DEGREE PROJECT, IN ENGINEERING PHYSICS , FIRST LEVEL
STOCKHOLM, SWEDEN 2015

Reinforcement Learning and the Game of Nim

PAUL GRAHAM & WILLIAM LORD

KTH ROYAL INSTITUTE OF TECHNOLOGY

CSC, COMPUTER SCIENCE

Reinforcement Learning

and the game of Nim

Paul Graham prgraham@kth.se

William Lord wlord@kth.se

Degree Project in Engineering Physics

First Cycle (SA104X, 15 ECTS)

05-2015

School of Computer Science (CSC)

Royal Institute of Technology

Stockholm

Supervisor: Pawel Herman (CSC)

Examinator: Mårten Olsson (SCI)

Abstract

This paper treats the concept of Reinforcement Learning (RL) applied to finding the winning strategy of the mathematical game Nim. Two algorithms, Q-learning and SARSA, were compared using several different sets of parameters in three different training regimes. An analysis on scalability was also undertaken.

It was found that tuning parameters for optimality is difficult and time-consuming, yet the RL agents did learn a winning strategy, in essentially the same time for both algorithms. As for scalability, it showed that increased learning time is indeed a problem in this approach.

The relevance of the different training regimes as well as other conceptual matters of the approach are discussed.

It is concluded that this usage of RL is a promising method, although laborious to optimize in this case and quickly becomes ineffective when scaling up the problem. Ideas are discussed and proposed for future research on solving these limiting factors.

Abstrakt

Denna rapport behandlar konceptet Reinforcement Learning (RL) och RL-agenters förmåga att lära sig den vinnande strategin i det matematiska spelet Nim. Två algoritmer, Q-learning och SARSA, med flera olika parameterinställningar jämfördes i tre olika träningsregimer. Därutöver analyserades effekterna av storleksökning av spelet.

I undersökningen visade det sig att bestämmandet av parametrar för ett optimalt beteende var väldigt svårt och tidskrävande, även om RL-agenterna med de funna parametrarna lyckades lära sig den vinnande strategin, och båda algoritmerna verkade lära sig ungefär lika snabbt. Att ökningen av inlärningstid vid växande uppgifter är ett problem verifierades också i undersökningen.

Relevansen av de olika träningsregimerna behandlas, likväl andra konceptuella frågor.

Som slutsats kan sägas att denna tillämpning av RL är en lovande metod men komplicerad att optimera och med nackdelen att den lätt blir ineffektiv vid större problem. I rapporten diskuteras idéer om föreslagen forskning på lösningar till de begränsande faktorerna.

Contents

1 Introduction

Since the first computer was created, its importance to humankind has grown rapidly. By making calculations and simulations easier, computers propel technological development. As most of the things we use in our everyday lives have become digitized, our modern society has become dependent on computer systems.

Fascinated as humankind is by the power of our own minds, a growing interest has presented itself in what intelligence is and if we can create it. In trying to answer these questions fields of science are rapidly growing which cover modelling of neural networks and brain-inspired computations [1, 2, 3].

In psychology, the theory of behaviourism describes how animal behaviour can be changed as a consequence of reinforcement due to reward or punishment. Analogously to this [1], a branch of artificial intelligence has developed that makes use of rewards in order to let an agent learn to perform a certain task. This method is called Reinforcement Learning (RL) and will be studied in this paper.

RL is a method of programming a computer (the agent) in order for it to be able to learn how to carry out tasks autonomously. Learning can be carried out either under the supervision of a human (supervised learning) or the agent can be left alone to explore its environment (unsupervised learning) [4]. To enable the agent to learn, rewards are given to encourage the agent to find the optimal solution to a given task.

RL provides vast potential to the area of artificial intelligence, with opportunities for human-like behaviour in robots, search algorithms and more. One example of human-like behaviour in robots is the torque-controlled robotic arm learning the task of flipping a pancake [5]. This task, which may seem trivial, is hard to perform even for some humans and even harder to implement in code, was successfully implemented using RL. However, the purpose of RL is not necessarily to imitate human behaviour. The fact that the agent teaches itself by trial-and-error means that it will try many different strategies before finding the optimal. This can lead the way to previously unknown solutions, as in the case of the early RL agent TD-gammon which came up with its own completely new strategies of playing backgammon [6].

Applications of artificial intelligence are already widely spread. What is unique in RL however, is that it does not require explicit programming of each situation [7], rather the agent learns from the ground up how to perform a task. Therefore, complex tasks can be learnt with much less programming. Furthermore, this creates a much more flexible agent than one coded with a single purpose since in this way agents can learn to perform more tasks in many different ways [8] as well as how to perform tasks that humans might not know how to describe in code [5].

A focus of contemporary research in RL (and of this paper) is on letting agents learn to play games. A recent example published in the journal *Nature* was work carried out by Deep Mind concerning an agent learning to play 49 classic Atari games [8].

This paper will focus on the mathematical game Nim. It is of interest as it is a rather simple game, yet representative for combinatorial games [9] and has a known solution [10]. Having a known solution makes measuring performance easier.

Applicability of RL to Nim has already been subject to research [11], however the feasibility of using RL with Nim as the dimensions of the game are increased was not studied. Scalability is a known problem of RL [7]. As the number of degrees of freedom of the agent increases, the time taken for the agent to complete its training period also increases, which is known as the *curse of dimensionality* [12]. This leads to the question of how effective RL applied to Nim will be when scaling up the game. Being a rather simple game, Nim is easily scaled up and suitable for studying scalability.

RL can be implemented using different algorithms with different parameter tunings, which leaves much room for investigation. In the preceding research on RL applied to Nim [11], one algorithm was studied. In this paper it is investigated how a slightly different algorithm would perform in comparison.

1.1 Problem Statement

We will implement RL agents using two algorithms in order to let them find the winning strategy of the mathematical game called Nim. In doing so, which algorithm and set of parameters will have optimal performance in terms of learning time and precision? Since dimensionality is known to be a problem to RL, an important question is raised of how much longer it takes an agent to find the optimal solution as the number of states increases.

1.2 Scope

Carrying out this project, the RL algorithms Q-learning and SARSA were used. There are other algorithms available to RL problems and many interesting areas to study, however as Q-learning has been previously studied this was considered a suitable benchmark. SARSA is a variant form of Q-learning with a slight difference and was therefore considered a possible and interesting subject of research.

