DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2016

Deep reinforcement learning compared with Q-table learning applied to backgammon

PETER FINNMAN

MAX WINBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Computer Science

Deep reinforcement learning compared with Q-table learning applied to backgammon

Finnman, Peter Winberg, Max

May 13, 2016

DD143x - Bachelor's Thesis in Computer Science at CSC
Supervisor: Pawel Herman
Examiner: Örjan Ekeberg

Abstract

Reinforcement learning attempts to mimic how humans react to their surrounding environment by giving feedback to software agents based on the actions they take. To test the capabilities of these agents, researches have long regarded board games as a powerful tool. This thesis compares two approaches to reinforcement learning in the board game backgammon, a Q-table and a deep reinforcement network. It was determined which approach surpassed the other in terms of accuracy and convergence rate towards the perceived optimal strategy. The evaluation is performed by training the agents using the self-learning approach. After variable amounts of training sessions, the agents are benchmarked against each other and a third, random agent. The results derived from the study indicate that the convergence rate of the deep learning agent is far superior to that of the Q-table agent. However, the results also indicate that the accuracy of Q-tables is greater than that of deep learning once the former has mapped the environment.

En jämförelse mellan deep reinforcement learning och Q-tabeller i spelet backgammon

Sammanfattning

Reinforcement learning härmar hur människor reagerar på sin omgivning genom att ge en sond återkoppling baserat på handlingarna som den tar. Forksare har sedan länge testat sondernas prestanda genom att låta de anpassa sig till brädspelsmijöer. Med hjälp av brädspelet backgammon jämför denna avhandling två stycken tillvägagångssätt till reinforcement learning, Q-tabeller och deep reinforcement learning. Det framgick tydligt vilket tillvägagångssätt som var överlägset det andra med tanke på inlärningshastighet samt träffsäkerhet. Algoritmerna utvärderades genom att träna sonderna med hjälp av självinlärning. Efter varierande mängder träningsepisoder spelade sonderna dels mot varandra och dels mot en tredje, slumpmässig sond. Resultaten antyder att deep learning sonden har mycket snabbare inlärningshastighet än Q-tabell sonden. Dock har Q-tabellerna större träffsäkerhet när den har fått tillräckligt mycket återkoppling från sin omgivning.

Contents

1 Introduction

Reinforcement learning is a general-purpose approach to artificial intelligence. In this paradigm, agents (software probes that act on behalf of a program) explore an environment and attempt to optimize the program based on rewards from that environment. A reinforcement learning method is defined as any method that is suitable for solving a learning problem, which sets it apart from supervised learning since a correct mapping between input and output is never presented [1]. There is prodigious value in exploring reinforcement learning. It does not only provide us with an avenue for better understanding of learning in nature, but has potent practical applications as well [6][22]. Thanks to recent advances in deep learning, resulting in improvements of both speech recognition and computer vision, the subject is more relevant than ever. Ever since the inception of machine learning, board games have been regarded as a strong test domain for benchmarking algorithms [10][1].

Reinforcement learning takes inspiration from behaviourist psychology in attempts to mimic how humans learn, by so called "learning by doing". Alternatively, it can be described as an approach to sequential decision problems, where an agent is faced with multiple decisions that contribute to a cumulative reward [12]. Q-learning is an approach to reinforcement learning based on finding the optimal action-selection policy for an environment. The simplest form of Q-learning stores state-action pairs in tables and assigns a value to each pair. This value represents how favourable the action is in this particular state. By combining the principles of Q-learning and deep neural networks (DNNs), it is possible to generalize the environment and optimize agents suited for solving tasks within that environment. It is valuable to explore this paradigm as it is a different approach to artificial intelligence, and takes us one step further when it comes to understanding the cognitive decision making of humans and how we react to our environment [19]. Previous attempts at learning board games include using neural networks with the sigmoidal logarithmic function and backwards propagation for learning [13]. More recent attempts include value networks to assess values of board positions and policy networks in order to make moves [6].

This thesis explores how deep reinforcement learning compares to Q-tables on complex decision problems that contain stochastic elements. It is particularly compelling to study the magnitude of recent algorithmic improvements in the field, and if Q-tables has any advantages over deep reinforcement learning.

1.1 Problem statement

This report studies two different methodologies for reinforcement learning. In this process we consider a board game with a stochastic component (backgammon, where players roll dice). The first approach examined is classical Q-tables, which determines a future optimal score by analyzing a Markov decision process. In contrast, the second approach operates with deep reinforcement learning which trains a network instead of using a look-up table. Therefore, our problem statement is as follows:

"How does classic Q-table learning compare to deep reinforcement learning in environments with stochastic components (backgammon) in terms of convergence rate and results?"

The intention is to judge the two methodologies on learning speed and the score, based on a head-to-head match-up of the two agents.

1.2 Scope

Q-table learning and deep reinforcement learning are the techniques covered by this paper. The two algorithms learn the game of backgammon by using self-play training. In self-play training, the agent sets up two instances to play against each other. After each state transition the algorithms update the weights of their lookup tables and networks based on encountered values. This approach differs from traditional reinforcement learning as it does not require any pre-training.

This study is limited by a variety of factors, such as the size of the game and the algorithms themselves. Q-tables are dependant on a lookup table to determine which moves are favourable in any given situation. An optimal move in backgammon depends on a plethora of variables including the position of the player's checkers, the opponent's checkers and the rolls of each of the dice. We focus solely on strategy in single matches. Therefore, certain parameters of the game have been eliminated or reduced. Alterations to the environment are elucidated following a brief introduction to backgammon in section 3.1. Due to time constraints, the analysis briefly glances at how the algorithms relate in alternative settings, such as larger environments.

1.3 Purpose

The potential learnings of machine learning transcend trivial problems such as board games. However, that does undermine the usefulness of understanding how state-of-the-art approaches to artificial learning compare to more outdated ones. It is indeed valuable to get a sense of direction from the past to better understand the future. The purpose of this report is to compare two algorithms that approach learning differently. More specifically, this thesis examines how deep reinforcement learning outperforms Q-table learning and if there are any indications of what the future may bring.

1.4 Thesis Outline

The following section serves to acquaint the reader with the rules of backgammon as well as the algorithms studied in this thesis. In addition to this, an overview of the history of machine learning within board games provides the context of our work. The third section introduces the methodology of building an environment and agents as well as collecting results. The results are presented in section 4 and discussed in section 5. Finally, we summarize and conclude the study in section 6.

2 Background

There are certain prerequisites in order understand and develop learning algorithms for board games. Initially, this section delves into the intricacies of backgammon, giving a general overview of the rules. Furthermore, the section provides an explanation of Q-table based learning and how the decision-process takes future states into account when evaluating moves. It is important to fully grasp the concepts of Q-table learning in order to follow the shift towards deep reinforcement learning. This section aims to familiarize the reader with backgammon, Q-tables, deep reinforcement learning and their past applications to board games.

2.1 Backgammon

Backgammon dates back to ancient Egypt circa 5000 years ago and is played by two players. The board consists of 24 board positions, called *points*, represented by triangles divided into 4 equal quadrants. Each player starts out with 15 game pieces, called *checkers*, that are placed according to figure 1. The checkers are moved towards descending numbers. The numbers for the white player are reversed in this case. Point 1 is considered black's home board and point 24 is considered white's home board. The objective is to move checkers toward the home board of the color you are playing. Once all of the player's checkers are confined in the home board, a player can start *bearing off*. Bearing off is the process of removing the checkers from the board by rolling the number which corresponds to the point of the checker. A player wins a game by bearing off all of their checkers.

The game is played with two dice and the numbers rolled correspond to the moves to be made. The moves represented by a die cannot be split and must be taken by a single checker. Furthermore, a checker cannot be moved to a point where the opponent has 2 or more checkers. Additionally, if the opponent has a single checker at a point (also known as a *blot*) it is possible to *hit* that checker by moving a checker to the same point. This puts the checker onto the *bar* which is a strategical advantage. The bar can be seen as the 0-point for each player. Checkers on the bar must be moved before any other move becomes legal.

There is a third die called the *doubling die* which displays the current stake of the game. At the beginning of a player's turn, he may propose to double the stakes. Their opponent can either refuse or accept doubling of the stakes. If a player refuses a proposed doubling, they concede the current game. Although the doubling die only displays up to 64, there is no limit to the stakes of the game. Lastly, if a player loses without bearing off a single checker he is *gammoned* and loses twice the amount shown on the doubling die. If he has any checkers on the bar or in the winner's home board he is *backgammoned* and loses three time the amount shown on the doubling die.

