3rd Year Project Report

Catherine Vlasov

May 2018

Abstract

This is my abstract

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 Reinforcement Learning

2.2 Learning Algorithms

2.2.1 Monte Carlo Learning

Chapter 3

Design & Implementation

This section describes the implementations of Tic-Tac-Toe and Chung Toi as well as the game-playing agents used in this project's experiments. The source code is not provided in this report, but is available upon request.

3.1 Overview

The project is built in a modular way, using an object-oriented style. All actions, agents, games, and states implement general interfaces, called *Action*, *Agent*, *Game*, and *State* respectively, that serve as Facades for the complex implementations.

The *Game* interface has a method *play()* that simulates a single game. This is where the agents are asked to choose actions, receive returns, and complete other tasks, depending on the specific game. Agents do not distinguish between different games and only use methods defined in the general interfaces, which means that the *MonteCarloAgent* class, for example, can be used to play both Tic-Tac-Toe and Chung Toi. This modularity provides a nice separation of concerns that enables simpler testing and debugging.

3.2 Agents

As part of this project, two agents were implemented: *RandomAgent* and *MonteCarloAgent*.

MonteCarloAgent implements the ϵ-soft on-policy Monte Carlo control algorithm. In its *chooseAction(State s)* method, it chooses an action according to its policy for states it has encountered in previous games, and randomly otherwise. In its *gameOver()*, method it calls two private helper methods *policyEvaluation()* and *policyImprovement()* that update Q and π, respectively, as described in section 2.2.1. Hash maps are used to store π and Q for constant time access and insertion.

As its name suggests, *RandomAgent* randomly selects an action from the list of available actions at any given state. In total, the logic for this agent is equivalent to one line of code and this agent is only used for training and benchmarking purposes.

3.3 Tic-Tac-Toe

The first game used in this project is Tic-Tac-Toe. It was chosen because it has a small state space (less than $3^9 = 19,683$ states) and it is a conceptually simple game that most people are familiar with.

3.3.1 Rules

Although this is a widely-known game, the rules are still included here for reference and clarity.

First, one of the two players is randomly selected to use **X** tokens (the other player uses **O** tokens). From now on, these players are referred to as the "X-player" and "O-player", respectively. The game then consists of the two players alternately placing their tokens in empty spaces on a 3x3 grid, starting with the X-player. Figure 3.1 is an example of a possible state of the grid.

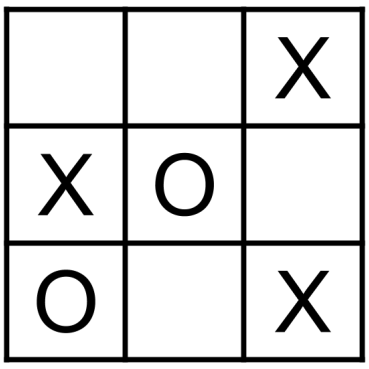

Figure 3.1: Example of a Tic-Tac-Toe grid

The game has three possible endings:

- **X-player wins**: three **X** tokens form a horizontal, vertical, or diagonal row
- **O-player wins**: three **O** tokens form a horizontal, vertical, or diagonal row
- **Draw**: the grid is filled and neither player has won

In all Tic-Tac-Toe implementations described in Section 3.3.2, agents receive a return of 1 when they win, -1 when they lose, and 0 when the game ends in a draw or the game is not yet over.

3.3.2 Implementation

The aim of this project is to investigate the effect of eliminating a game's state symmetries on the Monte Carlo algorithm's learning rate and optimal learning parameters. In the game of Tic-Tac-Toe, a state can be symmetrical with up to seven other states as result of flipping it in the following ways:

- horizontal axis
- vertical axis
- horizontal axis and then vertical axis
- major diagonal (i.e. top left to bottom right)
- major diagonal and then horizontal axis
- minor diagonal (i.e. top right to bottom left)
- minor diagonal and then horizontal axis

Of course, for some states, the result of some of these flips are the same, which is why a state might have strictly less than seven other symmetrical states.

