextState:nextState + 1]})
188         # Obtain the max of the next Q values and set our target value
                for chosen action
189         maxNextQValues = np.max(nextQValues)
190         targetQ = qValues
191
192         # Q-learning formula that directly sets the "new Q value"
193         targetQ[0, action[0][1]] = reward + gamma * maxNextQValues
194
195         # Train our network using target and predicted Q values
196         sess.run(trainer, feed_dict={X:
                np.identity(numStates)[state:state + 1], Y: targetQ})
197
198         state = nextState
199         stepsPerEpisode += 1
200
201         # If the next state is the goal state, finish this episode
                (goal reached)
202         if nextState == goalState:
203             break
204
205     if accumulatedReward == 1.0 and not reward1Reached:
206         firstEpisodeReward1 = episodes
```

```
207         reward1Reached = True
208
209     episodes += 1
210     listOfStepsPerEpisode.append(stepsPerEpisode)
211     accumulatedRewards.append(accumulatedReward)
212
213 # writer.close()
214
215 elapsedTime = time.time() - startTime
216
217 ''' TESTING '''
218 print(Fore.BLUE + "Activations of neurons of the first hidden layer:"
        + Style.RESET_ALL)
219 for i in range(0, 16):
220     print("From", chr(i + 65), ":", sess.run(h1, feed_dict={X:
            np.identity(numStates)[i:i + 1]}))
221 print(Fore.BLUE + "\nActivations of neurons of the second hidden
        layer:" + Style.RESET_ALL)
222 for i in range(0, 16):
223     print("From", chr(i + 65), ":", sess.run(h2, feed_dict={X:
            np.identity(numStates)[i:i + 1]}))
224 print(Fore.BLUE + "\nActivations of neurons of the third hidden
        layer:" + Style.RESET_ALL)
225 for i in range(0, 16):
226     print("From", chr(i + 65), ":", sess.run(h3, feed_dict={X:
            np.identity(numStates)[i:i + 1]}))
227
228 # Testing with Q matrix
229 print(Fore.BLUE + "\nTesting with Q Matrix" + Style.RESET_ALL)
230 a = sess.run(tf.matmul(weightsHL1, weightsHL2)) # 16x8
231 b = sess.run(tf.matmul(a, weightsHL3)) # 16x2
232 Q = sess.run(tf.matmul(b, weightsOL)) # 16x4
233 print(Fore.BLUE + "Final Q matrix (weights of the NN):\n" +
        Style.RESET_ALL, Q)
234
235 paths = []
236 for i in range(numStates):
237     state = i
238     path = []
```

```
239
240     shouldBreak = False
241     steps = 0
242
243     while state != goalState and not shouldBreak:
244         possibleActions = []
245         for j in range(numActions):
246             possibleActions.append((Q[state][j], j))
247         # print("Possible actions:", possibleActions)
248
249         action = [max(possibleActions)]
250         # print("Action:", action)
251
252         stateChar = R[state][action[0][1]]
253         # print("State:", stateChar)
254         state = ord(stateChar) - 65
255
256         path.append(stateChar)
257
258         steps += 1
259         if steps > numStates:
260             shouldBreak = True
261     paths.append(path)
262
263 print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
264 for i in range(len(paths)):
265     print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
        + 65), ":", paths[i])
266
267 # Testing with activations of neurons of the output layer
268 print(Fore.BLUE + "\nTesting with activations of neurons of the
    output layer" + Style.RESET_ALL)
269 print(Fore.BLUE + "Activations of neurons of the output layer:" +
    Style.RESET_ALL)
270 for i in range(numStates):
271     print("From", chr(i + 65), ":", sess.run(YPredicted,
        feed_dict={X: np.identity(numStates)[i:i + 1]}))
272
273 activationsOL = []
```

```
for i in range(numStates):
    activations = sess.run(YPredicted, feed_dict={X:
        np.identity(numStates)[i:i + 1]})
    activationsOL.append(activations.tolist())
activationsOL = [activationsOL[x][0] for x in range(numStates)]

fromStateToNextState = []
for i in range(len(activationsOL)):
    best = max(activationsOL[i])
    fromStateToNextState.append([chr(i + 65),
        R[i][activationsOL[i].index(best)]])

paths = []
for i in range(numStates):
    shouldBreak = False
    steps = 0

    nextState = fromStateToNextState[i][1]

    path = []
    path.append(nextState)

    while nextState != chr(goalState + 65) and not shouldBreak:
        for stateAndNextState in fromStateToNextState:
            if stateAndNextState[0] == nextState and
                stateAndNextState[0] != chr(goalState + 65):
                nextState = stateAndNextState[1]
                path.append(nextState)

                steps += 1
                if steps > 15:
                    shouldBreak = True

    paths.append(path)

print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
for i in range(len(paths)):
    print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
        + 65), ":", paths[i])
```