The main problem studied was scalability, being a relevant and interesting subject. However, restrictions in computational power did not leave very much room for scaling. As number of states and actions increased, computation times increased accordingly and simulations soon became too long to be carried out within the frame of this project. Another aspect of the project subject to restrictions was tuning the parameters of the algorithms. In theory, there is an infinite amount of parameter combinations available for each algorithm. The entire spectrum of parameter combinations was not studied. Instead, a few interesting combinations were chosen. Furthermore, performance measures were made on data from one simulation run. Averaging over multiple runs was not done.

2 Theory & Background

2.1 Reinforcement Learning

In RL, learning is done by what is known as an agent. The agent exists in an environment and is equipped with sensors to allow it to observe and interact with the environment. Through a process of a trial-and-error the agent explores its environment, making decisions on the next action to take based on the rewards it receives and on its past experience. As an example, Deep Mind’s game-playing agent uses only the score and pixels from the screen as inputs [8]. Most interestingly, this agent learns different tasks independently, which is considered a big step towards genuine artificial intelligence.

RL consists of four main components: the policy, reward and value functions as well as a model, which are described below and are treated more thoroughly by Barto [4]. The policy guides the agent's behaviour outlining which action to take given a specific state. While learning to perform a certain task, the agent looks for an optimal policy that will provide the agent with the maximum reward going from a starting state to a terminal state.

The reward function assigns reward to the agent either after each action or after a series of actions depending on the task. The reward function is there to define the main goal of the agent.

The value function is a measure of how advantageous it is to be in a certain state or to take a certain action whilst in a certain state, if following the policy. It controls whether or not the agent should seek short-term gain or pursue a long-term strategy, and makes way for finding an optimal policy.

Finally, a model would consist of any prior information of the environment. The agent and environment can be coded model-free [4, 7]. In the model-free case it is necessary for the agent to discover the value and reward functions by exploring the environment.

In the model-free case, a question of exploration vs exploitation is raised when it comes to acquiring a maximum reward [4, 7]. The agent can choose to exploit what is already known and perform the actions it knows to generate the highest reward, or it can choose to explore new actions with unknown rewards for a chance of finding a better policy.

By combining these four main components the agent can teach itself the optimal behaviour given any state it will encounter.

2.2 Markov Decision Processes

The mathematical basis of RL rests on Markov Decision Processes (MDP) [4,7]. An MDP consists of a set of states S and a set of actions A. In each state $s \in S$ the agent may execute one of a number of actions $a \in A$. The reward function R maps a reward to each action carried out from a state. The state transition function T maps the probability of arriving in state s' from s after having taken action a. Further one makes the assumption of the *Markov property* which states that the result of an action depends only on the current state in which the agent finds itself and is independent of the history of actions and states leading up to the current state.

Each state has a value *V(s)* which is the expected reward the agent will receive if it follows the current policy from this state. Throughout the training stage the value of each state *V(s)* is updated, as is the policy which the agent follows. The policy and the value function *V(s)* are updated simultaneously. Over time these converge to V^* and π^*, the optimal solutions for the value and policy functions respectively.

2.3 Reinforcement Learning Algorithms

RL algorithms search for optimal policies by updating the expected value of a state based on the expected value of successor states. In other words they make estimations based on estimations which is known as bootstrapping [4]. While updating these value estimations, policies are also updated and agents can then easily be programmed to follow an optimal policy, that is, a policy generating maximum reward.

Among different RL algorithms Q-learning has become popular [7]. This algorithm, as well as another one named SARSA, are focused on in this paper.

2.3.1 Q-Learning

In Q-Learning the expected value of states and actions are stored in what is known as the Q-table. In the Q-table every element (state-action pair) is assigned a value *Q(s,a)* indicating to the agent the expected reward for taking action a in state s. This is illustrated in figure 1.

The values of the Q-table are updated using equation (1). In Q-learning the values are updated with respect to the action that maximizes expected reward in the successor state with equation (2). The parameters of the update formula (1) must be tuned in order for the agent to converge to the optimal policy as fast as possible.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\, max_{a'} Q(s',a') - Q(s,a)] \qquad (1)$$

$$max_{a'} Q(s',a') \qquad (2)$$

$$\begin{array}{c|cccc} \text{States\\Actions} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & -10 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 5 & 0 & 0 & 5 \\ 4 & 0 & 0 & -10 & 0 \end{array}$$

Figure 1. A Q-table of state-action pairs. This example has four states and four possible actions per state. The elements contain values for taking a specific action in a specific state, for example the value -10 of taking action 2 in state 1.

The parameter α is the learning rate parameter and takes a value between 0 and 1. The closer α is to 1 the more emphasis is placed on recent rewards, where $\alpha = 1$ means that the new state-action value completely overwrites the old. An α value of 0 means that no learning will take place while a value close to 0 means altering the old values slightly with every update. This could be likened to long-term memory.

The discount factor γ also takes a value between 0 and 1. If $\gamma = 0$ then importance is placed on maximising current reward and no propagation of reward will occur, whereas $\gamma = 1$ means more importance is placed on future rewards. In this case, rewards will be propagated from state-action pairs to predecessor state-action pairs [13].

The Q-learning algorithm is rather straightforward (see figure 1). The agent starts in state *s*, it then chooses an action *a* according to a policy which controls its behaviour. The Q-table is updated as described above. This process is then repeated from the new state until a terminal state is reached.

