Optimizing Learning Parameters Used by Reinforcement Learning Algorithms

Catherine Vlasov

23 April 2018

Abstract

This is my abstract

Contents

Chapter 1

Introduction

Over the last few decades, reinforcement learning has seen a large increase in popularity. It amazes the world again and again, whether playing Go [5], Backgammon [8], or Atari [3] games, and its potential continues to grow. These successes were not lucky, but rather they were a product of studying the nature, size, and complexity of each task, choosing the algorithm best suited to it, and then setting up the algorithm appropriately. The last of these steps is crucial to the success of reinforcement learning for a particular task and must be done carefully. Thus, this project aims to elucidate how to set up a reinforcement learning algorithm by tuning its parameters, specifically for a particular class of algorithms called Monte Carlo algorithms.

1.1 Goals

The specific Monte Carlo algorithm this project investigates is called an "on-policy ε-soft Monte Carlo control algorithm". For those unfamiliar with reinforcement learning, Section 2.3 explains the basics as well as terminology used throughout this report, such as "on-policy" and "ε-soft". As the name suggests, the algorithm uses a parameter ε to guide its learning. Effectively, it determines the balance between exploitation (i.e. choosing actions that are best, according to what it currently knows) and exploration (i.e. trying new actions). Thus, the chosen value of ε can have a large effect on an agent's learning rate and therefore success at a particular task.

The on-policy ε-soft Monte Carlo control algorithm is not the primary subject of many research papers since it is simple, relative to other popular reinforcement learning techniques. However, this simplicity facilitates an interesting analysis of the effect of the value of ε on the algorithm's success rate. Hopefully some of the insights and results found along the way can be extended to other reinforcement learning algorithms with parameters that need to configured.

This project attempted to tackle a number of questions:

- Are there "optimal" values of ε that maximise the algorithm's performance?
- How quickly does the algorithm's policy converge? Does is converge to the theoretical optimal policy?
- How does the Monte Carlo with Exploring Starts algorithm (referred to as Monte Carlo ES [7]) affect the convergence rate of the policy?
- When playing games where states have axes of symmetry, how does the agent's performance and learning change when symmetry is eliminated?
- How does the performance of the algorithm differ for different games?

The three games selected for this project were an interesting mix since some had relatively long episodes, others were conducive to a study of the Monte Carlo ES algorithm, and some could be used to investigate symmetry. Thus, it was possible answer all of the above questions and derive some interesting results.

Chapter 2

Background

This section provides an overview of the theory behind reinforcement learning and the particular algorithm studied in this project.

2.1 Reinforcement Learning

Reinforcement learning – referred to from now on as RL – is an area of machine learning in which an agent interacts with an environment by choosing actions, with the goal of maximizing the cumulative reward it receives for these actions.

The beauty of RL is that the agent is not given any information about *how* to solve a problem, but rather it has to find out what works and what does not through trial-and-error. For this reason, RL is different from supervised learning because the agent is not provided with any input-output pairs to learn from. Instead, it learns and improves purely based on its own experiences.

RL is now described more formally. An *agent* is a learner and the *environment* is everything outside the agent, namely the thing that the agent interacts with. These interactions occur at discrete time steps $t = 0, 1, 2, 3, ...$ There is a set of states $\mathcal{S}$ and a function $\mathcal{A}$ such that for any state $s \in \mathcal{S}$, $\mathcal{A}(s)$ is the set of actions that are available from state s. There is a function $\mathfrak{R}$ that specifies the *reward* (or *return*) r that an agent receives for choosing an action a at a state s. The ultimate goal of an agent is to maximize the total sum of all the rewards it receives. Based on its experience, an agent builds a *policy* π, which maps states to the probabilities of selecting each of the possible actions.

It is worth making a quick digression to explain what a Markov Decision Process (called MDP from now on) is. First, we define the *Markov property.*

$$\mathbb{P}(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0) \tag{2.1}$$

$$\mathbb{P}(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t) \tag{2.2}$$

A RL task has the Markov propety if and only if, for all states s', returns r, and all possible previous states, actions, and returns s_t, a_t, r_t, ... , r_1, s_0, a_0, 2.1 and 2.2 are equal [7].