With this goal in mind, Tic-Tac-Toe was implemented in three ways:

- **"Normal"**: This version does not break any symmetry and was used as a baseline.
- **"Limited Actions"**: This version breaks the symmetry between states by limiting the actions available from each state such that no two actions result in symmetrical states.
- **"Symmetric Equality"**: This version breaks the symmetry between states by considering two objects representing states to be "equal" if they are symmetrical.

An interesting result of the Limited Actions implementation is that it slightly alters the way the game is played. Suppose a *RandomAgent* is asked to select a move from the game's initial state (i.e. empty grid). In the Normal implementation, there are nine possible actions: putting an **X** token in one of the nine grid cells. In the Limited Actions implementation, there are only three possible actions: putting an **X** token in a corner, in the middle of the grid, and in the middle of one of the sides. To give a specific

example of how this changes the game, notice that the probability of the agent choosing the action that involves placing their token in the middle of the grid is $\frac{1}{9}$ in the Normal implementation and $\frac{1}{3}$ in the Limited Actions implementation. Thus, the agent will end up choosing this action three times as frequently in the latter version. However, this effect is ignored for the purposes of this project because in the limit of infinitely many games, the *MonteCarloAgent* will encounter all states and choose all possible moves an equal number of times, thus computing an accurate expected return for each action.

Implementing the desired behaviour for the Symmetric Equality version was not as simple as initially expected. Overriding the *equals()* method inherited from Java's *Object* class was straightforward. However, overriding the inherited *hashCode()* method was tricky.

A Java *HashMap* places key-value mappings in buckets depending on the value returned by calling *hashCode()* on the keys. Thus, if two keys have different hash codes, then they are placed in two different buckets, even if they are "equal" according to both of their *equals()* methods. So, two "equal" keys (and their respective values) can both end up being stored in a hash map at the same time if they have different hash codes. This is why it was crucial for the *hashCode()* method in *TicTacToeStateWithSymmetricEquality* to produce exactly the same result for symmetrical states. After considering a few solutions, this behaviour was implemented by internally converting each state to a canonical form in such a way that symmetrical states have the same canonical form. Each state's canonical form is used to compute the state's hash code, which results in all symmetrical states having the same hash code, as required.

The idea behind the Limited Actions implementation is that it is expected to speed up the *MonteCarloAgent*'s learning since only a fraction of the states will be reachable. Thus, after a fixed number of games, each possible state will be encountered more often than in the Normal implementation and so the expected return for each state will be more accurate. On the other hand, the Symmetric Equality implementation is expected speed up learning by allowing the *MonteCarloAgent* to combine its learning from symmetric states therefore increasing the accuracy of its expected returns. In both cases, the agent's policy and action-value function will be smaller, which reduces the agent's memory requirements.

3.4 Chung Toi

The second game used is in the project is Chung Toi, a more complex version of Tic-Tac-Toe that has been studied in the context of reinforcement learning before. Its state space is larger than that of Tic-Tac-Toe (i.e. less than $5^9 = 1,953,125$ states) which leads to interesting results.

3.4.1 Rules

The main idea behind Chung Toi is the same as that behind Tic-Tac-Toe in the sense that the game is played on a 3x3 grid and each player's goal is to get three of their pieces in a row. However, that is the entire extent of their overlap. Although the two games may seem similar, finding a good strategy for Chung Toi is not at all intuitive, unlike in Tic-Tac-Toe.

Before the game begins, one of the two players is randomly selected to use red tokens (the other player uses white tokens). From now on, these players are referred to as the "red player" and "white player", respectively. Each player has three tokens at their disposal and each token can be used in two possible orientations, as shown in Figure 3.2.

