Deep Distributional Temporal Difference Learning for Game Playing

Frej Berglind

Master's thesis
2019:E66

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

Abstract

Temporal difference learning is considered one of the most successful methods in reinforcement learning. Recent developments in deep learning have opened up a new world of opportunities. In this project, we compare classic scalar temporal difference learning with three new distributional algorithms for playing the game of 5-in-a-row using deep neural networks: distributional temporal difference learning with constant learning rate, and two distributional temporal difference algorithms with adaptive learning rate. All these algorithms are applicable to any two-player deterministic zero sum game and can probably be successfully generalized to other settings.

As it turned out, all algorithms performed well and developed strong strategies. The algorithms implementing the adaptive methods learned more quickly in the beginning, but in the long run, they were outperformed by the algorithms using constant learning rate which, without any prior knowledge, learned to play the game at a very high level after 200 000 games of self play.

Acknowledgements

I wish to thank my advisors Professor Jianhua Chen of Louisiana State University and Professor Alexandros Sopasakis of Lund University for providing invaluable feedback, cherished guidance and wonderful support throughout the project. I always left my weekly meetings with Dr. Chen excited about the research and inspired to carry on with the work.

Thanks to the computer science department of Louisiana State University for providing me with a pleasant workplace, efficient computational resources and friendly colleagues. Thanks to Doug Lafield for helping me get started with running my code on the GPU server. It was essential for the success of this project.

Thanks to my family for supporting me throughout my education, helping me find my place in life. Thanks to Jing Tan for sharing this time with me and for her unparalleled love of cabbage. Thanks to Carl Henrik Dahmén and Johannes Kasmir for keeping me company through countless hours of exercises and discussions by the chalkboards in the mathematics building of Lund University.

Contents

Chapter 1

Introduction

Reinforcement learning is a branch of machine learning inspired by how humans learn by practicing. Through trial and error, an agent gradually learns to maximize a reward or win a game. These techniques could be applied to a wide range of sequential or adversarial optimization problems and are predicted to be an essential part of the development of artificial intelligence [3].

In recent years, many great breakthroughs have been made by combining reinforcement learning with deep neural networks. A neural network is an approach to machine learning inspired by the human brain. It consists of layers of mathematical neurons that can be trained to recognize patterns and make predictions. Deep learning is the application of deep neural networks, using many layers of neurons to learn abstract features and perform complex tasks. The field of deep learning has shown a dramatic rise in the past ten years due to more powerful computers, access to larger datasets and new techniques for training deep networks [5].

Furthermore, deep learning combined with reinforcement learning has led to great achievements in game playing. The reinforcement learning algorithm acts as a link between the game and the neural network, helping the neural network make sense of the game. This is illustrated in Figure 1.1. The neural network can be set up to make decisions in the game, but in the beginning, it does not know anything about the game and will play very poorly. In this project, it was set up to predict the outcome of the game based on a state and used this prediction to select the best move. The reinforcement learning algorithm can analyse games played by the neural network and generate training data that is used to improve the neural network. By iterating between playing games, analysing them and training the network using the new data, the neural network can gradually learn to play the game.

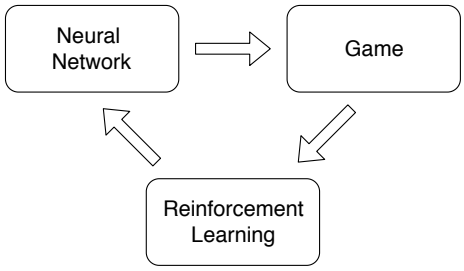

Figure 1.1: The reinforcement learning algorithm helps the neural network learn the game. The neural network makes decisions and plays the game. The games played by the neural network are analysed by the reinforcement learning algorithm which generates training data. The training data is used to train and improve the neural network. Through this process, it gradually develops a strategy for the game.