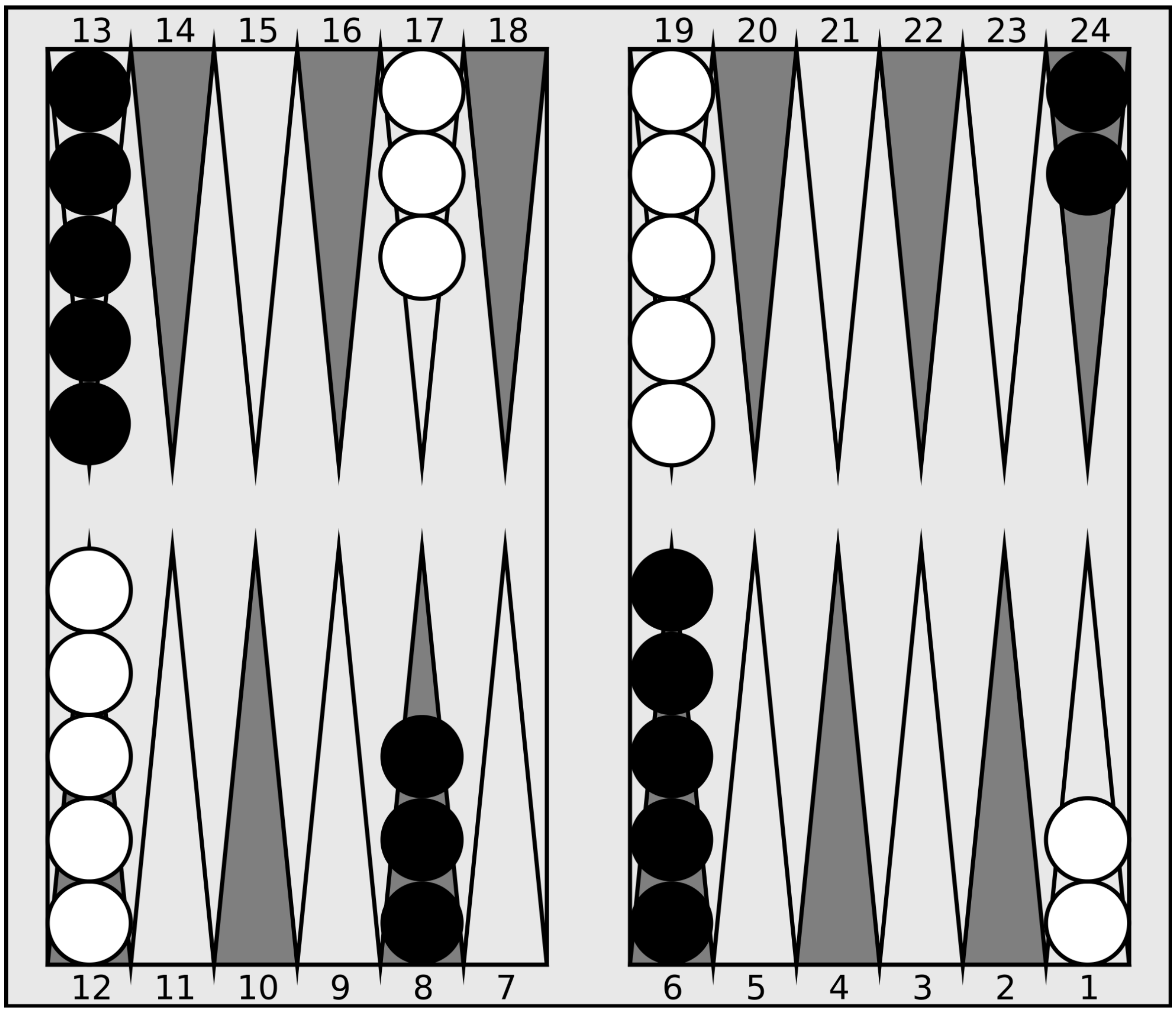

Figure 1: Starting positions for backgammon.

2.2 Application to board games

Early studies conducted, comparing deep reinforcement learning and highly specialized networks in the game of checkers, revealed the possibility to devise learning schemes that can outperform humans [1]. On July 15, 1979, Berliner's backgammon computer program BKG9.8 beat the reigning world champion Luigi Villa in an exhibition match to 7 points. BKG was based on finely tuned heuristic functions containing hand-crafted features to resemble and counter expert level strategy [11][8]. Playing against BKG, Berliner would ask it for the best move while having a move of his own in mind. Occasionally the agent would suggest a move superior to Berliner's, something he considered to be an indication of the potential in machine learning. However, BKG did not learn the features that allowed it to play at world class level. It was never addressed whether the methodology used in BKG could be learned [11].

Tesauro and Sejnowski[8] applied the back-propagation algorithm to train a feedforward neural network based on a data-base of "expert" positions and moves [4][5]. Even without look-ahead the neural network was able to achieve levels matching intermediate level of play. A decade thereafter, the same algorithm trained through self-play starting from random action selection managed to reach expert levels of play. When combined with shallow look-ahead (2-ply and 3-ply) as well as a stake-doubling algorithm the agent surpassed even the best human players [10][7].

Since reaching superhuman levels of play in backgammon, researchers turned their attention to games of greater complexity such as chess and GO. In these games exhaustive search is infeasible. Therefore, in order to limit the search area two general principles are applied. The optimal value function is approximated by truncating the search tree and then using Monte Carlo tree search to assess the value in each state. This approach has been proved to converge on optimal play [6][10]. In March, 2016 a GO agent called "AlphaGo" played against South Korean GO grandmaster Lee Sedol and won the series 4-1. All things considered, it may be more impressive that a human player can match the agent in a single game.

In the realm of simple computer games Mnih et al.[22] managed to develop an agent based on a deep Q-network (Q-learning applied to DNNs) that could learn Atari 2600 games from raw sensory data (pixels and game score in this case)[2]. The agent was able to reach a level comparable to that of a professional human games tester across a set of 49 different games. Generalized learning is the very essence of artificial intelligence and this agent serves as a first stepping stone in that direction.

2.3 Q-table learning

Any game of backgammon can be visualized as a sequence of states, actions and rewards. In each state a participating player takes an action that leads to a reward. The total reward of a game of backgammon is the sum of rewards that the actions induced,

$$R = r_1 + r_2 + \ldots + r_n \tag{1}$$

where r_1 is the reward for the first action a_1 and so on. Subsequently, we can calculate the total future reward R_f in a state t as

$$R_f = r_t + r_{t+1} + \ldots + r_n. \tag{2}$$

Due to the fact that backgammon games are non-deterministic we discount future rewards exponentially and instruct the agent to value immediate rewards above future rewards [17]. Introducing a discount factor γ to equation 2, the future reward in any given state s_t is

$$R_f = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-t} r_n. \tag{3}$$

If we denote $maxR_f$ (the maximum future reward function based on a state and an action) as $Q(s,a)$ and factorize the discount factor, the iterative nature of the function becomes apparent. The Q-function can be approximated as

$$Q(s_t, a^\pi) = r_t + \gamma Q(s_{t+1}, a^\pi), \tag{4}$$

where π is the policy of selecting the actions that lead to the maximum future reward. This equation is also known as the *Bellman Equation.* Using the Bellman equation, we can iteratively approximate the true Q-function [20]. Applying the theory behind Q-learning to backgammon, it is reasonable to expect the approximate Q-function to be completely wrong initially. As we iterate, the approximate function converges on the true value of the Q-function. In each state (turn) of backgammon we evaluate the reward for the next state. Environment parameters that affect reward includes opening and covering 'blots' as well as grouping of checkers (as it is advantageous to move the checkers together instead of having them spread out).

A Q-table stores state-action pairs and assigns them an initial value. As backgammon rounds are played, the Q-table is traversed and updated.

$$\textit{Q-table} = \begin{pmatrix} Q(1,1) & Q(1,2) & \dots & Q(1,m) \\ Q(2,1) & Q(2,2) & \dots & Q(2,m) \\ \vdots & \vdots & \ddots & \vdots \\ Q(n,1) & Q(n,2) & \dots & Q(n,m) \end{pmatrix} \tag{5}$$

In the matrix representing a Q-table in equation 5, there are m possible actions for each of the n states. In a state transition the previous state is updated based on the Q-value of the present state according to equation 12 presented in section 3.3.