More intuitively, having the Markov property means that the environment's response (i.e. state it transitions to and return it gives to the agent) at a particular time step should only depend on what happened during the previous time step. This response should be independent of what happened before the previous time step.

A task that satisfies the Markov property is called a MDP and furthermore, if it has finitely many states and actions (i.e. $\mathcal{S}$ and $\mathcal{A}$ are both finite), then it is called a *finite* MDP.

The reason finite MDP's are important is that it has been shown that they all have an optimal policy π^* [7]. Since the games used in this project all have finite state spaces and action spaces, this result is very useful since it facilitates the study of the convergence of an agent's policy as it gains experience playing the games.

2.2 Learning Algorithms

The primary goal of RL algorithms is to estimate a *state-value function* that indicates the "strength" of being in a particular state. This "strength" refers to the expected return that the agent will receive from the environment if it follows a certain policy. Formally the state-value function for a policy π is $V^\pi : \mathcal{S} \to \mathbb{R}$.

Similarly, RL algorithms also try to estimate an *action-value function* that indicates the "strength" of choosing a particular action from a particular state. The action-value function for a policy π is $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$.

RL algorithms estimate V^π and Q^π from experience and their goal is to build estimates as close to V^* and Q^* – the optimal state-value and action-value functions – as possible. We know that such optimal functions exist because, as explained in Section 2.1, finite MDP's have an optimal policy.

A final distinction to make between RL algorithms is between *on-policy* and *off-policy* algorithms. On-policy algorithms update their estimates of the value functions solely using the experience they gained from choosing actions and interacting with the environment. On the other hand, off-policy algorithms update their estimates based on actions they might not event take.

There are a number of well-known RL techniques that estimate V^π and improve their policies in different ways. One of the most popular and well-understood [6] algorithms is Q-learning [9], an off-policy algorithm. Most notably, a variant of Q-learning was used to train a convolutional neural network to play Atari 2600 games, outperforming human experts [3]. Q-learning is based in part on a category of RL algorithms called Temporal Difference, which are on-policy algorithms. One called TD(λ) gained notoriety when it was successfully used to train a neural network to play Backgammon at a level that surpassed other similar networks as well as human experts [8]. Temporal Difference, in turn, combines techniques from dynamic programming approaches to RL as well another category of algorithms called Monte Carlo, which are mostly on-policy algorithms. The latter category is the focus of this project since such algorithms are simple and intuitive, but still reveal a lot about how policies converge and how performance varies from game to game.

2.2.1 Monte Carlo Algorithms

Monte Carlo algorithms (referred to as MC from now on) are online algorithms that try to solve RL problems by computing the averages of returns. The term *online* means that as the agent gains new information by interacting with the environment, this new information is immediately put to use. In particular, MC algorithms update their value function estimates and policies at the end of each episode. For this reason, MC algorithms are only used for tasks that can be divided into discrete episodes that are guaranteed to terminate eventually.

MC learning works by saving the average of all returns received for choosing action a from s, for each state s and action a available from s, over the course of all episodes. This is stored in the action-value function Q^{π}.

There are two variations of MC with respect to averaging returns: *every-visit* MC and *first-visit* MC. Every-visit MC computes its estimate of each expected return $Q(s, a)$ using the average of the returns after *all* the times a was chosen from s. In contrast, first-visit MC does so using only the returns received the *first* time a was chosen from s in each episode. Both versions converge to the true expected values in the limit of infinitely many encounters with each state-action pair, so the first-visit version was arbitrarily chosen for this project.

A problem arises when a policy is completely deterministic because many state-actions pairs will never be encountered, regardless of how many episodes take place. More specifically, only one action from each state will be chosen whenever the state is encountered, so the algorithm will never improve. Thus, a concept called *exploring starts* is assumed. This means that the each episode begins with a random state-action pair, where all pairs have a non-zero probability of being selected. In this way, all states and actions are guaranteed to be encountered infinitely many times over the course of infinitely many episodes.

However, exploring starts cannot be assumed to hold for all RL tasks and one solution to this problem is an on-policy control method. Policies resulting from these methods are called *ε-soft* if $\pi(s, a) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$. A specific type of ε-soft policy is an ε-greedy policy and this is the type used in this project. The way they work is that the majority of the time, the agent chooses the best action, but with probability ε it chooses an action randomly. Specifically, the best action is assigned a probability of $1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$ and all other actions are assigned a probability of $\frac{\varepsilon}{|\mathcal{A}(s)|}$ in the policy.