For an agent to learn from its environment it is necessary for it to explore. The policy assigned to the agent tells it how much exploring it should do and how much it should exploit the knowledge it has already gained, also known as the *exploration/exploitation trade-off* [4]. At the beginning of training it pays for the agent to explore as much as possible in order to find out what rewards are available.

```
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Figure 2. The Q-learning algorithm ([4]).

A standard exploration policy is known as ε-greedy [4,7]. In ε-greedy the agent chooses the best action (action with highest Q-value) with a probability of ε. Conversely with probability 1 - ε it chooses an action randomly from a uniform distribution. The purpose of this policy is to encourage exploration. As the number of training rounds increases the value of ε can be increased until no further exploration is made. In this way the agent will explore in the beginning of its training regime before beginning to exploit its knowledge of the environment. Interestingly, Q-learning is exploration insensitive. In other words the agent will find the optimal policy if it is allowed to visit each state-action pair an infinite number of times [4]. The role of an exploration policy in this light is to speed-up the learning process [7].

The Q-learning algorithm is what is known as an *off-policy* algorithm [7]. In other words the agent estimates the optimal policy while following another (for example ε-greedy).

2.3.2 SARSA

SARSA is a variant form of Q-learning also using a Q-table although it differs from Q-learning in the updating of the Q-table [7]. Whereas Q-learning updates the Q-table using the state-action pair with the highest value, SARSA updates the table according to the policy being followed. In this way, the exploration policy becomes a part of the optimal policy. SARSA is known as an *on-policy* learning algorithm [7]. The algorithm for SARSA is given in figure 3.

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

Figure 3. The SARSA algorithm [4].

2.4 Nim

The game of Nim is a mathematical game in which two players take turns on removing an arbitrary amount of counters from one of the heaps that make up the game (for example four heaps of 1, 3, 5 and 7 counters). At least one marker must be removed by the player in each move. In each move the player is allowed to remove counters from only one heap. The players continue taking alternate turns until all the heaps are empty. The player who takes the last marker, i.e. leaves the board empty, is the winner (see figure 4).

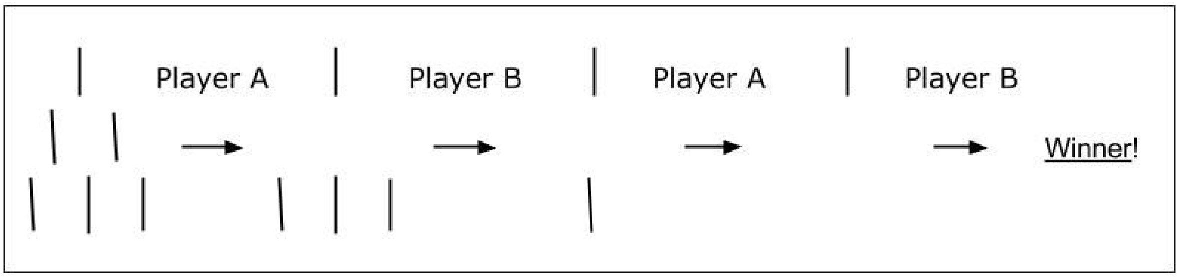

Figure 4. A complete game of Nim of three heaps. The game proceeds with moves from left to right. Player B makes the last move and wins.

2.4.1 Binary Digital Sum And Winning Strategy Of Nim

A digital sum works as an ordinary sum with the exception that carry-overs are rejected. This is best illustrated by writing the numbers that are to be summed above one another. Writing numbers in binary notation this way produces the binary digital sum. This way of finding the digital sum is illustrated along with a completed game of Nim in figure 5.

Nim has a winning strategy based on binary digital sums [10]. This winning strategy is, when possible, to make moves such that the binary digital sum of the heaps becomes zero. A combination of heaps with binary digital sum zero is called a safe combination [10]. Leaving the board with a safe combination of heaps in every move guarantees winning [10]. If player A makes a move leaving a safe combination, player B cannot do so. A safe combination means that there is an even number of ones in every column of the binary digital sum as illustrated above. Any valid move now removes a one from at least one column and makes it impossible to leave a safe combination. Instead, this leaves an uneven number of ones in at least one column which can easily be compensated for in the next move by reducing a heap so that each column again contains an even number of ones, and thus again leaves a safe combination.

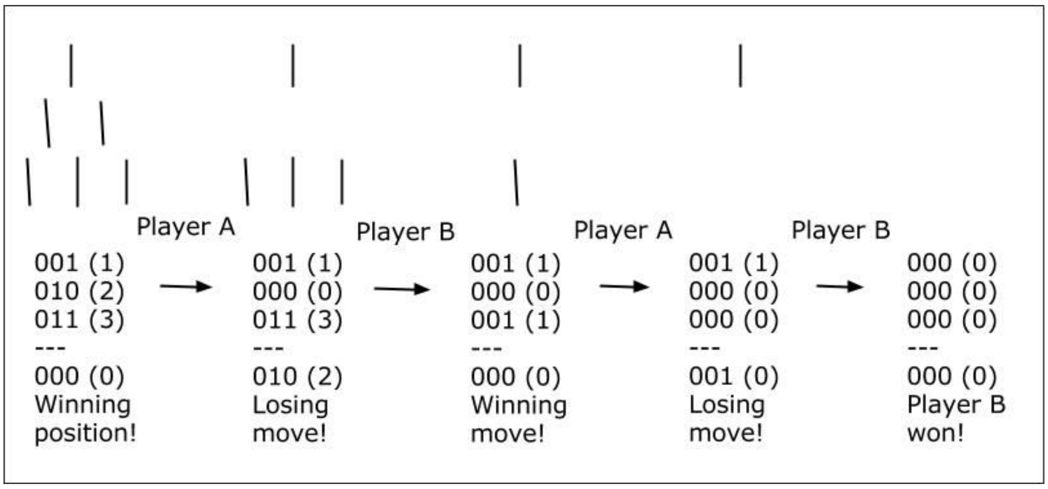

Figure 5. A completed game of Nim of three heaps including binary digital sums. Heaps are illustrated as rows of counters and binary digital sums are written below each state. The game proceeds with moves from left to right. Player B makes the last move and wins.

Accordingly, the player starting in a winning position, having the opportunity to leave the board with a safe combination of heaps, will win the game, if familiar with the winning strategy. Conversely, if the board comprises a safe combination in the initiation of a game round, the second player will win, if familiar with the winning strategy. The existence of this strategy will allow us to observe the performance of the agent as it converges to the optimal solution.

3 Methodology

In this work, the environment and the agent were implemented in Python 3.4.2. For the interested reader, code is available at https://github.com/NimQ/NimRL.git.

