Quarto as a Reinforcement Learning problem
Frode Stig Nørgaard Pedersen, 201405114

Master's Thesis, Computer Science
June 2019
Advisor: Christian Nørgaard Storm Pedersen
Abstract

Reinforcement Learning has proven itself through a recent history of superhuman level performances in various tasks. Board games are of particular interest because they can be simulated completely, as the rules are pre-determined. Reinforcement Learning is a machine learning method, wherein the problem space is explored such that actions leading toward beneficial situations reward an Agent and situations that lead to unfavourable positions are punished. The aim of this thesis is to implement different learning methods and compare how well they play the game of Quarto. Training is done through self-play, where neural networks approximate the value-functions that are used to estimate how good a given position or action is. The results of my experiments indicate that differences in hyper-parameter settings lead to different performances. While the choice of learning algorithm did not change much in the way of overall performances, some differences in behaviour throughout training could be measured.
Acknowledgments

I would like to thank my supervisor, Christian S. N. Pedersen, for his immense patience with me and for his continuous support.

I want to thank Nicolas N. Guerrero for spending time discussing fundamental ideas regarding Reinforcement Learning.

For his expertise and time, I would like to thank Frederik Hvilshøj, who helped me understand and design various neural network architectures.

For their willingness to spend time reading my thesis and catch errors, I would like to thank Nikolaj C. S. Knudsen, Mathias Haudgaard and Malene S. Sønnichsen.

I would like to thank my friends and family for their patience and interest in my project. To my son Patrik A. Pedersen, and daughter Kamilla B. Pedersen, for reminding me that life goes on, even when everything seems to fail.

Finally, I want to thank Malene S. Sønnichsen for her loving support. Her efforts in the month leading up to the deadline have been of paramount importance in all things practical and emotional.

Frode Stig Nørgaard Pedersen,
Aarhus, Friday 14th June, 2019.
## Contents

1 Introduction ................................................. 3
   1.1 Quarto ................................................. 4
   1.2 Reinforcement Learning in Board Games ................. 5
   1.3 Related Work ........................................... 6
       1.3.1 Examples of Reinforcement Learning in Real world settings 6
       1.3.2 Examples of Reinforcement Learning in Games .............. 7
       1.3.3 Quarto using Search Trees ........................... 8
   1.4 Outline of thesis ....................................... 8

2 Reinforcement Learning .................................... 11
   2.1 Agent, Environment and Reward signals ................. 11
   2.2 Markov Decision Processes .............................. 13
   2.3 Value functions ........................................ 15
   2.4 Prediction and Control ................................. 21
       2.4.1 Monte Carlo Learning ............................... 21
       2.4.2 Temporal Difference Learning ..................... 24
       2.4.3 TD(λ) ............................................. 27
   2.5 Eligibility traces ....................................... 30
   2.6 Exploration ............................................. 32
   2.7 Function approximation ................................. 34
       2.7.1 Training of neural networks in RL .................. 35
       2.7.2 Feature Representation ............................... 37
       2.7.3 Stability issues .................................... 37
   2.8 Summary of Part 2 ..................................... 37

3 Reinforcement Learning in Practice ....................... 39
   3.1 Quarto as a Reinforcement Learning problem ............ 39
   3.2 Reduction of State-Space ............................... 42
   3.3 Learning Algorithms .................................... 45
   3.4 Function Approximation in a practical sense .......... 47
   3.5 Exploration in Board games ............................. 49
   3.6 Training of a Board game playing Agent ............... 49
   3.7 Summary of Part 3 ..................................... 50
## Experiments

### 4.1 Experimentational setup

### 4.2 Experiments

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Finding Variant Representatives</td>
<td>53</td>
</tr>
<tr>
<td>2: Performances of Gamma parameter: $\gamma$</td>
<td>55</td>
</tr>
<tr>
<td>3: Performances of Lambda parameter: $\lambda$</td>
<td>56</td>
</tr>
<tr>
<td>4: Effect of Agent History</td>
<td>57</td>
</tr>
<tr>
<td>5: Effects of Fix-Q-Networks, Regularisation of Networks, and Input normalisation</td>
<td>58</td>
</tr>
<tr>
<td>6: Best Overall Agent</td>
<td>58</td>
</tr>
<tr>
<td>7: Out of Sample Testing</td>
<td>60</td>
</tr>
</tbody>
</table>

### 4.3 Summary of Part 4

## Conclusion

### 5.1 Future Work

## Bibliography
Chapter 1

Introduction

In this part I shall give a brief overview of the role that Reinforcement Learning has within the machine learning field, how it differentiates itself from other methods, such as supervised learning, and introduce the problem, Quarto, that shall be used to test different learning methods of Reinforcement Learning.

Reinforcement Learning typically attempts to have a machine learn from its own experience, instead of relying on expert knowledge within the domain of the problem, as is the case with Supervised Learning. For Supervised Learning, a model is told what its goals are in the form of labels, such that for given data points it can approximate the output to that of the labels. Now, if Supervised Learning can be used to approximate ground truths, one might ask, why do we need Reinforcement Learning? Unfortunately, what empowers Supervised Learning schemes are also what limits them. The main problem is that data has to be supplied, and as such, an expert has to provide the labelling. For cases where data is sparse, or simply too expensive to produce, Supervised Learning generally cannot create a satisfying model. Another issue is that problems solved using Supervised Learning are inherently limited to only ever become as good as the data and labels that are available, meaning, if we hope to create an Agent that can perform at superhuman levels, it needs to be fed data that are of superhuman character. Reinforcement Learning, on the other hand, will search the problem-space, aiming to create a sense of what constitutes a good situation and use this sense to perform actions, navigating through the specific problem that it attempts to find a solution to. Reinforcement Learning is based on iterative methods, where value-functions are updated to represent this sense of quality.

Another approach to learning through experience, is Evolutionary Learning, which is based on the idea of natural selection. While both Reinforcement Learning and Evolutionary Learning methods find solutions to problems through trial-and-error, they arrive at solutions in different ways. Reinforcement Learning focuses on one Agent and, through iterations of function evaluation, estimates how good it is to be in a given situation compared to that of the ground truth. An Agent trained using an Evolutionary Learning approach will instead be the fittest Agent among a series of reproductions, where the offspring will be variations of their parents, with mutation added to increase exploration
Even though they are different, there is no clear divide between the two, and it is possible to combine them, as done by K. Pathak [2018].

Finally, Reinforcement Learning distinguishes itself from Unsupervised Learning, as the latter seeks to discover patterns or give meaning to data through clustering [Yaser S. Abu-Mostafa and Lin, 2012].

This introduction to Reinforcement Learning might give the reader the idea that it is limitless, and in general could find solutions to most problems, simply through exploring the problem space. Why bother using Supervised Learning if Reinforcement Learning could achieve similar goals without the dependency on pre-labelled data. The issues with Reinforcement Learning relates to the size of the problem space, as for larger problems, there is inherently more actions to be explored. Unfortunately, most interesting problems are also too large to be explored effectively, so the task becomes that of approximating a model towards an optimal policy through samples, which of course, cannot hope to achieve optimality in any practical setting. This also leads to issues where the Agent spends a long time exploring many bad actions, as it is unaware of what the good ones are, and with the potential hazard that it can become stuck on a low local maxima. Another problem that is often associated with Reinforcement Learning is that there are a lot of hyper-parameters associated with the learning methods. Not only are there the classical ones relating to that of function-approximation models, such as neural networks, but each learning algorithm might also be seen as a hyper-parameter. Furthermore, each learning algorithm comes with its own set of hyper-parameters, resulting in a very big hyper-parameter field that needs to be searched in the hopes of finding the best performing Agent.

For the following sections, I will describe the problem in this thesis, illustrate how others have used Reinforcement Learning to find solutions to similar problems and lastly, define the general outline of the thesis.

### 1.1 Quarto

For this thesis I have used Quarto as my test bench for various Reinforcement Learning methods. Quarto is a board game consisting of a 4x4 cell grid board and 16 unique pieces played by two players. Each piece has 4 attributes, namely Height, Color, Shape and Indentation. To illustrate this, figure 1.1 shows the game.
Figure 1.1: The game of Quarto, with its 4x4 board after 10 moves.

An example of a piece could then be (TALL, BLACK, SQUARE, INDENTED). This specific piece can also be seen in the lower left corner of 1.1. The aim of the game is to place a piece such that there are 4 in a row of the same type of attribute; e.g. if there are 4 TALL pieces along a row, a column or the diagonals, the player that placed the fourth TALL piece wins the game, regardless of the other features. There is no color or other attribute assigned to the players, there is only a pool of shared pieces. The first player then selects a piece from the pool and hands it to the opponent, after which that player places it on the board. Following this, the opponent selects a new piece from the pool and gives that to the first player to place. In this manner the game continues until either a player places a piece that creates a win condition, or there is no more pieces to give. In the latter case, this would mean that each cell of the board has been occupied without a winning configuration being found, leading to a draw. Below is an enumeration of the game sequence.

1. The game is initialised by placing all the pieces in a pool next to the board.
2. Player 1 selects a piece, which is given to player 2.
3. Player 2 places that piece as they see fit, after which player 2 selects a piece to give to player 1.
4. Player 1 places that piece.
5. Repeat point 2-4 until a termination condition has been found, either by a win configuration or a draw configuration.

1.2 Reinforcement Learning in Board Games

Board games, or games in general, are great test benches for Reinforcement Learning algorithms, as it is possible to model them completely as illustrated
by [Ghory, 2004]. Through these models, simulations can be made such that training can be done on simulated experience instead of having an Agent explore real world situations, which would incur a significant increase in training speed.

Furthermore, a solution to a board game problem can also be found based on these models, through 'virtual steps', resembling that of what we would typically call 'reading ahead'. In this fashion, an Agent would, before actually performing a real action, simulate possible game sequences and through these select an action that seems the most beneficial to its objective. This is called modelled-learning, which is in contrast to that of model-free learning. The strengths of model-free learning is that it is more general in nature, and does not depend on perfect knowledge about the environment and underlying state-transitions. With that said, introducing a model to the learning objective often improves the performance of the Agent, through methods, such as state-space reductions as illustrated in section 3.2. Models can also be used, through simulations, to perform virtual steps, where a game-tree can be created using either Mini-max techniques such as the one used in [Jochen Mohrmann, 2013], or Monte Carlo Search Trees as illustrated in [Guillaume Chaslot and Spronck, 2008] and [Tome Vodopivec, 2017]. For this thesis, I shall use a model of the game to simulate the action space of each state as illustrated in figure 3.1 in section 3.1 but without the 'reading ahead' aspect that modelled learning supply. This was done to emphasise the learning methods over that of learning a model. The methods in this thesis can then later be expanded to include models without much restructuring.

1.3 Related Work

Before settling on the approaches I used for implementing Quarto as a Reinforcement Learning problem, a series of considerations were made, based on the following works done within the field.

1.3.1 Examples of Reinforcement Learning in Real world settings

Andrew Y. Ng and Sastry [2006] made the steering of a model helicopter possible using Monte Carlo learning as described in section 2.4.1 combined with a random number generator to increase exploration. They modelled the effects of various physical attributes such as velocity and world positions in a computer, so that learning could be done on a simulation instead of on an expensive model helicopter. As Helicopters work in a 3D Environment, its facing could be that of many directions. Symmetries were used to normalise the direction it was facing, so that it only had to take into account of what was forward, sideways and down. Finally, once training was complete, the Agent trained was set up to fly a real world model helicopter, with greater than human level control, as can be seen in their video.
1.3.2 Examples of Reinforcement Learning in Games

One of the first groundbreaking results is that of Tesauro’s Backgammon playing Agent, TD-Gammon, which managed to learn to play at an expert human level back in 1995. It did so utilising both hand-crafted features and a raw encoding of a state. As the game has a stochastic element, namely the throwing of dice, they did not find it necessary to include other exploration techniques. Unfortunately, although it is based on a TD(\(\lambda\)) learning approach, they do not specify what values of \(\lambda\) they found to perform the best. They do, however, note that research has been made into the effect of \(\lambda\) in board games [Nicol N. Schraudolph and Sejnowski, 2000].

Chess has always been one of the big games in western cultures used to measure the prowess between people, effectively working as a sort of gauge of intelligence. Of course, this is a very simplistic view on intelligence, but when IBM created Deep Blue, a program that could beat a reigning world champion, Gerry Kasparov in 1996, the world was shook as this represented the fact that computers were becoming smarter than people [Murray Campbella and hsiung Hsu, 2002]. Deep Blue was not trained using Reinforcement Learning methods but definitely ignited the curiosity within the field of AI with respect to board games. Deep Blue was not the first attempt at creating chess playing programs. Sebastian Thrun had some success in creating a Reinforcement Learning Agent using TD(0) and value-function approximation through a neural network in 1995 Thrun. A more recent success story is that of Deep Mind’s Alpha Zero program, originally designed to play the game of Go but applied to the problem of Chess. As they showed in [David Silver and Hassabis, 2018] their algorithms were of a more general nature and could be applied to multiple problems, performing at superhuman levels of game play.

Finally, as this was one of the breakthroughs motivating my interest in Reinforcement Learning, I will include work done on the game of Go. It is particularly interesting, as it has a long history of programs that attempted to play the game to a degree that could rival that of professional players but without much luck. Schraudolph et al created an Agent in 1994 based on TD-learning with a neural network, trained using expert moves [Nicol N. Schraudolph and Sejnowski, 2000]. They found that the problem of self-play was helped by the use of expert moves, so that the Agent did not have to go through a long training period of terrible moves. They also utilised other programs to perform the moves that were not played by the Agent, and in this way circumventing the issues of self-play (as illustrated in section 3.6). It accomplished a decently strong level of play. Building upon this success, Deep Mind created an Agent using Monte Carlo Tree Search to perform virtual simulations based on Reinforcement Learning techniques to approximate the heuristic used in these trees. This agent was AlphaGo [David Silver, 2016], with convolutional networks trained to approximate the action- and state-value functions used when calculating the heuristics. They were trained to imitate that of human experts first, to alleviate the bootstrapping issues mentioned by Schraudolph et al, after which they applied the Reinforcement Learning aspects. AlphaGo managed to beat, arguably, the strongest go player alive, Lee Sedol, in 2016, with a score
of 4-1, a feat thought to be decades away. AlphaGo was also the strongest
Go-playing program of its time. Later, a version called AlphaGo Zero was
made by Deep Mind, wherein they removed the dependency on expert moves
and trained their heuristics purely with Reinforcement Learning [David Silver,
2017]. It beat the previous versions of AlphaGo after just 36 hours of training
compared to that of several months which was used by AlphaGo to beat Lee
Sedol. For both AlphaGo and AlphaGo Zero, they use an exploration technique
called softmax exploration, which is a variant of $\epsilon$-greedy exploration mentioned
in 2.6, where the actions are chosen in a weighted manner with respect to how
good the Agent believe they are.

1.3.3 Quarto using Search Trees
In [Jochen Mohrmann, 2013], they created an Agent based solely on modelled
training. To do this, they created a game tree at each step of the game by
performing Mini-Max search combined with $\alpha$- and $\beta$-pruning and a heuristic
function based on the configurations of the board state, which was used when
depths were too large. This created an Agent that did not utilise standard
Reinforcement Learning techniques, such as MC or TD learning but illustrates
the effectiveness of a modelled environment, making look-aheads through sim-
ulations possible. This Agent is particularly interesting, as this represents the
other end of the spectrum with regards to modelled training than the Agents I
have created for this thesis, which relies solely on MC and TD-learning methods.
As illustrated in [David Silver, 2017], combining the two, using both a search
tree and using a value-function approximation to learn the heuristic function
used in the search tree, seems to create the best Agents.