2.4 Deep reinforcement learning

Neglecting the integration with reinforcement learning momentarily, deep neural networks can be described as a stack of representation layers. The first layer represents the input from the problem at hand, while subsequent hidden layers abstract that information progressively until the output layer. The output layer is the neural network's solution to the given problem. The enacted abstractions are not explicitly constructed by the engineer. Instead, the network of abstraction-nodes rearrange its weights based on performance, whilst evaluating training data, through back-propagation. The back-propagation procedure starts by computing the gradient at the output layer and then propagates backwards, using the chain rule for partial derivatives [4]. The weights can then be adjusted according to the gradient of each layer [25]. The technique is alluring to researchers due to its wide variety of applications [16][18][21].

As stated previously, deep learning networks rearrange their weights while training on datasets. This procedure is triggered by giving the network feedback as it selects actions deemed optimal. By replacing the Q-table (explained in section 2.3) with a deep learning network you trade accuracy for abstraction. The network needs to be tuned in order to operate efficiently. Hyperparameters such as the number of hidden layers and nodes as well as the learning rate of the network needs to be set. Increasing the number of hidden layers and nodes therein increases the learning capacity of the network [14]. Tuning of learning rate and network architecture is presented in section 3.4. The studies conducted in this thesis utilize a feedforward neural network, characterized by one-directional, forward flow of information.

Figure 2 shows an example of how neural networks can be applied to backgammon. The network is not a DNN since these are defined to have two or more hidden layers.

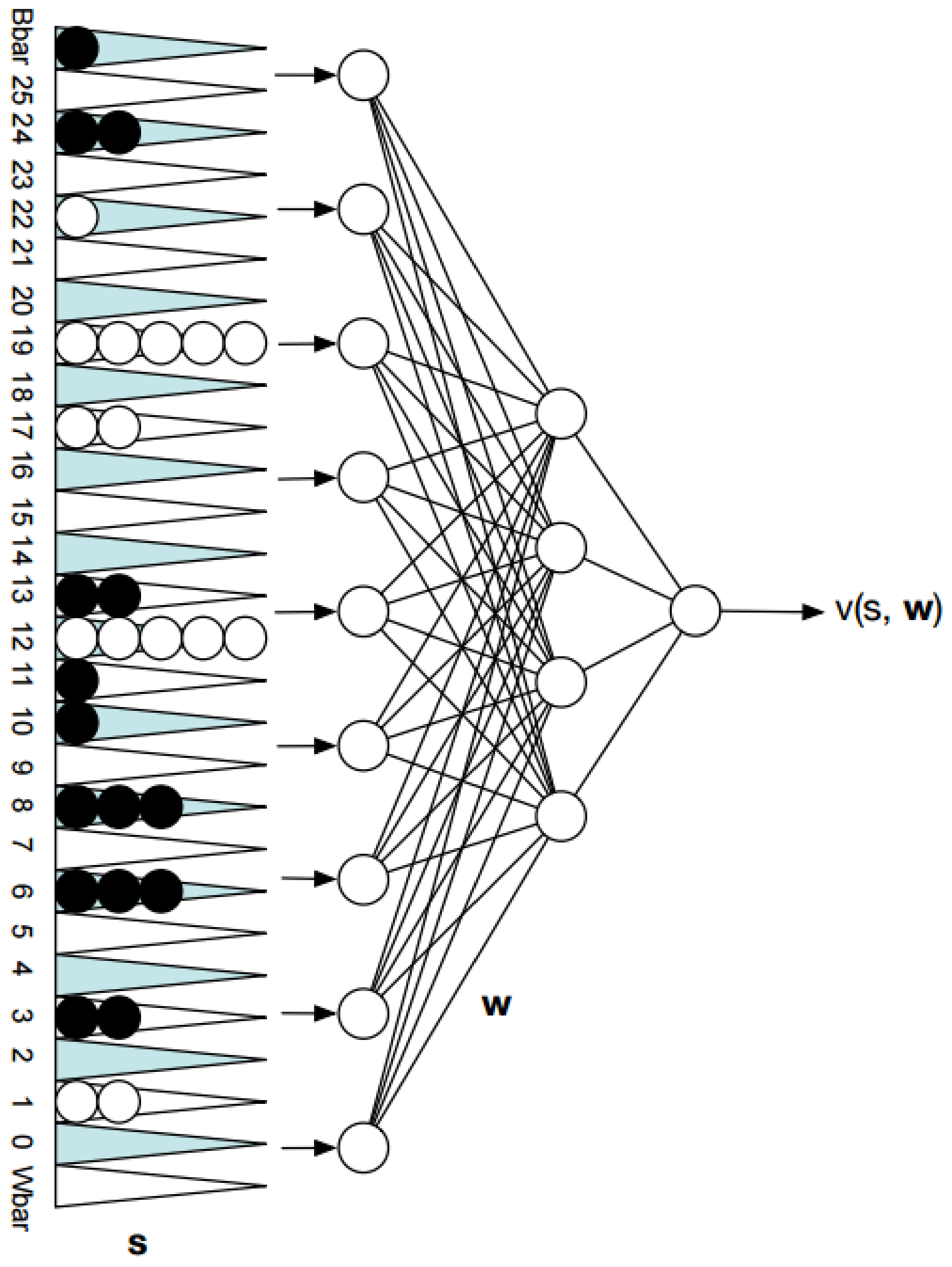
Bbar 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Wbar
s
w
v(s, w)

Figure 2: Applying neural networks to backgammon. The board positions and the encompassed checkers are passed to the input layer. In between the input layer and the output layer is one hidden layer consisting of four nodes [5].

2.5 Exploration versus exploitation

In the realm of reinforcement learning, exploration is the term used to describe the extent to which an agent seeks alternate solutions to a problem. In contrast, exploitation describes the degree that the agent selects actions identified as optimal. An agent that always selects the optimal action in terms of reward is likely to fail in discovering the optimal strategy. Thus, there is a need for selecting actions which are seemingly suboptimal, in essence, exploring the environment [3]. In certain scenarios it is paramount to select a suitable exploration strategy. In the case of an aerial glider agent that relies on energy-gaining flight trajectories, exploration has a direct impact on the distance the glider is able to travel [15].

A simple exploration strategy is the ϵ-greedy strategy, where there is a set parameter ϵ such that

$$a = \begin{cases} random(a), & if \quad x \leq \epsilon \\ max_a Q(s,a), & if \quad x > \epsilon \end{cases}, \quad where \quad 0 \leq \epsilon \leq 1 \tag{6}$$

which tunes how often we select actions at random. After the agent has received dice rolls it retrieves a random value $x, 0 \leq x \leq 1$. As equation 6 suggests, if $x \leq \epsilon$ then the agent selects an action at random, otherwise it performs the action that has the largest Q-value.

Another approach to exploration is to take into account the relative values of the actions in a given state. Boltzmann selection involves probability relative to the action value. The probability p of selecting an action a can be calculated as

$$p = \frac{e^{V(s,a)}}{\sum_a e^{V(s,a)}}, \quad V(s,a) = \frac{Q(s,a) - max_b Q(s,a)}{T} \tag{7}$$

where the temperature T, decides the significance of differences in state-action values. Higher temperatures entail equiprobability between actions while lower temperatures distinguishes more between the values of actions [19]. Other approaches actively prioritize exploration of unknown parts of the state space [15].

2.6 Benchmarking

Traditionally, agents learn tasks by training on data-sets with dynamic weights. Once the agents receive feedback from the environment they adjust their weights to better approximate the Q-function. Benchmarking progress of the agents can be done during training or by copying the weights into a static instance and playing the agent against a static benchmark. The former, referred to as generalized policy iteration, implements an additional process in the training algorithm called a critic. The critic evaluates the agent's current policy, the current approximation of the Q-function [3]. However, in this thesis we concern ourselves with the latter. The advantage of static benchmarking is the uncomplicated procedure of extracting progress information. Additionally, it is easier to diagnose an algorithm with static benchmarking. In section 3.2, benchmarks are proposed and rationalized.

3 Method

In this section we explain the method used for this thesis. Initially, we explain how we change the environment and how each alteration impacts game-play. Next, we consider setting up training and our approach to testing our problem statement.

3.1 Backgammon environment

Backgammon is an infinitely complex game incorporating many variables that determine whether a strategy is successful or not. Q-table learning relies on a lookup table consisting of states and possible actions. Unfortunately, this imposes restrictions on the environment as each variable exponentially increases the size of the lookup table. Our goal is to downsize the environment while still maintaining the core integrity of the game.