3.1 Environment, states and actions

The environment in the game of Nim is the board itself. A state is any permutation of the board. The number of individual states possible in a game of Nim can be calculated quite easily. For a board containing three heaps of x_1, x_2, and x_3 counters the total number of states is given by $(x_1 + 1)(x_2 + 1)(x_3 + 1)$.

Agents interact with the environment by removing counters from the board. From a heap of x counters an agent can take anything from 1 up to x counters. The agent must however only take counters from one heap in each move. The total number of actions available to the agent is therefore equal to $x_1 + x_2 + x_3$.

Unless otherwise stated the board used in this work was of four heaps with 1, 3, 5 and 7 counters in each heap respectively. The total number of states in this case is $(1+1)(3+1)(5+1)(7+1) = 384$. The total number of actions is therefore $1+3+5+7 = 16$.

According to Barto [4] rewards are regarded as part of the environment. In this work the agents received a reward of +1 if it won a game, -1 if it lost and 0 for each transitioning state during the games.

3.2 The Agent

At the beginning of training the entries in the agent's Q-table were set to zero. The agents were programmed to read the board and choose an action according to one of the algorithms given in section 2.3. If a legal action was chosen the agent performed the action and the board was changed accordingly. In both Q-learning and SARSA the policy sometimes instructs the agent to choose an action based on the state-action pair with the largest value. At the start of training when many entries in the Q-table have the same value this can lead to the agent choosing an illegal move. In such a case the move was ignored while the Q-table was updated with a value of -10 and the board remained unchanged, much like getting hit on the fingers when reaching for a cookie. The agent then chose a new valid move. In this way the agent ignored these actions in the future and as these invalid actions could never be chosen they have no bearing on the agent's ability to learn.

The reward was given to the agent after each pair of moves: the agent moved, then the opponent moved, in this way the agent saw the ultimate consequences of its actions and its Q-table was updated. If on the other hand the opponent made the first move of the game, the agent's Q-table was first updated after the opponent made its second move.

In this work there exist two criteria one can observe when looking for optimal behaviour in the agent: either the percentage of matches won or how often the agent makes a safe combination move when one is possible. A safe combination is checked for in code with the XOR operator. The XOR operator returns 0 if the heaps add up to a safe combination and 1 if not.

3.3 Training the Agent

Several methods of training the agents were looked at. These are listed in sections below.

In Nim the initial configuration of the board can leave the player who starts in an advantageous or disadvantageous position [10]. To avoid letting this become a limiting factor in the agent's learning process the player to make the first move was chosen at random. Furthermore, the random generator was seeded to ensure a better comparison of the results. In finding out which set of parameters were best for the task at hand, the agents were tested over a large number of training games for different tunings. These tunings were all possible combinations of $\alpha = \{0.1, 0.5, 0.99\}$, $\gamma = \{0.1, 0.5, 1\}$ and $\varepsilon = \{0.2, 0.8\}$ with ε either being constant, ε_c, or increasing regularly over time ε_i .

3.3.1 Agent vs. Smart

The smart opponent played a 'perfect' game. If a move was possible which left the board with a binary sum of 0, in other words a winning move, then the smart opponent would make this move. If a winning move was not possible, the smart opponent made a random move.

3.3.2 Agent vs. Random

Analogously to the method above, the agents were also trained against an opponent who made only random moves. That is, the opponent took a valid number of counters from a randomly chosen non-empty heap on the board.

3.3.3 Agent vs. Agent

In this regime, the Q-learning and SARSA algorithms underwent training against each other.

3.4 Evaluating the Agent

At regular intervals the agents were tested by competing against the smart opponent for 250 games using their latest estimations of the optimal policy. From these games the agent's winning rate and ratio of moves leaving a safe combination when possible were recorded. As mentioned in Section 2.4.1, an agent can start in a winning position. Starting in this winning position is the only chance the agent has of winning against the smart opponent. For this reason, the winning rate was defined as the number of games won to the number of games starting in such a winning position. Also from Section 2.4.1 it is known that making optimal moves will guarantee staying in winning position and eventually winning the game. Thus the ratio of winning moves was defined as the number of winning moves made to the number of times a winning move was a possible action.