```
309
310 ''' STATS '''
311 print(Fore.BLUE + "\nStats:" + Style.RESET_ALL)
312
313 minutes, seconds = divmod(elapsedTime, 60)
314 print(Fore.BLUE + "Training time:" + Style.RESET_ALL, minutes,
        "minutes,", seconds, "seconds")
315
316 print(Fore.BLUE + "Accumulated reward over time:" + Style.RESET_ALL,
        sum(accumulatedRewards) / numEpisodes, "%")
317
318 print(Fore.BLUE + "First episode with reward 1.0:" + Style.RESET_ALL,
        firstEpisodeReward1)
319
320 plt.suptitle('Rewards per episode', fontsize=14, fontweight='bold')
321 plt.plot(accumulatedRewards)
322 plt.xlabel('Episodes')
323 plt.ylabel('Rewards')
324 plt.show()
325
326 plt.suptitle('Steps per episode', fontsize=14, fontweight='bold')
327 plt.plot(listOfStepsPerEpisode)
328 plt.xlabel('Episodes')
329 plt.ylabel('Steps')
330 plt.show()
```

Listing 5: SourceCode/4FeatureExtraction2.py

```
1 from __future__ import absolute_import, division, print_function
2
3 import itertools
4 import math
5 import os
6 import time
7
8 import matplotlib.pyplot as plt
9 import numpy as np
10 import tensorflow as tf
```

```
from colorama import init, Fore, Style

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

if 'PYCHARM_HOSTED' in os.environ:
    convert = False
    strip = False
else:
    convert = None
    strip = None

init(convert=convert, strip=strip)

print(Fore.BLUE + Style.BRIGHT + "4FeatureExtraction2.py" +
    Style.RESET_ALL)

''' INITIAL PARAMETERS '''
environment = np.array([
    ('A', 'B', 'C', 'D'),
    ('E', 'F', 'G', 'H'),
    ('I', 'J', 'K', 'L'),
    ('M', 'N', 'O', 'P')
])
print(Fore.BLUE + "\nEnvironment:\n" + Style.RESET_ALL, environment)

initialState = ord('A') - 65
print(Fore.YELLOW + "Initial state:", chr(initialState + 65))
goalState = ord('P') - 65
print(Fore.GREEN + "Goal state:", chr(goalState + 65))
listOfHoles = np.array(['F', 'H', 'L', 'M'])
print(Fore.RED + "List of holes:", listOfHoles, Style.RESET_ALL)

e = 0.1

''' INITIALIZATION '''
R = []
states = []
for i in range(environment.shape[0]):
    for j in range(environment.shape[1]):
```

```
        row = []

        # left
        if i - 1 >= 0:
            row.append(environment[i - 1][j])
        else:
            row.append(environment[i][j])
        # right
        if i + 1 < environment.shape[0]:
            row.append(environment[i + 1][j])
        else:
            row.append(environment[i][j])
        # down
        if j + 1 < environment.shape[1]:
            row.append(environment[i][j + 1])
        else:
            row.append(environment[i][j])
        # up
        if j - 1 >= 0:
            row.append(environment[i][j - 1])
        else:
            row.append(environment[i][j])

        R.append(row)

        states.append(environment[i][j])
R = np.array(R)
# print("R:\n", R)
# print("States:\n", states)

numStates = R.shape[0]
numActions = R.shape[1]
# print("numStates:", numStates, ", numActions:", numActions)

# NOT REDUCED DEEP NEURAL NETWORK
# Implementing the network itself
tf.reset_default_graph()

# Input: 1x16
```

```
X = tf.placeholder(shape=[1, numStates], dtype=tf.float32, name="X")
# Output: 1x4
Y = tf.placeholder(shape=[1, numActions], dtype=tf.float32, name="Y")

# First hidden layer
# Weights: 16x12
# Biases: 12
hiddenLayer1 = 12
weightsHL1 = tf.Variable(
    tf.random_uniform((numStates, hiddenLayer1), -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
    name="weightsHL1")
biasesHL1 = tf.Variable(tf.zeros([hiddenLayer1]), name="biasesHL1")
h1 = tf.nn.tanh((tf.matmul(X, weightsHL1) + biasesHL1),
    name="hiddenLayer1")