1.4 Outline of thesis
Following is an outline of what the rest of this thesis will contain, and an
overview of my contribution.

Part 2 is a theoretical section, wherein I attempt to go through the un-
derlying theory, strengths and weaknesses of different learning methods. It is
based on explanations found in the literature and as such is not part of my
contribution. Theory included in this part of the thesis is extracted such that
only information that I have deemed relevant to my solutions to Quarto has
been showcased. It is by no means an exhaustive description of all learning
methods available within the Reinforcement Learning domain.

Part 3 is my solution to Quarto, and as such my primary contribution,
illustrating how a board game such as Quarto can be made into a Reinforce-
ment Learning problem. I go through various considerations with regards to
modelling the game, algorithmic choices and practical issues related to using
neural networks as value-function approximators. During Part 3, Part 2 is used
to argue why I selected the methods involved in my solutions.

Part 4 is an experimental analysis of the Agents created. The first part
is variant testing, showing that different choices of hyper-parameters result in
different outcomes of Agents. The Agents are being tested in a greedy manner
through a tournament setting such that the Agent with the highest win-ratio is deemed the best performing one. This is done, as there is no concrete way of testing performance levels of the Agents. The last part are tests that does not test Agents between variants but are Agents tested against outside players such as Random Agents, Human players and Other computer programs.

Finally, **Part 5** is my conclusion on the thesis and my reflections on what could have been done to find Agents that performed better, as the resulting Agents unfortunately did not grow very strong.
Chapter 2

Reinforcement Learning

In this part of the thesis I shall go into details with Reinforcement Learning and what it entails, what the objectives are for the learning algorithms and to give an in-depth view of how various methods can be used to find a solution to problems that can be modelled as a Markov Decision Process. I will illustrate how the classical learning methods resemble each other and how they differ, while discussing the strengths of each of these. The following chapter will emphasise the practical aspects more so than illustrating proofs, as including these would take up too much space. References to proofs have been made where I found necessary. Finally, the chapter ends in an extension to the basic methods, where function-approximation is introduced, to illustrate how large problems may be handled in a Reinforcement Learning sense.

2.1 Agent, Environment and Reward signals

For Reinforcement Learning problems, the key components that are at play is the Agent and the Environment. Before we can get into details with the various methods of which Reinforcement Learning problems is solved or approximated, the Agent, Environment and the Reward signal have to be defined.

The Agent is the learning component. It is the model that we wish to train to perform actions within a world such that it optimises the amount of reward that it can achieve. This world is the Environment and it is important to make distinction between what is part of the Agent, and what is part of the Environment, as the Agent should not have any control over the Environment, i.e. it should not be able to dictate how the Environment changes with respect to the actions chosen. It should be able to affect it by performing actions upon which the Environment changes in some fashion and returns the so called reward signal. In other words, the Agent performs actions within this world and observes whether said actions were beneficial or disadvantageous to it after which it should learn that given same or similar situations it will achieve similar rewards. These interactions between the Agent and the Environment are done at discrete time steps denoted $t \in \mathbb{N}$, and for many problems these are finite, giving a sequence of interactions happening at time $t_0, t_1, ... T$. Now, for each of these time steps, the Agent is presented a representation of the Environment
also called a State, which is denoted $s_t$. These states may for many problems be a complete representation of the Environment as is the case when the Environment is fully observable by the Agent, e.g. for games such as Chess, Go or Backgammon, or incomplete representations where only some aspects are known, e.g. in a game of Blackjack the cards that the dealer has is not observable by the Agent, but they are still a part of the Environment that the Agent interacts with. A single state is then a representation of the Environment at a given time. For the full problem, a series of states will be observed, each being only one of many states that could have occurred, depending on what the action chosen by the Agent was. Therefore we say $s_t \in S$ where $S$ is the set of all possible representations of the Environment throughout the time period in which the Agent lives. For each of these states, a set of actions are possible for the Agent to perform, denoted $a_t \in A(s_t)$, with $a_t$ being a specific action and $A(s_t)$ being the set of all actions available at state $s_t$. Each of these are ways of which the Agent can affect some aspect of the Environment, resulting in a new situation that the Agent has to deal with. Lastly, as mentioned, a scalar reward signal $r_t \in \mathbb{R}$, with $\mathbb{R}$ being the set of all possible rewards in a given problem, is associated with each interaction between the Agent and the Environment, without which no learning could be achieved.

With the states, actions and rewards defined we are able to see how problems within the Reinforcement Learning domain can be modelled as a sequence of states, actions and rewards using discrete time steps:

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$$

with $T$ being the terminal state. This interaction is also typically illustrated as in figure 2.1. This gives rise to the question, what happens when the specific problem that we wish to solve does not have a terminal state? Luckily there are also ways of dealing with these types of problems but those are outside of the scope of this thesis as I have decided to look at finite problems also called episodal tasks. These are problems where the Agent is shown some start state $s_0$ and has to find a series of actions that end in a terminal state $s_T$ yielding the some reward $r_T$. Such a sequence is also called a trajectory. Once a terminal state has been reached, The Agent can start anew, and possibly search other state-action pairs in the hopes that a better performing trajectory may be found. In this manner the episodal aspect comes into play.

![Diagram](image)

Figure 2.1: Diagram typically used to illustrate the interaction between the Agent and Environment as described in [Sutton and Barto, 2017, p. 80]
As the learning methods within Reinforcement Learning very much depend on the sets $S$, $A$ and $R$, deciding how these should be designed is one of the major hurdles as small changes might lead to big differences in behaviour. For instance, take a game of 4-in-a-row, one might assign a -1 reward per move in the hopes that this would minimise the amount of moves the Agent would use and thus strive to win as early as possibly. Unfortunately, this would end up leading the Agent to learn how to end a game using the fewest moves, regardless of who the winner is. On the other hand, using a reward function $r(s_t)$ that would assign a reward signal of 1 on winning moves and 0 for the rest might result in the Agent learning how to win but without the urgency of doing so as early as possible.

As mentioned, the trajectories might be infinitely long introducing its own set of problems. In the same manner, the sets $S$, $A$ might also be infinite, as is often the case with most real-world problems where the Environment cannot be described using discrete values. An example of this could be the helicopter controlling mentioned in Andrew Y. Ng and Sastry [2006], where $S$ is described using continuous values, rendering $S$ infinite. Once again, luckily there exists methods of handling such problems as well, but in this thesis I shall only describe problems with finite $S$ and $A$ sets, although these are by no means small for most problems. The combined set $S \cup A$ is also referred to as the state-space of a problem, and the size of it indicating how large the problem is. The size of the state-space of a problem is interesting as small problems can be solved using non-Reinforcement Learning methods by means of brute-forcing, which simply searches the entire state-space for an optimal trajectory. Of course, this is an exponential task in the size of the state-space and therefore is often an impractical solution technique. Fortunately, for many problems (particularly smaller ones), there exists Reinforcement Learning techniques that can find guaranteed optimal solutions through dynamic programming as the algorithms satisfy the Bellman equation requirements as will be described in section 2.3. The term solved in this domain of machine learning means that no matter the state the Agent is represented, a function can calculate the optimal sequence of actions that leads to the highest reward available from said state in constant time. Now, one might say that all problems having a finite state-space is solvable, and while in theory this is correct, the sheer magnitude of large problems is typically a detergent for this type of method. To illustrate this, games such as Go has a state-space of $10^{170}$ [David Silver, 2016] and Backgammon has $10^{20}$ [Tesauro, 1995]. These values are simply too large to calculate using contemporary computing techniques and would require more memory and compute time than is feasible.

2.2 Markov Decision Processes

Now that States, Actions and Rewards have been defined it is time to see how they are used in Reinforcement Learning. Because a trajectory is a sequence of states, actions and rewards we can model it as a Markov Decision Process
(MDP) if it has the Markov property [Sutton and Barto, 2017]. If it can be modelled as an MDP, this opens up a wide array of theory that Reinforcement Learning utilises to perform convergence proofs etc. Making a Reinforcement Learning problem an MDP is one of the fundamental approaches to many of the solutions within the literature and as such should be one of the first tasks at hand when deciding whether a problem should be solved using Reinforcement Learning or not.

As mentioned, MDPs require the problem to have the Markov property, which means that all the information contained within a given state is enough to choose what action to perform. In other words, the Agent does not need to keep a history of states that it has visited when performing an action. Only the state $s_t$ is required. To see how most board-games are MDPs is easy, as a given board-layout is typically independent of the sequence of moves.

Formally, a Reinforcement Learning problem has the Markov property if and only if:

$$p(s', r | s, a) = Pr[s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0] = Pr[s_{t+1} = s', r_{t+1} = r | s_t, a_t]$$

for all $s', s \in S$, $r \in R$ and $a \in A(s)$.

As $p$ is a probability distribution for each choice of $s$ and $a$ we are given:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r | s, a) = 1 \quad (2.1)$$

for all $s \in S$ and $a \in A(s)$.

From this the the probabilities of the state-transition can be extracted as:

$$p(s' | s, a) = Pr[s_{t+1} = s' | s_t, a_t] = \sum_{r \in R} p(s', r | s, a) \quad (2.2)$$

and the expected reward for state-action pairs as:

$$r(s, a) = E[R_{t+1} | s_t, a_t] = \sum_{r \in R} \sum_{s' \in S} r \cdot p(s', r | s, a) \quad (2.3)$$

These equations can be found in [Sutton and Barto, 2017]. Finally, for completeness sake, a finite MDP is an MDP where the sets of states, actions and rewards are finite, and as such have well defined discrete probability distributions. As the problems I will be dealing with in this thesis are finite problems, they can be modelled as finite MDPs. Furthermore, the benefit of finite MDPs is that they can be solved using dynamic programming approaches as they fulfil the requirements of being Bellman equations, and as such can guarantee to converge on an optimal solution [Sutton and Barto, 2017].
2.3 Value functions

So far, the role of the reward signal and its value I have described have been informal definitions. Therefore I shall be using this subsection to more clearly define what it is the Agent wants to maximise.

For a sequence of actions yielding reward $r_t$ at time $t$, we want to maximise the cumulative reward following time $t$. In other words, we wish to maximise the expected return, denoted $G_t$:

$$G_t = R_{t+1} + R_{t+2} + ... R_T$$

(2.4)

Now, this is a naive return, as the further into the future we go, the less certain we are that the reward is correct, as for each step we sample from a distribution of actions. Also, for non-episodal tasks, and especially for problems that are not finite MDPs, there would be no $R_T$, as there would be no terminal state. Therefore, a discounted version of the return is used, where more immediate rewards contribute more to $G_t$:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

(2.5)

with $\gamma \in [0, 1]$ representing the discount factor. This discount factor is one of the most used hyper-parameters within Reinforcement Learning, as it changes the behaviour of the Agent. For $\gamma$ values close to 0, the Agent will focus on immediate reward signals without looking at future rewards, whereas $\gamma$ values closer to 1 will take into account future rewards and as such might forego a more immediate reward if it results in a higher return in the long run.

To come back to episodal tasks we have to remove summation of infinite terms in equation 2.5. This can be done by modelling terminal states with self-loops, such that once a terminal state has been reached it is no longer possible to leave it, regardless of the actions chosen. Furthermore, the reward signal will be 0 for all transitions. Using this, the return can be rewritten as:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

(2.6)

For the Reinforcement Learning methods I will be describing in section 2.3 I shall use the notion of value-functions, which is something that is used within most Reinforcement Learning algorithms. These value functions estimate how good a specific action performed in a specific state is, or simply how beneficial it is to be in a given state in terms of the expected return $G_t$ at time $t$. For this to make sense, $G_t$ very much depends on the actions along the trajectory as mentioned earlier and therefore might change drastically depending on the trajectories from time $t$ and on-wards. Therefore a value-function is coupled with a policy $\pi$, indicating the behavioural pattern, so that we can calculate
the expected return. A policy $\pi$ is simply a mapping from states to actions, also denoted $\pi(a|s)$, meaning the probability of selecting action $a$ given a state $s$, making it a probability distribution over all $a \in A(s)$ for each $s \in S$. A policy might be either deterministic, as is the case with greedy policies or stochastic, which is illustrated in section (REF SECTION) about exploitation vs exploration.

Armed with the knowledge of the behavioural patterns of the Agent we can calculate the value-functions. The value of a state $s$ under a policy $\pi$ is denoted $v_\pi(s)$ and is the expected return $G_t$ given the state $s$ and following policy $\pi$. For finite MDPs $v_\pi(s)$ is then formally:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1}R_k|s\right]$$  \hfill (2.7)

for all $s \in S$. $v_\pi(s)$ is also called the state-value function under policy $\pi$.

Likewise, the value for the action-value function is formally:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|s, a] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1}R_k|s, a\right]$$  \hfill (2.8)

The magic within Reinforcement Learning then lies behind the methods of estimating these value-functions, as once we have a decent estimate of these, the policy can decide the course of actions leading to the highest expected return. Ultimately, this is the goal of all of the algorithms within the domain of this type of learning, as the one with the best estimate of these value-functions will perform the best. The reason why Reinforcement Learning is called so is then maybe also more obvious once we consider how the value-functions are estimated. As the type of problems that we attempt to tackle are those where we very often do not want to rely on supervised data or maybe none is available to us, the Agent has to bootstrap from its own experience. Therefore one might get the intuition that starting without any knowledge about the true value-functions it is possible for the Agent to estimate them by simply visiting all states for $v_\pi(s)$, or performing all actions to estimate $q_\pi(s, a)$. In other words, learning from experience in cases where the state-space is large or using dynamic programming when the transition probabilities are known. When states / actions are beneficial they are rewarded by the Environment and those that are detrimental are punished. Of course, this would all be under some policy $\pi$ but it would find an estimate to $v_\pi(s)$ or $q_\pi(s, a)$ for that particular policy. More formally, a fundamental property of the value functions, as with the Markov Property, is that they satisfy recursive relationships and therefore is a Bellman equation. This means that the value of a state $s_t$ is related to that of state $s_{t+1}$:
\[ v_\pi(s) = \mathbb{E}_{\pi}[G_t|s] \]
\[ = \mathbb{E}_{\pi}[r_{t+1} + \gamma G_{t+1}|s] \]
\[ = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[ r + \gamma \mathbb{E}_{\pi}[G_{t+1}|s'] \right] \] (2.9)
\[ = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) \left[ r + \gamma v_\pi(s') \right] \]

with \( s' \in S \), \( a \in A(s) \) and \( r \in \mathbb{R} \). The meaning behind equation 2.9 is that the value of state \( s \) is the expected return from each subsequent state, weighted by the probability of transitioning to those states. Furthermore, the value function is the unique solution to the Bellman equation [Sutton and Barto, 2017]. A similar proof holds for \( q_\pi(s, a) \).

Using equation 2.9 we can then find an estimate for \( v_\pi(s) \) by repeatedly applying it on its own output under policy \( \pi \), creating a better understanding of the Environment based on the experience the Agent achieves. The above mentioned approach is also called \( \text{iterative policy evaluation} \) [Watkins, 1989] and is illustrated in the example below.