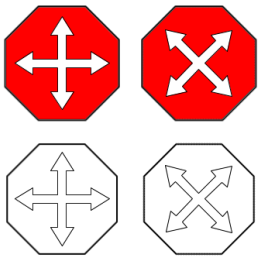

Figure 3.2: Red and white Chung Toi tokens in both possible orientations

The game consists of the players alternately making a move, starting with the red player. Throughout the game, the red player can only move red tokens and the white player can only move white tokens. The game has two phases and both players start in Phase 1. Once a player places all three of their tokens, that player proceeds to Phase 2. The following actions are available in each phase:

- **Phase 1**
  - Place a token (in either orientation) in an empty grid cell
  - Pass
- **Phase 2**
  - Slide a token on the grid in the direction of any of the arrows on that token. When the token reaches the final cell, it can be rotated (to switch its orientation) if the player chooses to do so. *(Note: all grid cells in the path from the original cell to the final cell must be empty, meaning that the token cannot "jump" over other tokens)*

– Rotate one token (to switch its orientation)
– Pass

At any point in the game, if both players pass, one right after the other, the game ends and is declared a draw. Otherwise, the game continues until one of the players makes their tokens form a horizontal, vertical, or diagonal row and that player is declared the winner. The orientation of the three tokens in the row does not matter.

Figure 3.3 is an example of a possible state of the grid.

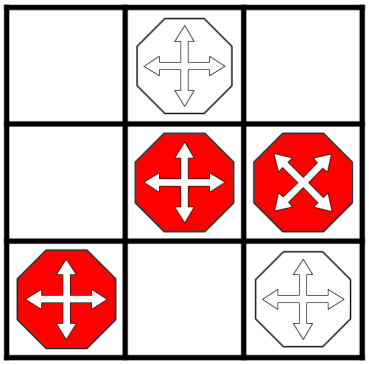

Figure 3.3: Example of a Chung Toi grid

3.4.2 Implementation

The role of Chung Toi in this project was to learn more about the effect of the Monte Carlo learning parameter ϵ on an agent's learning rate and winning rate. Chung Toi has roughly 100 times more states than Tic-Tac-Toe, which is expected to have an effect on the optimal value of ϵ and therefore *MonteCarloAgent*'s learning rate.

Chung Toi has the same axes of symmetry as Tic-Tac-Toe, so similar results would be expected from a Chung Toi implementation that breaks symmetries. For this reason, such an implementation was not developed to reduce redundancy and instead only a normal implementation with all symmetrical states present was developed.

Chapter 4

Results & Analysis

This section describes the outcomes of a variety of experiments and discusses why these outcomes were expected, or unexpected, and what can be learned from them.

4.1 Types of Experiments

Two types of experiments were used to investigate the behaviour of the Monte Carlo agent when playing games:

- **"Epsilon" Experiment**: This type of experiment iterates through all values of epsilon in a particular range and makes a Monte Carlo agent (initialised with that value of epsilon) play a specific game against another agent a fixed number of times. Then it records the results (i.e. wins, losses, draws) for each value of epsilon in a CSV file. Thus, it is possible to produce graphs with the tested values of epsilon on the x-axis and the number of episodes resulting in a win, loss, and draw on the y-axis. These graphs are used to determine the optimal value of epsilon for a particular game.
- **"Convergence" Experiment**: This type of experiment makes a Monte Carlo agent (initialised with a particular value of epsilon) play a specific game against another agent a fixed number of times. In the process, it records the cumulative results (i.e. wins, losses, draws) at regular intervals, as more and more games are played, in a CSV file. Thus, it is possible to produce area charts with the cumulative number of games on the x-axis and percentages on the y-axis to show the proportions of wins, losses, and draws. These graphs are used to visualise the convergence rate of a Monte Carlo agent's policy with a specific value of epsilon.

The data from running these experiments on different games and with agents playing against each other different numbers of times was combined

to produce a number of interesting graphs that reveal interesting properties of the Monte Carlo control algorithm used.

4.2 Hypothesis

Literature discussing on-policy ϵ-soft Monte Carlo control algorithms tend not to discuss the effect of the value of ϵ on an agent's learning rate or how to determine a good value for a particular application. Thus, this project was quite exploratory in nature and did not attempt to empirically prove certain results.

However, the following questions guided the experimentation:

- Are there "optimal" values of epsilon that maximise an agent's performance?
- How quickly does the agent's policy converge to the theoretical optimal policy?
- How does the performance of the algorithm differ for different games?
- When playing games where states have axes of symmetry, how does the agent's performance and learning change when symmetry is eliminated?