Games provide a suitable setting to explore different decision making and learning algorithms. The combination of simple rules and complex strategies makes board games especially appealing. It might seem frivolous to waste time and resources on game playing, but the knowledge gained in game playing can be used in many other domains. Just like a chemist makes experiments in the controlled environment of a laboratory, AI research is performed in the artificial environments of games. Games offer absolute definitions of the choices the agent has to make and measurements of it's performance; there is no doubt about the winner or the score in a well defined game.

In the long run, the goal is not to create the best player for a certain game. The goal is to gain knowledge about the decision making and learning processes. The game is simply a testbed for comparing the performance of different methods. Algorithms used for game playing can be used in many applications that require efficient search methods, complex reasoning or long term planning such as autonomous driving, traffic control, recommendation systems, robotics and DNA analysis.

One of the most common and successful methods for reinforcement learning in game playing is temporal difference learning (TD) [21]. It is a group of reinforcement learning algorithms applicable to a wide range of problems. However, most real world problems require more data efficient learning and better performance than what is currently possible with reinforcement learning [9]. The classic approach to TD is learning a strategy by approximating the expected reward, but recent research [1] has shown greatly improved results using distributional TD, which approximates the distribution of the reward.

The goal of this project is to study temporal difference learning algorithms combined with deep neural networks and explore the possible advantages of distributional TD and an adaptive learning rate. As a framework for comparing different algorithms, I chose the game of 5-in-a-row. In order to create good conditions for the algorithms to learn, I also put some efforts in neural net-

work design and hyperparameter tuning and these results will also be covered in this thesis. A probabilistic measure of an action's optimality is proposed, which served as an adaptive learning rate for distributional temporal difference learning.

1.1 Related Works

Game playing is a classic area of AI research. Games provide a suitable setting to explore different decision making and learning algorithms. Since the dawn of the field of AI research games such as Checkers, Chess and Go have been very common settings for AI experiments. Here, I will provide a short overview of such experiments.

The earliest example of reinforcement learning in game playing is Arthur Samuel's checkers experiments from 1959, where he constructed a reinforcement learning algorithm that managed to beat him in the game of checkers after 10 hours of practice [17]. That is quite an achievement considering the limitations of a 1950's computer.

In 1992, the Backgammon program TD-Gammon reached master level in Backgammon by using temporal difference learning and a neural network [23]. In 1997, IBM's chess computer DeepBlue based on highly optimized Minimax search and a handcrafted evaluation function beat the world champion Garry Kasparov [23].

In 2014 Google Deepmind combined deep learning with Q-learning (a kind of TD-learning) and created an algorithm that learned to play several Atari video games from the raw images [12]. In 2017, they published the paper "A Distributional Perspective on Reinforcement Learning" [1] introducing an algorithm that combine deep learning and distributional temporal difference learning showing greatly improved performance on Atari games. It is similar to the distributional temporal difference learning algorithm used in this project.

In 2016, Google Deepmind's AlphaGo [19] using a new reinforcement learning algorithm and a deep neural network, beat a professional Go player and a year later beat the world's top ranked Go player [23]. A later version of the algorithm, AlphaGoZero [20] greatly outperformed it's predecessor and learned the game without any prior knowledge. A generalized version of this algorithm, AlphaZero [18] reached superhuman performance in Go, Chess and Shogi without any additional tuning for the different games. Since it has achieved very impressive results in these complex games, the AlphaZero algorithm is likely to outperfrom TD-learning in playing the quite simple game of 5-in-a-row. However, the AlphaZero algorithm is specialized on board games and is computationally expensive. TD-learning has a wider range of applications and even though it might not beat AlphaZero for playing board games, progress in this field can be very useful.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement leaning is a branch of machine learning where an agent learns to make decisions that optimize a reward, similar to how humans learn by practising. A common setting for reinforcement learning is a *Markov decision process* (MDP). An MDP is centered around an *agent* interacting with an *environment*. I will use a simplified description of an MDP restricted to deterministic processes. A deterministic MDP is defined by a tuple: (S, A, r, f) where S is a set of *states*, A is a function returning the set of all possible *actions* in a given state, r is a function returning the reward for selecting an action, and f is the transition function.