**Example 1 - 4x4 Grid World.**

In this example the Environment is a 4 x 4 grid world with the top-left cell (0,0) and bottom-right cell (3,3) being terminal states. The Agent then occupies one of the cells and is tasked with finding one of the terminal states using the fewest possible steps. As such, the reward function adds a reward of -1 per step taken. The steps, or actions, available to the Agent at each state is to move left, right, up or down. The Agent starts off with a policy \( k \) that is completely random, meaning it assigns equal probability to each action happening and therefore the state-transition function is weighted by 0.25, as there are 4 actions per cell. For simplicity, each cell has all 4 actions, and if an action takes the Agent out of the grid-world, it is reinstated at the same position but still penalised with the reward of -1. Under this policy the objective is to evaluate the state-value function \( v_k(s) \). One could attempt to apply a simple brute-force solution method which would require exponential time, or since we know the transition function, \( \pi(a|s) \), a dynamic programming approach can be utilised. For this example the latter will be used and as can be seen in figure 2.2 The left column illustrates the value function after iteration \( k \), whereas the right column illustrates the best greedy policies with regards to the state-value function at iteration \( k \). For \( k = 0 \), each state is initiated with a value of 0. At \( k = 1 \) we see that the terminal states remain at 0, as the Bellman equation 2.9 will add 0 for the reward, and the discounted value of the next state is also 0, multiplied with the probability of ending up in the terminal state, which is 0.25 per action. For each of the non-terminal states, the reward is -1 + 0 (the initial value at \( k=0 \)) times 0.25 per action, of which there were 4, resulting in a net return of -1. Once each cell have been evaluated, iteration \( k=2 \) begins and the values of each cell is re-evaluated again and again until convergence. As can be seen by the policies in the right column, policy evaluation does not need to
improve the policy after a few iterations. Therefore, in practice, algorithms are
typically designed to not wait till convergence before updating the policy.

Figure 2.2: diagram showcasing value-function evaluation and the best greedy
policy with regards to the value-function at iteration k.

<table>
<thead>
<tr>
<th>$V_k$ for the Random Policy</th>
<th>Greedy Policy w.r.t. $V_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 0$</td>
<td></td>
</tr>
<tr>
<td>0.0 0.0 0.0 0.0</td>
<td></td>
</tr>
<tr>
<td>0.0 0.0 0.0 0.0</td>
<td></td>
</tr>
<tr>
<td>0.0 0.0 0.0 0.0</td>
<td></td>
</tr>
<tr>
<td>0.0 0.0 0.0 0.0</td>
<td></td>
</tr>
<tr>
<td>$k = 1$</td>
<td></td>
</tr>
<tr>
<td>0.0 -1.0 -1.0 -1.0</td>
<td></td>
</tr>
<tr>
<td>-1.0 -1.0 -1.0 -1.0</td>
<td></td>
</tr>
<tr>
<td>-1.0 -1.0 -1.0 -1.0</td>
<td></td>
</tr>
<tr>
<td>-1.0 -1.0 -1.0 0.0</td>
<td></td>
</tr>
<tr>
<td>$k = 2$</td>
<td></td>
</tr>
<tr>
<td>0.0 -1.7 -2.0 -2.0</td>
<td></td>
</tr>
<tr>
<td>-1.7 -2.0 -2.0 -2.0</td>
<td></td>
</tr>
<tr>
<td>-2.0 -2.0 -2.0 -1.7</td>
<td></td>
</tr>
<tr>
<td>-2.0 -2.0 -1.7 0.0</td>
<td></td>
</tr>
<tr>
<td>$k = 3$</td>
<td></td>
</tr>
<tr>
<td>0.0 -2.4 -2.9 -3.0</td>
<td></td>
</tr>
<tr>
<td>-2.4 -2.9 -3.0 -2.9</td>
<td></td>
</tr>
<tr>
<td>-2.9 -3.0 -2.9 -2.4</td>
<td></td>
</tr>
<tr>
<td>-3.0 -2.9 -2.4 0.0</td>
<td></td>
</tr>
<tr>
<td>$k = 10$</td>
<td></td>
</tr>
<tr>
<td>0.0 -6.1 -8.4 9.0</td>
<td></td>
</tr>
<tr>
<td>-6.1 -7.7 -8.4 -8.4</td>
<td></td>
</tr>
<tr>
<td>-8.4 -8.4 -7.7 -6.1</td>
<td></td>
</tr>
<tr>
<td>-9.6 -8.4 -6.1 0.0</td>
<td></td>
</tr>
<tr>
<td>$k = \infty$</td>
<td></td>
</tr>
<tr>
<td>0.0 -14.0 -20.0 -22.0</td>
<td></td>
</tr>
<tr>
<td>-14.0 -18.0 -20.0 -20.0</td>
<td></td>
</tr>
<tr>
<td>-20.0 -20.0 -18.0 -14.0</td>
<td></td>
</tr>
<tr>
<td>-22.0 -20.0 -14.0 0.0</td>
<td></td>
</tr>
</tbody>
</table>

Now, the Agent will be stuck if no change is made with regards to the policy $\pi$ as it does not matter how accurate we estimate the value-functions if we keep using the same policy. In figure 2.2 we see that the value-function converged but if we still use the random policy we know we can do better, as is also visible in the right side. Therefore an update must also happen to the policy with respect to the estimates of the value-functions. This gives rise to generalised policy iteration (GPI), which has two parts to it: the policy evaluation part which was described above, that converges the value-function under some policy $\pi$ with regards to the true value function and then the policy
improvement part that changes the policy with regards to the value-function. This is done by looking at the newly estimated value-functions and then greedily mapping states to actions that yield the best overall return. This will lead to a new policy $\pi'$ after which a new value-function evaluation begins again, and $v_{\pi'}(s, a)$ is to be estimated under the new policy $\pi'$. A policy $\pi'$ is defined to be better than or equal to a policy $\pi$ if the expected return of $\pi'$ is greater than or equal to that of $\pi$ for all states. In other words, $\pi' \geq \pi \iff v_{\pi'}(s) \geq v_{\pi}(s)$ for all $s \in S$. Policy improvement is more formally described as:

$$
\pi'(s) = \arg\max_a q_\pi(s, a)
= \arg\max_a \mathbb{E}[r_{t+1} + \gamma v_\pi(s + 1)]
= \arg\max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')]$$

(2.10)

What equation 2.10 essentially describes is that the new policy $\pi'$ now updates the actions per state in a deterministic fashion based on which action will lead to the highest return in state $s'$. To see that equation 2.10 holds, we need to define what it means to be an improvement. Therefore the above equation needs to use the policy improvement theorem that states that for any pair of deterministic policies $\pi$ and $\pi'$ such that, for all $s \in S$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s).$$

(2.11)

then policy $\pi'$ must be as good as $\pi$ or even better. Therefore the following must also hold:

$$v_{\pi'}(s) \geq v_{\pi}(s)$$

(2.12)

Which is what was needed for equation 2.10. The goal is then to find policies such that the maximum expected return is found for the problem in general. Fortunately, by virtue of equation 2.11 and 2.12, there always exists a policy that is better or equal to all other policies. This policy is called the optimal policy and even though it might not be the only optimal policy, it is denoted $\pi_*$. All policies $\pi_*$ share the same state-value function, aptly named the optimal state-value function, denoted $v_*$ and is defined as:

$$v_*(s) = \max_\pi v_\pi(s), \text{ for all } s \in S$$

(2.13)

Similarly, optimal action-value function denoted $q_*$ and is defined as:

$$q_*(s, a) = \max_\pi q_\pi(s, a), \text{ for all } s \in S \text{ and } a \in A(s)$$

(2.14)

For a state-action pair $(s, a)$, these equations then gives the expected return for performing action $a$ in state $s$, and then following an optimal policy. This means we can write them in terms of each other.

$$q_*(s, a) = \mathbb{E}[r_{t+1} + \gamma v_*(s_{t+1}) | s, a]$$

(2.15)
with equation 2.15 stating that the value of the best action of a given state is the discounted expected return of the next state plus the reward associated with it.

Silver illustrates General Policy Iteration as in figure 2.3, which shows how alternating between policy evaluation and policy improvement results in an optimal policy \( \pi_* \).

Figure 2.3: Illustration of the GPI pipe-line, showing first the policy evaluation under some policy \( \pi \) followed by policy improvement updating the policy greedily with respect to the value-function \( v \).

While the above approach to finding the optimal policy \( \pi_* \) and the optimal value-function \( v_* \) is guaranteed to work, GPI is unfortunately, for almost all interesting problems, an infeasible method to solving a Reinforcement Learning problem, although it is of course better than the brute-force alternative. The reason why is of course the same as earlier, that to be able to perform policy evaluation, we must store each value in a table and for large problems, although finite, this is not possible. Therefore other methods have to be utilised inspired by GPI, with the Agent creating trajectories throughout the state-space, meaning the Agent will collect experience in only a small subset of the overall state-space, boot-strapping upon its own experience more so than analytical calculations. This will result in an approximation of the solution and as such is not guaranteed to find an optimal solution. Now, although we are not guaranteed optimality, we are by no means without power to find good solutions. The nature of learning from experience leads the Agent to learn more from states (gather a better approximation of the value-functions) that is visited more. This results in some areas where the Agent might be really strong and others where it is less so. As an example of this, AlphaGo had, for many situations, superior play to that of humans but in some special occasions it played moves that did not make sense at all [David Silver, 2016, Game 5]. This illustrates that although some states are less trained, and these might even be a large fraction of the overall states, it played well in an expert setting, as it had learned to focus on situations that humans played, without the need to train on the entire state-space. This is one of the strengths of Reinforcement
Learning that distinguishes it from other approaches to approximately solving MDPs [Sutton and Barto, 2017].

2.4 Prediction and Control

There are two categories when it comes to Reinforcement Learning problems. There is the prediction problem and the Control problem [Watkins, 1989]. Prediction is the problem of estimating the state-value function $v_\pi(s)$. In other words, a prediction algorithm should be able to predict the cumulative collected return from a given state and on-wards. This can be thought of as a measure of "How good is it to be in this situation". For Control problems, the task is to find the best policy $\pi$, so we can get the best performing Agent, in other words, a control algorithm must be able to estimate the cumulative collected return of successive actions taken from a given state. If we are able to simulate the effects of actions, meaning the Agent has a model of the Environment (which is one of the strengths of games, as the world is designed), we can find a solution to the Control problem by looking at which state has the highest prediction score for each action taken from the current state. Otherwise, if the Agent does not have any knowledge about the domain it is supposed to act within, a solution to the Control problem can be found by estimating the action-value function $q_\pi(s,a)$.

The following subsections describe different methods used to approximate these value-functions and finding the best policy $\pi$ inspired by GPI. Furthermore, for simplicity, they use the simpler case of tabular learning, meaning the state-spaces are finite and it is possible to store it in memory together with the value-function’s estimate in a table. In section 2.7 I will describe how tabular learning can be extended to problems that are too large to store.

2.4.1 Monte Carlo Learning

For Monte Carlo (MC) learning the idea is to estimate the value-functions via experience throughout a complete episode under some policy $\pi$. This learning approach simply creates a trajectory from some starting state $s_0...s_T$ to a terminal state $s_T$. Once it reaches this terminal state, we can calculate the discounted return for each state (or action) involved in the trajectory, backwards in time from the terminal state. Then, for each time a state (or action) is involved in a trajectory, the mean of each of the returns are calculated, therefore a counter $N(s)$ keeps track of the amount of times a given state has been visited. Remember the return from equation 2.6 was: $G_t = R_{t+1} + \gamma R_{t+2} + ...\gamma^{T-t-1}R_T$. Therefore, to be able to calculate the return at time $t$, we must wait till we have the return at time $t + 1$, which results in the need for the episode to terminate before we can calculate any returns at all. A dependency for MC learning methods is then the requirement that the problem at hand is an episodal problem.

A typical pipeline for using MC learning in finding an approximation to the prediction problem then looks like:

1. Given a policy $\pi$, run a trajectory using $\pi$: $s_0, s_1, ..., s_T$. 
2. Once a complete trajectory is computed, observe the return $G_t$ for each state involved in the trajectory.

3. Update a counter associated to a given state, by the amount of times it has been visited: $N(s_t) = N(s_t) + 1$

4. Update the value-function’s value by the mean of the return for the state: $V(s_t) = \frac{V(s_t) + G_t}{N(s_t)}$.

As we are dealing with a tabular learning method, the value-function is then merely a table lookup, such that $V(s)$ is an entry into a table, updated by: $V(s_t) = \frac{V(s_t) + G_t}{N(s_t)}$. Using this approach, $V(s)$ is an approximation of $v_\pi(s)$ as instead of looking at all states and their state-transition probabilities under policy $\pi$, we sample trajectories based on the policy $\pi$. Fortunately, by law of large numbers, as $N(s) \to \infty$, $V(s) \to v_\pi(s)$.

Another way of writing the update rule is the more common expression:

$$V(s_t) = V(s_t) + \alpha(G_t - V(s_t))$$  \hspace{1cm} (2.16)

, with $\alpha \in [0, 1]$. Instead of calculating the mean each time, we update $V(s_t)$ based on the error $\delta$: $\delta = G_t - V(s_t)$ multiplied by some learning parameter $\alpha$. The learning parameter $\alpha$ is introduced as to remove the dependency on historic data, as the problems might be non-stationary. In other words, if old data becomes less relevant due to some change in the Environment or Agent behaviour, using the learning parameter $\alpha$ will effectively scale more recent values of $V(s)$ higher than historic ones. $\alpha$-values that are closer to 0 will result in the Agent not wanting to update the value-function much, emphasising historic data, whereas $\alpha$-values closer to 1 will encourage the Agent to learn more from recent actions. Together with $\gamma$, $\alpha$ is also one of the three major and recurring parameters in almost all Reinforcement Learning algorithms.

This way of updating will resemble the update rules for all future algorithms that I shall describe in this thesis. With that said, even though the different methods look similar, small changes in the update rule will result in major differences in update- and convergence speed while also affecting the behaviour of the Agent. The main point is simply that each of the approaches have their own way of estimating the value-functions, but their goal is the same, namely to approximate $v_\pi(s)$ or $q_\pi(s,a)$.

Now, this was policy evaluation for MC learning, and as is described in section 2.3 policy improvement also has to take place, as otherwise we do not update the policy $\pi$ and the Agent’s behaviour stays the same. Luckily, policy improvement works the same way for MC learning as it does for GPI. It is an iterative approach, alternating between evaluation and improvement, where the improvement step takes the greedy actions according to the respective Control approach. For simplicity, let us assume the Agent does not have a model of the Environment, and the Control approach is to take the action with the highest $q_\pi(s,a)$ value. In other words, the greedy policy is then, for each $s \in S$ deterministically choose an action $a \in A(s)$ with maximal action-value:

$$\pi(s) = \arg\max_a q(s,a).$$  \hspace{1cm} (2.17)
Policy improvement is then at iteration $k + 1$ taking the greedy policy $\pi_{k+1}$ with respect to $q_{\pi_k}$. The Policy improvement theorem in equation 2.10 then applies to $\pi_k$ and $\pi_{k+1}$, as for all $s \in S$:

$$q_{\pi_k}(s, \pi_{k+1}(s)) = q_{\pi_k}(s, \arg \max_a q_{\pi_k}(s, a))$$

$$= \max_a q_{\pi_k}(s, a)$$

$$\geq q_{\pi_k}(s, \pi_k(s))$$

$$\geq v_{\pi_k}(s).$$

(2.18)

Using this, the greedy update rule in equation 2.17 for MC in conjunction with the policy evaluation is then guaranteed to converge to the optimal policy $\pi^*$ and value-function $q^*$ [Sutton and Barto, 2017]. Unfortunately, for equation 2.18 to hold, an assumption has to be made, and that is that we can run infinite episodes per policy evaluation to ensure we have a proper approximation of the value-function, which, of course, is impossible in any practical manner. The approach to circumvent this issue can be done in different ways. One way of avoiding infinite episodes per policy evaluation is to create a threshold, such that we stop evaluating once the $\delta$-error is below this threshold. If the threshold is small enough, this will result in a good enough approximation for most practical solutions. Unfortunately this might still take a while as there is no guarantee on the speed of which the $\delta$-error converges towards 0, so another approach is typically favored. Instead, the idea of approximating the value-function well enough is forgone and we simply stop after some arbitrary $n$-iterations. This will move the value-function toward $q_{\pi_k}$, but without the expectation that we will get an accurate approximation except after many steps. In reality, for most applications, policy evaluation is done using only one episode, after which policy improvement takes place. Unfortunately, there is still no formal proof that this works [Sutton and Barto, 2017], but the idea seems to make sense, intuitively, and is backed up by empirical evidence, through the many success stories of Reinforcement Learning.