The games selected for this project were an interesting mix since some had relatively long episodes (Chung Toi), others could be easily configured to vary the number of possible states (Nim), and some were used to investigate symmetry (Tic-Tac-Toe). Thus, it was possible answer all of the above questions and derive some interesting results.

4.3 Optimal Value of ϵ

It is not unreasonable to expect that each game might have a specific value of epsilon that produces the fastest convergence of the learning agent's policy. If this was the case, then one would expect the graph of the winning rate produced by an epsilon experiment with ϵ ranging from 0.0 to 1.0 (inclusive) to be convex with a global maximum. This belief is reinforced by running an epsilon experiment on any game for some number of training games. For example, Figure 4.1 displays the results of training the learning agent on one million episodes of Chung Toi.

However, with some experimentation it becomes clear that there is another key factor in the equation: the number of training episodes. If the results of any particular epsilon experiment are considered without the context of the results of the same experiment with different numbers of training

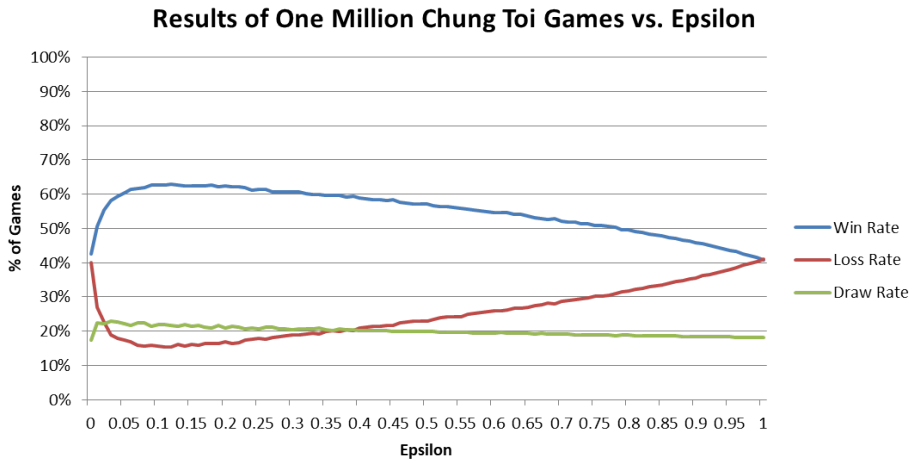

Figure 4.1: Results of training learning agent with one million episodes of Chung Toi

episodes, then the belief that each game has an optimal epsilon value seems to be true. However, when these results are considered in context, an interesting relationship between the number of training episodes and the optimal value of epsilon is uncovered. Figure 4.2 demonstrates this relationship for Nim. The same relationship is also visible in the graphs for Tic-Tac-Toe and Chung Toi.

As we can see, as the number of training episodes increases, the optimal value of epsilon decreases and approaches zero. In particular, if we plot the optimal value of epsilon versus the number of training episodes, for any game, we get a graph that looks like Figure 4.3, which exhibits an inverse relationship (??) between the two. In order to obtain more accurate optimal epsilon values for Figure 4.3, each epsilon experiment was run three times and the results were averaged.

This behaviour actually makes sense and should, in fact, be expected because of the role of ϵ in the Monte Carlo control algorithm. As discussed in Section 2.2.1, ϵ is used during policy improvement. In particular, the probability of choosing an action (other than the best action encountered so far) is $\frac{\epsilon}{N}$ where N is the number of available actions. On the other hand, the probability of choosing the best action encountered so far is $1-\epsilon+\frac{\epsilon}{N}$. Thus, as ϵ approaches zero, the probability of choosing the best action encountered so far approaches one.