# Second hidden layer
# Weights: 12x8
# Biases: 8
hiddenLayer2 = 8
weightsHL2 = tf.Variable(
    tf.random_uniform((hiddenLayer1, hiddenLayer2), -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
    name="weightsHL2")
biasesHL2 = tf.Variable(tf.zeros([hiddenLayer2]), name="biasesHL2")
h2 = tf.nn.tanh((tf.matmul(h1, weightsHL2) + biasesHL2),
    name="hiddenLayer2")

# Third hidden layer
# Weights: 8x2
# Biases: 2
hiddenLayer3 = 2
weightsHL3 = tf.Variable(
    tf.random_uniform((hiddenLayer2, hiddenLayer3), -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
    name="weightsHL3")
biasesHL3 = tf.Variable(tf.zeros([hiddenLayer3]), name="biasesHL3")
h3 = tf.nn.tanh((tf.matmul(h2, weightsHL3) + biasesHL3),
    name="hiddenLayer3")
```

```
121
122 # Output layer
123 # Weights: 2x4
124 # Biases: 4
125 weightsOL = tf.Variable(
126     tf.random_uniform((hiddenLayer3, numActions), -1.0 /
        math.sqrt(numStates), 1.0 / math.sqrt(numStates)),
127     name="weightsOL")
128 biasesOL = tf.Variable(tf.zeros([numActions]), name="biasesOL")
129 YPredicted = tf.nn.tanh((tf.matmul(h3, weightsOL) + biasesOL),
    name="YPredicted")
130 maxPredictedY = tf.argmax(YPredicted, 1, name="maxPredictedY")
131
132 loss = tf.reduce_sum(tf.square(Y - YPredicted), name="loss")
133 trainer = tf.train.GradientDescentOptimizer(learning_rate=0.05,
    name="GDOptimizer").minimize(loss)
134
135 # Some parameters
136 gamma = 0.99
137
138 episodes = 0
139 numEpisodes = 2000
140
141 firstEpisodeReward1 = 0
142 reward1Reached = False
143
144 listOfStepsPerEpisode = []
145 accumulatedRewards = []
146
147 startTime = time.time()
148
149 ''' TRAINING '''
150 print(Fore.BLUE + "\nTraining:" + Style.RESET_ALL)
151 sess = tf.Session()
152 initVariables = tf.global_variables_initializer()
153 sess.run(initVariables)
154
155 # TensorBoard
```

```
156 # writer = tf.summary.FileWriter('./4FeatureExtraction21_graphs',
        sess.graph)
157
158 while episodes < numEpisodes:
159     state = initialState
160
161     stepsPerEpisode = 0
162     accumulatedReward = 0
163
164     if episodes % 250 == 0:
165         print(Fore.BLUE + "Episode:" + Style.RESET_ALL, episodes)
166
167     while stepsPerEpisode < 99:
168         # Choose an action by greedily (with e chance of random
              action) from the Q-network
169         a, qValues = sess.run([maxPredictedY, YPredicted],
              feed_dict={X: np.identity(numStates)[state:state + 1]})
170
171         action = [(R[state][a], a)]
172         if np.random.rand(1, 1) < e:
173             random_action = np.random.randint(0, numActions)
174             action = [(R[state][random_action], random_action)]
175         # print("Action:", action)
176
177         # Next state
178         nextState = states.index(action[0][0])
179
180         # Reward
181         reward = 0
182         if nextState == goalState:
183             reward = 1
184         if action[0][0] in listOfHoles:
185             reward = -0.25 * len(listOfHoles)
186         accumulatedReward += reward
187
188         # Obtain the next Q values by feeding the next state through
              our network
189         nextQValues = sess.run(YPredicted, feed_dict={X:
              np.identity(numStates)[nextState:nextState + 1]})
```