Plots of the winning move ratio against number of training games were studied as a measure of performance since a value of 1 for this ratio ensures optimal moves are always taken and thus that a winning strategy has been learned.

3.5 Scaling

With the environment and agent already implemented, scaling the game was simple. With an increasing number of heaps the above mentioned training regimes were repeated. More heaps were added to the board in steps taking it from {1,3,5,7} to {1,3,5,7,9,11}. When adding more heaps to the game the total amount of training games and regular interval at which the agents were tested was made longer.

4 Results & Discussion

4.1 Parameter Selection

The parameters α, γ and ε play a very important role in the how quickly the agent finds the optimal policy. However, tuning the parameters is a very time-consuming and inexact process. The difficulty lies in finding a pattern in exactly how different combinations of the parameters affect the learning process.

Before training a presumption was made. The Nim environment is static in the sense that an optimal move will always be an optimal move. A state-action pair representing an optimal move in the Q-table should therefore keep a high value when one has been acquired. This suggests setting α rather low (for example $\alpha = 0.1$).

Considering how the Nim environment looks - the goal is to win and this is where the agent is rewarded - it would be reasonable to put emphasis on future rewards to improve early game performance. This means a $\gamma = 1$ should deliver the best results.

A low value for ε would be considered a good thing in the beginning of training to make sure all (or at least as many as possible) states are visited by the learning agent. However, later during training the agent should have encountered enough situations to have figured out the optimal strategy and exploring would now lead to making bad moves when the strategy is already known. To avoid this, ε should be increased over time to 1.

By analysing collected data trends within the parameter selection were discerned. An example of how this appeared is shown in figure 6. The best combination of parameters causes the agent to learn quickest and thereby converge to a winning move ratio of 1 fastest. Figure 7 compares the effect on convergence of a good set of parameters with a not so good set of parameters.

We found that our presumptions were somewhat verified. That is, α should be low and γ high. The parameter controlling exploration ε was found to provide best results when rather high and increasing ε_i rather than constant ε_c.

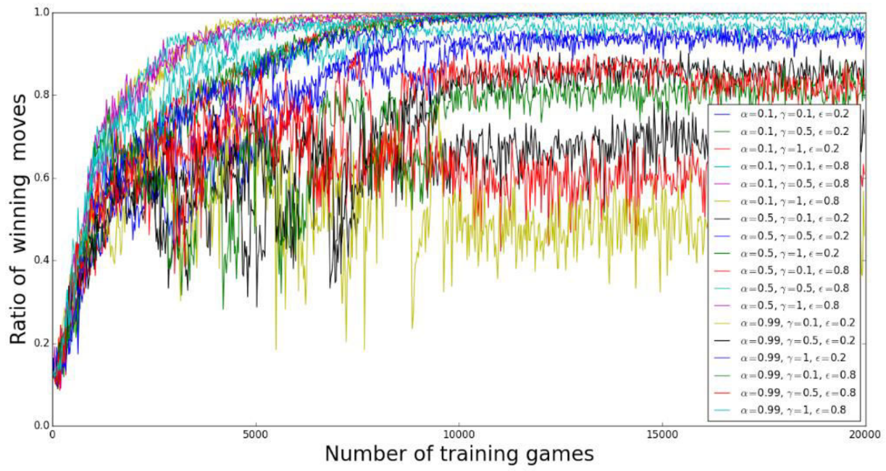

Figure 6. A plot of performance of the SARSA agent with all sets of parameters trained against smart opponent on a board of four heaps. Some groups of similar parameter sets showed better performance than others.

A further difficulty was noted when switching on or off the seed function of the random number generator. During training the random generator was seeded after initialising the agent. The idea was that agents with different combinations of parameters would face the same pattern of numbers created by the random generator. From this one can compare the performance of different parameter combinations in roughly the same situations. However, once the random generator was not seeded, noticeable differences in convergence could be seen for the same parameter combinations as illustrated in figure 8. This behaviour was observed in both the Q-learning and SARSA algorithms. This can be taken as evidence of the difficulty in tuning parameters for both these algorithms as well as suggesting that the differences between the graphs for different parameters could be due to differences in how training turns out rather than due to changed parameter values alone. One solution to this problem could be to take an average over multiple simulations of the same parameter combination. However, this would increase simulation times significantly and was not performed in this work.

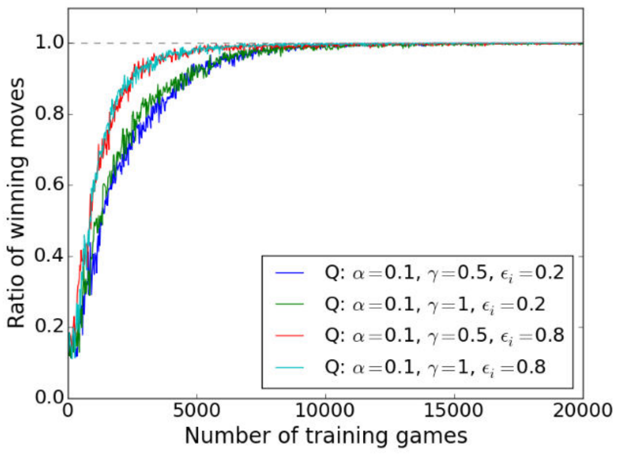

Figure 7. Graphs of Q-learning agents trained against the smart opponent on a board of four heaps using good and not-so-good parameter combinations.

As the optimal set of parameters is unknown and is laborious to find the one wonders if possibly it may be a task for an RL agent to carry out. Would it be possible to have RL agents learn to find optimal parameters?

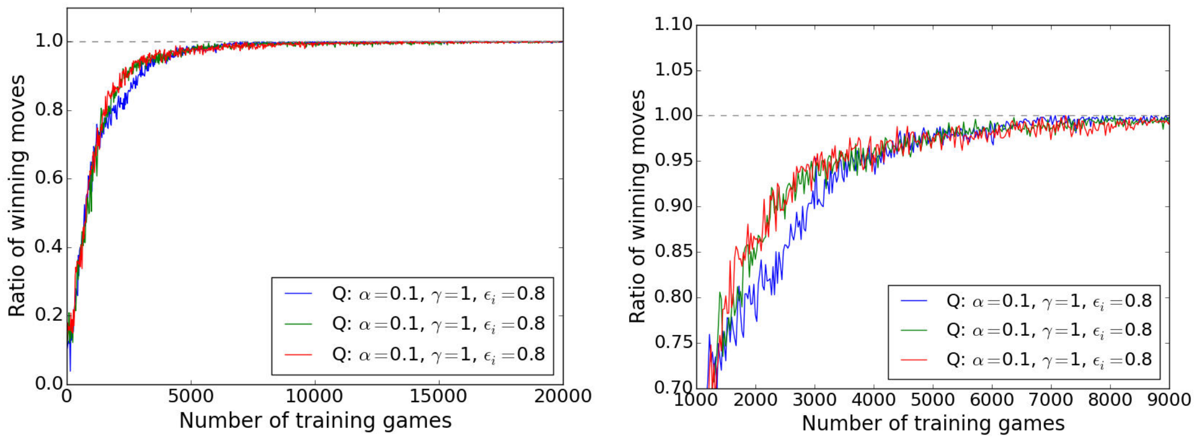

Figure 8. Graphs of Q-learning agent trained against a smart opponent on a board of four heaps using same set of parameters for three training simulations. The right plot is a close-up of the plot to the left. One can see randomness in performance between the different simulations.

4.2 Agent vs. Smart

Following the training method outlined in Section 3.3 the best parameters were found to be close to what was presumed in Section 4.1. For both Q-learning and SARSA these were α = 0.1, γ = 1 and ε = 0.8, with ε increasing to 1 after 10000 games. A comparison of the Q-learning and SARSA algorithms performance is shown in figure 9.