3.1.1 Environment reduction

The amount of states on a backgammon board can be calculated by the combination of checkers over the board positions. Next, this number is multiplied by the number of unique dice rolls to obtain the matrix that represents the lookup table. This gives us

$$S(p, n, a) = a \times \binom{p+n-1}{p-1}^2 = a \times \left(\frac{[p+n-1]!}{[p-1]! \times n!} \right)^2 \tag{8}$$

unique state-action pairs, where p, n, a are the number of board positions, checkers of each player and actions respectively. Allow us to consider the backgammon board depicted in figure 1. Viewing the bar as 2 board positions (essentially position 0 for both players) we have a total of 26 board positions. If we disregard rules restricting board positions of checkers we have a 30 checkers (15 for each player), spread out over the 26 board positions. Given the 21 unique dice combinations we end up with

$$21 \times \binom{26+15-1}{26-1} = 21 \times \binom{40}{25}^2 = 21 \times \left(\frac{40!}{15! \times 25!} \right)^2 \approx 34 \times 10^{21} \tag{9}$$

states, or in other terms, 34 sextillion unique state-action pairs. To restrict the amount of unique states we decided to use 9 board positions (discarding the bar), 12 checkers (6 checkers for each player) and 3-sided dice (6 dice combinations). This amounts to

$$6 \times \binom{9+6-1}{9-1}^2 = 6 \times \binom{14}{8}^2 = 6 \times \left(\frac{14!}{8! \times 6!} \right)^2 \approx 54 \times 10^6 \tag{10}$$

states. 54 million states is a clear improvement and results in more rapid computation rates.We limit the states further by applying the *Egyptian rule* only allowing 3 checkers to be present on any point. In addition to boosting the rate of computation, we also preserve expert tactics such as *blockade* (preventing opponent's checkers from moving by occupying sequential points).

Further reduction has been done by removing the bar. When a checker is hit, it is now moved to the first position on the opponent's home board that is vacant. Vacant in this case denotes that there are no opposing checkers on the point. Additionally, the point is not occupied by three of the agent's checkers. Otherwise, moving the hit checker to this position would be in violation of the Egyptian rule. Lastly, we remove stakes with the motive that agents are learning to win single games of backgammon, nullifying the effect of stakes.

3.1.2 Representation of reduced environment

The reduced environment is represented in vector form containing nine integers. Each integer represents which player the checkers belong to and the number of checkers on the position. Positive integers represent the current player's checkers while negative represent the opponent's checkers. For instance, a state in the environment could be represented as

$$\{-1, 0, 3, -2, 0, 2, -3, 0, 1\}. \tag{11}$$

This particular state is also the starting state of our backgammon environment.

3.2 Training & Benchmarking

Self-training Similar to the approach of Tesauro[10] we decided to train our agents through self-play as opposed to expert level agents. This has several implications for the learning procedure of the agents. Firstly, they develop generalized strategies in contrast to playing against experts. This is due to the fact that expert agents have developed strategies that they deem optimal, which leads to learning agents developing counter-strategies. It is problematic to measure performance of these agents as they may not be suitable for general-purpose play. Secondly, it increases the convergence rate of the agents as they perform twice the amount of Q-value updates.

Mirror state One implementation decision for our learning algorithm was the choice to employ mirror states in the self-learning procedure. Instead of constructing the underlying understanding of playing two sides at once the board is flipped in between turns. An indirect consequence of this decision is the fact that it doubles the rate of convergence on the Q-function. This is due to

$$S_{black} = -1 \times reverse(S_{white}),$$

which effectively doubles the amount of updates of state-action pairs per game. The end result is the same since the color the agent is playing has no impact on decision making, much like in reality. In summary, each state presented to the agent is one sided and the agent only considers positive values as its own checkers. An example of mirror transformation of state x, S_{x_b} to S_{x_w} is presented below

$$S_{x_b} = \{0, 0, 3, -3, 0, 2, -2, 1, -1\} \xrightarrow{f_x} S_{x_w} = \{1, -1, 2, -2, 0, 3, -3, 0, 0\},$$

where f_x is an involution that transforms a black state to white and vice versa.

Benchmarking We intend to match the agents against each other. Due to the reduced state space of the backgammon environment, sheer luck plays a considerable part in the outcome of a match. In order to counter-act random variability in benchmarking each match-up is performed in sets of 10000 matches. Additionally, we introduce a third benchmarking agent that selects random actions in each state. By allowing the deep reinforcement agent and the Q-table agent to play against the random agent after variable amounts of training sessions, it is possible to extract reliable statistics, highlighting the effect of learning the environment.

Furthermore, the deep reinforcement agent plays against a fully trained Q-table agent and vice versa. This allows for effective measuring of relative accuracy and convergence rate.

3.3 Q-table learning

In this subsection a concrete explanation of the implementation is described and how components mentioned in section 2.3 have been modified and applied to fit our research.

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{Old value}} + \underbrace{\alpha}_{Learningrate} \times [\overbrace{\underbrace{r_{t+1}}_{\text{Feedback}} + \underbrace{\gamma}_{\text{Discount factor}} \times \underbrace{max_a Q(s_{t+1},a)}_{\text{Max future reward}}}^{\text{Expected reward}} - \overbrace{Q(s_t,a_t)}^{Oldvalue}] \tag{12}$$

Equation 12 explains how Q-values are updated after state transitions. The individual components of the equation are broken down in section 3.3.1.

Code 1 Q-table learning procedure

```
procedure PERFORMACTION($QValue$)
    $state \leftarrow backgammon.getState()$
    $dice \leftarrow (random(1,3),random(1,3))$
    $x \leftarrow randomFloat(0,1)$
    if $x \leq \epsilon$ then
        $action \leftarrow randomAction()$
    else
        $action \leftarrow getMaxValueAction(state, dice)$
    $QValue \leftarrow getValue(state, action)$
    $updateQValues(QValue)$
    $QPrevious \leftarrow QValue$

    $backgammon.updateState(action)$

procedure UPDATEQVALUES($maxQPrime$)
    if ($QPrevious \neq null$) then
        $QPrevious \leftarrow QPrevious + \alpha \times [r_{t+1} + \gamma \times maxQPrime - QPrevious]$
```

The pseudo-code describes the reinforcement learning process agents perform to explore and learn the environment. One important concept of the algorithm is the delayed updating of Q-values. Due to the stochastic environment predicting $max_a Q(s_{t+1}, a)$ is too computationally demanding. Instead, we update the previous $Q(s_t, a_t)$ once we know the outcome of the $max_a Q(s_{t+1}, a)$.

3.3.1 Parameters

In equation 12 new parameters are introduced in order to account for various phenomena attributed to Q-learning. This subsection elaborates on these parameters and the role they play in updating the lookup table.

Exploration tuning The exploration-exploitation trade-off described in section 2.5 requires a lot of fine-tuning. Exaggerating exploration in Q-learning leads to slow learning or diverging Q-values. On the other hand, overvaluing exploitation may lead the agent to settle for sub-optimal strategies. We chose to use the ϵ-greedy strategy and set the exploration parameter to 0.2.

Learning rate The parameter α in equation 12 regulates the magnitude of the update function. The learning rate is a trade-off between accuracy and the rate at which values propagate through the matrix in equation 5. If the learning rate is set to 1, then $Q(s_t, a_t)$ is updated with the value of the expected reward. If set to 0, values are not updated at all. The learning rate is to be set to reflect the tolerance of substandard strategy, where lower values generally increase accuracy. In the research conducted in this thesis the α-parameter was set to 0.5.

Immediate reward In equation 12 this parameter is denoted as feedback. It is a static reward value that is received by the agent for taking an action in a given state. Think of the immediate reward as a rule-set for the agent. Setting predetermined rewards for actions dictate how the agents behave once the values of the lookup table converge on the Q-function. For example, setting an immediate reward of 0.3 for a state where the agent hits an opposing checker results in a strategy where the agent attempts to disrupt the opponent. On the other hand, setting a negative immediate reward for exposing a blot results in a strategy where the agent values moving checkers together in a coordinated fashion.

In this comparison we opted to set the immediate reward for moving checkers to the negative value -0.0011 and the terminal state to 1. This has various implications on the strategy of agents. Strategies adopted by agents are indirect consequences of the immediate reward for a terminal state. The negative reward for moving checkers forces exploration in the early phases of learning.

Discount factor The γ-parameter, referred to as the discount factor, determines the value of potential future rewards in relation to immediate ones. Due to the distance between starting state and terminal states this parameter was set to 0.9 to allow positive Q-values to propagate faster.

3.3.2 Lookup table

Normally, lookup tables used for storing the Q-values are implemented as a matrix, mentioned in section 2.3. Rows and columns represent states and actions where an action is performed at a given state. The Q-value is the reward for performing the combination. However, applying the matrix approach results in unused storage quantities. In section 3.1.1, a rough estimation of the total amount of state-actions is calculated. To limit the amount of storage used by the lookup table and still keep the performance, the lookup table is implemented as a hashmap. This allows for a rapid lookup procedure with no wasteful storage. The state-action pair is embedded and represented in a single hashed key.