```
190         # Obtain the max of the next Q values and set our target value
                for chosen action
191         maxNextQValues = np.max(nextQValues)
192         targetQ = qValues
193 
194         # Q-learning formula that directly sets the "new Q value"
195         targetQ[0, action[0][1]] = reward + gamma * maxNextQValues
196 
197         # Train our network using target and predicted Q values
198         sess.run(trainer, feed_dict={X:
                np.identity(numStates)[state:state + 1], Y: targetQ})
199 
200         state = nextState
201         stepsPerEpisode += 1
202 
203         # If the next state is the goal state, finish this episode
                (goal reached)
204         if nextState == goalState:
205             break
206 
207     if accumulatedReward == 1.0 and not reward1Reached:
208         firstEpisodeReward1 = episodes
209         reward1Reached = True
210 
211     episodes += 1
212     listOfStepsPerEpisode.append(stepsPerEpisode)
213     accumulatedRewards.append(accumulatedReward)
214 
215 # writer.close()
216 
217 elapsedTime = time.time() - startTime
218 
219 ''' TESTING '''
220 # Testing with Q matrix
221 print(Fore.BLUE + "\nTesting with Q Matrix" + Style.RESET_ALL)
222 a = sess.run(tf.matmul(weightsHL1, weightsHL2)) # 16x8
223 b = sess.run(tf.matmul(a, weightsHL3)) # 16x2
224 Q = sess.run(tf.matmul(b, weightsOL)) # 16x4
```

```
225 print(Fore.BLUE + "Final Q matrix (weights of the NN):\n" +
        Style.RESET_ALL, Q)
226
227 paths = []
228 for i in range(numStates):
229     state = i
230     path = []
231
232     shouldBreak = False
233     steps = 0
234
235     while state != goalState and not shouldBreak:
236         possibleActions = []
237         for j in range(numActions):
238             possibleActions.append((Q[state][j], j))
239         # print("Possible actions:", possibleActions)
240
241         action = [max(possibleActions)]
242         # print("Action:", action)
243
244         stateChar = R[state][action[0][1]]
245         # print("State:", stateChar)
246         state = ord(stateChar) - 65
247
248         path.append(stateChar)
249
250         steps += 1
251         if steps > numStates:
252             shouldBreak = True
253     paths.append(path)
254
255 print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
256 for i in range(len(paths)):
257     print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
          + 65), ":", paths[i])
258
259 # Testing with activations of neurons of the output layer
260 print(Fore.BLUE + "\nTesting with activations of neurons of the
        output layer" + Style.RESET_ALL)
```

```
261 print(Fore.BLUE + "Activations of neurons of the output layer:" +
        Style.RESET_ALL)
262 for i in range(numStates):
263     print("From", chr(i + 65), ":", sess.run(YPredicted,
            feed_dict={X: np.identity(numStates)[i:i + 1]}))
264
265 activationsOL = []
266 for i in range(numStates):
267     activations = sess.run(YPredicted, feed_dict={X:
            np.identity(numStates)[i:i + 1]})
268     activationsOL.append(activations.tolist())
269 activationsOL = [activationsOL[x][0] for x in range(numStates)]
270
271 fromStateToNextState = []
272 for i in range(len(activationsOL)):
273     best = max(activationsOL[i])
274     fromStateToNextState.append([chr(i + 65),
            R[i][activationsOL[i].index(best)]])
275
276 paths = []
277 for i in range(numStates):
278     shouldBreak = False
279     steps = 0
280
281     nextState = fromStateToNextState[i][1]
282
283     path = []
284     path.append(nextState)
285
286     while nextState != chr(goalState + 65) and not shouldBreak:
287         for stateAndNextState in fromStateToNextState:
288             if stateAndNextState[0] == nextState and
                    stateAndNextState[0] != chr(goalState + 65):
289                 nextState = stateAndNextState[1]
290                 path.append(nextState)
291
292                 steps += 1
293                 if steps > numStates:
294                     shouldBreak = True
```

```
295
296     paths.append(path)
297
298 print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
299 for i in range(len(paths)):
300     print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
            + 65), ":", paths[i])
301
302 # Last hidden layer theory
303 # Activations of neurons of the last hidden layer
304 print(Fore.BLUE + "\nActivations of neurons of the last hidden
        layer:" + Style.RESET_ALL)
305 for i in range(numStates):
306     print("From", chr(i + 65), ":", sess.run(h3, feed_dict={X:
            np.identity(numStates)[i:i + 1]}))
307
308 activationsLHL = []
309 for i in range(numStates):
310     activations = sess.run(h3, feed_dict={X:
            np.identity(numStates)[i:i + 1]})
311     activationsLHL.append(activations.tolist())
312 activationsLHL = [activationsLHL[x][0] for x in range(numStates)]
313
314 # Activations of neurons of the third hidden layer -> 0 or 1
315 # threshold = float(input("Set the threshold: "))
316 threshold = 0.0
317 for i in range(numStates):
318     for j in range(hiddenLayer3):
319         if activationsLHL[i][j] <= threshold:
320             activationsLHL[i][j] = 0
321         else:
322             activationsLHL[i][j] = 1
323 print(activationsLHL)
324
325 print(Fore.BLUE + "\nUnique activations of neurons of the last hidden
        layer:" + Style.RESET_ALL)
326 uniqueActivations = []
327 for i in range(numStates):
328     if activationsLHL[i] not in uniqueActivations:
```