So, as the number of training episodes approaches infinity, the probability that the agent has encountered the best actions for all states approaches one. Once the best action from a state has been encountered, the agent no longer benefits from randomly trying other actions since they can only lead

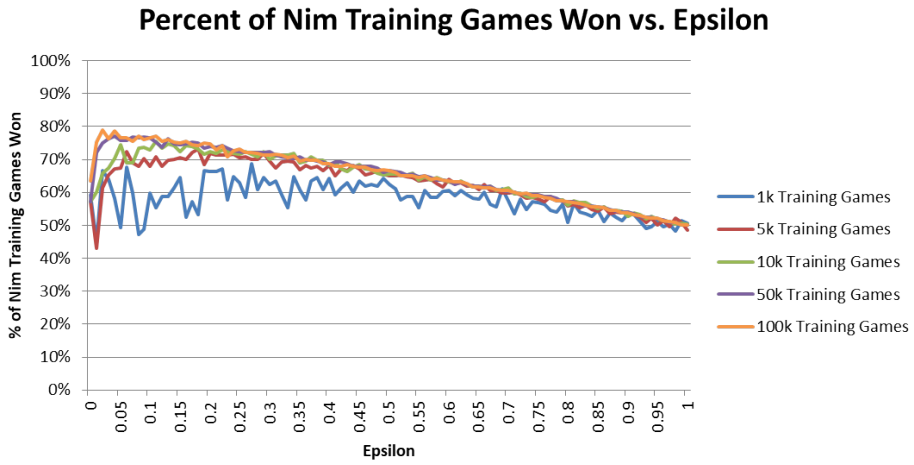

Figure 4.2: Comparison between results from different numbers of training episodes for Nim

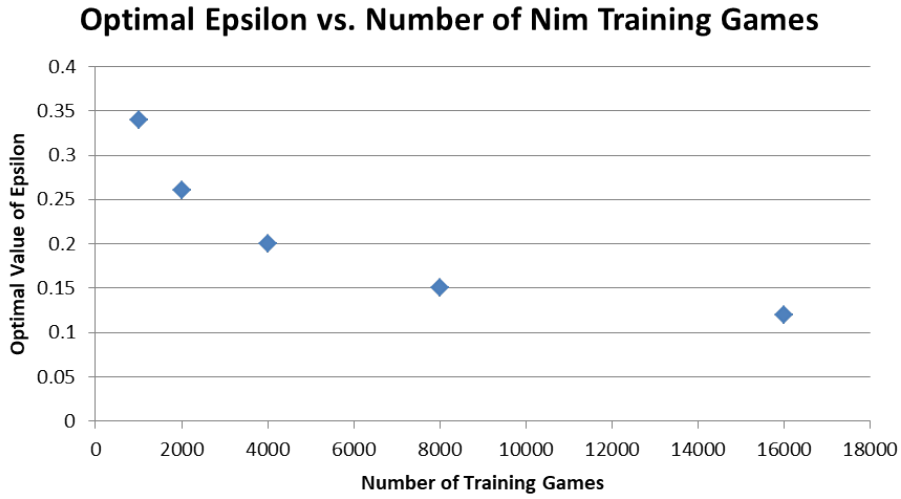

Figure 4.3: Relationship between optimal epsilon value and number of training episodes for Nim

to smaller (or equal) returns. Therefore, from then on, the agent will earn the largest overall return by choosing the best action from that state with a very large probability whenever that state is encountered.

A larger value of ϵ benefits an agent in the short-term because it speeds up learning (i.e. finding the best actions). However, it is disadvantageous in the long-term because even though the agent knows which actions produce the largest returns, the agent will still be forced to choose other actions relatively often.

Overall, there is no single value of ϵ that maximises an agent's performance in a particular game. However, for any specific number of training episodes, there *is* a value that results in the best winning rate by striking the right balance between being greedy (i.e. choosing the best action so far) and trying new actions that results for *that* amount of training.

4.4 Convergence to the Optimal Policy

In the context of training a Monte Carlo agent to play a game, the "speed" at which the agent's policy converges is the number of training games the agent needs until its policy, π, converges. Naturally, this varies depending on the value ϵ used by the agent.

Since the policy only converges to the theoretically optimal policy in the limit of infinitely many training episodes, in this project we consider the policy to have converged after x training games if the agent's winning rate changes by less than 0.1% after x training episodes compared to after $x + 1000$ training episodes.