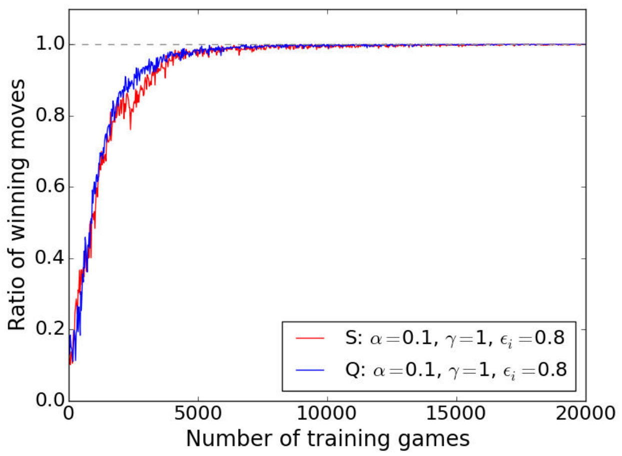

Figure 9. Plots of Q and SARSA agents' ratio of winning moves against number of training games when trained against a smart opponent on a board of four heaps.

From figure 9 one can see that convergence to 1 occurs around 10000 training games for both algorithms, although it seems that SARSA lags somewhat behind Q-learning in reaching the optimal strategy. It should be noted however that these are not the definitive optimal parameters for these algorithms rather they are the optimal parameters of all the combinations implemented in this project.

4.3 Agent vs. Random

When the RL agent was in training against an opponent making random moves it was found that performance was lower than for an agent training against a smart opponent (see figure 10). Convergence speed (if it ever learned completely) was evidently much lower. This can be explained by the nature of the environment for this task. Since the agent trained against an opponent making random moves, the agent would win many games when making non-optimal moves. Making these non-optimal moves and still winning, the agent would propagate positive reward to non-optimal moves, and thus miss the optimal strategy. The agent would be confused as to why it was rewarded. Most noticeable was that the Q-learning agent almost reached convergence, but with a lower value of ε than in the agent vs. smart regime, meaning much more exploration of the state space was needed.

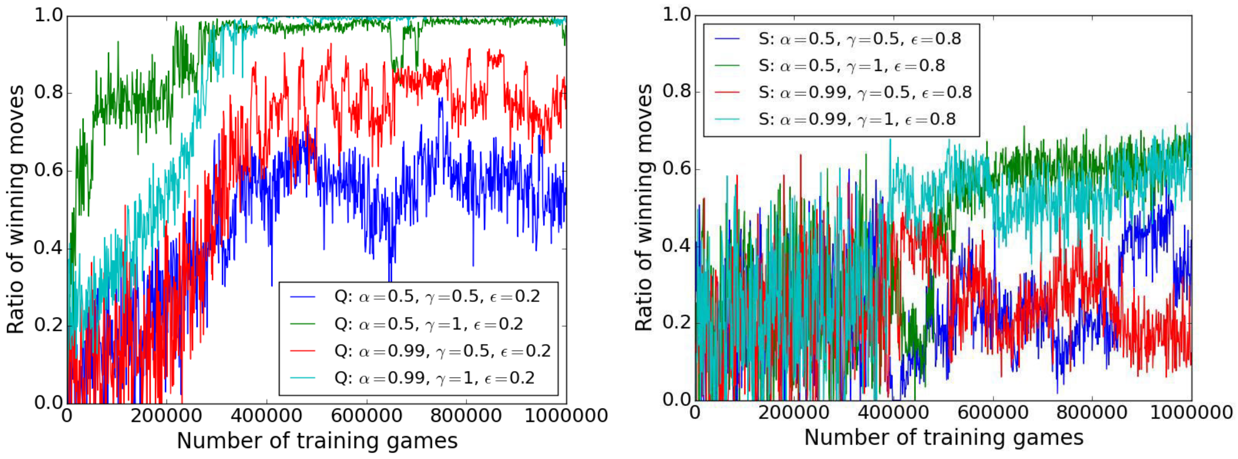

Figure 10. Performance of Q-learning (left) and SARSA (right) agents with different parameter sets (ε reaches 1 at 400000 training games) trained against an opponent making random moves. Convergence toward optimal behaviour could be seen for some of the parameter sets for the Q-learning agent. SARSA did not converge to optimal behaviour.

Unfortunately, this training regime did not provide satisfactory levels of performance on a board of four heaps. Consequently, this training regime was discarded when scaling up the number of heaps.

4.4 Agent vs. Agent

A comparison of the behaviour of a SARSA agent and a Q-learning agent trained by playing against each other and tested against the smart opponent is shown in figure 11. Both agents reached convergence after approximately 8000 training games. From this, one cannot observe a large difference in performance of the algorithms.

A very interesting behaviour occurred in this training regime. In contrast to figure 9 in Section 4.2 Agent vs. Smart, the agents winning move ratio no longer fluctuated below 1. When reaching optimal behaviour the agents continued making optimal moves when possible for the rest of the simulation. The agents learned better in this regime of playing against an opponent who was also in the process of learning the game than in both previous regimes.

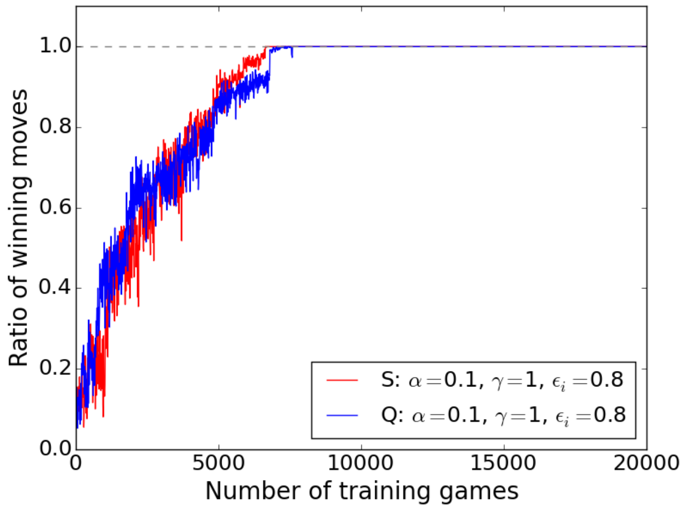

Figure 11. Performance of Q and SARSA when trained against each other in a game of four heaps using the same set of parameters (ε reaches 1 at 10000 training games). Both agents learn optimal behaviour in roughly the same time. Once optimal behaviour is learned both agents continue playing optimally.

4.5 Scaling

Scaling up Nim to larger boards increased an agent's learning time. This is illustrated for boards of five and six heaps respectively in figure 12, showing plots of performance of both the Q-learning and SARSA agents trained against the smart opponent. In this regime, on a board of five heaps, the Q-learning agent converged to a winning move ratio of 1 at approximately 13000 training games. SARSA was a little slower, just as in the case of four heaps, and converged to 1 at approximately 17000 training games. On a board of six heaps, SARSA climbed toward the optimal strategy with a gentler slope, yet settled at a winning move ratio of 1 before the Q-learning agent which converged more quickly but continued to fluctuate slightly. For both agents optimal behaviour occurred at approximately $1.5 \cdot 10^6$ training games. For a better perspective on increased learning time with added heaps, the convergence for four, five and six heaps respectively is plotted in figure 13.

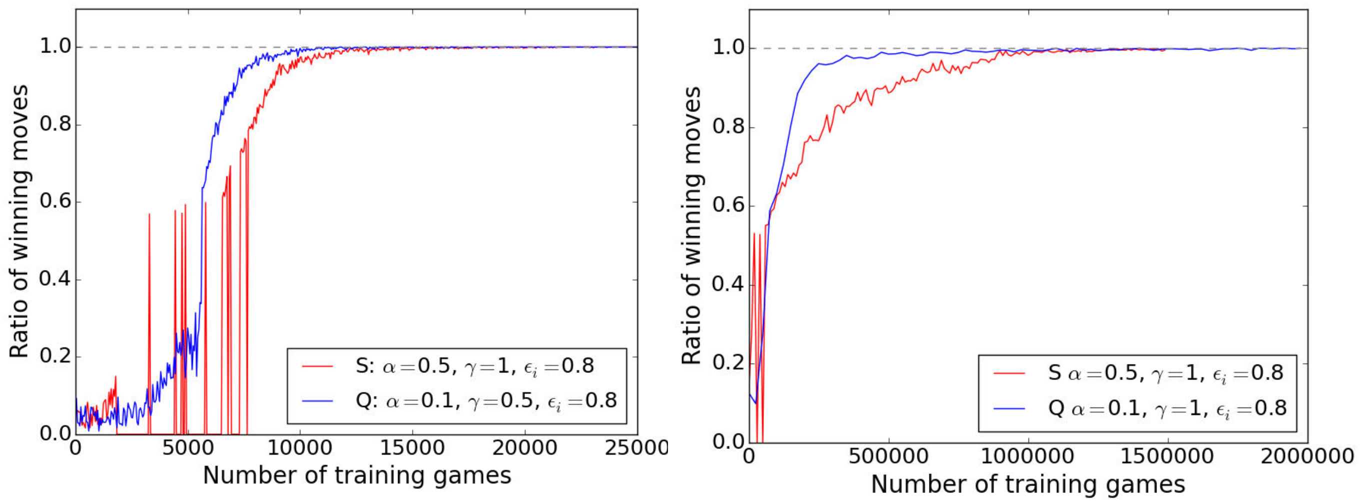

Figure 12. Q and SARSA agents vs smart on boards of 5 heaps (left) and 6 heaps (right).

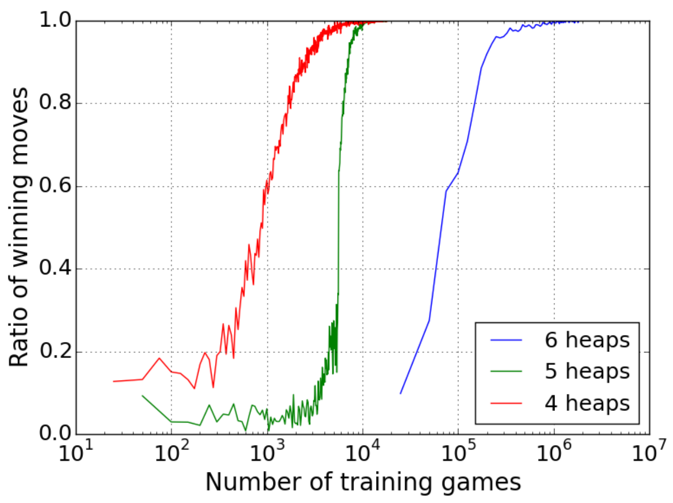

Figure 13. Plots of Q-learning vs. Smart for boards of four, five and six heaps showing ratio of winning moves plotted against number of training games on a logarithmic scale. Convergence time is much higher on a board of six heaps.

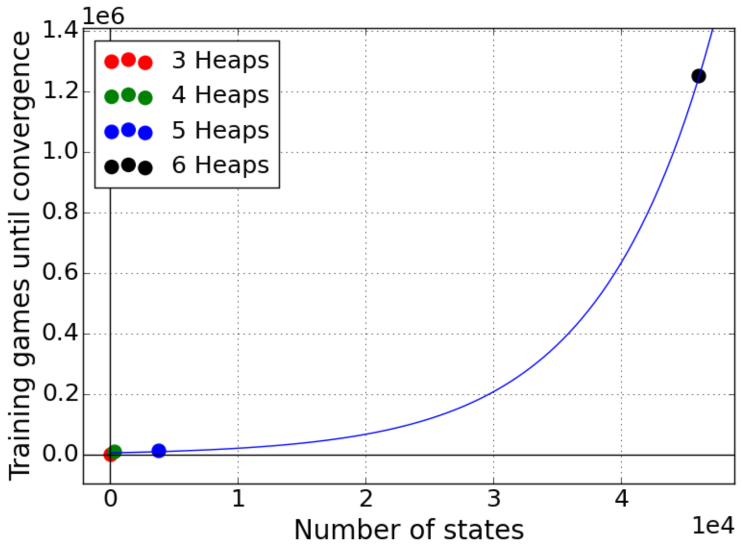

Figure 14. Plot showing the exponential relation between number of states and number of games required for an agent to converge to a winning move ratio of 1.