```
329         uniqueActivations.append(activationsLHL[i])
330 print(uniqueActivations)
331
332 allStatesAndIndex = []
333 for i in range(len(uniqueActivations)):
334     stateAndIndex = []
335     for j in range(numStates):
336         if uniqueActivations[i] == activationsLHL[j]:
337             stateAndIndex.append([chr(j + 65), j])
338     allStatesAndIndex.append(stateAndIndex)
339 # print(allStatesAndIndex)
340
341 listOfActions = [0, 1, 2, 3]
342 possibleActions = [list(x) for x in
        itertools.permutations(listOfActions, len(listOfActions))]
343
344
345 def translateAction(x):
346     return {
347         0: 'Up',
348         1: 'Down',
349         2: 'Right',
350         3: 'Left',
351     }.get(x, 'None')
352
353
354 combinationFound = False
355 if len(allStatesAndIndex) <= numActions:
356     for i in range(len(possibleActions)):
357         fromStateToNextState = []
358
359         for j in range(len(allStatesAndIndex)):
360             for a, b in allStatesAndIndex[j]:
361                 fromStateToNextState.append([a,
                        R[b][possibleActions[i][j]]])
362
363         paths = []
364         for k in range(numStates):
365             shouldBreak = False
```

```
366             steps = 0
367
368             state = fromStateToNextState[k][0]
369             nextState = fromStateToNextState[k][1]
370
371             path = []
372             path.append(state)
373             path.append(nextState)
374
375             while nextState != goalState and not shouldBreak:
376                 for stateAndNextState in fromStateToNextState:
377                     if stateAndNextState[0] == nextState:
378                         nextState = stateAndNextState[1]
379                         path.append(nextState)
380
381                         steps += 1
382                         if steps > numStates:
383                             shouldBreak = True
384
385             paths.append(path)
386
387         lastStates = [isGoalState[-1] for isGoalState in paths]
388         if all(isGoalState == chr(goalState + 65) for isGoalState in
                lastStates):
389             print(Fore.GREEN + "\nTheory demonstrated with this
                    combination of actions:")
390             for l in range(len(listOfActions)):
391                 print(Fore.GREEN + "Group of states", l, "taking this
                        action:", translateAction(possibleActions[i][l]))
392             print(Style.RESET_ALL)
393             combinationFound = True
394             break
395 if not combinationFound:
396     print(Fore.RED + "\nTheory NOT demonstrated with ANY combination
            of actions.", Style.RESET_ALL)
397
398 ''' STATS '''
399 print(Fore.BLUE + "\nStats:" + Style.RESET_ALL)
400
```

```
401 minutes, seconds = divmod(elapsedTime, 60)
402 print(Fore.BLUE + "Training time:" + Style.RESET_ALL, minutes,
        "minutes,", seconds, "seconds")
403
404 print(Fore.BLUE + "Accumulated reward over time:" + Style.RESET_ALL,
        sum(accumulatedRewards) / numEpisodes, "%")
405
406 print(Fore.BLUE + "First episode with reward 1.0:" + Style.RESET_ALL,
        firstEpisodeReward1)
407
408 plt.suptitle('Rewards per episode', fontsize=14, fontweight='bold')
409 plt.plot(accumulatedRewards)
410 plt.xlabel('Episodes')
411 plt.ylabel('Rewards')
412 plt.show()
413
414 plt.suptitle('Steps per episode', fontsize=14, fontweight='bold')
415 plt.plot(listOfStepsPerEpisode)
416 plt.xlabel('Episodes')
417 plt.ylabel('Steps')
418 plt.show()
419
420 # REDUCED DEEP NEURAL NETWORK (using as inputs the activations of the
        first hidden layer)
421 # Activations of neurons of the first hidden layer
422 print(Fore.BLUE + "\nActivations of neurons of the first hidden
        layer:" + Style.RESET_ALL)
423 for i in range(numStates):
424     print("From", chr(i + 65), ":", sess.run(h1, feed_dict={X:
            np.identity(numStates)[i:i + 1]}))
425
426 activationsHL1 = []
427 for i in range(numStates):
428     activations = sess.run(h1, feed_dict={X:
            np.identity(numStates)[i:i + 1]})
429     activationsHL1.append(activations.tolist())
430 activationsHL1 = [activationsHL1[x][0] for x in range(numStates)]
431
432 # Activations of neurons of the first hidden layer -> 0 or 1
```