In the experiment framework developed for this project, convergence experiments provide the data needed to compare different values of ϵ and to analyse the speed of convergence. Convergence experiments produce the proportions of wins and losses (and draws, if applicable) at regular intervals as an agent participates in more and more training episodes, using a specific value of epsilon. Thus, by plotting these ratios as an area chart and looking at the borders between the regions, we see how quickly the policy changes.

For instance, in Figure 4.2, we see that the agent's winning rate when playing Nim with $\epsilon = 0.1$ changes quite significantly as the number of training episodes increases. This suggests that running a convergence experiment for Nim with this value might provide insight into the policy's convergence rate. Figure 4.4 shows these results.

As we can see, after about twenty or thirty thousand training episodes, the agent's performance stabilises. In fact, the performance (and therefore policy) converges – by the definition of convergence for the purposes of this project – after $21,000$ training episodes.

On the other hand, Figure 4.2 shows that the agent's winning rate in Nim when $\epsilon > 0.5$ does not not change significantly with more training episodes.

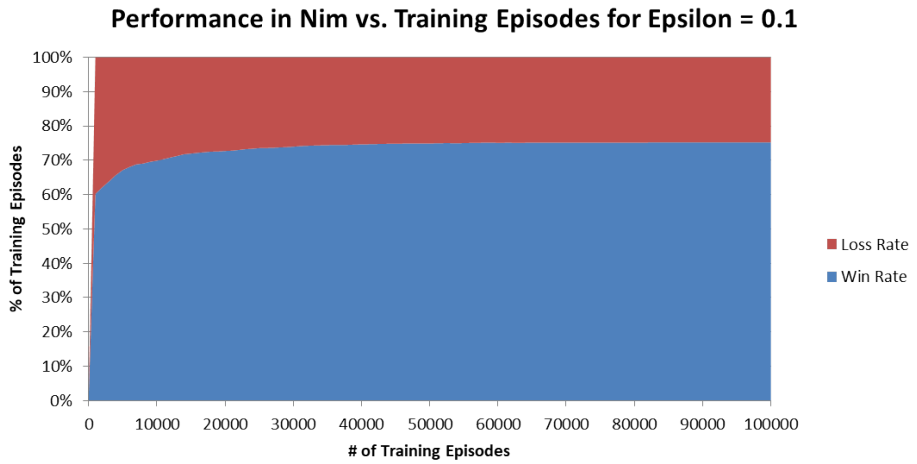

Figure 4.4: Changes in agent's performance in Nim with $\epsilon = 0.1$

So, we would expect that the agent's policy converges faster. This intuition is confirmed by Figure 4.5, which shows the results of a convergence test on $\epsilon = 0.5$.

More specifically, the performance converges after only $10,000$ training epsidodes.

Finally, the part of Figure 4.2 that seems to show the most changes is where ϵ is very small. Figure 4.6 displays the results of a convergence test on $\epsilon = 0.01$. This shows that the policy takes much longer to converge – $69,000$ training episodes, to be precise.

Epsilon	Training Episodes	Winning Rate
0.01	42,000	70.7%
0.1	19,000	72.6%
0.5	7,000	64.7%