The process begins in some state, $s_0 \in S$. The agent selects an action $a_0 \in A(s_0)$, receives the reward $r(s_0, a_0)$ and moves to the state $s_1 = f(s_0, a_0)$. From there, the agent will make a new action, $a_1 \in A(s_1)$, receive a reward $r(s_1, a_1)$ and move on to another state, $s_2 = f(s_1, a_1)$. The process continues like this. The goal of the agent is to maximize the cumulative reward [11]. A function that maps each state to an action is called a *policy* [13]. It represents a strategy for the agent. A policy can be developed gradually by letting the agent practice and learn from its mistakes.

A central equation in reinforcement learning is the recursive Bellman Equation [22],

$$V(s) = \max_{a_i \in A(s)} \Big[r(s, a_i) + \gamma V(f(s, a_i)) \Big] \tag{2.1}$$

where $V(s)$ is the expected reward in state s, $\gamma \in [0, 1]$ is the discount factor and $f(s, a_i)$ is the successor of s after the action a_i. The Bellman Equation states that the expected reward in s is the sum of the immediate reward and the discounted reward of the successor state for the best action in s. This version of the Bellman Equation is simplified for a deterministic process.

Exploration & Exploitation

The dilemma of exploration and exploitation is central in reinforcement learning. In this context, exploration means trying out suboptimal actions to learn more about the environment. Exploitation is choosing the best action using the current knowledge of the environment [14]. In order to make progress it is often necessary have some exploration, otherwise the agent might get stuck repeating the same actions, similar to an optimization algorithm getting stuck at a local minimum. A common approach is the epsilon greedy strategy: with probability ϵ select a random move, otherwise select the best move [2]. As the agent practises and learns more about the environment it is common to decrease ϵ to gradually shift the focus from exploration to exploitation.

Temporal Difference Learning

Temporal difference learning is a class of reinforcement learning algorithms that approximate the expected reward for the states. The expected reward is called the value of the state and is denoted $V(s)$. It is learned by using the estimated value of the successor state bootstrapping back through the decision process [22].

The simplest Temporal Difference algorithm is called one-step TD or TD(0). According the its current knowledge, it takes the best action a at state s and updates the value of s using the value of the successor state:

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha\big(r(a) + \gamma V(f(s,a))\big) \tag{2.2}$$

where $\alpha \in [0,1]$ is the learning rate.

2.2 The Game of 5-in-a-row

5-in-a-row is a two-player strategy game traditionally played on squared paper. The players take turns placing markers in an empty cell on the paper. One player uses "X" and the other player uses "O". You win by getting 5 in a row horizontally, vertically or diagonally. If the grid is filled up without anyone having 5-in-a-row the game ends in a tie.

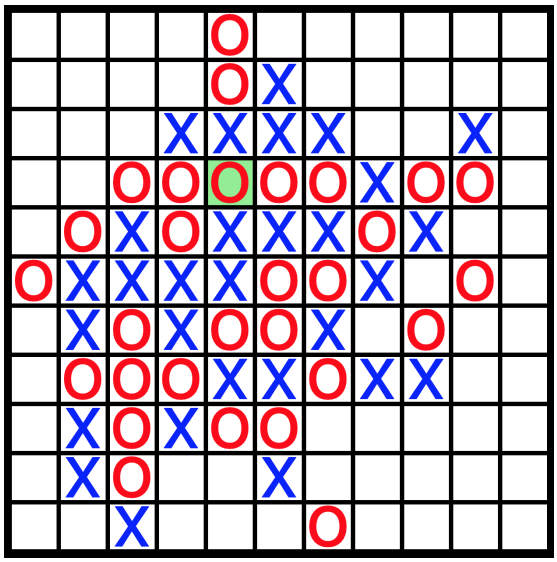

Figure 2.1: A game of 5-in-a-row won by "o".