```
433 for i in range(numStates):
434     for j in range(hiddenLayer1):
435         if activationsHL1[i][j] <= 0.0:
436             activationsHL1[i][j] = 0
437         else:
438             activationsHL1[i][j] = 1
439 print(activationsHL1)
440
441 newNumStates = len(activationsHL1[0])
442 # goalState = ord('N') - 65
443
444 # Activations of neurons of the second hidden layer
445 print("\nActivations of neurons of the second hidden layer:")
446 for i in range(numStates):
447     print("From", chr(i + 65), ":", sess.run(h2, feed_dict={X:
        np.identity(numStates)[i:i + 1]}))
448
449 activationsHL2 = []
450 for i in range(numStates):
451     activations = sess.run(h2, feed_dict={X:
        np.identity(numStates)[i:i + 1]})
452     activationsHL2.append(activations.tolist())
453 activationsHL2 = [activationsHL2[x][0] for x in range(0, 16)]
454
455 # Activation of neurons of the second hidden layer -> 0 or 1
456 for i in range(numStates):
457     for j in range(hiddenLayer2):
458         if activationsHL2[i][j] <= 0.0:
459             activationsHL2[i][j] = 0
460         else:
461             activationsHL2[i][j] = 1
462
463 # Implementing the network itself
464 tf.reset_default_graph()
465
466 # Input: 1x12
467 X2 = tf.placeholder(shape=[1, newNumStates], dtype=tf.float32,
    name="X2")
468 # Output: 1x4
```

```
469 Y2 = tf.placeholder(shape=[1, numActions], dtype=tf.float32,
        name="Y2")
470
471 # First hidden layer
472 # Weights: 12x8
473 # Biases: 8
474 hiddenLayer12 = 8
475 weightsHL12 = tf.Variable(
476     tf.random_uniform((newNumStates, hiddenLayer12), -1.0 /
            math.sqrt(newNumStates), 1.0 / math.sqrt(newNumStates)),
477     name="weightsHL12")
478 biasesHL12 = tf.Variable(tf.zeros([hiddenLayer12]), name="biasesHL12")
479 h12 = tf.nn.tanh((tf.matmul(X2, weightsHL12) + biasesHL12),
        name="hiddenLayer12")
480
481 # Second hidden layer
482 # Weights: 8x2
483 # Biases: 2
484 hiddenLayer22 = 2
485 weightsHL22 = tf.Variable(
486     tf.random_uniform((hiddenLayer12, hiddenLayer22), -1.0 /
            math.sqrt(newNumStates), 1.0 / math.sqrt(newNumStates)),
487     name="weightsHL22")
488 biasesHL22 = tf.Variable(tf.zeros([hiddenLayer22]), name="biasesHL22")
489 h22 = tf.nn.tanh((tf.matmul(h12, weightsHL22) + biasesHL22),
        name="hiddenLayer22")
490
491 # Output layer
492 # Weights: 2x4
493 # Biases: 4
494 weightsOL2 = tf.Variable(
495     tf.random_uniform((hiddenLayer22, numActions), -1.0 /
            math.sqrt(newNumStates), 1.0 / math.sqrt(newNumStates)),
496     name="weightsOL")
497 biasesOL2 = tf.Variable(tf.zeros([numActions]), name="biasesOL2")
498 YPredicted2 = tf.nn.tanh((tf.matmul(h22, weightsOL2) + biasesOL2),
        name="YPredicted2")
499 maxPredictedY2 = tf.argmax(YPredicted2, 1, name="maxPredictedY2")
500
```

```
501 loss2 = tf.reduce_sum(tf.square(Y2 - YPredicted2), name="loss2")
502 trainer2 = tf.train.GradientDescentOptimizer(learning_rate=0.05,
        name="GDOptimizer2").minimize(loss2)
503
504 # Some parameters
505 gamma = 0.99
506
507 episodes = 0
508
509 firstEpisodeReward1 = 0
510 reward1Reached = False
511
512 listOfStepsPerEpisode2 = []
513 accumulatedRewards2 = []
514
515 startTime2 = time.time()
516
517 ''' TRAINING '''
518 print(Fore.BLUE + "\nTraining:" + Style.RESET_ALL)
519 sess2 = tf.Session()
520 initVariables2 = tf.global_variables_initializer()
521 sess2.run(initVariables2)
522
523 # TensorBoard
524 # writer2 = tf.summary.FileWriter('./4FeatureExtraction22_graphs',
        sess2.graph)
525
526 while episodes < numEpisodes:
527     state = initialState
528
529     stepsPerEpisode = 0
530     accumulatedReward = 0
531
532     if episodes % 250 == 0:
533         print(Fore.BLUE + "Episode:" + Style.RESET_ALL, episodes)
534
535     while stepsPerEpisode < 99:
536         # Choose an action by greedily (with e chance of random
              action) from the Q-network
```