With all of this in mind, we can now understand the on-policy ε-soft MC control algorithm used in this project:

[will figure out how to use the algorithm package to write pseudocode here]

2.3 Terminology

In addition to the terms described in Section 2.1, the following terms are used throughout this project:

- **Random agent**: Agent that randomly chooses an action from the list of available actions at any given state
  - In particular, an instance of the *RandomAgent* class described in Section 3.2)
- **MC agent**: Agent that uses the on-policy ε-soft Monte Carlo control algorithm for learning
  - In particular, it refers to an instance of the *MonteCarloAgent* class described in Section 3.2
- **Game**: Examples are Tic-Tac-Toe, Connect Four, and Chess
- **Training episode**: One iteration of a game used to train a MC agent

Chapter 3

Design & Implementation

This section describes the implementations of Tic-Tac-Toe, Chung Toi, and Nim, as well as the game-playing agents used in this project's experiments.

3.1 Structure

The project is written entirely in Java and 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. Figure 3.1 shows this structure.

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 all three games. 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 MC control algorithm, as discussed in Section 2.2.1. 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^π and π, respectively, as described in Section 2.2.1. Hash maps are used to store Q^π and π for constant time access and insertion.

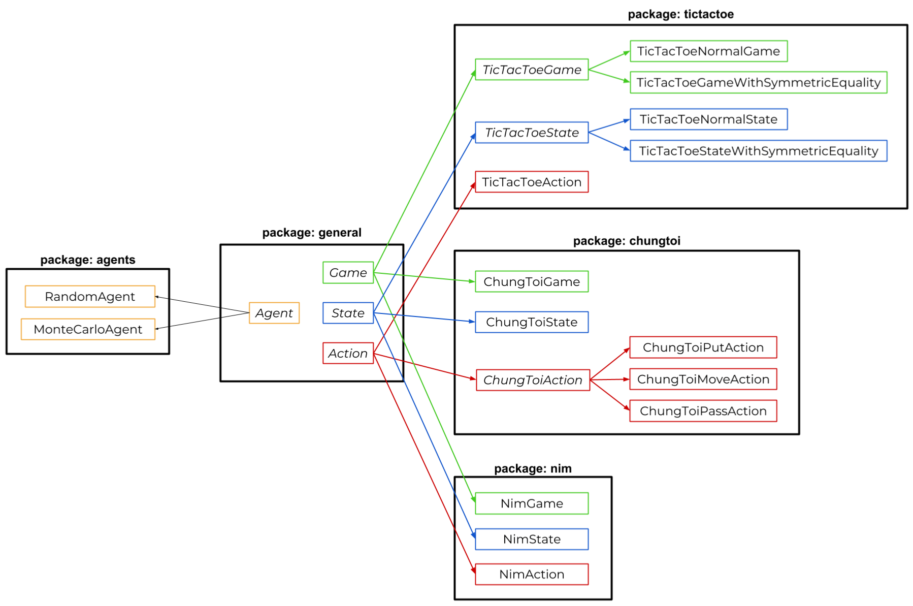

Figure 3.1: Structure of classes in project.

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 to play against the MC agent during training.

3.3 Tic-Tac-Toe

The first game used in this project is Tic-Tac-Toe. It was chosen as a first experiment because it has a small state space 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 [4] 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, and they both have an unlimited number of tokens at their disposal. 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 3.2 is an example of a possible state of the grid.

The game has three possible endings:

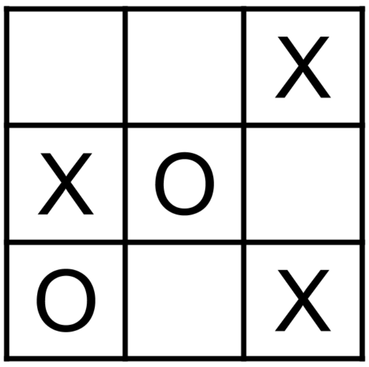

Figure 3.2: Example of a Tic-Tac-Toe grid. Each of the nine cells can be in one of three states: empty, X, or O.