This is the version of the game I learned growing up in Sweden. There are many names for this game: Gomuko, Gobang, Luffarschack (Swedish meaning poor man's chess) to name a few. Gomuko is usually played on a 15 by 15 board and has many variations of the rules to reduce the first player advantage. Gomuko without any such extra rules was proven to be a first player win in 1993 [10].

I have chosen to restrict the game to a board size of 11 by 11. It is large enough to make the game complex while keeping the state and action spaces reasonably small. Since an 11 by 11 board is more restricted than the 15 by 15 Gomuko board, the game should, if played perfectly, either end in with the first player winning or a tie.

The number of actions in each state of the game decreases by 1 each move. Starting with 121, 120, 119... Therefore, if we ignore any winning actions ending the game, there are $121! \approx 10^{201}$ possible games, but this is of course a grave overestimation. If we only look at games with the reasonable length of 40 moves there are $\frac{121!}{81!} \approx 10^{80}$ possible games. The game has the symmetry of square, i.e any state is equivalent under rotation of 0, 90, 180, 270 degrees and horizontal, vertical and diagonal reflections.

2.3 Alternating Markov Games

To define the algorithms in a more general setting, I will use an abstract framework for games similar to 5-in-a-row. It includes games like Chess, Othello, Checkers and Go. These are *deterministic alternating Markov games* with a final reward of 1 for winning 0 for a tie and -1 for losing and no other reward

during the game. An alternating Markov game is similar to a Markov decision process (see Section 2.1), but differs by having several adversarial agents.

A game is defined by a state space S, an action space $A(s)$ for each state $s \in S$, a transition function $f(s, a)$ defining the successor state when selecting a in s, an initial state s_0 and a function $r(s)$ which determines if a state is final and in that case returns the reward. A final state either has reward 1 (win), 0 (tie), or -1 (loss). The reward for a player is -1 times the reward of the opponent. This symmetric view on the reward is used in my implementation of the algorithms, but sometimes it is better to think if one player trying to maximize the reward and its opponent trying to minimize the same reward.

The game is played by letting the players take turns selecting actions until they reach a final state. A game can be seen as a sequence of states $[s_i]$ connected by actions $[a_i]$ where even numbered actions are played by the first player and the odd numbered actions are played by the opponent. This is illustrated in Figure 2.2 and can be the described by the recurrence relation

$$s_{i+1} = f(s_i, a_i), a_i \in A(s_i), \tag{2.3}$$

saying that the next element in the sequence of states is the successor of the current state depending on which action the agent selects.

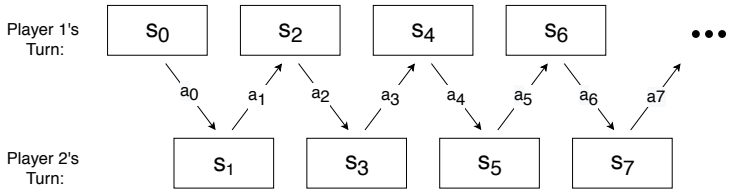

Figure 2.2: The game is a sequence of states connected by actions. The states are produced recursively according to Equation 2.3. The actions of the first player are the even numbered downward arrows and the moves done by the second player are the odd numbered upwards arrows.

Minimax Search

A common approach for analysing adversarial games is minimax search which searches through the game tree looking for optimal choices for both players [16]. The root node of the game tree is the initial state, s_0. The children of a node are its successor states. In minimax search the two players are called MAX, who is maximizing the reward and MIN, who is minimizing the reward.