```
        a, qValues = sess2.run([maxPredictedY2, YPredicted2],
            feed_dict={X2: activationsHL1[state:state + 1]})

        action = [(R[state][a], a)]
        if np.random.rand(1, 1) < e:
            random_action = np.random.randint(0, 4)
            action = [(R[state][random_action], random_action)]
        # print("Action:", action)

        # Next state
        nextState = states.index(action[0][0])

        # Reward
        reward = 0
        if nextState == goalState:
            reward = 1
        if action[0][0] in listOfHoles:
            reward = -0.25 * len(listOfHoles)
        accumulatedReward += reward

        # Obtain the next Q values by feeding the next state through
            our network
        nextQValues = sess2.run(YPredicted2, feed_dict={X2:
            activationsHL1[nextState:nextState + 1]})
        # Obtain the max of the next Q values and set our target value
            for chosen action
        maxNextQValues = np.max(nextQValues)
        targetQ = qValues

        # Q-learning formula that directly sets the "new Q value"
        targetQ[0, action[0][1]] = reward + gamma * maxNextQValues

        # Train our network using target and predicted Q values
        sess2.run(trainer2, feed_dict={X2: activationsHL1[state:state
            + 1], Y2: targetQ})

        state = nextState
        stepsPerEpisode += 1
```

```
571         # If the next state is the goal state, finish this episode
                (goal reached)
572         if nextState == goalState:
573             break
574 
575     if accumulatedReward == 1.0 and not reward1Reached:
576         firstEpisodeReward1 = episodes
577         reward1Reached = True
578 
579     episodes += 1
580     listOfStepsPerEpisode2.append(stepsPerEpisode)
581     accumulatedRewards2.append(accumulatedReward)
582 
583 # writer2.close()
584 
585 elapsedTime2 = time.time() - startTime2
586 
587 ''' TESTING '''
588 # Testing with activations of neurons of the output layer
589 print(Fore.BLUE + "\nTesting with activations of neurons of the
        output layer" + Style.RESET_ALL)
590 print(Fore.BLUE + "Activations of neurons of the output layer:" +
        Style.RESET_ALL)
591 for i in range(numStates):
592     print("From", chr(i + 65), ":", sess2.run(YPredicted2,
            feed_dict={X2: activationsHL1[i:i + 1]}))
593 
594 activationsOL2 = []
595 for i in range(numStates):
596     activations = sess2.run(YPredicted2, feed_dict={X2:
            activationsHL1[i:i + 1]})
597     activationsOL2.append(activations.tolist())
598 activationsOL2 = [activationsOL2[x][0] for x in range(numStates)]
599 
600 fromStateToNextState = []
601 for i in range(len(activationsOL2)):
602     best = max(activationsOL2[i])
603     fromStateToNextState.append([chr(i + 65),
            R[i][activationsOL2[i].index(best)]])
```

```
604
605 paths = []
606 for j in range(numStates):
607     shouldBreak = False
608     steps = 0
609
610     state = fromStateToNextState[j][0]
611     nextState = fromStateToNextState[j][1]
612
613     path = []
614     path.append(nextState)
615
616     while nextState != chr(goalState + 65) and not shouldBreak:
617         for stateAndNextState in fromStateToNextState:
618             if stateAndNextState[0] == nextState and
                    stateAndNextState[0] != chr(goalState + 65):
619                 nextState = stateAndNextState[1]
620                 path.append(nextState)
621
622                 steps += 1
623                 if steps > numStates:
624                     shouldBreak = True
625
626     paths.append(path)
627
628 print(Fore.BLUE + "\nOptimum paths" + Style.RESET_ALL)
629 for i in range(len(paths)):
630     print(Fore.GREEN + "Optimum path from:" + Style.RESET_ALL, chr(i
            + 65), ":", paths[i])
631
632 # Last hidden layer theory
633 # Activations of neurons of the last hidden layer
634 print(Fore.BLUE + "\nActivations of neurons of the last hidden
        layer:" + Style.RESET_ALL)
635 for i in range(numStates):
636     print("From", chr(i + 65), ":", sess2.run(h22, feed_dict={X2:
            activationsHL1[i:i + 1]}))
637
638 activationsLHL = []
```