- **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

3.3.2 State Space

Since each cell has three possible states, there are at most $3^9 = 19683$ states. In practice, strictly fewer than 19,683 possible states can be reached because technically the grid can be filled with exclusively X tokens, with more O tokens than X tokens, or even with a row of three X tokens *and* a row of three O tokens. None of these examples are valid game states so none of them will ever be encountered in a real game.

Preliminary experiments reveal that, in fact, only 4520 states are reachable, without taking into account the four axes of symmetry of a Tic-Tac-Toe grid. If these are taken into account, we find that there are only 640 unique (quotiented by equality by symmetry), reachable states. This is a bit more than one eighth of 4520 because reflecting a state (for example a state with only one token in the center of the grid) along some axes of symmetry produces the original state itself.

3.3.3 Implementation

One of the aims 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 two ways:

- **"Normal"**: This version does not break any symmetry and was used as a baseline.
- **"Symmetric Equality"**: This version breaks the symmetry between states by considering two objects representing states to be "equal" if they are symmetrical.

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.

In both Tic-Tac-Toe implementations, 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.

3.4 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 which leads to interesting results.

3.4.1 Rules

The main idea behind Chung Toi [2] 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 3.3.

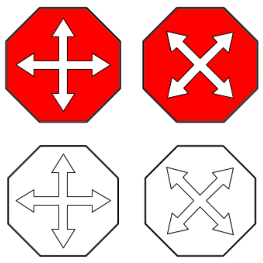

Figure 3.3: Red and white Chung Toi tokens in both possible orientations. The tokens in these orientations are referred to as: red-straight (top-left), red-diagonal (top-right), white-straight (bottom-left), and white-diagonal (bottom-right).

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 3.4 is an example of a possible state of the grid.

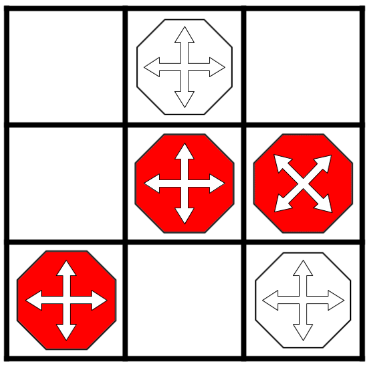

Figure 3.4: Example of a Chung Toi grid. Each of the nine cells can be in one of five states: empty, red-straight, red-diagonal, white-straight, or white-diagonal.

3.4.2 State Space

Since each cell has five possible states, there are at most $5^9 = 1,953,125$ states. For the same reasons outlined in Section 3.3.2, strictly fewer than this many states are reachable in practice and roughly an eighth of the reachable states are unique with respect to the four axes of symmetry.

3.4.3 Implementation

A standard implementation of Chung Toi was developed for this project. It enables two agents to play the game, exactly as described above.

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 the MC agent's learning rate. It 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.

Just like in Tic-Tac-Toe, 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 has not yet ended.

3.5 Nim

The third game used is in the project is Nim, a simple game with an interesting mathematical theory behind it. It has a large number of variations and can be configured in many ways, so its state space can vary widely. For the purposes of this project, a "small" version of the game is used in order to deal with a small state space.

3.5.1 Rules

The general idea of Nim [1] is that two players alternate removing objects from several piles. The first player is chosen randomly and then on each turn, a player can remove any number of objects from any one of the piles. Each player must remove at least one object on each turn. The game ends when all of the piles are empty.

Which player wins depends on which version of the game is being played. In the "normal play" version the last move is the winning move, whereas in the "misère" version (where misère is used in a game theoretic sense) the player who moves last loses. The misère version is more common, so that is the version implemented in this project.

3.5.2 State Space

As explained in Section 3.5.3, the "small" version of Nim used in this project consists of three piles, initially with seven objects each. Since each pile can have between zero and seven objects, inclusive, there are $8^3 = 512$ possible states.

3.5.3 Implementation

Nim is traditionally played with three piles of objects. The first of two versions of Nim implemented in this project is standard and initialises all piles with a fixed number of objects. In particular, in order to keep the state space relatively small, all three piles start off with seven objects.

A second version of Nim was implemented since the game is well-suited to investigating the performance of the Monte Carlo control algorithm with exploring starts. This is because instead of initialising all piles with a fixed number of objects, all piles can be initialised with random numbers of objects.