The minimax search is performed recursively,

$$\text{MINIMAX}(s) = \begin{cases} r(s) & \text{if } s \text{ is a terminal state.} \\ \max_{a_i \in A(s)} \text{MINIMAX}(f(s, a_i)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a_i \in A(s)} \text{MINIMAX}(f(s, a_i)) & \text{if PLAYER}(s) = \text{MIN} \end{cases} \tag{2.4}$$

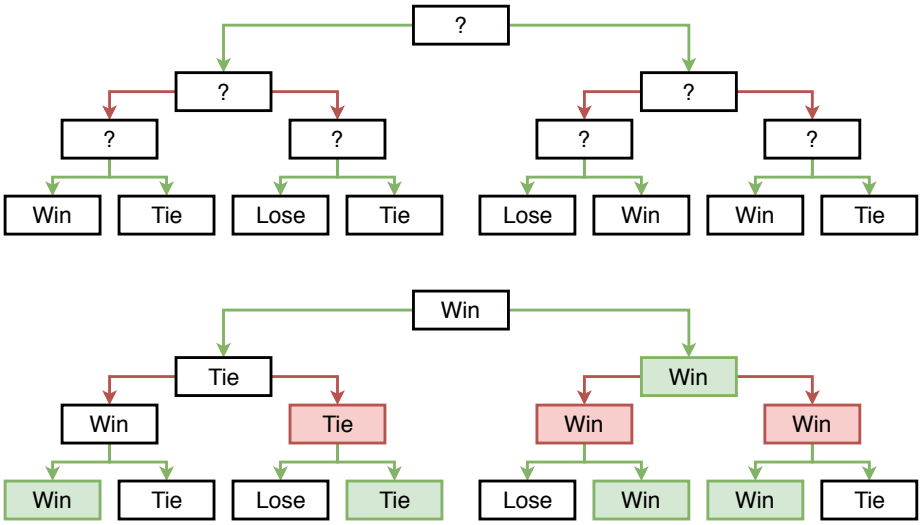

Figure 2.3: The minimax search tree for a game of three actions, two by the maximizing player and one by the minimizing player. The top tree is before the search and the lower tree show the search results. The minimax algorithm is described in Equation 2.4. Red arrows represent the minimizing player's options and the green arrows are the maximizing player's options. Their choices are colored in red and green respectively. This game is shown to be a first player win.

and this process is illustrated in Figure 2.3. In the end, you have a win, tie or lose label for each state. Making a decision in the game can be seen as a classification problem: labeling each successor state as win, tie or lose and based on this selecting the best option. I will get back to this later when I define the distributional algorithms.

Chapter 3

Method

In this section, I will describe the algorithms starting from basic TD-learning and define the optimality measure used in my new algorithms. Furthermore, I will explain how I implement the algorithms, the training procedure and the evaluation of the results. The algorithms are defined within the framework of the Deterministic Alternating Markov Games described in Section 2.3.

3.1 Temporal Difference Learning

A classic reinforcement learning approach is directly approximating the value function, $V^*(s) \approx V(s)$. This is what is usually referred to as temporal difference learning. In deep temporal difference learning, $V^*(s)$ is calculated by a neural network. The result of this algorithm will serve as a reference point for my other more experimental methods. To distinguish it from the distributional algorithms, I will sometimes refer to it as scalar TD. I will call the agent using this algorithm TDBot.

Decision Making

The agent simply selects the move with the highest expected reward,

$$a(s) = \underset{a_i \in A(s)}{\operatorname{argmax}} V^*(f(s, a_i)). \tag{3.1}$$

Training

After the agent has played a game it uses the result to assign new values to V^*. The update starts from the end of the game by assigning a new value [1] to the

[1] The arrow ($\leftarrow$) is a pseudo code notation for assigning a new value to the function. In my implementation, the new value is used immediately to create new values for preceding states and the input/output pair is used as training data for the neural network at the end of the training iteration.

final state,

$$V^*(s_{final}) \leftarrow r(s_{final}) \tag{3.2}$$

and recursively updates V^* based on the Bellman Equation (Equation 2.1), radually stepping back through the game,

$$V^*(s) \leftarrow (1-\alpha)V^*(s) + \alpha\gamma \max_{a_i \in A(s)} \Big[V^*(f(s,a_i))\Big]. \tag{3.3}$$

There are two training parameters: $\alpha \in [0,1]$ is the learning rate and determines how the agent values old and new knowledge. With $\alpha = 1$ the agent completely discards old knowledge and with $\alpha = 0$ the agent does not change at all. $\gamma \in [0,1]$ is the discount factor. It determines how the agent prioritize between quick and delayed rewards. $\gamma = 0$ would mean the agent only cares about immediate reward and $\gamma = 1$ makes the agent value all rewards equally.