This alternate version of GPI is also illustrated by David Silver as in figure 2.4, except he is using an $\epsilon$-greedy version instead of a deterministic version for the policy improvement step, but the idea is the same. I shall go into detail with exploration and what it means to have an $\epsilon$-greedy policy in section 2.6:
2.4.2 Temporal Difference Learning

Another type of learning is the Temporal Difference (TD) learning, which addresses some of the issues with MC learning. One issue with MC learning is, that it depends on full trajectories to be computed. The idea behind TD learning is that instead of waiting till the end of the episode to update the return of a state (or action), a target is created based on the value of the next state (or action). In this manner, TD learning bootstraps its targets toward that of the next step instead of waiting the full episode, creating biased targets as we update a guess towards another guess. One can then ask the question, is basing the update step on merely the next value a good enough estimate of the current state, and the answer is unfortunately rather unclear. Given a policy $\pi$, TD based methods are still guaranteed to converge to $v_\pi$ [Sutton and Barto, 2017], with a decreasing $\alpha$ value, but the speed of which this is done might drastically change based on how many steps we use during the look-ahead step. Instead of just one-step look-ahead, we could base the update rule on the next n-steps to be sure that the update reflects the actual return $G_t$, with a high enough n to possibly encompass all steps from now till the end. How far ahead should the Agent look for the update targets to be the most stable? If we look all the way to the end, we end up with MC learning, as that is specifically what it does. On the other hand, we might look at only the next step. To handle all the cases in between one-step look-ahead and MC-learning we have TD($\lambda$), with the $\lambda$-parameter defining how much we should look-ahead, with $\lambda$-value of 0 being a simple on-step look-ahead, and a $\lambda$-value of 1 being that of MC-learning. For this section, I shall introduce TD(0), as it is the simplest case of TD-learning and has many successful applications. The power of TD(0) compared to that of MC learning is the fact that it only uses one-step look-ahead as its target return, making it more biased but introducing much less variance, as each step in the episode introduces its own uncertainty. Furthermore, updating immediately makes it possible to perform on-line training, which can speed-up learning
by quite a lot and even enables learning from incomplete trajectories. An added benefit of immediate updates is that TD(0) exploits the Markov Property due to the one-step dependency and therefore is a more efficient choice in Markov Environments, whereas MC learning clearly does not, as it needs to complete the entire trajectory.

The two main questions are then, how do we perform policy evaluation and policy improvement? For policy evaluation of the Prediction problem, we evaluate some policy \( \pi \) using the following general pipeline:

1. Given a policy \( \pi \), perform one-step look-ahead based on \( \pi \), that is, perform one virtual step in accordance with \( \pi \).
2. Use value of the look-ahead as the TD-target for the current state:
   \[ r_{t+1} + \gamma V(s_{t+1}) \]
3. Find the difference between the target and the current value:
   \[ \delta_t = \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \]
4. Update the value-function’s value by:
   \[ V(s_t) = V(s_t) + \delta. \]

Here \( r_{t+1} + \gamma V(s_{t+1}) \) is the target value that we wish to boot-strap towards, namely, the estimated discounted value of being in the next step plus the reward associated with the action that leads us there. The expression \( r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \) is also called the TD-error, as it is the error between the target value and the value of the current state. The update rule is then the following:

\[
V(s_t) = V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)). \tag{2.19}
\]

, with \( \alpha \in [0, 1] \). As can be seen, the update rule resembles that of MC learning but the return \( G_t \) has been exchanged with the TD-target \( r_{t+1} + \gamma V(s_{t+1}) \).

Now that we have a way to evaluate a policy, policy improvement needs to happen. For the Control problem, I shall use a model-free approach, once again, as with the description of MC learning, meaning the Agent cannot simulate the Environment when it comes to predicting outcomes of actions, and therefore has to approximate the action-value function \( q^\pi(s,a) \). To do this, a choice stands between two categories, namely whether to be off-policy or on-policy. Off-policy methods are learning methods where the target is independent of the policy currently being estimated, whereas On-policy takes behavioural aspects of the Agent into account. The difference is that off-policy methods directly approximates the value function \( q^* \), whereas on-policy might not find the overall optimal solution but one that takes into account how the Agent behaves. To illustrate this, the following example should shed some light onto this.

**Example 2: Cliff walking:**

Given a 4 x 12 grid world, with the lower left cell (3,0) being the Starting point, and the lower right (3,11) being the Goal, the task for the Agent is to find the Goal using the fewest possible steps. Furthermore, a cliff is introduced, with all cells in the bottom row being the cliff, except the start cell and the goal cell. If the Agent, at any point, performs an action that takes it into one of the cliff cells, then it has to start over again, and is placed in the starting cell. Each
cell has 4 possible actions, namely to move up, left, down or right. If an action takes the Agent outside of the grid world, it is simply reinstated into the cell that it attempted to leave. To motivate the Agent to perform its task in as few steps as possible, a reward of -1 is given per step, while performing an action that takes the Agent into one of the cliff cells incurs a reward of -100. Figure 2.5 shows the Cliff Walking example with two paths, namely the optimal path, which is to traverse the world next to the edge of the cliff, while the safe path would take the Agent as quickly from \( S \) to \( G \) as it can while keeping the Agent as far away from the Cliff as possible.

![Figure 2.5:](image)

Now, imagine we introduce a 10% chance of performing a random action to the policy. Then following the optimal path will have, at each step 10% chance of selecting a random move, of which there is 25% chance of falling down the cliff. The probability of not falling down the cliff, following the optimal path decreases the further we have to travel along the cliff. If instead the world was not 4 x 12 but 4 x 100, the odds of not falling down the cliff would be almost 0, and even though the Agent is following the optimal path, it would incur a far lower reward than if it took into account this behavioural aspect of random moves. Therefore, using an On-policy method would be beneficial here, as it would, overall, perform the task without risking to fall down the cliff and end up with a higher score than the Agent trained using Off-policy methods. For further analysis of this, see my project on Cliff Walking [Pedersen, 2019].

As I have chosen to utilise an Off-policy TD(0) algorithm, I shall only illustrate how Control is done through off-policy.

The update rule for the pipeline, given an approximation \( Q_\pi(s, a) \) of \( q_\pi(s, a) \), is then:

\[
Q_\pi(s_t, a_t) = Q_\pi(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a) - Q_\pi(s_t, a_t)]
\]  

(2.20)

, with \( \gamma \) and \( \alpha \in [0, 1] \). Now that the policy evaluation for the Control Problem has been made, the policy improvement is simply to make greedy updates to the policy with regards to \( Q_\pi(s, a) \), as have been done so far. Alternating between policy evaluation where the updates are performed using equation 2.20 followed by policy improvement. This has shown to converge towards the optimal solution \( q^* \) given enough iterations, even with single episodal policy evaluations [Sutton and Barto, 2017]. An algorithm for running Off-policy TD(0)
Learning is Q-learning, which was originally made by Watkins, and is shown below in figure 2.6:

### Q-learning (off-policy TD control) for estimating \( \pi \approx \pi^* \)

<table>
<thead>
<tr>
<th>Initialize ( Q(s,a) ), for all ( s \in S, a \in A(s) ), arbitrarily, and ( Q(\text{terminal-state}, \cdot) = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat (for each episode):</td>
</tr>
<tr>
<td>Initialize ( S' )</td>
</tr>
<tr>
<td>Repeat (for each step of episode):</td>
</tr>
<tr>
<td>Initialize ( S' )</td>
</tr>
<tr>
<td>Choose ( a ) from ( S ) using policy derived from ( Q ) (e.g., ( \epsilon )-greedy)</td>
</tr>
<tr>
<td>Take action ( A ), observe ( R, S' )</td>
</tr>
<tr>
<td>( Q(S', A) \leftarrow Q(S', A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)] )</td>
</tr>
<tr>
<td>( S \leftarrow S' )</td>
</tr>
<tr>
<td>until ( S ) is terminal</td>
</tr>
</tbody>
</table>

Figure 2.6: Q-learning algorithm, as described in \cite{Watkins, 1989}

#### 2.4.3 TD(\( \lambda \))

For the previous section, I went into detail with how well TD(0) dealt with learning, and how one could apply a specific algorithm, namely Q-Learning, to find a solution to the Control problem. Now, instead of performing a one-step look-ahead I will describe the trade-off between looking \( n \)-steps ahead compared to that of TD(0) and MC learning. In reality, this is a matter of two things: Bias-variance trade-off and learning-rate. As mentioned previously, TD(0) gives us a biased target, as the target very much depends on next-step’s value, whereas MC learning gives us the actual return \( G_t \), meaning there is no bias. On the other hand, MC learning introduces a high amount of variance, as for each step, depending on the policy, we might perform a random choice of actions and as we are waiting till the end of the episode before we get the return, the variance is large for initial states, less so for later states. The task at hand is then, how do we balance this bias-variance trade-off. The second matter of trade-offs is, that for TD(0) we only effectively affect one value of the value-function that we wish to evaluate in cases where there is sparse reward signals, meaning for most steps rewards are 0. To understand this, imagine a grid world with a value-function whose values are all initialised to zero. Then performing one-step look-ahead will use the return of the next step + the reward signal associated with the action. If the reward signal is 0 for all steps except when the Agent reaches its goal, upon which it is rewarded by some arbitrary large reward, then TD(0) would only update the second-to-last state (or action). This is because 0 change was discovered throughout the entire trajectory except for the last action. For the next episode, the entire world now has two states in which a target differs from any given state \( s_t \), namely the goal state, and the one just previous to it, meaning we now update 3 states instead of 2. This will continue until we have run enough episodes so that every step in the trajectory has received a target. Remember, the target is not the real target after just one update, so this might seem like a wasteful use of episodes. Why should only the states closer to the terminal state, or states exhibiting a reward signal, receive a high degree of training, whereas states far from them receive much less? This
is the motivation that sparked the \textit{n-step} return, which looks similar the the $G_t$ return used in MC, except it only uses the next n-steps from time $t$, and will not necessarily wait until termination before calculation of the return. More formally it looks like:

\begin{align*}
TD(0) & \quad n = 1 : \quad G^1_t = r_{t+1} + \gamma V(s_{t+1}) \\
& \quad n = 2 : \quad G^2_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) \\
& \quad n = k : \quad G^k_t = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{k-1} r_{t+k} + \gamma^k V(s_{t+k}) \\
MC & \quad n = \infty : \quad G^\infty_t = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{T-1} r_T \tag{2.21}
\end{align*}

To simplify it even further, the n-step return takes the form:

$$G^n_t = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}) \tag{2.22}$$

As can be seen in equation 2.21, there is a close relationship between the return and the target used in TD learning, making both TD(0) and MC special cases of the n-step TD method.

The update rule for n-step TD is then a slight modification to that of the MC update:

\begin{align*}
MC : V(s_t) &= V(s_t) + \alpha (G_t - V(s_t)) \\
n - \text{step TD} : V(s_t) &= V(s_t) + \alpha (G^n_t - V(s_t)) \tag{2.23}
\end{align*}

, with $\alpha \in [0, 1]$ as usual. Going back to the grid-world problem before, illustrating the wasteful nature of TD(0) with regards to the episodes, we can see that if we substituted TD(0) with n-step TD we would get more use of the information per episode, as illustrated in figure 2.7.

Figure 2.7: Grid world illustrating the efficiency of an episode with regards to one-step look-ahead learning and n-step. \textbf{LEFT} diagram shows a trajectory of the Agent, where a reward of 0 is given except when the goal has been reached. \textbf{MIDDLE} diagram shows what element would be updated using one-step learning. \textbf{RIGHT} diagram shows 10 elements along the trajectory that would be updated using 10-step learning. [Sutton and Barto, 2017]

Of course, it is not evident that wasteful episodes, as is shown in figure 2.7 would impact the learning of the Agent. To show that there is a difference, example 3 below will illustrate this:
Example 3: n-step TD methods on Random Walk.

Given a world of 19-states connected as a string, meaning each state has two transitions, one to the left neighbour and one to the right, with the end states being terminal. A reward of 0 is given for each action, of which there are two: move left or move right. A move into the left most terminal state incurs a reward of 0, whereas a move into the right most terminal state incurs a reward of 1. The policy to evaluate is then a random-walk policy, meaning at each state we have equal probability of moving right or left, except once a terminal state has been reached, then it is impossible to move out of them. Furthermore, by virtue of being terminal, once reached, no further actions will grant any reward. [Sutton and Barto, 2017] illustrates that the average RMS error over the 19 states and the first 10 episodes is lowest for n=4, with an $\alpha$-value of 0.4 as can be seen in figure 2.8:

![Figure 2.8: Root Mean Squared Error of each state, between that of the n-step TD method and ground truth. As can be seen, n=4 results in the lowest error [Sutton and Barto, 2017]](image)

From figure 2.8 it is clear that there is a difference in how accurate the n-step TD methods manage to approximate the real value-function under the random-walk policy. This also, unfortunately, means that a need to find the correct value of n is necessary, resulting in yet another parameter-field that needs to be searched. To reduce this problem, another approach which attempts to combine multiple values of n is used. One such approach could be to average over all the values of n that we wish to use and use that as the n-step return. This would lead to an increase in algorithm complexity which is unfortunate as it would not scale well for tasks having long episodes. Instead, a geometric mean is used called the $\lambda$-return, which is what the $\lambda$-parameter in TD($\lambda$) refers to, and is the most common approach to finding a proper value of n. The $\lambda$-return looks like:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^n$$  \hspace{1cm} (2.24)
, with $\lambda \in [0, 1]$. For episodal tasks we can rewrite the lambda-return in terms of non-terminal and terminal states as:

$$G^\lambda_t = [(1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1}G^n_t] + \lambda^{T-t-1}G_t.$$  \hspace{1cm} (2.25)

Effectively what this return does, is applying a weight on the return after 1 step, 2 steps, ..., n-steps according to $\lambda$. For small values of $\lambda$ we get that more emphasis is put on smaller values of n, with $\lambda = 0$ being equal to TD(0), as can be seen if we substitute $\lambda$ with 0 in equation 2.25, where the return would then be equal to that of $[(1 - 0)0^0G^1_t] + 0 = 1 \cdot G^1_t$, as the sum would yield 0 for all terms except the first, and the terminal term would also yield 0. $G^1_t$ is exactly that of the one-step look-ahead TD(0) method, as can be seen in equation: 2.21. On the other end of the spectrum, a $\lambda$-value of 1 is the MC return, as the summation would be multiplied by a factor of 0, and only the final return is left, which is $G_t$, and therefore identical to the MC return. In this manner, TD($\lambda$) encompasses all the different values of n, weighted by what we as the designers might believe is a good value of $\lambda$. Of course, this didn’t relieve us from an extra hyper-parameters but it reduces the complexity of running each value of n separately and keeping a mean, as now we can test all n at once using the $\lambda$ parameter.

For completeness sake, policy improvement for TD($\lambda$) is identical to that of TD(0) and MC learning, as they are merely special cases of TD($\lambda$).