3.4 Deep reinforcement learning

This research uses a deep neural network, described in section 2.4, in order to elect the best action to perform in the current state.

3.4.1 Deep neural network

The input layer takes the state of the environment. This state representation, described in section 3.1.2, is composed of an integer array ranging from -3 to 3 including the 9 board positions. This array is fed to the neural network.

The input is passed to the two hidden layers of the neural network. In this thesis, we opted to use Tesauro as a source of inspiration for DNN architecture. TD-gammon 2.1 had tremendous success with only 80 hidden nodes and one layer [9]. Goodfellow [14] suggests that increasing the amount of hidden nodes leads to increased representational capacity of the DNN. However, increasing the amount of hidden nodes also increases the time it takes to train one session of backgammon. Therefore, our first hidden layer consists of 100 nodes and the second layer consists of 50 nodes. The activation function of the hidden layers is rectified linear unit (ReLU). ReLU is a non-linear function $f(z) = max(0, z)$ and is popular in deep neural networks due to faster training [25].

Finally, the output layer is represented as a two-unit vector. Softmax is used as the activation function resulting in a normalized output vector. This vector is the probability for each player to win the game in the given state. The procedure of obtaining an optimal action from the neural network consists of presenting the network with connected states in regard to the dice. Next, the state that produces the highest probability to win is selected.

The DNN is set to use a learning rate of 0.05. This learning rate is an inherent hyperparameter of the network and is not to be confused with the learning rate presented in equation 12. Stochastic gradient descent is used as the optimization function combined with Nestrov's updating function with the momentum parameter set to 0.9. This procedure offers improved stability and convergence over regular gradient descent [24].

Furthermore, the DNN is initialized using Xavier initialization which arranges the weights in the network appropriately for signals to reach deep through layers [23]. The layers are not pre-trained.

3.4.2 Training deep neural network

The application of the Q-learning update equation explained in section 3.3 is elucidated in this section. Parameters included in equation 12, are also used for training the DNN.

The procedure of training the DNN is similar to that of the training process explained for Q-table learning in the pseudo-code at section 3.3. The difference is that a particular input state is fitted with a modified output changing the weights in the DNN to an improved approximation of the value function. Previous input-output vector pairs are saved. As the next state is encountered, the previous output vector from the DNN is modified using the Q-learning update equation. Finally, the modified output vector is fitted to the previous input vector.

4 Results

In this section we provide the results of our study and describe each result briefly.

4.1 Training time

Table 1 presents the time it takes to train a DNN and a Q-table. It takes roughly 3,5 thousand times longer to play 100 thousand sessions of backgammon using the neural network than the same amount using a Q-table.

Table 1: Training time of DNNs and Q-tables

(a) DNN

Sessions	Hours	Minutes	Seconds
1	0	00	00.43
20	0	00	05.01
50	0	00	10.39
100	0	00	18.69
200	0	00	36.70
500	0	01	30.01
1 000	0	02	56.56
2 000	0	05	50.89
5 000	0	14	44.42
10 000	0	29	35.53
25 000	1	16	20.12
50 000	2	33	40.63
100 000	5	08	05.87

(b) Q-table

Sessions	Hours	Minutes	Seconds
1	0	00	00.00
100	0	00	00.02
500	0	00	00.05
1 000	0	00	00.07
5 000	0	00	00.31
10 000	0	00	00.61
50 000	0	00	02.69
100 000	0	00	05.30
1 000 000	0	00	54.76
5 000 000	0	04	38.05
10 000 000	0	09	17.51
50 000 000	0	48	58.75
100 000 000	1	51	14.45

Below is a mapping of the learning process in Q-tables. Positive Q-values indicate learning, as the algorithm becomes more familiar with the environment. As the agent plays more training sessions, the number of Q-values converges. According to equation 10 Q-tables explore a fraction of the state space.

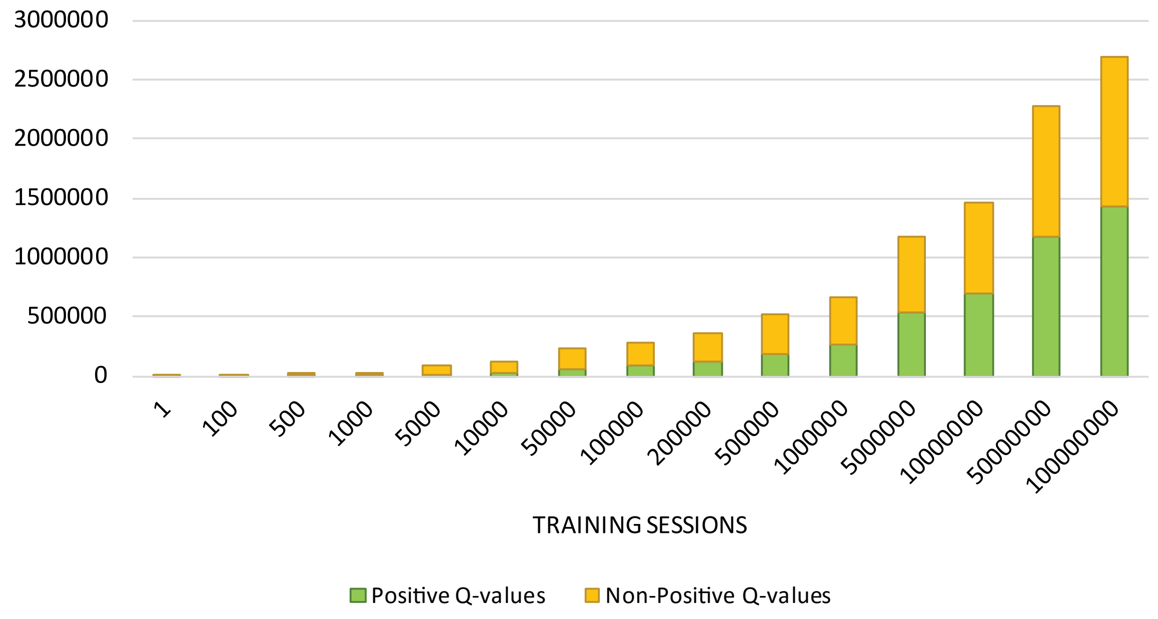

Figure 3: Mapping of state-actions in Q-tables after variable amounts of training sessions. Lists the amount of positive and non-positive Q-values.

4.2 Comparison of agents

This subsection highlights strengths and weaknesses of the algorithms. Additionally, we showcase interesting outcomes and summarize these. At each benchmarking interval illustrated in this section, the agents played 10 thousand matches against each other. The 95% confidence interval for population proportions

$$(\overline{x} - \beta, \overline{x} + \beta) = \hat{p} \pm z(\alpha/2) \times \sqrt{\frac{\hat{p} \times (1 - \hat{p})}{n}} \tag{13}$$

gives us

$$(0.4902, 0.5098) = 0.5 \pm 1.96 \times \sqrt{\frac{0.5^2}{10000}}$$

when the value of $(\hat{p} \times (1 - \hat{p}))$ is the largest (for $\hat{p} = 0.5$). Equation 13 illustrates that the error is no larger than 0.01 with 95% certainty. Therefore, we do not include confidence intervals in the result graphs as they limit the visibility of our results.

4.2.1 Random agent benchmarking

Figure 4 and 5 depict the win rate of Q-tables and deep reinforcement learning after training sessions listed in table 1. Q-tables start at 50% win rate and increase slowly towards 80% win rate. The deep reinforcement agent wins 63% of the games after one training session and increases to roughly 67%. The deep reinforcement agent displays higher uncertainty in deciding upon an optimal action-selection policy.