In this second implementation, piles are initialised with a random number of objects between zero and seven, in such a way that at least two piles are non-empty. This last condition was necessary because if all piles are empty or if there is only one non-empty pile and the other agent goes first and removes all the objects, then the game ends and the MC agent must receive a return without having chosen any action. This creates problems in the policy improvement phase of the algorithm since there is an outstanding return relative to the number of actions taken (i.e. zero).

Since the concept of exploring starts relies on games starting with random state-action pairs, the implementation of the MC agent (as well as the *Agent* interface and random agent) had to be modified and a slightly modified version of Nim was created. This is because in addition to randomly initialising the piles, an action has to be randomly selected. Then, the agent that is randomly selected to go first has to be forced to choose that random action. In this way, in the limit of infinitely many games, all actions from all states are explored and the agent is able to determine the overall best return it can receive from each state.

At the end of the game, the winner receives a return of 1 and the loser receives a return of -1. Throughout the rest of the game, both players receive returns of 0 for all their actions.

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 MC agent when playing games:

- **"Epsilon" Experiment**: This type of experiment iterates through all values of epsilon in a particular range and makes a MC 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 MC 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 MC 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 Number of Training Episodes

One decision that was made in the design and execution of these experiments was the number of training episodes to make a MC agent play for each experiment type and each game. Through a process of trial-and-error, it was determined that for all three games, the agent needed at least one hundred thousand training episodes before having it would be able to reach a good portion of the states

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.

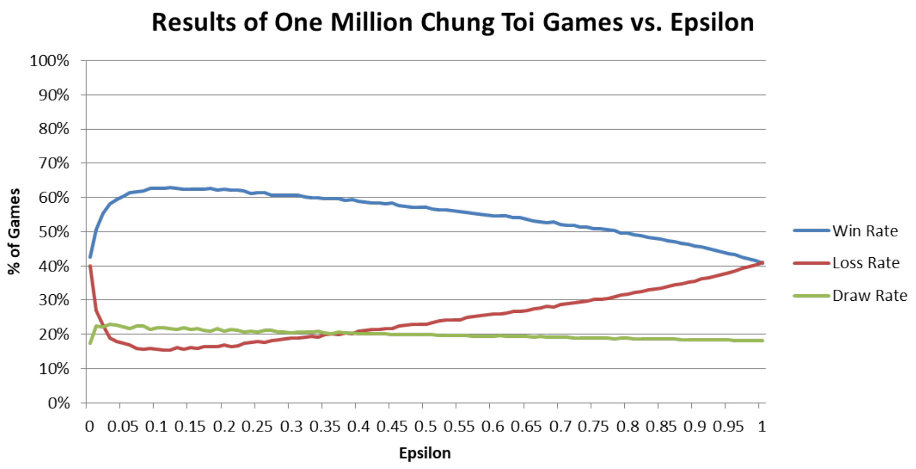

Figure 4.1: Results of training learning agent with one million episodes of Chung Toi

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 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.

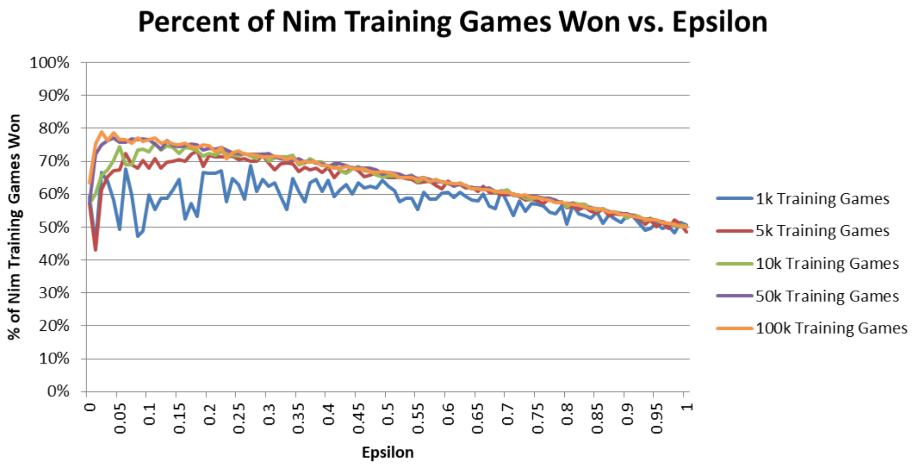

Figure 4.2: Comparison between results from different numbers of training episodes for Nim