```
for i in range(numStates):
    activations = sess2.run(h22, feed_dict={X2: activationsHL1[i:i +
        1]})
    activationsLHL.append(activations.tolist())
activationsLHL = [activationsLHL[x][0] for x in range(0, 16)]

# Activation of neurons of the last hidden layer -> 0 or 1
# threshold = float(input("Set the threshold: "))
threshold = 0.0
for i in range(numStates):
    for j in range(hiddenLayer22):
        if activationsLHL[i][j] <= threshold:
            activationsLHL[i][j] = 0
        else:
            activationsLHL[i][j] = 1

print(Fore.BLUE + "\nUnique activations of neurons of the last hidden
    layer:" + Style.RESET_ALL)
uniqueActivations = []
for i in range(numStates):
    if activationsLHL[i] not in uniqueActivations:
        uniqueActivations.append(activationsLHL[i])
print(uniqueActivations)

allStatesAndIndex = []
for i in range(len(uniqueActivations)):
    stateAndIndex = []
    for j in range(numStates):
        if uniqueActivations[i] == activationsLHL[j]:
            stateAndIndex.append([chr(j + 65), j])
    allStatesAndIndex.append(stateAndIndex)
# print(allStatesAndIndex)

listOfActions = [0, 1, 2, 3]
possibleActions = [list(x) for x in
    itertools.permutations(listOfActions, len(listOfActions))]


def translateAction(x):
```

```
    return {
        0: 'Up',
        1: 'Down',
        2: 'Right',
        3: 'Left',
    }.get(x, 'None')


combinationFound = False
if len(allStatesAndIndex) <= numActions:
    for i in range(len(possibleActions)):
        fromStateToNextState = []

        for j in range(len(allStatesAndIndex)):
            for a, b in allStatesAndIndex[j]:
                fromStateToNextState.append([a,
                    R[b][possibleActions[i][j]]])

        paths = []
        for k in range(numStates):
            shouldBreak = False
            steps = 0

            state = fromStateToNextState[k][0]
            nextState = fromStateToNextState[k][1]

            path = []
            path.append(state)
            path.append(nextState)

            while nextState != goalState and not shouldBreak:
                for stateAndNextState in fromStateToNextState:
                    if stateAndNextState[0] == nextState:
                        nextState = stateAndNextState[1]
                        path.append(nextState)

                        steps += 1
                        if steps > numStates:
                            shouldBreak = True
```

```
713
714             paths.append(path)
715
716         lastStates = [isGoalState[-1] for isGoalState in paths]
717         if all(isGoalState == chr(goalState + 65) for isGoalState in
                lastStates):
718             print(Fore.GREEN + "\nTheory demonstrated with this
                    combination of actions:")
719             for l in range(len(listOfActions)):
720                 print(Fore.GREEN + "Group of states", l, "taking this
                        action:", translateAction(possibleActions[i][l]))
721             print(Style.RESET_ALL)
722             combinationFound = True
723             break
724 if not combinationFound:
725     print(Fore.RED + "\nTheory NOT demonstrated with ANY combination
            of actions.", Style.RESET_ALL)
726
727 ''' STATS '''
728 print(Fore.BLUE + "\nStats:" + Style.RESET_ALL)
729
730 minutes2, seconds2 = divmod(elapsedTime2, 60)
731 print(Fore.BLUE + "Training time:" + Style.RESET_ALL, minutes2,
        "minutes and", seconds2, "seconds")
732
733 print(Fore.BLUE + "Accumulated reward over time:" + Style.RESET_ALL,
        sum(accumulatedRewards2) / numEpisodes, "%")
734
735 print(Fore.BLUE + "First episode with reward 1.0:" + Style.RESET_ALL,
        firstEpisodeReward1)
736
737 plt.suptitle('Rewards per episode', fontsize=14, fontweight='bold')
738 plt.plot(accumulatedRewards2)
739 plt.xlabel('Episodes')
740 plt.ylabel('Rewards')
741 plt.show()
742
743 plt.suptitle('Steps per episode', fontsize=14, fontweight='bold')
744 plt.plot(listOfStepsPerEpisode2)
```

```
plt.xlabel('Episodes')
plt.ylabel('Steps')
plt.show()
```