In order to study the trend of training time with respect to number of states, data was added from a simulation of Q vs. Smart on a board of three heaps of one, three and five counters. Convergence on this board of three heaps was only on the order of several hundred training games. The data from the number of states per board plotted to the number of training games required to reach optimal performance was found to agree with an exponential curve (see figure 14). This suggests that the agents quickly become rather ineffective in finding the optimal strategy when the game is scaled up.

Training Q-learning agents against SARSA agents on boards of more heaps showed similar results to those of Q vs. Smart. Convergence time increased with added heaps and quickly became inefficient. Interestingly, in some cases one could for a board of five heaps see convergence time of one agent very close to that of an agent learning on a board of four heaps. In these cases the other agent's performance was lower, see plot for five heaps in figure 15. This suggests that possibly, agents may be coupled so that one learns faster while the other works as a catalyst. However, these results were clearly subject to how the training sessions turned out. As mentioned in Section 4.1, the sequence of random numbers can affect the performance of the agent under its training period. Running the training again with a different value for the seed in the random number generator at times resulted in SARSA outperforming Q-learning. Hence, it is difficult to draw conclusions on the effects of scaling in this regime.

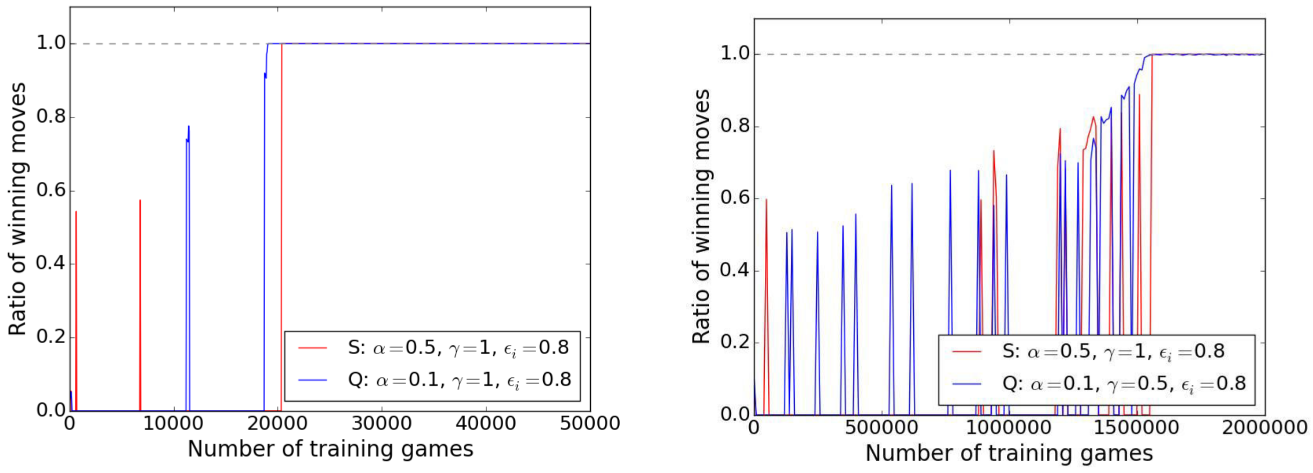

Figure 15. Q and SARSA in training against one another on a board of five heaps (left) and six heaps (right). Parameters for both agents are the best obtained from training against a smart opponent. Both agents converge to optimal behaviour. On the five heap board (left plot) the Q agent converges fast while the SARSA agent converges slower.

As in the case of boards of four heaps, once the optimal strategy was found the agents continued playing with a 100% optimal move rate. However, a peculiar behaviour could be noted for both algorithms when training against each other on boards of 5 and 6 heaps. At times an agent's performance fell drastically to a ratio of 0 optimal moves before suddenly rising towards convergence. The exponential behaviour seen in figure 14 can be explained by examining the number of entries in the Q-table as the number of heaps on the board increases (see table 1). Nim is a simple game with a strategy that an RL agent can learn, yet the seemingly trivial action of adding extra heaps to the board quickly poses problems in terms of computational time required and storage of large matrices in a computer's memory.

A possible remedy to decrease the time taken to convergence when adding extra heaps could be to use state-action values from an agent who has already undergone training on smaller boards and knows the optimal strategy. Optimal moves will always be optimal moves regardless of the number of heaps on the board. In other words training the agent in stages may speed up the learning process.

Table 1. Increase in size of Q-table corresponding to size of board.

Board	**States**	**Actions**	**Entries in Q-table**
1,3,5,7	384	16	6144
1,3,5,7,9	3840	25	96 000
1,3,5,7,9,11	46 080	275	$1.27 \cdot 10^7$
1,3,5,7,9,11,13	645 120	3575	$2.31 \cdot 10^9$
1,3,5,7,9,11,13,15	10 321 920	53 625	$5.5 \cdot 10^{11}$

4.6 Other Observations and Remarks

A known problem in RL is where during a training episode one should assign credit to the agent [7]. During the initial phase of programming, the Q-table was updated after the agent had completed its move but before its opponent's move. We found however that the agent was not learning optimally and would win at most around half of its matches, even against a naive random player. This result led us to instead let the agent see the consequences of its actions and update the Q-table after the opponent's move. This slight alteration enabled the agent to learn the optimal strategy.

It is clear that the RL agents can learn the optimal strategy of Nim, although performance differs between the training regimes. When faced with an opponent using the optimal strategy the agents soon learned which actions are optimal. However, it can be argued how representative of a relevant RL problem this regime is since the optimal strategy is already known and implemented, whereas RL has been classified as suitable for solving problems with unknown solutions. Yet, it has been shown that RL agents applied to problems with known solutions can discover completely novel solutions [6]. Additionally, RL can be relevant when a solution is known, but difficult to implement as in the case of the pancake flipping robot [5]. These examples suggest the Agent vs. Smart regime might be of some relevance.

For problems with completely unknown solutions the other two training regimes are more relevant since in these regimes both players start with no knowledge of how the game is won.