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. [*In this particular version, the experiment was only run once – I will run it twice more to make a smoother graph*]

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 $\frac{\varepsilon}{N}$ where N is the number of available actions. On the other hand, the probability of choosing the best action encountered so far is $1-\epsilon+\frac{\varepsilon}{N}$. 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 probabil-

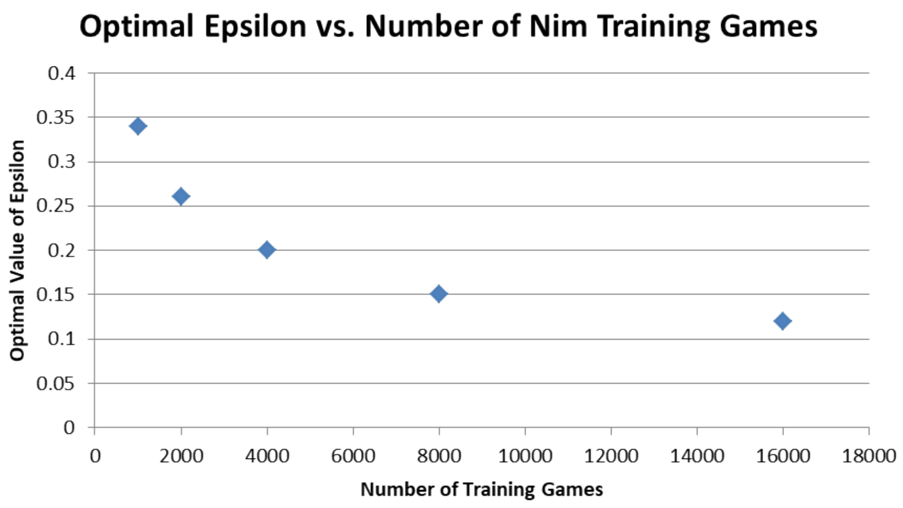

Figure 4.3: Relationship between optimal epsilon value and number of training episodes for Nim

ity 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 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.

[*Maybe mention that optimal values vary from game to game due to varying state spaces. Also I could briefly mention why the behaviour at 0.0 and 1.0 is the same in all experiments.*]

4.4 Convergence to the Optimal Policy

In the context of training a MC 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 of 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 $\varepsilon = 0.1$ 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.

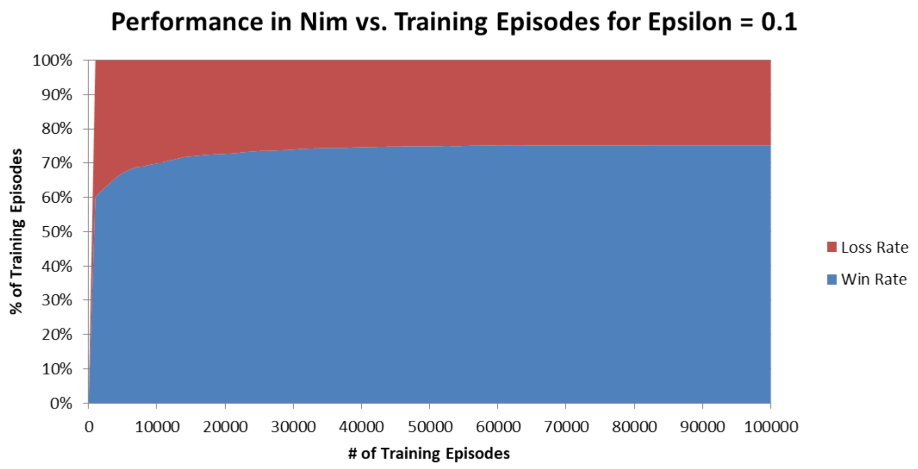

Figure 4.4: Changes in agent's performance in Nim with $\varepsilon = 0.1$

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 19, 000 training episodes.