2.5 Eligibility traces

Having finally arrived at the last part of the approaches I shall be using in this thesis, I will briefly describe some of the issues with TD($\lambda$). It may seem that TD(0) and MC learning are both inferior to that of TD($\lambda$) as it can be used to find a proper length of look-ahead steps. The truth is that both MC and TD(0) has their place in Reinforcement Learning, as even with TD($\lambda$) there are certain issues. The first glaring problem is of course, that it looks-ahead, meaning the return $G^\lambda_t$ is dependent on finishing the episode just as with MC learning. It can be truncated of course, when we feel that the $(1 - \lambda)$ factor becomes small enough, as it will decay the value further from a given state $s_t$ (or action) like the $\gamma$ parameter, using some threshold mechanism. Although adding a threshold will reduce the wait time, the problem still remains that we cannot update the value until we have the return $G^\lambda_t$. In other words, this is the exact same problem as with MC learning. Fortunately, there is a way to alleviate this problem through the use of eligibility traces, so that instead of looking forwards in time (that is, for each state we have to wait till some n steps have computed before updates can happen), we look only backwards in time, to the previous steps. What eligibility traces attempt to do, is weighting each element along a trajectory based on how frequent and how recent the element contributed towards the goal. The longer time has passed from when we observed that element (state or action) being used its value decays and the current state’s, $s_t$, influence on these less recent states become less relevant,
as we might have been able to get to \( s_t \) from various other trajectories and as such we become more uncertain that the return of \( s_t \) correctly reflects that of the previous states. More formally, eligibility traces is a value per state containing the weight of that state resembling its relevancy compared to the current state \( s_t \), meaning we add a scalar weight \( e \in [0, 1] \) to the memory of the value-function’s table storing the states. The eligibility trace is defined as:

\[
E_0(s) = 0 \\
E_t(s) = \gamma \lambda E_{t-1}(s) + 1(S = s)
\]  

(2.26)

Where \( E_0(s) \) is the initial weighting of states at time 0, where no returns have been observed yet, and \( E_t(s) \) is the decayed value of the previous steps + the full value of the current state, meaning the current state is fully weighted (given a value of 1). If we combine these, we see that the eligibility traces for e.g. a trajectory containing 4 steps would look like:

\[
E_0(s) = 0 \\
E_1(s) = E_0(s) + 1(s) \\
E_2(s) = \gamma \lambda E_1(s) + 1(s) \\
E_3(s) = (\gamma \lambda)^2 E_1(s) + \gamma \lambda E_2(s) + 1(s) \\
E_4(s) = (\gamma \lambda)^3 E_1(s) + (\gamma \lambda)^2 E_2(s) + \gamma \lambda E_3(s) + 1(s)
\]  

(2.27)

, where at each time step \( t \) we recompute the eligibility trace of each state according to their importance. We do not have to recompute the value of each previous state as shown above when calculating the weight \( E_t \), as they will all be combined into a single value, namely \( E_t \), using the definition as seen in equation: 2.26. Therefore, each step leading up to \( s_t \) is already a part of the value for \( E_{t-1}(s) \), meaning example 2.27 can be reduced to simply:

\[
E_0(s) = 0 \\
E_1(s) = 0 + 1(s) \\
E_2(s) = \gamma \lambda E_1(s) + 1(s) \\
E_3(s) = \gamma \lambda E_2(s) + 1(s) \\
E_4(s) = \gamma \lambda E_3(s) + 1(s)
\]  

(2.28)

Using this simple calculation, each state has an eligibility trace and we can update the value-function \( V(s) \) for every state \( s \), backwards in time instead of forwards, meaning we can change the value of the entries of the value-function already seen instead of waiting for some future return. The update rule to our value-function then looks like:

\[
\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)
\]  

(2.29)

\[
V(s) = V(s) + \alpha \delta_t E_t(s)
\]  

(2.30)

What the update rule in equation 2.30 says is that at any time step \( t \), we find the TD-error \( \delta_t \) by performing one-step look-ahead and then use this error to
update each previous state in the direction of this error based on $\alpha$, and weighted with that state’s eligibility trace. This will result in more recent states to be updated more than states further down in the trajectory. It is an update rule that updates all states at once, each step of the trajectory [Lee, 2005, Silver, 2015, Sutton and Barto, 2017].

What eligibility traces really accomplishes is not to achieve faster convergences but to create a mechanical approach to TD($\lambda$) methods, that can be programmed as the dependency on future-looking is removed, and instead relies solely on states already seen. For proofs of why these are exactly equivalent look at [Sutton and Barto, 2017].

2.6 Exploration

Now that the fundamental differences in Reinforcement Learning approaches and methods have been discussed, there is one aspect that I have left out, for simplicity’s sake in the MC, TD(0) and TD($\lambda$) based solutions. And that is the topic of exploration. One more assumption had to be made when it came to policy improvement, and that is that every state is visited. Of course, performing policy evaluation of a policy based on sample trajectories will not look at every state, meaning it is rather easy to imagine scenarios where the Agent would greedily choose actions during policy improvement that would lead to a local maxima. Example 4 will illustrate this issue further.

Example 4: k-armed bandit problems

Imagine a simple case of having a k-armed bandit, i.e. a slot machine with k-levers. If we tasked the Agent with finding an optimal strategy of playing these machines, how should it go about doing so? We do not have any knowledge about the underlying reward distribution of each lever, and therefore do not know anything about the real action-value function $q^*(s,a)$. In this example, the action space is simply pulling one of the levers at any given time $t$ based on the Agent’s approximation $Q_t(s,a)$ of $q^*(s,a)$. The trajectories in each episode are then the rewards given by the machine at time $t$ combined with the estimated reward $Q_t(s,a)$. Using tabular learning methods, we can store each action-value in a table, with each row representing the estimated value of pulling the corresponding lever. Before training, we might initialise the values as 0, as we do not have any experience regarding the outcomes of any lever. If we then update the action-values based on the result (the reward given by the machine) and the $\delta$-error for a specific learning method, be it MC or TD based, we would at any time always have an action whose action-value, or q-value, is the highest, with ties broken arbitrarily. Now, as mentioned we do not know anything about the underlying reward distribution of the k-armed bandit. This could result in an unfavourable emphasis on arms that we have observed did well. To illustrate this, imagine an arm being pulled yielding a reward of 1. Now the action-value for that arm would increase as we now expect it to be better than the remaining arms. If we then performed policy improvement based on this one-step policy evaluation, using a greedy deterministic approach, the new
improved policy would claim that pulling the lever that gave the reward of 1 is the best course of action. Now it is time to evaluate the new policy, and following the newly updated policy, the Agent would pull the arm that has the highest action-value and observe some reward. If we say that each time the Agent pulls this arm it observes some positive value larger than 0 but less than 10, the Agent would always only approximate this one arm, as the other arms had a q-value (action-value) of 0 due to the fact that they were never tested. The Agent would have approximated the value function to some expected value of the true reward distribution for the one arm. At the end of the day, we would have an Agent, based on some arbitrary choice of initial arm-pulling, that would always decide to pull just that one arm. Now, imagine another arm had a reward distribution ranging from 10 to 20, then that arm is obviously better than the one ranging from 1 to 10. Using a purely deterministic approach to policy improvement would likely not have found this arm. Therefore, there is a need to sometimes perform non-greedy actions, so that the Agent can explore actions that it is less certain about, in the hopes that it might find a better solution.

This problem is the one of balancing exploitation and exploration, where exploitation is the Agent’s willingness to exploit current knowledge and perform greedy actions with regards to its estimated value-functions and exploration is to take a non-greedy option, in the hopes that the less explored actions might yield a better overall score. Example 4 illustrated the issue with pure exploitation, but of course, there is also problems related to exploration. If the Agent follows a purely exploratory approach, it would never utilise its estimated value-functions and in essence be reduced to a random Agent. Therefore exploration typically starts high, and then the more certain the Agent is that its estimates are approximating the true value-functions well enough, exploration decreases. In the end, the goal is to have an Agent that has explored the state-space while simultaneously ending with a deterministic policy. To what end exploration should be done, how high exploration should be initialised as, when it should end are all questions that, unfortunately only has theories based upon empirical data, meaning there is no fixed way of determining to what degree an Agent should explore to converge on the optimal policy $\pi^*$.

As mentioned, during Policy Iteration of the MC- and TD-learning based methods, we needed to introduce some stochasticity into the greedy choice of action during policy improvement, and to that end $\epsilon$-greedy is one very common approach. An $\epsilon$-greedy policy takes a greedy choice of action based on the value-function for state $s$ with a probability of $1 - \epsilon$, and performs a random action with probability of $\epsilon$, for $\epsilon \in [0, 1]$ as shown below:

$$\epsilon - greedy \pi(a|s) = \begin{cases} Pr[\arg\max_a Q(s,a)] & = 1 - \epsilon \\ Pr[\text{random action}] & = \epsilon \end{cases}, \epsilon \in [0, 1] \quad (2.31)$$

Using the above formulation of $\epsilon$-greedy policy improvement, we can search the state-space in a way that allows for more accurate estimations of values that are not explored as much. In the cliff-walking example, example 2, I showed how an off-policy would find the optimal path but risk falling down the cliff. The
stochastic element of a 10% chance of a random action is exactly what would happen if the Agent used an $\epsilon$-greedy policy with an $\epsilon$-value of 0.1. The value $\epsilon$ is the last general parameter of almost all algorithms related to Reinforcement Learning, together with $\gamma$ and $\alpha$. Of course, other ways of exploring can be attempted but as far as I have seen, the simplest and most common in the literature is the $\epsilon$-greedy policy.

### 2.7 Function approximation

Reinforcement Learning is interesting because it is not confined to finding a solution to problems that are limited in scope with regards to the state-space. While previous examples in this thesis have had manageable state-spaces, this is not so for many real life problems. For instance, Backgammon has a state-space of $10^{20}$ states [Tesauro, 1995], Go has a state-space of $10^{170}$ states [David Silver, 2016] and many problems even have a continuous state-space, like the steering of a helicopter [Andrew Y. Ng and Sastry, 2006]. With this in mind, it becomes clear that it is no longer a viable option to store each state-value or state-action-value in a table, as the tables for the finite problems simply become too large, and for the ones having a continuous state-space it is by virtue impossible. Now, even if we could store a table for large problems and use table-lookup methods, it would be intractable to learn from this, as MDPs inherently rely on visiting all states multiple times to learn the values of each of these individual states. Instead, we want to be able to generalise values based on the similarity of states, such that two similar states would yield similar values, which is typically the case in many real world problems, generalising seen states to unseen states. This is done through *function approximation*, that estimates the value functions $v_\pi(s)$ and $q_\pi(s, a)$ as:

\[
\hat{v}(s, w) \approx v_\pi(s) \\
\hat{q}(s, a, w) \approx q_\pi(s, a)
\]

where $w$ is some parameter vector of weights, which is much smaller than the state-space for a given problem. The weights $w$ are then updated using learning methods such as MC or TD learning. The idea is then for the model to learn the Agent’s current estimate of its value-function.

There are three types of value function approximations as described by Silver:

1. Input is a state $s$, and output is an approximated value of $s$. Also called state-in.
2. Input is a state-action pair $(s, a)$ and output is an approximated value of $(s, a)$. Also called action in.
3. Input is a state $s$, and output is a vector of approximated values, one for each action available from state $s$. Also called action out.

For prediction problems, the first type is used, whereas for control problems in model-free Environments the other two are both viable. The three types are illustrated in figure 2.9
The specific function can be one of many classical machine-learning functions such as a linear combination of weights, decision trees etc. For my thesis I have chosen to use neural networks as they are well known and have been successfully applied in other board games [Tesauro, 1995, David Silver, 2016, Candocia, 2017, Nicol N. Schraudolph and Sejnowski, 2000]. Two reasons why neural networks work so well is because they are able to capture non-linearity in the input, they are differentiable and therefore lends itself to efficient back-propagation algorithms that can find gradients of the weights \( w \) with regard to the input, which is useful as this directly shows us how we can change \( w \). In addition to this, training an RL Agent produces highly dependent data as the trajectories of states is a Markov chain. Therefore the specific training methods of the function approximator is required to also be able to handle non-iid data.

#### 2.7.1 Training of neural networks in RL

For supervised learning, the pipeline is typically that we have some data consisting of a feature vector \( X \) and a label \( Y \). The network is then trained by using some error-measure (also known as loss function) such as Mean Squared Error (MSE) for regression problems or Cross-Entropy for classification. This loss function is used to measure a difference \( \delta \) between the output of the network and the label \( Y \). The gradient is then found of the loss function with regard to the input \( X \) using back-propagation and finally the weights \( w \) are changed so as to lessen the difference \( \delta \) using some method of gradient descent [Silver, 2015].

If we were to use the same pipeline for Reinforcement Learning for the prediction problem (the same would go for the estimation of \( Q(s,a) \)), the Mean Squared Error would look like:

\[
J(w) = \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, w))^2]
\]

we can then find the gradient of \( J(w) \) via back-propagation and in turn
perform gradient descent as one would normally. The difference in $w$ might then look like:

$$\Delta w = \alpha (v_\pi(S) - \hat{v}(S,w)) \nabla_w J(w)$$

, with $\alpha$ being a learning rate parameter.

The problem of doing this is that it requires an oracle $v_\pi(S)$ which returns the true value of the value-function $v$ under policy $\pi$ that we wish our network $\hat{v}(S,w)$ to learn. Of course, if we were to have $v_\pi(S)$, then we would not need a function approximator and the whole problem of large state-spaces is avoided.

The way to get around the need for the oracle, a target for $v_\pi(s)$ is used. This target takes different forms depending on the learning algorithm used. For Monte Carlo learning, the target is the return $G_t$:

$$\Delta w = \alpha (G_t - \hat{v}(S,w)) \nabla_w J(w) \quad (2.32)$$

For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1},w)$:

$$\Delta w = \alpha (R_{t+1} + \gamma \hat{v}(S_{t+1},w) - \hat{v}(S,w)) \nabla_w J(w) \quad (2.33)$$

and for TD($\lambda$), the target is the $\lambda$-return $G^\lambda_t$:

$$\Delta w = \alpha (G^\lambda_t - \hat{v}(S,w)) \nabla_w J(w) \quad (2.34)$$

Armed with this new target for a given type of learning it is then possible to complete the pipeline without the need of an oracle. This is done by running trajectories and using each step involved as a data point. These can then be supplied, as in a typical supervised learning setting, along with the 'label' that is the targets shown in equation 2.32-2.34. The network is updated after each action involved, as the updates happen incrementally along the path of the trajectory but conceptually, the data set looks like a series of pairs containing the states and their targets, e.g. for TD($\lambda$) it would look like: $\langle S_1, G^\lambda_1 \rangle, \langle S_2, G^\lambda_2 \rangle,...,\langle S_{T-1}, G^\lambda_{T-1} \rangle$. The problem of the forward-view of TD($\lambda$) mentioned in section 2.5 is still present for these cases, and as such, for any practical case, have to be converted into a backwards-view method using eligibility traces. In that case, the update to TD($\lambda$) resembles that of equation 2.30 but using function approximation instead as follow:

$$\Delta w = \alpha \left( R_{t+1} + \gamma \hat{v}(S_{t+1}) - \hat{v}(S,w) \right) z \quad (2.35)$$

, where $z$ is the eligibility trace of the weights $w$:

$$z = \gamma \lambda z + \nabla_w J(w). \quad (2.36)$$

The last problem is then that of feature representation. As a function approximator has to be given a feature vector $X$, the outcome of the learning process is then very much dependent on this feature encoding $X(S)$ of a state $S$. 
2.7.2 Feature Representation

For many problems within RL, the feature representation of a state $S$ is something to be considered very carefully. How one represents the Environment that the Agent acts within determines to a big degree how well it performs. Take the AlphaGo Agents as an example of this, where the stronger Agent had a much smaller feature vector than that of the older version.