Unfortunately, the results of the agent vs. Random regime did not provide satisfactory levels of performance. However, when letting Q and SARSA undergo training against each other it was found that performance was even better than when training agents against the smart opponent. This result is very pleasing since it shows that two RL agents can be played against each other to find the optimal strategy, even if it were previously unknown.

In the cases of training and evaluating the performances of agents in the Agent vs. Agent regime with more than four heaps it seems like the agent suffers from something akin to ‘amnesia’ (see figures 12 and 13). This may be due to the large state space and many state-action pairs having the same value in the Q-table while at the same time learning against an opponent making occasionally sub-optimal moves. It is possible that after having learned a somewhat good strategy an agent continues to explore when it should have made an optimal move. In this way it overwrites and reduces the value of what previously were good actions to take while in a given state.

As mentioned in 1 Introduction, RL is based on behaviourism [1]. With the results in mind it can be questioned if this really is a good model of animal learning. It is true that this works for a computer program. However, the human brain is a very complex organism. RL may be seen to be applicable as a model for simple cases: for example teaching a very young child how to behave or learn to ride a bike. In terms of more complicated tasks such as learning a musical instrument it is harder to see where the immediate rewards or even long term rewards fit in such a scenario. RL agents demanded a very large number of training games for learning an optimal strategy when scaling up. However, when learning a complicated task or skill, humans also require a large amount of practice.

5 Conclusion

It is completely feasible for an RL agent to learn the optimal strategy of Nim in all three training regimes treated in this paper. This verifies previous work [11] and expands on it with the use of SARSA as an RL algorithm applied to an agent learning Nim.

Tuning parameters was difficult but the initial presumption of low α and high γ was confirmed while it was found that ε should take a rather high value.

Training against a smart opponent was initially thought to be the best way for the agents to learn the winning strategy. However, this training regime is not very relevant to problems with no previously known solution, although it might be of relevance to problems for which a solution is known but too complicated to implement or for finding alternative solutions to problems. Nevertheless, it turned out that when training Q-learning agents against SARSA agents learning was not only faster, the agents also performed with higher stability once the optimal strategy was learned. This training regime is more representative for problems with no previously known solution. However, this result was extremely sensitive to how games were played during training and the results were clear only on a board of 4 heaps.

The *curse of dimensionality* proves itself to be an apt term for what happens with Nim as the number of heaps is increased. The exponential growth of the state space makes the learning time for an agent unfeasibly long. For this reason, it may be worthwhile to carry out research into training an agent in stages, using Q-tables from smaller dimensional problems and building on this previous knowledge when scaling up.

6 References

[1] Beierholm, U. R. (2014). Bayes Optimality of Human Perception, Action and Learning: Behavioural and Neural Evidence. In *Brain-Inspired Computing* (pp. 117-129). Springer International Publishing.

[2] Beck, J. M., Ma, W. J., Kiani, R., Hanks, T., Churchland, A. K., Roitman, J., ... & Pouget, A. (2008). Probabilistic population codes for Bayesian decision making. *Neuron*, *60*(6), 1142-1152.

[3] Berkes, P., Orbán, G., Lengyel, M., & Fiser, J. (2011). Spontaneous cortical activity reveals hallmarks of an optimal internal model of the environment.*Science*, *331*(6013), 83-87.

[4] Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press.

[5] Kormushev, P., Calinon, S., & Caldwell, D. G. (2013). Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, *2*(3), 122-148.

[6] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *38*(3), 58-68.

[7] van Otterlo, M., & Wiering, M. (2012). Reinforcement learning and markov decision processes. In *Reinforcement Learning* (pp. 3-42). Springer Berlin Heidelberg.

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533.

[9] Grundy, P. M. (1938). Mathematics and games. Eureka, 2(6-8)

[10] Bouton, C. L. (1901). Nim, a game with a complete mathematical theory. *The Annals of Mathematics*, *3*(1/4), 35-39.

[11] Järleberg, E. (2011). Reinforcement learning on the combinatorial game of Nim.

[12] Gosavi, A. (2009). Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, *21*(2), 178-192.

[13] Nissen, S. (2007). Large scale reinforcement learning using q-sarsa (λ) and cascading neural networks. *Unpublished masters thesis, Department of Computer Science, University of Copenhagen, København, Denmark*.