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

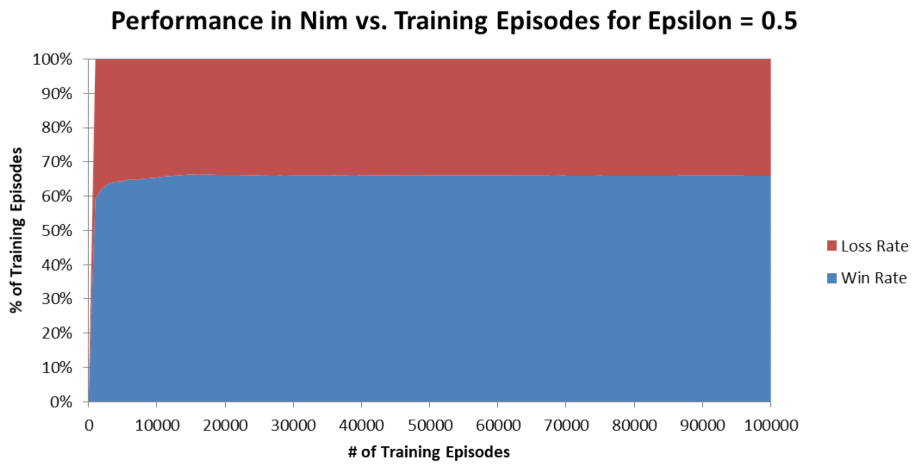

Figure 4.5: Changes in agent's performance in Nim with $\varepsilon = 0.5$

More specifically, the performance converges after only 7, 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 experiment on $\varepsilon = 0.01$. This shows that the policy takes much longer to converge – 42, 000 training episodes, to be precise.

Convergence experiments were run on a several values of epsilon and the results are summarised in Table 4.1. As we can see, the results empirically verify the following intuitions:

- **ε should not be too small (otherwise new actions will rarely be chosen) or too large (otherwise the best actions will rarely be chosen).** This is indeed the case since the best value of epsilon is not quite at either extreme of the range of values tried – the value with the largest winning rate at convergence is $\varepsilon = 0.04$.

- **In general, as ε gets larger, the number of training episodes until convergence decreases.**

- **As ε gets larger, the agent is more likely to encounter all possible states (assuming a reasonably small state space).** This is

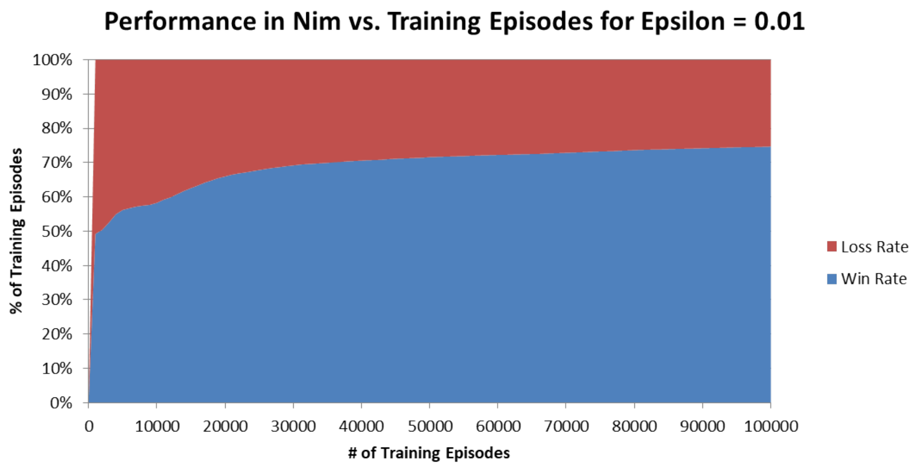

Figure 4.6: Changes in agent's performance in Nim with $\varepsilon = 0.01$

clearer when looking at the actual data produced by the convergence experiments since there we can see how many states were encountered after every additional 1,000 training episodes. In the case of $\varepsilon = 0.64$ all states were encountered after only 5,000 episodes whereas it took tens of thousands of episodes for the smaller values of ε that were tested.

Epsilon	Training Episodes	States Encountered	Winning Rate
0.01	42,000	494	70.7%
0.02	47,000	507	75.7%
0.04	38,000	511	76.5%
0.08	21,000	508	76.1%
0.16	19,000	507	72.7%
0.32	9,000	509	70.4%
0.64	8,000	511	61.5%

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

4.5 Monte Carlo with Exploring Starts

The game that is most conducive to applying the Monte Carlo with exploring starts algorithm is Nim because it is easy to create a random starting state. Since Monte Carlo with exploring starts does not have any learning parameters like ε, the only relevant type of experiment is a convergence experiment. Normally, a convergence experiment takes a value of ε as an argument, but since such a value is not used by an agent that implements the exploring starts algorithm, the argument is ignored. Thus, the only experiment parameter that can be modified is the total number of training episodes. For consistency with the other experiments on Nim, the convergence experiment for Nim with exploring starts was run 100,000 times and the results can be seen in Figure 4.7.