A state gets updated using its best successor state. This process is described in Figure 3.1. When applied to a 2-player game, the value of the successor state will be calculated from the opponent's perspective. In this case, one would use

$$V^*(f(s,a_i)) = -V^*_{\text{opponent}}(f(s,a_i)) \tag{3.4}$$

to change it to the correct perspective.

The new input-output (state-value) pairs assigned in Equation 3.2 and 3.3 were used as training data for the neural network.

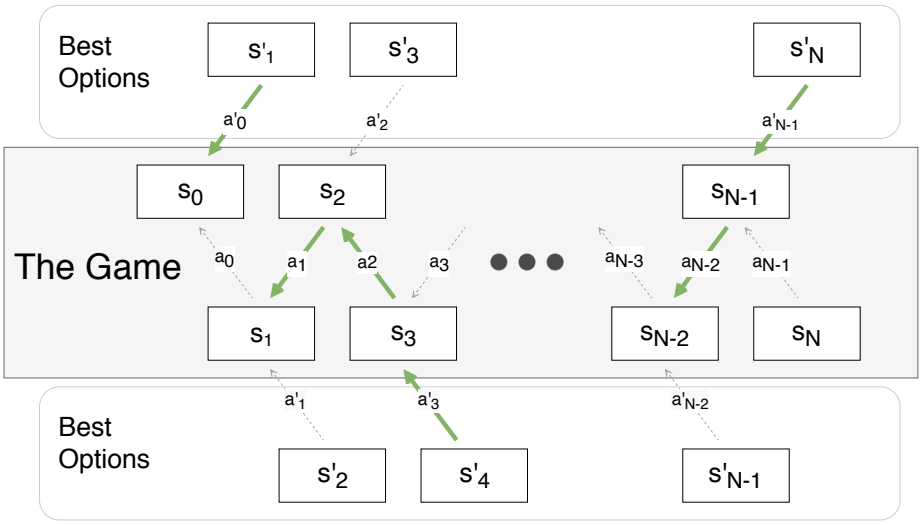

Figure 3.1: The training result backpropragates through the game, starting from the right in this diagram gradually moving to the left, back through the decisions made in the game. The value of the final state is assigned according to Equation 3.2 and the preceding states are updated recursively using Equation 3.3. The green arrows represent the use of the best successor states for the update. Either the value of a state visited in the game is used, or a better option.

3.2 Distributional Temporal Difference Learning

Distributional Temporal Difference Learning (DTD) is very similar to scalar TD-learning. The only difference is that it learns the distribution of the reward instead of the expectation value. I have used a kind of DTD where $\gamma = 1$ and the distribution consists of probabilities for ending the game with a win, tie or loss. This discrete distribution is approximated using a neural network,

$$g(s) \approx (p(win|s), p(tie|s), p(lose|s)) \tag{3.5}$$

where each component of the output vector is the probability of an outcome for the game in state s . The approximated distribution D^* is:

$$\begin{cases} D^*(win|s) & = g(s)_1 \\ D^*(tie|s) & = g(s)_2 \\ D^*(lose|s) & = g(s)_3 \end{cases} \tag{3.6}$$

In DTD g takes role of V^* as the function which is learned through the training process . The state value can be approximated using g:

$$V(s) = p(win|s) - p(lose|s) \approx g(s) \cdot (1, 0, -1) \tag{3.7}$$

where "$\cdot$" is a dot product.