There are different types of encodings. One is the raw-encoding which is all the features for a given problem, for instance in Tic-Tac-Toe the raw representation could be a vector of size 9, with elements indicating whether a slot was free, occupied by a cross or occupied by a nought. Another type of features is the hand-crafted ones, where domain knowledge experts dictate what they think are important such as potential threats, value gain of actions etc [Thor Bagge, 2016, David Silver, 2016, Tesauro, 1995]. For instance how many stones can be captured (and as a result how many points can be gained) in the game of go with a certain move.

Unfortunately, the conclusion is that there is no clear indication of which approach is the best when it comes to feature encoding, as there have been successful research using different types of encodings for various problems.

2.7.3 Stability issues

Reinforcement Learning comes with its own set of problems regarding non-stationary targets, as the policies are updated. Combining that with value-function approximation and it becomes a recipe for instability. The fact that we are no longer performing updates on the Agent’s estimated value-function but a model that attempts to learn the Agent’s estimate of the value-function means that the effect of basing the target "on a guess", as was shown in section 2.4.2, then becomes basing the target 'on a guess of a guess'. Furthermore, using the same parameters $w$ for an approximation to both estimate the current state-value (or action-value) and as the target means that once the $\delta$-error has been computed and the model has been updated, the weights have changed and as such the target is also affected, which can lead to a spiralling effect of model updates, where the weights can grow larger and larger, resulting in a breakdown of the model as overflows might happen [Volodymyr Mnih, 2013, Sutton and Barto, 2017]. Fortunately, there are some practical tricks one may apply to help stabilise the models but there are no proofs and therefore no guarantee that they will work. In section 3.4 I will go into detail with how to combat these instability issues through practical means.

2.8 Summary of Part 2

In this part 2 of the thesis, I have described what the key components of Reinforcement Learning are, namely the Agent, Environment and the Reward signals. It was shown how the Environment can be made concrete through states and how the Agent navigates through the Environment via actions in section 2.1.
As the learning methods are modelled using Markov Decision Processes, section 2.2 described the fundamental requirements that a problem has to fulfil to become an MDP, followed by how MDPs help in finding a solution to Reinforcement Learning in section 2.3.

The general pipeline of learning was illustrated for the case where dynamic programming could be utilised to find a solution in section 2.3, using the underlying idea of General Policy Iteration and how it iteratively uses Policy Evaluation and Policy Improvement to converge towards an optimal solution to the Control and Prediction problems illustrated in section 2.4. During this section, three methods were described, namely MC, TD(0) and TD(λ), which, using different targets, attempts to approximate the optimal solution through sample trajectories. Eligibility traces were shown in section 2.5 which helped convert the forward-view issues of TD(λ) to that of backwards-view, making it possible to implement in a practical sense.

As Agents inherently attempt to search the state-space of a problem for the correct course of actions, exploration is essential, and as such is described in section 2.6.

Finally, the extension of tabular learning to that of value-function approximations is shown in 2.7, showcasing both the strengths and problems associated with doing so.
Chapter 3

Reinforcement Learning in Practice

For this part 3 of the thesis, I shall explain how Quarto is made into a Reinforcement Learning problem, and illustrate each consideration that was made with regards to the theory described in part 2.

3.1 Quarto as a Reinforcement Learning problem

Quarto is a strategic board game in the same way that Go, Chess, Backgammon and even Crosses-and-Noughts are. Every move has a consequence and finding a good move is often much harder than finding a wrong move. It is therefore paramount that one considers his or her moves carefully before settling on how to proceed. For a short introduction to the game, see section 1.1. To model Quarto as a Reinforcement Learning problem, the first task at hand is to find a way to convert it into an MDP. Fortunately, as with most board games, the way the rules are set up, this can be done relatively naively. Each game configuration can be modelled independently of each other, with different actions leading to the same configuration being independent. In other words, each configuration represents the trajectory of actions taken so far completely without the need to store the history. The only thing that the Agent needs to be presented is the current game configuration. Furthermore, the game has a finite amount of configurations, making it possible to model it as a finite MDP.

The Second thing that needs to be done is to determine what is part of the Agent and what is part of the Environment. Again the game can be split up quite naively. The Agent should be one of the players, whose actions will be to pick pieces to give the opponent, and to place a specific piece. The Environment should be the specific game configurations that the Agent can act upon. To clarify, a game configuration consists of two components: The board configuration, which entails what cells of the board are occupied by pieces and then the second component, which is the piece pool, storing the remaining pieces. The longer a game lasts, the more cells will be occupied, while the piece pool will be reduced.

The Third thing to be considered is how one should construct the reward
function. As the Agent acts in a designed Environment, we are in full control over the reward signal. As mentioned in section 2.1, the reward signal determines how the Agent will learn to act. Therefore I decided to design the reward function such that for any non-terminal state, the reward is 0, and for terminal states the reward is 1 if a winning configuration has occurred, and 0 for a loss or draw configuration. This reward signal is inspired by [Tesauro, 1995], where they used the same approach. Rewarding draws and losses the same is done in the hope that the Agent will attempt to go for winning moves. Attempts were also made to differentiate between losses and draws by rewarding losses with -1 and draws with 0, keeping the victory of 1. Unfortunately, I did not have much luck using this reward scheme as will be explained during section 3.3. For now, the reward function stays 1 for winning configurations and 0 for all non-winning configurations, and I shall use this for further analysis in this thesis.

Now that the fundamental aspects of the game has been modelled in such a way that it can be presented as a finite MDP, it is time to look at how the states of the Environment should be represented, and how the actions should be done.

Since a state is a snapshot of the Environment in a given situation, I found that it made sense to model the board configuration as a 4x4 matrix. Each cell in the matrix corresponds to a position on the game board. The values in the cell should then represent the individual pieces. To do this, I encoded them as a 4-bit vector, with the bits representing the attributes. The bit-vector then looks like: [Height, Color, Shape, Indentation]. Each attribute takes a Boolean, as a piece is either Tall or Short, Black or White, Square or Round, Indented or Filled. An example encoding could be [0,1,0,0], which would be a Short, Black, Round and Filled piece. To avoid storing a bit-vector per cell they were converted into integers such that a piece would be represented using the following encoding function $e(p)$, with $p$ being the bit-vector and $p_i$ being the i’th bit of the bit-vector:

$$e(p) = 1^{p_0} + 2^{p_1} + 4^{p_2} + 8^{p_3} + 1 \quad (3.1)$$

In this way $e \in [1,16]$, and will represent each piece uniquely by a singular value. Furthermore, a cell in the matrix can also be 0, representing no piece occupancy. Following this approach, each cell of the 4x4 matrix can hold one of 17 values, of which 16 are unique. Now, a state should also represent the piece pool. As there are 16 pieces in the piece pool, it was stored as a bit-vector with 16 entries, one for each piece. A mapping function then maps each unique piece to the piece pool vector by taking the e-encoding of a piece, and use that value - 1 as the indexing into the piece pool vector. To find the index of a piece into the piece pool is then: $\text{index} = e(p) - 1$ as it is zero-indexed. The piece Pool contains a Boolean per piece, representing whether a specific piece is in the Piece Pool. Once taken, the bit for that specific piece is set to 0, meaning in the beginning of the game, the Piece Pool will be filled with 1s, and at the end of the game there will be less 1s, and zero 1s in case of a draw. Due to
the framework used (Pytorch), I opted to represent the Piece Pool bit-vector as a 4x4 matrix as well to keep the dimensions the same as for the board state. Indexing is done almost identically so I shall not go into further details with how this is done. It is enough to think of the Piece Pool as a bit-vector storing whether a given piece is in or out of the pool. Lastly, I also included a vector, like the Piece Pool, except this vector should represent which piece was handed to the player by the opponent. As such, only one entry into this vector was set to 1, and all else set to 0. This was also modelled as a matrix to retain dimensions, but as with the piece pool, thinking of it as a vector simplifies the concept. To recap, a state \( s \) is then represented by the following:

- A 4x4 matrix, with values \( v \in [0, 16] \). This matrix keeps track of the positions of pieces on the game board.
- A 16-bit vector, storing which piece is in the Piece Pool based on the encoding function \( e \) of a piece \( p \).
- A 16-bit vector, storing which piece has been picked by the opponent. This piece is the piece that needs to be placed by the acting player. This stores 1 for the entry found by the encoding function \( e \), and 0 for the rest.

Now that the states have been defined, it is time to define what an action is. Since every algorithm that I have found in the literature selects actions that perform a singular action per time step \( t \), the actions have to somehow reflect this. The reason why this is a problem is because for each state represented to the Agent, two actions have to be taken. The Agent has to choose where to place the piece it is given, and it has to select a piece from the piece pool to give to the opposing player. I found that encoding multiple actions as one action could be done by taking each combination of actions, and represent them as one. In this way, an action becomes a tuple of (placement index, piece index), where the first entry holds the index of the cell that the Agent wants to place the piece it is given, and the second holds the index of the piece that it gives the opponent. This approach is motivated by the ones used in [Nishiyama, 2016, Marcelo J. Karanik]. This way of representing multiple actions as one, unfortunately does not scale well, but it is not a big issue for this problem as the amount of actions per game state does not grow too large. As can be seen in figure 3.1 below.
The reason why the beginning of the game has 16 and the next move has 240 is that in the beginning a player only has to pick what piece to give and not consider where a piece has to be placed. In the other end of the graph, only 1 action remains, and that is to place the piece given without considering which piece to give, as there are no more pieces in the Piece Pool. During the flow of a game, a player then has to consider at most the sum of these moves, which is 1377. Of course, this does not take into account the act of "reading ahead", but merely the combination of possible moves throughout the game if a player was to only consider immediate moves. If we allowed a player to "read ahead", that is, perform virtual moves to see what the result might become from a specific action, the action space would explode exponentially in the amount of steps that we wish to read. For games it is typical to find a solution to them using search-trees, which does exactly this but introducing search based methods is outside of the scope of this thesis. Using a single step look-ahead, a max evaluation of 1377 actions is not bad in a practical manner in the greater scheme of things. Although the action-space is manageable throughout a game, the state-space should not be underestimated. The state-space for Quarto is $16^{12} \approx 4.378 \cdot 10^{26}$, which by all means is impossible to store or solve.

### 3.2 Reduction of State-Space

Now that everything is setup and ready to be plugged into a Reinforcement Learning algorithm, one might consider if there are ways to reduce the state-space of a given problem, as the fewer states the Agent has to look through during learning, the quicker it converges towards an optimal policy $\pi_*$. For Quarto there are two things that can be done, one more obvious than the other. The first reduction is to find symmetrical board positions, which is shown in Anuj Mahajan [2017], whereas the second is to model the actions as after-states instead.
Symmetries is finding states that are represented differently, but result in the same outcome. In Quarto, a symmetry is, for instance, the mirroring of a board. A winning position in the original board is also a winning position in the mirrored position. Further more, the board can be flipped along the horizontal axis, meaning the bottom row becomes the top, and the middle rows are exchanged. In the same manner, a flipped board will result in exactly the same way that the original board would. Lastly, a combination can also be found, such that a flipped, mirrored board is the same as the original board. An example of these symmetries can be seen in figure 3.2. Using this, the state-space is effectively reduced to $\frac{1}{4}$ of the original problem. Of course, while this sounds like a lot, I will remind the reader that the state-space was in the vicinity of $10^{26}$. While the reduction is substantial, it is by far still too large to store in memory.

Figure 3.2: TOP LEFT is the original board matrix. TOP RIGHT is the flipped version. BOTTOM LEFT is the mirrored version. BOTTOM RIGHT is the flipped and mirrored version

By keeping track of where the first move was placed, the game can be flipped and mirrored so as to always keep the same orientation. For instance, we could say that no matter the placement of the first move on the real board, we can force a representation of the state to have this move be placed in the top-left quarter of the matrix. To do this, a move in the lower left quarter will result in a flipped board, a move in the top right quarter would result in a mirrored board, and finally, a move in the lower right quarter would result in a flipped and mirrored board. Knowing which quarter the first move was placed, means we can find the correct symmetry for the rest of the moves by simply doing the same to all successive moves.

The second approach to reducing the state-space is to use after-states as in [Thor Bagge, 2016, Sutton and Barto, 2017]. An after-state is the state after an action has been performed. To explain what this means, I’ll remind the reader that for the Control problem, as defined in section 2.4, for model-free methods, we estimate the $Q(s,a)$ value function, and select actions based on greedy choices once training is complete, i.e. we find the action with the highest
value for a given state: $\arg\max_a Q(s, a)$, for all $a \in \mathcal{A}(s)$. As the problem at hand is a board game, and it is being designed completely, we do not have to rely on model-free methods. Because we are able to simulate the result of each action at any given state, we can utilise this to find a solution to the Control problem through a solution to the Prediction problem. In other words, we can look at each action for a given state $s$, and see which state $s'$ gave the highest predicted score, without needing to know the scores of the individual actions. Now, transforming a Control problem into a Prediction problem is by itself not enough to reduce the state-space, but by doing so for Quarto it will. To illustrate this, the following example should help to explain it.

Figure 3.3: After-state as a result of actions $s_1$ and $s_2$, leading to the same state $s'$. The values in each cell corresponds to the pieces occupying them. Empty cells are represented as empty instead of 0.

If we were to use the Q-value estimation, the action from figure 3.4 (A) would be approximated differently than that of (B), even though they would lead to the same state (C). If instead we look at the after-states of actions, and attempt to find a solution to the Control problem through Prediction, then both (A) and (B) with an action leading to state (C) would find the same value, as it is the state-value that is being used now, instead of the action-value.

To summarise, using both symmetries and after-states I have reduced the state-space of the Quarto problem. There might exist further reduction techniques but the chosen methods seemed to illustrate the point and as such didn’t feel necessary to continue further investigation into.
3.3 Learning Algorithms

The next step is to implement the learning algorithms so that the Agent can learn to play Quarto. To do this, I chose two different approaches but both with value-function approximation, as the state-space, while reduced, is still way too large to fit into memory in any tabular fashion. Firstly, utilising after-states I attempted to find a solution to the Prediction problem using TD(\(\lambda\)) as I wanted to see the effect of different values of \(\lambda\) ranging from 0 to 1, making it function equally to that of both TD(0) as described in section 2.4.2 and MC learning as described in section 2.4.1. To understand why this is interesting, see section 2.4.3. As I wanted to avoid the hassle of forward-view dependencies, I used eligibility traces as described in section 2.5, so that I only had to perform back-wards view updates. For the specific algorithm I used the Semi-gradient TD(\(\lambda\)) method for estimating \(\hat{v} \approx v_\pi\), which is illustrated by Sutton and Barto in the pseudo code for the algorithm as follows:

<table>
<thead>
<tr>
<th>Semi-gradient TD((\lambda)) for estimating (\hat{v} \approx v_\pi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: the policy (\pi) to be evaluated</td>
</tr>
<tr>
<td>Input: a differentiable function (\hat{v}: S^+ \times \mathbb{R}^d \rightarrow \mathbb{R}) such that (\hat{v}(\text{terminal}, \cdot) = 0)</td>
</tr>
<tr>
<td>Initialize value-function weights (w) arbitrarily (e.g., (w = 0))</td>
</tr>
<tr>
<td>Repeat (for each episode):</td>
</tr>
<tr>
<td>Initialize (S)</td>
</tr>
<tr>
<td>(z \leftarrow 0) (a (d)-dimensional vector)</td>
</tr>
<tr>
<td>Repeat (for each step of episode):</td>
</tr>
<tr>
<td>. Choose (A \sim \pi(\cdot</td>
</tr>
<tr>
<td>. Take action (A), observe (R, S')</td>
</tr>
<tr>
<td>. (z \leftarrow \gamma \lambda z + \nabla \hat{v}(S,w))</td>
</tr>
<tr>
<td>. (\delta \leftarrow R + \gamma \hat{v}(S',w) - \hat{v}(S,w))</td>
</tr>
<tr>
<td>. (w \leftarrow w + \alpha \delta z)</td>
</tr>
<tr>
<td>. (S \leftarrow S')</td>
</tr>
<tr>
<td>until (S') is terminal</td>
</tr>
</tbody>
</table>

Figure 3.4: Pseudo code for Semi-gradient TD(\(\lambda\)) as stated in Sutton and Barto [2017].