Table 4.1: Data About Policy Convergence for Multiple Values of Epsilon

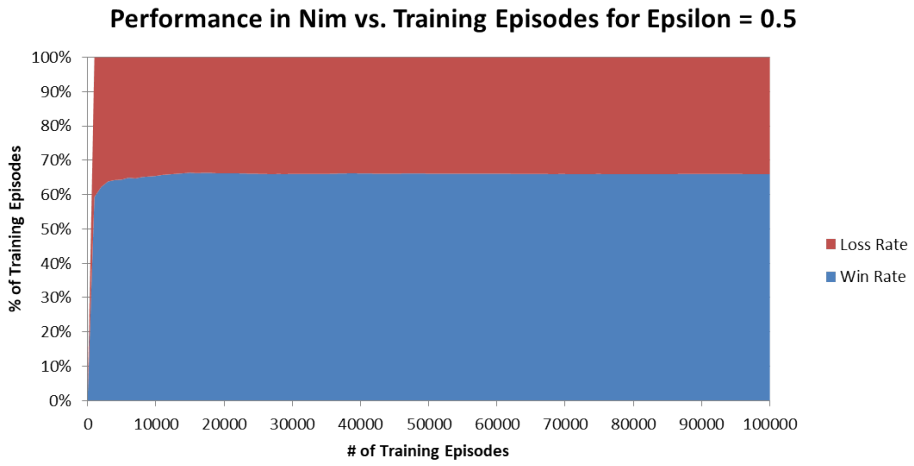

Figure 4.5: Changes in agent's performance in Nim with $\epsilon = 0.5$

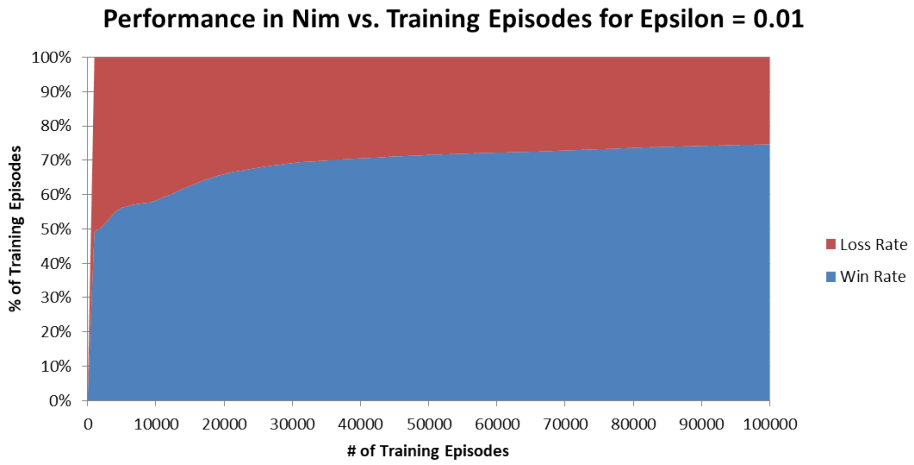

Figure 4.6: Changes in agent's performance in Nim with $\epsilon = 0.01$

Chapter 5

Conclusion

5.1 Future Work

5.2 Personal Reflections

This has been a really great learning experience, not only in terms of discovering the field of reinforcement learning and its applications, but also in terms of gaining experience with building up a large, complex programming project from scratch using a variety of technical tools.

When I was in high school, I designed and implemented the classic board game Nine Mens Morris where a user plays against an algorithm I wrote. My plan for the project was to ask my friends and family to play my game and to store the moves made in each game so that I could build up a database of sequences of moves and outcomes of the game. I hoped to somehow compute which moves were the best and get my algorithm to learn which moves made winning the most likely. Little did I know, I wanted to invent RL from scratch unaware of its existence or of the amount of research in the field. Several years later, I am delighted to have had the chance to work on a similar project, but this time on a deeper level using well-known computational algorithms.

When I started working on this project, I decided to treat it as an opportunity to develop many skills that are critical for a career in software engineering:

- Writing clean, maintainable, well-documented code
- Designing and implementing tests for my code
- Using a build tool to manage dependencies between packages in my project as well as with external libraries
- Using version control effectively

I tested the methods in my API using a unit-testing framework for Java called JUnit. I also used a testing framework for Java called Mockito this allowed me to verify the behaviour of objects with external dependencies by creating mock objects for theses dependencies, which mimic real objects but do so in a particular way that I can specify.

In order to save my experiment results in a format that would facilitate the creation of graphs, I used opencsv, a CSV parser library for Java.

To manage my projects dependencies, I used a build tool developed by Google called Bazel and I used Git for version control.

Overall, I really enjoyed learning about RL algorithms and exploring their applications and success rates, all while developing strong programming skills that will help me throughout my career.

Chapter 6

Acknowledgements

Thanks for reading.