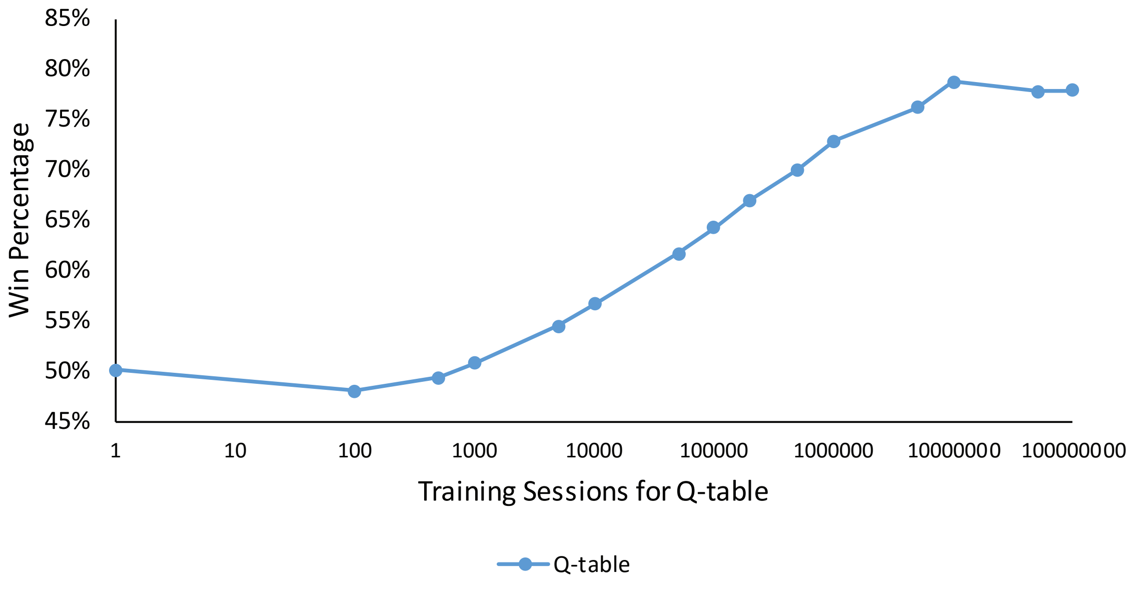

Figure 4: Q-Table win rate over random agent after training sessions listed in table 1.

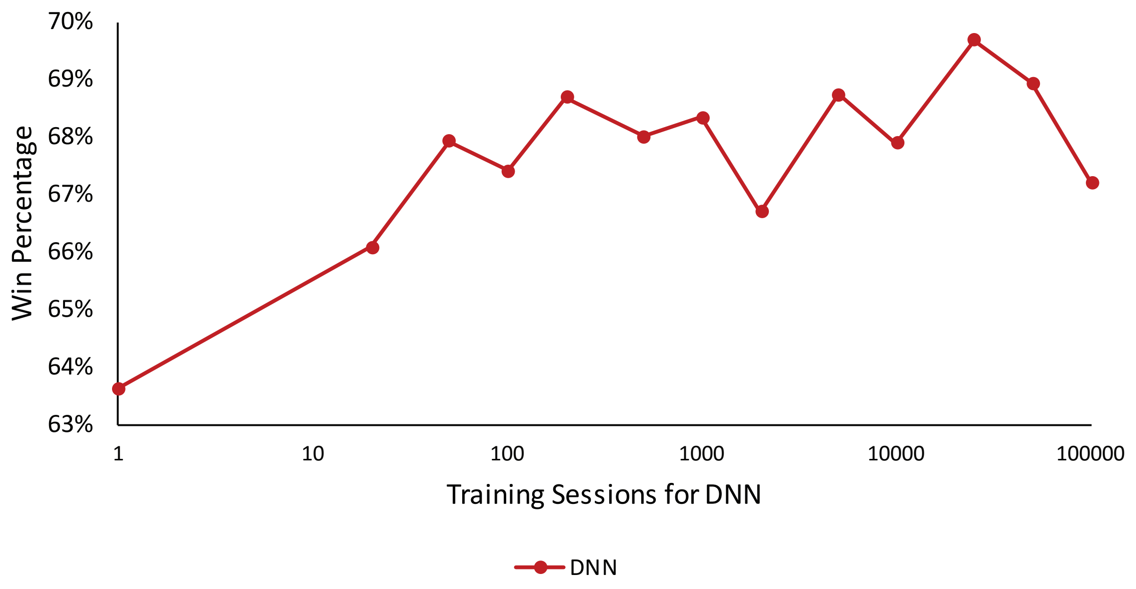

Figure 5: DNN win rate over random agent after training sessions listed in table 1.

4.2.2 DNN vs Q-table

Figure 6 and 7 depict a comparison between the Q-table and DNN when benchmarked against each other. Figure 6 illustrates the number of training sessions it takes for the Q-table to outplay a trained DNN. Against a trained DNN, this number is between 10 thousand and 100 thousand.

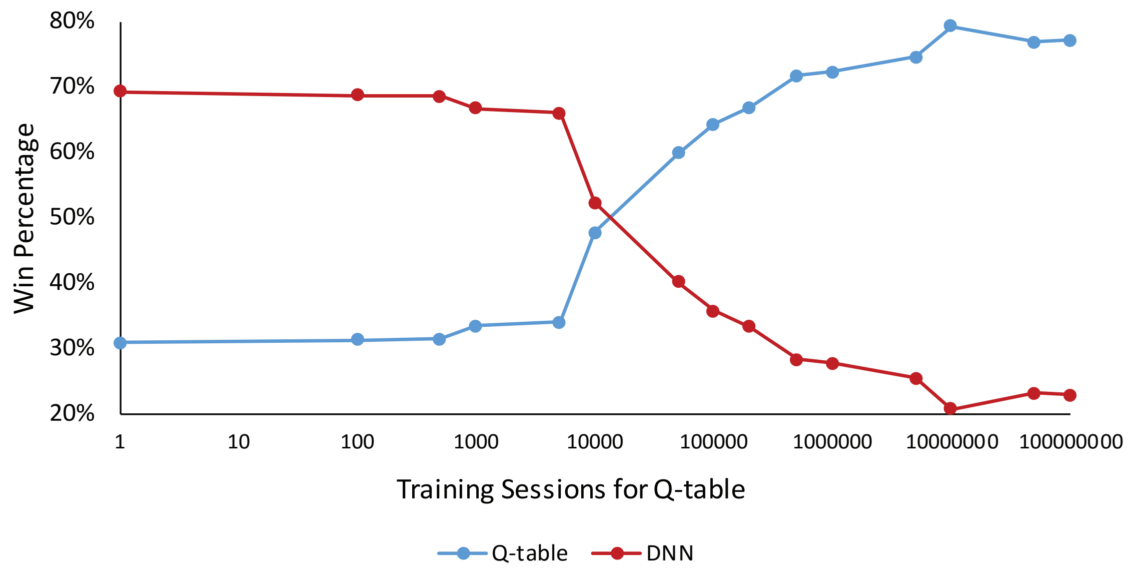

Figure 6: The relative win rate between a trained DNN versus a Q-table with training sessions listed in table 1. The DNN has trained 100 thousand training sessions.

Conversely, figure 7 depicts the win rate of a DNN against a trained Q-table. The DNN does not manage to reach above 35% win rate. The downward trend displayed at 100 thousand training sessions of the DNN, can be explained by the lack of convergence towards a perceived optimal action-selection policy, delineated in figure 5.

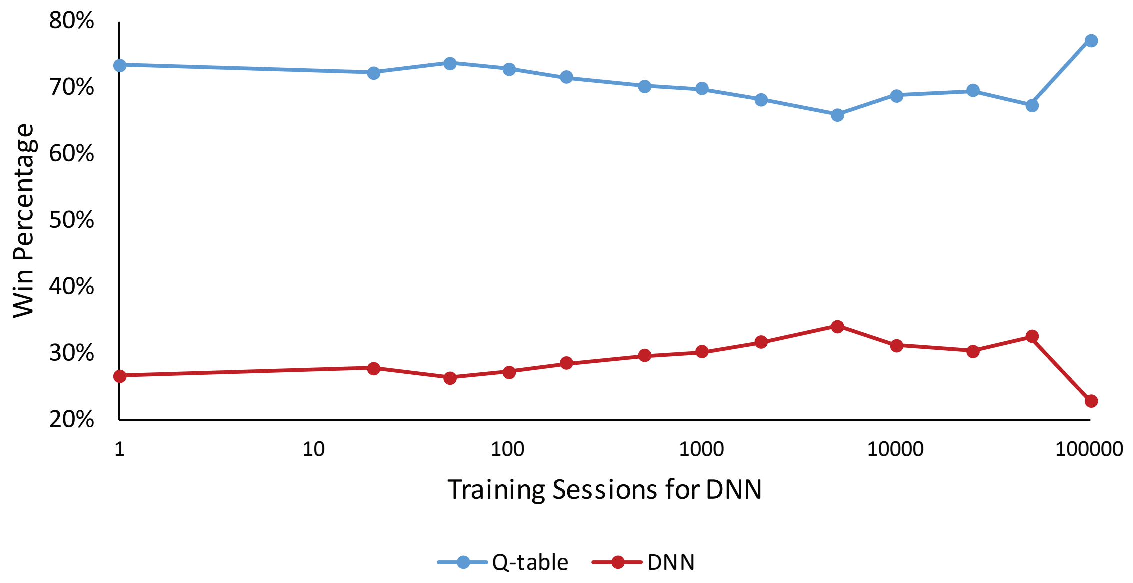

Figure 7: The relative win rate between a trained Q-table versus a DNN with training sessions listed in table 1. The Q-table has trained 100 million training sessions.