What this algorithm does, is that it finds the gradient of value-function \(\hat{v}(s, w)\) with respect to the input state \(s\) (the three matrices described in section 3.1). This gradient is used as the weighting for the weights involved in the approximation model. \(z\) is then updated so that the eligibility traces of the weights of the model is correctly decayed. The line \(z \leftarrow \gamma \lambda z + \nabla \hat{v}(S,w)\) is the update to the eligibility traces as in equation 2.35. The remaining part of the pseudo code is then just as with regular TD(\(\lambda\)) using approximation with eligibility traces, where the the update happens on the weights of the approximation model as described in 2.7. One detail that troubling me for some time is the rule in line: 2 that states: \(\hat{v}(\text{terminal}, \cdot) = 0\). This means, that when performing the one-step look-ahead, the TD-error \(\delta\) should be only \(R - \hat{v}(S,w)\) when at a position where a move bringing the Agent into a terminal state is found, as the \(\gamma \hat{v}(S',w)\) term is reduced to 0. Forgetting this leads to annoying problems.
The second learning algorithm I implemented was the DQN algorithm from Deepmind, which builds upon the Q-learning algorithm shown in section 2.4.2. This algorithm works by updating the approximation-function instead of a tabular value-function. In DQN the Agent produces samples for the training part to use, and stores them in something called a Replay Memory. Training can then run in parallel to that of the sample generation as samples are drawn from the Replay Memory and not directly from the Agent’s actions. Furthermore, it applies some methods to deal with the stability issues mentioned in section 2.7.3 which I will get into detail with in the next section, section 3.4. The update rule is very similar to Q-learning in the tabular case, which can be seen in equation 2.20 but to remind the reader, it is simply updated using the TD-error between the one-step look-ahead and the current state’s action-value: \[ \delta = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s, a_t). \] The differences are, that the TD-target value is substituted with one function-approximator \( \hat{Q}(s_{t+1}, a') \), while the current action-value with another function-approximator: \( Q(s_t, a_t) \), along with the update rule, which is now a gradient descent based method on the TD-error \( \delta \). The pseudo code for DQN is shown below but as the problem they attempted to tackle was playing video games, they needed to encode preprocessed sequences of game-screens. Not all processes are needed for the Quarto problem, namely the preprocessing and emulation steps:

```
For episode = 1, M do
  Initialize sequence \( s_1 = \{ x_1 \} \) and preprocessed sequence \( \phi_1 = \phi(s_1) \)
  For \( t = 1, T \) do
    With probability \( \epsilon \) select a random action \( a_t \)
    otherwise select \( a_t = \arg \max_a Q(\phi(s_t), a; \theta) \)
    Execute action \( a_t \) in emulator and observe reward \( r_t \) and image \( x_{t+1} \)
    Set \( s_{t+1} = s_t, a_t, x_{t+1} \) and preprocess \( \phi_{t+1} = \phi(s_{t+1}) \)
    Store transition \((\phi_t, a_t, r_t, \phi_{t+1})\) in \( D \)
    Sample random minibatch of transitions \((\phi_j, a_j, r_j, \phi_{j+1})\) from \( D \)
    Set \( y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases} \)
    Perform a gradient descent step on \( (y_j - Q(\phi_j, a_j; \theta))^2 \) with respect to the network parameters \( \theta \)
    Every \( C \) steps reset \( \hat{Q} = Q \)
  End For
End For
```

Figure 3.5: Pseudo code for DQN as introduced in [Volodymyr Mnih, 2013]. The problem handled by DQN was that of playing Atari games, introducing a few extra steps unnecessary for Quarto.

Because DQN use decoupling between sample generation and the approximation-function update, it made it stand out compared to that of classical Q-learning algorithms. As the two processes can run in parallel, the experience gained is
better utilised, as there is a probability of the same experience used multiple times when the sample is drawn during training. As a consequence of the parallel aspect of DQN, I wanted to try and see if it would perform better than the TD(\(\lambda\)) approach, even though it only performs one-step look-ahead learning as it is a TD(0) algorithm, and it approximates a Q-value function, meaning it does not utilise the after-states. To see why these points might be a hindrance towards learning, look at section 2.4.3 and 3.2 respectively.

3.4 Function Approximation in a practical sense

With all large problems, value-function approximation must be used as mentioned in section 2.7. The model I chose for approximation was neural networks as they have proven themselves useful in many cases such as [Tesauro, 1995, David Silver, 2016, Candocia, 2017, Nicol N. Schraudolph and Sejnowski, 2000]. The architectures of the networks I made depended on what learning algorithm was implemented. The TD(\(\lambda\)) algorithm used Sigmoid activation functions across all layers inspired by [Tesauro, 1995]. This would mean that for a given state, to handle the Control problem, the Agent would have to do an amount of forward passes equal to the size of the action-space of the current state (for a concrete amount, see fig 3.1). For the DQN algorithm I used Sigmoid activation functions across all layers except the last layer. The last layer was a LogSoftmax layer, with 287 logits, representing each possible combination of the (placement index, piece index) action pair as I wanted it to be an action-out function as described in section 2.7 and illustrated in figure 2.9. This would make it output a distribution over the actions available at a given state \(s\), meaning it would require only a single forward pass, regardless of the state it is in. A mapping from the output to a specific actions was then required, where the action with the highest value was chosen \(\epsilon\)-greedily. Furthermore, to prevent invalid actions from being selected, the logits representing invalid moves were masked out before being fed into the LogSoftmax layer. This approach was inspired by the policy-net of AlphaGo in [David Silver, 2016].

For both algorithms I used Dropout layers as a regularisation method. I also tried using different activation functions for both the TD(\(\lambda\)) Agent and the DQN Agent without luck as these resulted in very unstable networks, and training could not complete before they broke down. By breaking down, what is meant is that the weights of the networks escalated to such a degree that the output approached infinity or 0, regardless of the input, often resulting in overflow. The reason why I wanted to use other activation functions was that ReLU has been shown to converge quicker than Sigmoid [Alex Krizhevsky, 2012], while also not suffering from the vanishing gradient problem to the same extend. Leaky RelUs were also attempted as they are supposed to deal with vanishing gradients in an even better manner but these also seemed to break down the network. Finally, as mentioned in section 3.1, I would have liked to play around with other reward functions and as such changed the output layer to a tanh activation function instead of Sigmoid, putting the output of the network into the range \([-1, 1]\). Using tanh I would then be able to punish
the Agent more for losing. Tanh has been very successful in many cases, such as for the value-net of AlphaGo [David Silver, 2016] and seems to be preferred over Sigmoid in the literature. Unfortunately, tanh also broke down the nets, as all outputs became either -1 or 1, with no value in between. For some reason the networks stabilised when using a purely Sigmoidal neural network. Lastly, I attempted different initialisation values for the networks based on their activation functions and found that using the xavier_uniform initialisation, as described in [Xavier Glorot, 2010], worked the best. What caused the instability of any other type of network than Sigmoidal ones, I am unable to say but below are some concerns regarding instability and tools that typically help in stabilising them. I chalk it up to the fact that value-function approximation within Reinforcement Learning is inherently unstable.

As TD-based methods are not stable for neither the Prediction- nor the Control problem, some effort to stabilising them goes a long way. Therefore one must try to apply as many tricks as possible to stabilise learning. One such trick that Deep Mind used to reduce the effect of the non-stationary target problem, was to fix the target in a sense. To accomplish this, two networks were used; one to calculate the TD-target and one to evaluate the current state and action chosen. The former had its weights frozen, meaning no updates came to them, whereas the latter was updating its weights based on gradient descent methods towards the TD-error between the TD-target and the current (state-action) pair value. Then, after some arbitrary C steps, the frozen network’s parameters updated to be that of the active network. In this way, the target is still non-stationary, but its effect has been dampened as we do not directly affect the TD-target whenever we update the network evaluating the current action-value. The trick of freezing one approximator and use it for the TD-target and another for the action-value of the current state and action is called a Fixed-network approach. This method I implemented for both the TD(\(\lambda\)) and the DQN Agents.

Another trick to stabilising the networks is to decouple the training from the sequence of experiences. What is meant by that, is that for regular TD-learning, the learning aspect is very sequential, resulting in one state being directly affected by the next state and as such is highly correlated to one another. By sampling randomly from the Experience Replay memory, this correlated aspect is broken, as the TD-error is based on individual targets and we now update based on many targets that might not at all be near each other. Unfortunately, for TD(\(\lambda\)) this is impossible, as it is dependent on the sequence of events due to the eligibility traces. As powerful as these traces are, they prevent us from stabilising the network in this manner.

Now, even if the latter stabilisation technique described above is not possible for TD(\(\lambda\)), there are success stories, such as the TD-gammon Agent, using this learning algorithm, even without any stabilisation at all. The networks might become unstable but it is not a certainty.

For the time that I have been attempting to use neural networks as value-function approximators, I have experienced three types of destabilisation events:

1. The first type is of values exploding, such that they became too large to
bear any meaning and in some cases resulting in overflow problems. A sort of cascading effect happened, where the estimated values grew larger and larger, as the weights in the networks grew larger and larger.

2. The second type is of values imploding, such that they became infinitesimally small and ended up being rounded down to 0, as weights approached 0.

3. The third type of error was that of vanishing gradients. If the gradients become too small, no updates will be done, regardless of the $\alpha$ parameter.

In the end, I found that figuring out how to structure the neural networks became one of the most time consuming aspects of this thesis, as so many ideas which I thought had good reasons to work simply did not.

### 3.5 Exploration in Board games

As described in section 2.6, exploration has to happen. Therefore every policy used has been an $\epsilon$-greedy policy, initialised at 0.3 with a decay factor of 0.9999 multiplied per game, to a minimum of 0.1. Tesauro states that for deterministic environment such as games without any stochastic elements, simply using $\epsilon$-exploration is not enough, and suggests introducing more stochasticity into the states that the Agent is presented. As such, each episodes (or game) is initialised by a series of random moves, from 0 to 15 moves, from which the game is to be played out. Training the Agent using this method guarantees that not only does the Agent sometimes perform non-greedy actions, it is also shown board positions that it would otherwise never see, increasing the amount of unique states that it has seen.

### 3.6 Training of a Board game playing Agent

The final part to be considered is then how these games should be played, how the training dataset should be produced. Quarto is a game between 2 players alternating turns, so the Agent should only act every second turn. Therefore, training is structured so that the starting player is randomised to remove bias towards going first or second. The Agent will then, on its turn, perform an action based on an $\epsilon$-greedy choice of the action-space for a given state. If the resulting state from performing said action is a terminal state, the episode is terminated and a new episode / game is initialised. To play the moves that the Agent does not, an opposing player is added to the learning process. This player can take many forms, such as being a random agent, performing completely random legal actions regardless of the state presented. Another type of player could be an expert player, such as an expert Human player, while yet another type could be the Agent itself, where the actions performed would be completely greedy with regards to its policy. The last method is also called self-play and is the one used in [Tesauro, 1995, David Silver, 2016, 2017, David Silver]
and Hassabis, 2018]. Inspired by their successes, I have chosen to rely on self-play for my thesis. One concern regarding self-play is that an Agent learns to defeat itself and therefore might end up stuck in a local maxima. Fortunately, the random initialisation mentioned in previous section 3.5 alleviates this to some extend, as the states are more varied. Another approach used in this thesis is Agent History, so that the Agent does not play a current version of itself but a version, uniformly randomly chosen from the history as described in David Silver [2016].

3.7 Summary of Part 3

In this part 3 of the thesis I have shown how Quarto can be transformed into a Reinforcement Learning problem, how the state-space can be reduced in a practical sense and illustrated different learning methods, such as TD(\(\lambda\)) and DQN, using value-function approximations. In section 3.1 it is shown how exactly I modelled Quarto as a Reinforcement Learning problem, where the following sections 3.2, 3.3 and 3.4 dealt with the practical matters of implementation. Lastly, section 3.5 and 3.6 described techniques to ensure a high enough degree of exploration during training of Agents in deterministic settings such as board-games with no stochastic elements.
Chapter 4

Experiments

4.1 Experimentational setup

Before getting into the experimentation, I shall give a brief description of the experimental setup. Firstly, the hardware followed by the software will be presented and finally how the experiments was conducted.

I ran the training of Agents on two different computers, one being a server at the Computer Science department, with a high-end GPU and the other being my own standard-issue laptop. For the server, it had the following specs:

- CPU: Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz.
- GPU: Nvidia GV100 TITAN V
- Memory: 374GB was free to use, Hardware is missing.

My own computer has the following specs:

- CPU: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
- GPU: Intel(R) HD Graphics 520
- Memory: 8GB DDR3

All the experiments run on the server were utilising the GPU, whereas the ones running on my laptop used the CPU only.

The algorithms and testing were implemented using the following software:

- Python v 3.6 standard library.
- Pytorch v. 1.0.1 CPU for the Tensor calculus and modeling of the neural networks on the CPU Pytorch.
- Pytorch v. 1.0.1 GPU for the Tensor calculus and modeling of the neural networks on the GPU Pytorch.
- Conda v. 4.6.11 to keep track of python environments and handle package installation Anaconda.

Each Agent trained played a total of 100,000 games, with one Agent per 50 games being stored, resulting in a total of 2000 Agents per training session. For variant testing, the Agents were made completely deterministic, meaning their policies were completely greedy, setting $\epsilon = 0$. Two sets, $A$ and $B$ of tests were then done, $A$ starting from a blank board state and $B$ starting from 20 randomised initial states, as described in section 3.5. For the first set, only two tests are necessary, as the Agents are completely deterministic, meaning depending on who goes first, all games will be identical. This type of testing is meant to find the best overall agent playing in real scenarios, whereas the second set is meant to see which agent has the best understanding of the game. For the second set, each Agent gets to be player 1 50% of the time. When testing the effect of various hyper-parameters, only the tested parameters changes, while all others remain unchanged. The tests are therefore done to see if there is a measurable difference, but does not identify which is the optimal parameters, as a grid-search would have to be performed to do so.

4.2 Experiments

When testing variants, an Agent has to be chosen as its representative. Instead of just selecting the one that trained the longest, I greedily selected one Agent among the variants being tested. This is done by selecting the Agent with the highest win ratio in a tournament setting, as described below.

For both sets $A$ and $B$, finding a representative through testing every combination of Agents is infeasible, as for just one variant the combination of agents would be $(2000 \times 2001)/2 - 2000 = 20008000$, with each game lasting about a second, resulting in approximately 231 days of experimentation, and that is for the deterministic case only. As this is completely unrealistic, and would be far more time consuming than the training of the Agents themselves, a tournament setting was used to find the representatives, running over 2 rounds. For the initialisation of the tournament, one Agent in each training pool would randomly be selected, with each training pool consisting of 20 Agents, to represent that training pool. The pools were made such that the first pool contained the Agents 1-20, the second pool contained Agents 21-40, etc. This reduces the Agents participating in the tournament to 100. As the best agent is already being determined based on one greedy result per variant, no guarantee is made that it is indeed the best representation of the variant compared to other variants, therefore finding the representation through this tournament does not seem to change much, as long as no claim of optimality is being made. The first round of the tournament consisted of brackets containing 20 players each, resulting in a victor from each bracket with ties broken arbitrarily. The next round was then between the victors of the first round’s brackets, meaning there are $100/20 = 5$ players in the second round. These 5 players will then play each other, after which the winner is declared and will represent the variant that is being tested for the specific testing set ($A$ or $B$). In this way, the hope is
that a representative is found, reflecting the strongest Agent for a given variant instead of selecting one at random or merely using the one trained the longest. When testing variants using this scheme, the overall testing time is reduced to

\[2 \cdot 5 \cdot ((20 \cdot 21)/2 - 20) + 2 \cdot ((5 \cdot 6)/2 - 5) = 1920\] games played in total for the games played in set A and

\[8 \cdot 5 \cdot ((20 \cdot 21)/2 - 20) + 40 \cdot ((5 \cdot 6)/2 - 5) = 7600 + 400 = 8000\] games played in total for set B, which is a big enough reduction making variant testing possible, although still costly.