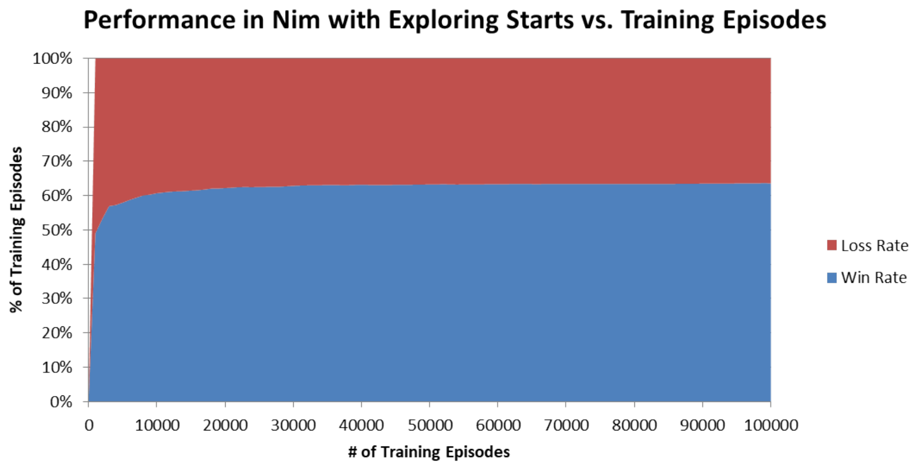

Figure 4.7: Changes in agent's performance in Nim with exploring starts

The policy converges after around 14,000 training episodes and at that point the winning rate is 61.3% and the number of states encountered is 511 (i.e. all states). This is very interesting because in the convergence experiment with the on-policy ε-soft MC agent, the largest winning rate *at the point when the agent's policy converges* of any value of ε (among the values in Table 4.1) was around 76.5%, meaning it won 15% more training episodes than the exploring starts agent.

[*I will add more discussion here. I could also make the two agents *after all the training* play another 100,000 games and see who wins more.*]

Chapter 5

Discussion

5.1 Learning Optimal Epsilon

[*Discussion of how I tried this but it didn't really work conceptually.*]

5.2 Alternative Implementations of Symmetry-Breaking

Another way of eliminating symmetry in Tic-Tac-Toe was also considered: limiting the actions available from each state such that no two actions result in symmetrical states. An interesting result of this idea is that it would slightly alter the way the game is played. Suppose a random agent 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 this "Limited Actions" version, 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 would change 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 version. Thus, the agent will end up choosing this action three times as frequently in the latter version.

In the limit of infinitely many games, however, the MC agent would encounter all states and choose all possible moves an equal number of times, thus computing an accurate expected return for each action. It would be interesting to compare the performance of such a Limited Actions implementation with the Symmetric Equality implementation. Both should perform equally well on any game given enough training, but it is possible that they might improve at different rates.

5.3 More... to be discussed

Chapter 6

Conclusion

6.1 Future Work

6.2 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. [*I don't know if this paragraph should stay in my final version...*]

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 7

Acknowledgements

Bibliography

[1] C. L. Bouton. Nim, a Game with a Complete Mathematical Theory. *Annals of Mathematics*, 3(1/4):35–39, 1901-1902.

[2] E. Cubukcuoglu. Unofficial Chung Toi Rules. `https://boardgamegeek.com/filepage/51629/unofficial-chung-toi-rules`.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] P. D. Schumer. *Mathematical Journeys.* Wiley-Interscience, 1st edition, 2004.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the Game of Go Without Human Knowledge. *Nature*, 529:484–489, 2016.

[6] R. S. Sutton. Introduction: The Challenges of Reinforcement Learning. *Machine Learning*, 8:225–227, 1992.

[7] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1st edition, 1998.

[8] G. Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

[9] C. J. C. H. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8:279–292, 1992.