Figure 8 illustrates the difference in level of play between a Q-table at various stages of learning. The DNN is benchmarked against the two Q-tables after variable amounts of training sessions. The difference in win rate of the DNN against both Q-tables is approximately 10%.

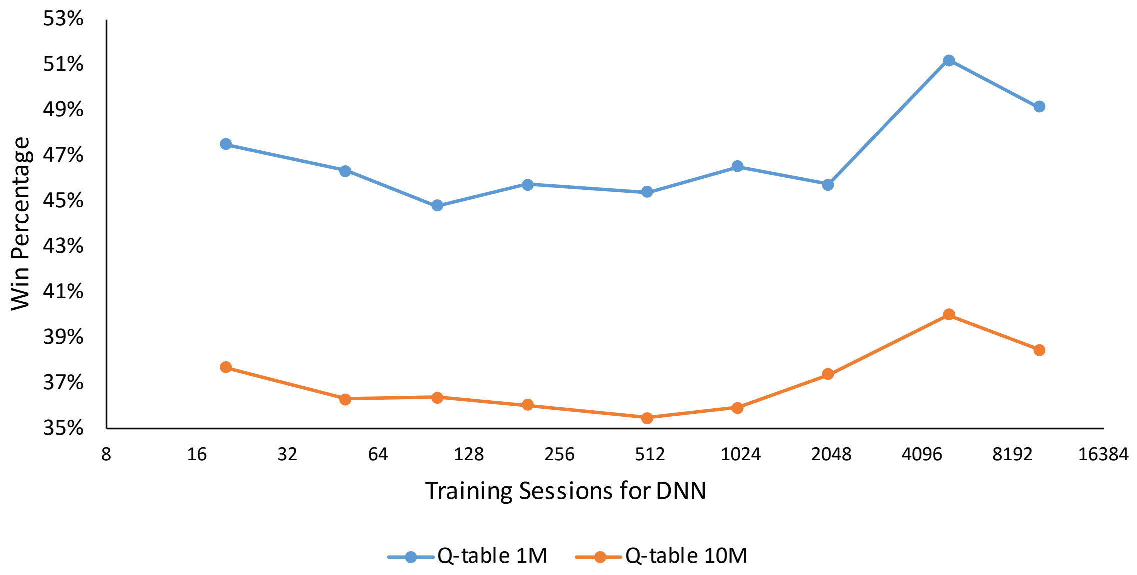

Figure 8: The DNN's win rate over Q-table learning after 20, 50, 100, 200, 500, 1000, 2000, 5000 and 10000 training games. The blue and orange lines represent the win rate over a Q-table with 1 million and 10 million training games respectively.

Lastly, figure 9 illustrates the potency of the DNN's convergence rate. A DNN that has trained 1 session is capable of outplaying a Q-table that has trained 100 thousand sessions.

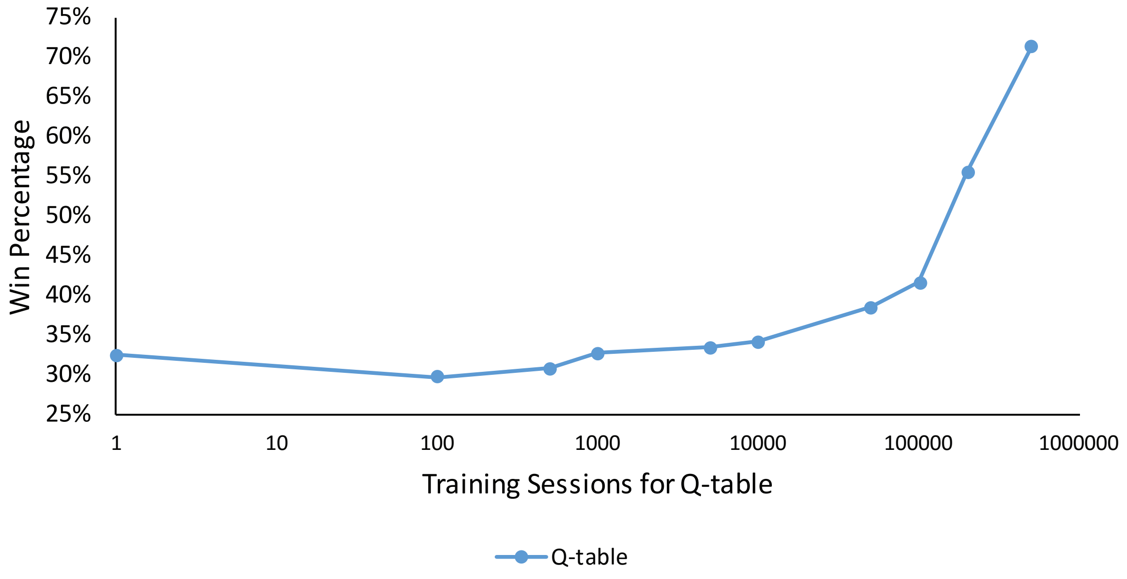

Figure 9: Q-table win rate over a DNN that has trained 1 session of backgammon. Depicted Q-table intervals are 1, 100, 500, 1000, 5000, 10000, 50000, 100000, 200000 and 500000.

5 Discussion

When conducting a comparative study of the nature covered in this thesis, there is tremendous significance in eliminating variable factors. We opted to use the same parameters for both algorithms, with the exception of the learning rate. There is a possibility that certain parameters affect one approach differently than the other. This in turn affects the validity of the results. Furthermore, analyzing results covered in the previous section, it is important to keep a holistic perspective due to the fluctuating tendencies of reinforcement learning during initial training sessions. Due to attention devoted to exploration, an agent may be in the middle of transitioning to a new strategy while being benchmarked. This may produce misleading results.

To answer the first element of the problem statement we consider figure 5 and figure 4 which depicts the performance of the algorithms after variable amounts of training sessions. By studying the win rate after 20 training sessions versus the random action-selection agent, it is evident that the neural network-equipped Q-learning agent is expeditiously capable of basic decision-making. On the other hand, the Q-table agent displays inadequate results after 20 training sessions, hinting at an action-selection policy resembling that of randomized selection. This is a result of values propagating backwards slower in a matrix than a neural network. Thus, the deep reinforcement learning approach has drastically altered the action-selection policy after minimal exposure to the environment with positive results.

However, by studying figure 4, 5 and 6, one can infer that the initial trend is misleading. The deep reinforcement agent is able to learn more quickly but has a smaller maximum capacity for learning. Therefore, it converges rapidly towards approximately 70% win rate against a random action-selection policy. The Q-table converges slower towards it's perceived optimal strategy with the benefit of convergence closer to the true optimal strategy. Figure 4 indicates a convergence towards approximately 80% win rate against the random action-selection policy. By comparison of the two algorithms through tug-of-war type match-ups, portrayed in figure 6, an interesting trend emerges. The generalizing characteristic of deep reinforcement learning proves superior in the initial 10 thousand sessions of learning. However, once reaching approximately 10 thousand sessions the accuracy of Q-tables develops into the dominating characteristic trumping deep reinforcement learning.

5.1 Smaller environments

Consider the scenario of applying reinforcement learning to an elevator controller, in a building that is 500 meters tall with 150 floors. The elevator can carry 8 passengers and can either go up or down. Additionally, it actively accounts for the 2 nearest floors where buttons are pressed and we have one state per 0.1 meters. Using the same principle of calculating the state space covered in section 3.1.1, there are less than 10^9 state-action pairs. In this scenario, the Q-table approach would be able to map the environment leading to greater accuracy. Therefore, it would be the preferred approach when it comes to optimization problems such as minimizing transportation time per person.

5.2 Larger environments

One of the fundamental strengths of neural networks is its adaptability to larger environments. The results presented based on the reduced environment accentuate the results of Q-table learning. The trade-off between accuracy and abstraction mentioned briefly in section 2.4, becomes intriguing in larger environments where Q-tables are unable to map the majority of an environment. This places emphasis on the generalization of neural networks and its ability to select advantageous actions in unfamiliar states.