### 4.2.1 Experiment 1: Finding Variant Representatives

Using the above scheme for finding representatives for test sets A and B, one Agent per variant has been found. As I have tested 21 different TD(\(\lambda\)) variants and 14 DQN variants, one entry in the table below shows the Agent index representing a given variant. Each Agent index can be multiplied by 2000 to get the amount of games trained. The specific variants are not important for this experiment, as this is used to illustrate whether training made any difference and to illustrate when training peaked.

![Figure 4.1](image)

**Figure 4.1**: Winners of the tournament settings under Test Set A and B, with Agent indices showing for TD(\(\lambda\)) variants.

![Figure 4.2](image)

**Figure 4.2**: Winners of the tournament settings under Test Set A and B, with Agent indices showing for DQN variants.

Figure 4.1 and 4.2 show that training for a longer period might not necessarily find the best Agents. Interestingly, the last variant tested for TD(\(\lambda\)) seemingly peaked after 5 Agent snapshots (250 games played). Another interesting observation is that some Agents performed best in both test set A and B, increasing the confidence that they are indeed the best representative of that variant. To see more clearly the relationship between training time and performance, fig 4.3 shows that for DQN, there was a tendency for Agents that have trained longer to also perform better, whereas for TD(\(\lambda\)) the winners tended to
be the ones trained using about half of the time.

Figure 4.3: Histogram with buckets spanning 400 Agent snapshots. The buckets represent the amount of Agents winning the tournament finals for a specific snapshot range. LEFT histogram shows that for DQN, the prevalent range of finalists lies within the range of 1600-2000 (80-100k training games), whereas RIGHT histogram shows that for TD(\(\lambda\)), the prevalent range of finalists lies within the range of 800-1200 (40-60k training games).

To further illustrate the confidence in the individual Agents selected, their win ratios within the finals and the difference between the victor and the poorest performing Agents are illustrated in the following figures 4.4 and 4.5. The reason why the poorest performing Agent was chosen to measure the difference, is to showcase the biggest difference between finalists.

### (A) Win-ratio of the DQN variants.

<table>
<thead>
<tr>
<th></th>
<th>Test Set A</th>
<th>Test Set B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>0.5</td>
<td>0.6</td>
<td>0.7</td>
</tr>
</tbody>
</table>

### (B) Difference between highest and lowest win-ratio for DQN variants.

<table>
<thead>
<tr>
<th></th>
<th>Test Set A Difference</th>
<th>Test Set B Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Figure 4.4: Win ratios of each DQN variant tested, with each cell corresponding to the Agent index illustrated in figure 4.2. Figure A shows the win ratio of the chosen Agent, whereas Figure B show the difference between the Agent and the worst performing competitor.
4.2.2 Experiment 2: Performances of Gamma parameter: $\gamma$

To test what the effect of changing the gamma parameter was, with regards to performances, games were played using only the randomised initial starts, as tests of type A would not confer any real meaning to this experiment. A tournament setting was also used for evaluating performances, with the Agent representatives found above and $\gamma \in \{0.1, 0.4, 0.7, 1\}$ for the DQN Agents, and $\gamma \in \{0.4, 0.7\}$ for the TD($\lambda$) Agents. Values of 0.1 and 1 were skipped for the TD($\lambda$) tests, as they broke the Agent during training. Both algorithms were tested with 40-game rounds, where each competitor acted as player 1 an equal amount of times. During testing, all other hyper-parameters were frozen, such that only the effects of $\gamma$ were measured.

Figure 4.5: Win ratios of each TD($\lambda$) variant tested, with each cell corresponding to the Agent index illustrated in figure 4.1. Figure A shows the win ratio of the chosen Agent, whereas Figure B show the difference between the Agent and the worst performing competitor.
Figure 4.6: this figure shows win ratio distributed over 4 $\gamma$ values for DQN and 2 for TD($\lambda$).

Change in $\gamma$ values did not seem to matter much for DQN, whereas TD($\lambda$) has benefited from having a value of 0.7 over that of 0.4.

4.2.3 Experiment 3: Performances of Lambda parameter: $\lambda$

For the TD($\lambda$) algorithm, the following tests were done to check whether a difference in lambda values result in different performances. Once again, a tournament setting was used with randomised initial board states. Every Agent plays the other Agents for 80 rounds, of which each competitor went first half of the time. During testing, all other hyper-parameters were frozen, such that only the effects of $\lambda$ were measured. Three different types of variants were tested for $\lambda$: 
Figure 4.7: figure shows win ratios distributed over 4 $\lambda$ values for three types of TD($\lambda$) variants.

As seen, a differences can be measured, illustrating that $\lambda$ has an effect on performances.

4.2.4 Experiment 4: Effect of Agent History

To see whether Agent History had an effect, the following tests were done. A tournament setting was used with randomised initial board states. Two variants using Agent history plays two variants without for 80 rounds, of which each competitor went first half of the time. During testing, all other hyperparameters were frozen, such that only the effects of Agent history were tested.

Figure 4.8: this figure shows win ratios with regards to two variants of $\gamma$ and the effect of using Agent History or not.

Results indicate that using an Agent history effects performances.
4.2.5 Experiment 5: Effects of Fix-Q-Networks, Regularisation of Networks, and Input normalisation

The following tests were conducted to see whether affecting the neural networks through fix-Q strategies, regularisation and input normalisation can be measured in performances. As such, the networks were changed every 50 games played and the networks were given a dropout rate of 50%, adding the highest amount of regularisation according to [Pierre Baldi, 2013]. Normalisation was done by dividing each input value by 16, to bring each value into the range of [0, 1]. This should help training neural networks according to [etc, 1997-2002].

4-variants of TD(\(\lambda\)) with regularisation will play the same variants without. Once again, matches lasted 80 rounds, of which each competitor goes first half of the time. During testing, all other hyper-parameters were frozen, such that only the effects of Regularisation and Input normalisation were tested for a specific value of \(\lambda\).

![Win Distribution for Regularisation and Input Normalisation](image1)

![Win Ratio between Variants](image2)

Figure 4.9: LEFT shows win ratio distribution between same type of variants. RIGHT shows win ratio distribution between Agents with Fix-Q-Networks, regularisation and input normalisation and those without, for same values of \(\lambda\).

The left plot of figure 4.9 shows that the Agents using no regularisation favored higher levels of \(\lambda\) whereas regularisation and input normalisation seems to perform better at lower values of \(\lambda\). Furthermore, using regularisation, input normalisation and fixed-networks performed better than that of the variants without. Again, lower levels of \(\lambda\) seems to be consistent with the left part of the figure.

4.2.6 Experiment 6: Best Overall Agent

Based on Experiment 1, an Agent shall be found to represent the degree of success that training has achieved. Therefore a tournament is set up, such that each of the 21 TD(\(\lambda\)) Agents shall play against each other in a tournament setting. As TD(\(\lambda\)) has to look through the action-space of each state, as mentioned in section 3.1, matches are limited to 30 games, as each game takes a while to complete. Each Agent gets to be player 1 half of the time. Figure 4.11 A illustrates the win ratios between the variant representatives, with the top three Agents highlighted in green. Figure 4.11 B shows an index combined with the six highest scored Agents.
Figure 4.10: A shows the win ratios between variants of TD(\(\lambda\)) and DQN, with the highest three highlighted in green. Figure B shows an indexing of the Agents and their respective win ratio.

The Agents of indices in figure 4.11 are as follows:

1. TD(\(\lambda\)) with \(\gamma = 0.7\), \(\lambda = 1\). It uses Fix-Q networks together with 50% Dropout rate and input normalisation.

2. TD(\(\lambda\)) with \(\gamma = 0.7\), \(\lambda = 1\). It uses Fix-Q networks together with 10% Dropout rate and input normalisation.

3. TD(\(\lambda\)) with \(\gamma = 0.4\), \(\lambda = 0\). No stabilisation techniques used.

4. DQN with \(\gamma = 0.7\), Fix-Q networks, Regularisation (Dropout 10%) and input normalisations were used.

5. DQN with \(\gamma = 0.7\), Fix-Q networks, Regularisation (Dropout 50%) and input normalisations were used.

6. DQN with \(\gamma = 0.4\), no stabilisation techniques used.

Following this is a final round to find the one representative of my thesis, in an all against all tournament setting. 100 Games were played, with each player starting as player 1 half of the time. Games were played from randomised initial starting configurations:
4.2.7 Experiment 7: Out of Sample Testing

To test how well the Agent found in Experiment 6 performed against current implementations of Quarto, it was tested against a browser based implementation which used minimax search trees. Unfortunately, the Agent I chose did not win against any of the difficulty settings available. The Agent was also tested against human opponents but could not win a single match. Therefore, no further tests were done in this manner, as the Agent had not achieved any interesting level of play.

4.3 Summary of Part 4

In this part 4 of my thesis I have tested various hyper-parameters to see whether they had an effect on performance. Performance was measured in tournament games, with each Agent participating playing against each other in either the full tournament or in a bracket. A representative was found per variant, based on how well it made it through the tournaments, and finally an Agent was selected to play against outside opponents, such as humans.
Chapter 5

Conclusion

For the most part, the experiments went as expected. I would have liked to spend more time conducting the search for representatives, as currently it relies on a random choice per training pool of twenty Agents, to reduce the amount of Agents playing from 2000 to 100. It is hard to estimate the performances of the Agents, as there is no ground truth. All that can be done is to rely on a biased choice of representatives. I hope to have reduced this bias by introducing the tournament setting but it is hard to know what effect it had. Therefore, the results should not be seen as definitive but rather illustrate whether a difference in performance could be measured with regards to different hyper-parameters.

For Experiment 1, finding a variant representative was done so as to not merely base every further tests on the last iteration of Agents. Interestingly, as shown in figure 4.3, Agents trained through the DQN learning approach seems to favour longer training whereas TD(\(\lambda\)) has the most representatives from half-way through training. This goes against what I expected, as I thought that longer training time would also equal stronger Agents. Of course, strength in this regard is very hard to estimate, as this experiment only found the Agents that performed the best within the variants. Therefore, Agents that have over-fitted towards a local maxima, may have to be pushed down from it (which is done through exploration), after which it needs to climb a new maxima. The fact that TD(\(\lambda\)) may have more Agents within the middle range may be indicating that they have learned some strategy that are particularly efficient at beating bad opponents.

Changing the \(\gamma\) parameter changes how the Agent perceives the target values. For lower values, it believes itself more than the new target, whereas it is more prone to believe the target for higher values. For DQN different values of \(\gamma\) seemingly doesn’t change much, as seen in figure 4.6, while TD(\(\lambda\)) seem to be greatly affected by it. Not only does TD(\(\lambda\)) show a big difference in win-ratio for \(\gamma\) values of 0.7 and 0.4 but increasing it to 1 made the Agent believe that every move was optimal. In other words, after only a few training sessions, the Sigmoid activation function of the output layer, regardless of the input, returned 1. Once this happened, it never changed and it continued to stay at 1 for the rest of the training session. On the other hand, reducing \(\gamma\) to 0.1 had the opposite effect, where output started to decline in value, as the TD-target

61
would be reduced significantly. For some reason, this did not seem to be a problem for the one-step look-ahead used in DQN, leading me to believe that the use of eligibility traces plays a big role, as big values of $\gamma$ would update the entire network constantly, depending on the $\lambda$ parameter.

Even though Sutton and Barto [2017] notes that the effect of $\lambda$ diminishes for problems with short episodes, figure 4.7 seems to illustrate that for two of the three variants tested, $\lambda$ values of 0.7 had a positive effect on performance. Furthermore, that the representative of the thesis was a TD($\lambda$) Agent, with a $\lambda$ value of 1, seems to further strengthen the idea that looking deeper is more beneficial to that of one-step look-ahead for this particular problem.

The effect of Agent History was as expected. Attempting to prevent overfitting is typically a good idea, and adding Agent History does just that. I am confident that Agent Histories should be used whenever a problem is modelled as a Reinforcement Learning Problem.

Experiment 5 was a pleasant surprise. I did not think adding all the stabilisation techniques to the neural networks would help to the degree that it did. All variants seem to have benefited from doing so. I now wish I had conducted more tests into this, to see which stabilisation made the biggest difference.

Having the thesis representative be a TD($\lambda$) Agent, suggests that learning based on state-values could be stronger than learning based on action-values when simulations are possible. The difference in win ratios illustrated in figure 4.11 is not big enough to conclude which approach is strongest though. On the other hand, it seems that for small training sessions, which 100k is, the choice of learning algorithm does not matter too much. Even though the thesis representative ended up being a TD($\lambda$) Agent, I would prefer going forward using DQN learning. The reason is due to the reduced calculations, as it was implemented as an action-out approach, something that is not possible for a Prediction algorithm, making only one pass of the network necessary per move. This enables bigger neural networks to be designed, which might lead to better value-function approximations. Furthermore, the fact that TD($\lambda$) produces very correlated training data, means that it is more susceptible to the instability issues related to function approximations.

Overall, the experience that I have gained through this process has made me realise that Reinforcement Learning involves a lot of trial and error. It is not simply a matter of deciding upon what learning algorithm that should be used but deciding how other hyper-parameters needs to be tweaked is a big part of it. On top of that, the problems are often unique in some fashion, making them require that the learning methods are modified in some sense. For Quarto this was expressed when I realised that the game has two actions per game-step. I must also conclude, that applying all the methods one can to help with the stability issues, is necessary.

5.1 Future Work

My goal was to compare model-free learning methods to that of Agents learning a model but as I ran out of time this is something that can be implemented in
future iterations. The reason is primarily that naive tree-search based applications exists for Quarto already, but I have found no Reinforcement Learning inspired programs so far. Combining Reinforcement Learning with tree-search algorithms such as Monte Carlo Tree Search should have a decent shot at outperforming mini-max methods. Of course, they will still be affected by the same problems that have plagued my implementation but hopefully the Agent would be able to combine both the strengths of "reading ahead" and the power of value-function approximations.

Another thing I would have liked to experiment with is different reward signal schemes. As mentioned, testing how much the Agent’s behaviour is affected by penalising losses, while rewarding wins could find interesting solutions. Because Quarto is a zero-sum game, the problem lends itself very well to this sort of experimentation.

While the overall experience has been rewarding, watching the Agents spent more than 2 days training per variant, without much improvement is disheartening. Therefore I would like to test how much introducing expert knowledge would effect learning behaviour and whether it is beneficial in any form. This would mean that I would have to play the Agents manually, as Quarto is a relatively unknown game, and as such does not have any database of expert games. Playing 100,000 games in this fashion would, of course, not be possible but experimenting using less could definitely be interesting.

Finally, it would be interesting to see how well the algorithms implemented for this game, including all the small improvements I applied to stabilise the learning process, would perform in other games. Seeing whether the practical approaches followed in this thesis is specific to Quarto, or if they can be applied in general.
Bibliography


David Silver. Reinforcement learning course. 2015. URL http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.


Kent Grigo Thor Bagge. Getting to know the captain’s mistress with reinforcement learning. 2016.

Sebastian Thrun. Learning to play the game of chess. 1995.