The equations presented in section 3.1.1 highlight the primary flaw with Q-table learning as a concept. As figure 4 illustrates, it takes a significant number of trainings sessions for the Q-table to adopt an effective strategy. It is reasonable to expect, as the environment expands, that the state-action pairs that the Q-table needs to explore in order to become adequate, expands as well. Applying the Q-table approach to full-scale backgammon is infeasible due to the gargantuan state space. If the Q-table does not have an exhaustive mapping of its environment, there is a greater probability that it encounters an unfamiliar state. This results in a randomly selected action which decreases performance.

In the previous section we proposed a small environment where Q-table's accuracy generates favourable results as opposed to a DNN. In the problem of finding an optimal strategy for the game of GO which averages 250 legal moves per position and a length of 150 turns [6], the state space is considerably larger. In fact, there are approximately 250^{150} unique state-action pairs, making exhaustive search of the state space infeasible. In this scenario Q-tables would roughly resemble a random action-selection policy. In contrast to the optimal elevator controller, a DNN would be far more practical as it's generalization characteristic would provide an above-average action selection policy.

5.3 Limitations

The study conducted in this thesis proved that there are useful applications of Q-tables as opposed to DNNs. It was limited by a few factors that interfere with the basis for conclusions of a wider perspective.

The parameter space of neural networks is massive and requires fine-tuning for each individual problem. This implies potential for constructing a DNN that outperforms the one used in this thesis. Specific parameters that have not been examined in satisfactory detail include learning rate and the network architecture. An optimally configured learning rate increases the learning capacity of the DNN. Likewise, increasing the amount of hidden layers and nodes in the network architecture has the same effect [14]. Therefore, it is possible to design a DNN that could match the Q-table at larger numbers of training sessions than depicted in figure 6.

Furthermore, the exploration strategy of the approaches is an integral component of finding favourable strategies. In this thesis, the ϵ-greedy strategy was selected as it is simple to apply. However, a static exploration strategy is seldom optimal. Considering the scenario where an agent has established the optimal action-selection policy the static ϵ-greedy strategy causes divergence from that optimal policy. Employing an exploration strategy that dissipates as the agent is trained may produce better results. This limitation is particularly applicable to the DNN as there are signs of uncertainty in selecting an optimal policy in figure 5.

Lastly, in section 3.3.1 we briefly discussed the implications of initializing certain states with an immediate reward. The terminal states were the only positively initialized states in our study. Therefore, the agents are given limited guidance as to what an optimal strategy for backgammon is. By initializing the states differently, the agents may find better strategies faster. For example, it is undeniable that hitting opponent's checkers is a favourable action in most states. By initializing the output state of a hit with a positive value the agents favour these actions.

5.4 Future research

There are multiple alternatives to future research within the area discussed in this thesis. Firstly, the impact that look-ahead has on the performance of these algorithms is a topic of considerable importance. Secondly, one can investigate the parameter space of the DNN for the possibility of improved performance or greater decisiveness in electing action-selection policies. In section 2.5, other exploration strategies than ϵ-greedy were mentioned. As discussed earlier, the ϵ-greedy strategy may interfere with the DNN's convergence towards an optimal strategy. Therefore, it is of particular interest to study how different exploration strategies affect the DNN's performance.

Lastly, it could be interesting to apply the approaches to a practical environment where Q-tables could thrive, as opposed to a DNN.

6 Conclusion

This thesis studied two approaches to reinforcement learning. Q-tables store values for action-state pairs and the Q-learning algorithm updates these as training sessions are played. On the contrary, deep reinforcement learning relies on a DNN to assess the value of future states. They were compared based on their accuracy in selecting optimal actions as well as their convergence rate, how quickly they managed to adopt favourable strategies. The agents produced by the algorithms were benchmarked against a third, random agent and against each other under varying circumstances.

It should be noted that the DNN used in this study is not guaranteed optimal configuration, since we have not explored the parameter space. Additionally, due to time constraints few training sessions were performed. This implicates a possibility for an agent to converge towards the global maximum. Results indicate that DNNs are effective for learning vast environments rapidly, adopting above average action-selection policies tremendously faster than Q-tables. However, once Q-tables have exhaustively explored the environment they adopt far more accurate action-selection policies than the DNNs. These results are derived in a reduced environment. In larger environments the convergence rate of DNNs are to be preferred over the accuracy in Q-tables, which is the case in full-scale backgammon.

The primary conclusion from this thesis is that the accuracy in Q-tables can improve results immensely if used in smaller environments. Elevator control, introduced in section 5.1, is an environment where Q-tables could prove superior to a DNN due to the limited state space. In larger environments such as optimal strategy for GO mentioned in section 5.2, there are too many state-action pairs for the Q-table to be useful. Therefore, DNNs are to be favoured in these situations.

References

[1] A.L. Samuel. 1969. "Some Studies in Machine Learning Using the Game of Checkers". *IBM Journal.* 3(3), pp 1-36.

[2] C. Watkins and P. Dayan. 1992. "Technical Note; Q-Learning". *Machine Learning.* 8(3), pp. 279-292.

[3] C. Szepesvári. 2010. "Algorithms for Reinforcement Learning". Edmonton, Alberta, Canada. Morgan & Claypool Publishers. pp. 1-89.

[4] D. Plaut and G. Hinton. 1987. "Learning sets of filters using back-propagation". *Computer Speech & Language.* 2(1), pp. 35-61.

[5] D. Silver. 2015. Lecture: Reinforcement Learning. Topic: "Case Study: RL in Classic Games". University College London, London, England.

[6] D. Silver et al. 2016. "Mastering the game of GO with deep neural networks and tree search". *Nature.* 529(7587), pp. 437-560.

[7] E. Keeler and J. Spencer. 1975. "Optimal Doubling in Backgammon". *Operations Research.* 23(6), pp. 1063-1071.

[8] G. Tesauro and T.J. Sejnowski. 1989. "A parallel network that learns to play backgammon". *Artificial Intelligence.* 39(3), pp. 357-390.

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

[10] G. Tesauro. 2002. "Programming backgammon using self-teaching neural nets". *Artificial Intelligence.* 134(1-2), pp 181-199.

[11] H. Berliner. 1980. "Backgammon computer program beats world champion". *Artificial Intelligence.* 14(2), pp. 205-220.

[12] H. van Hasselt et al. 2015. "Deep Reinforcement Learning with Double Q-Learning". *arXiv.* [Online] Available: `http://arxiv.org/abs/1509.06461`. [Accessed: 05 Apr 2016].

[13] I. Ghory. 2004. "Reinforcement learning in board games.". Deparment of Computer Science, University of Bristol, Bristol, England, Tech. Rep. CSTR-04-004.

[14] I. Goodfellow et al. "Deep Learning". unpublished.

[15] J.J. Chung et al. 2015. "Learning to Soar: Resource-constrained exploration in reinforcement learning". *The International Journal of Robotics Research.* 34(2), pp. 158-172.

[16] M. Riedmiller et al. 2009. "Reinforcement learning for robot soccer". *Autonomous Robots*, 27(1), pp. 55-73.

[17] O. Arvidsson and L. Wallgren. 2010. "Q-Learning for a Simple Board Game." B. Sc. Thesis, Royal Institute of Technology. Stockholm, Sweden.

[18] P. Baldi et al. 2014. "Searching for exotic particles in high-energy physics with deep learning". *Nature Communications*, 5(1), pp. 4308.

[19] R. Sutton and A. Barto. 1998. "Reinforcement Learning: An Introduction". *IEEE Transactions on Neural Networks*. 9(5), pp 1054-1054.

[20] T. Matiisen. 2015. "Guest Post (Part I): Demystifying Deep Reinforcement Learning - Nervana". *Nervana*. [Online]. Available: `http://www.nervanasys.com/demystifying-deep-reinforcement-learning/`. [Accessed: 08 May 2016].

[21] V. Jain et al. 2007. "Supervised Learning of Image Restoration with Convolutional Networks" in Proc. *IEEE 11th International Conference on Computer Vision*. pp. 1-8.

[22] V. Mnih et al. 2015. "Human-Level control through deep reinforcement learning." *Nature*. 518(7540), pp. 456-568.

[23] X. Glorot and Y. Bengio. 2010. "Understanding the difficulty of training deep feedforward neural networks". *Journal of Machine Learning Research*. 9(1), pp. 249-256.

[24] Y. Bengio et al. 2013. "Advances in optimizing recurrent networks" in Proc. *IEEE Int. Conf. Acoust. Speech Signal Process.* (ICASSP), pp. 8624–8628.

[25] Y. Lecun et al. 2015. "Deep Learning". *Nature*. 521(7553), pp. 436-444.