Like the MINIMAX search in Section 2.3, this algorithm attempts to label the states as WIN, TIE and LOSE, but unlike MINIMAX, it is using soft labels. For example, a state could be labeled 30% win, 20% tie and 50% lose. In machine learning, this is usually called a classification problem, whereas learning the continuous values of $V^*(s)$ as described in the previous section, is a regression problem.

Decision Making

The agent selects the move with the highest expected reward:

$$a(s) = \underset{a_i \in A(s)}{\operatorname{argmax}} \Big[g(f(s, a)) \cdot (1, 0, -1) \Big] \tag{3.8}$$

Training

Similarly to TD-learning, DTD uses the result of the game to assign new values to g.. The update starts from the end of the game by assigning a new value to the final state,

$$g(s_{final}) \leftarrow \begin{cases} (1, 0, 0) & \text{for } r(s_{final}) = 1 \\ (0, 1, 0) & \text{for } r(s_{final}) = 0 \\ (0, 0, 1) & \text{for } r(s_{final}) = -1 \end{cases} \tag{3.9}$$

and recursively updates the values according to

$$\begin{cases} a = \text{argmax}_{a_i \in A(s)} \left[g(f(s, a_i)) \cdot (1, 0, -1) \right] \\ g(s) \leftarrow (1 - \alpha) g(s) + \alpha \, g((f(s, a)) \end{cases} \tag{3.10}$$

where $\alpha \in [0, 1]$ is the learning rate.

To get the correct perspective in a 2-player game you need to reverse $g(f(s, a_i))$ since the successor state will be evaluated from the opponent's perspective and $p(\text{player 1 winning}) = p(\text{player 2 losing})$.

3.3 Optimality

An action is optimal if it leads to an optimal outcome from the current state. We can calculate the probability of an action being optimal using g. Let $a_i \in A(s)$ and $g(f(s, a_i)) = (w_i, t_i, l_i)$, then

$$p(a_i \text{ optimal}) = p(\text{win}|a_i) + p(\text{tie}|a_i) p(\text{can't win}|a_j, j \neq i) + p(\text{can only lose in } s). \tag{3.11}$$

$$\iff p(a_i \text{ optimal}) = w_i + t_i \prod_{j \neq i} (1 - w_j) + \prod_j l_j \tag{3.12}$$

I will refer to $p(a_i \text{ optimal})$ as the *optimality* of a_i. The optimality shows how strongly connected the values for two consecutive states are. If we choose an optimal action, then the current state and the successor state should have the same ranking. If we make a bad move —choose an action with low optimality— then the current state isn't necessarily bad, but the next is.

3.4 Adaptive Distributional Temporal Difference Learning

The optimality can be used as an adaptive learning rate for DTD. By using $\alpha = p(a(s) \text{ optimal})$, we get a new algorithm with adaptive learning rate. I will call the agent using this algortithm ADTDBot.

3.5 My Original Algorithm

With optimality as an adaptive learning rate, it is no longer necessary to learn from the best successor state. It is possible to simply use the actions and successor states from the game for training. That is the idea behind this algorithm. Since this agent is my own creation I will call it BerglindBot.

The decision making is performed like in DTD. The agent is trained using the following update rule,

$$g(s) \leftarrow (1 - p(a \text{ optimal}))g(s) + p(a \text{ optimal})g(f(s, a)) \tag{3.13}$$

where a is the action selected in state s during the game.

3.6 Exploration

If the agent always selects the best action it might not discover new and potentially better policies. Furthermore, a neural network needs diverse data to learn well and an agent using its best policy is likely to play quite repetitively. I devised a simple method for adding a good amount of randomization without completely ignoring prior knowledge. Let $\hat{V}(s)$ be the approximated expected reward, either calculated directly using $V^*(s)$ or as $g(s) \cdot (1, 0, -1)$. Let

$$\hat{M} = \max_{a_i \in A(s)} \hat{V}(f(s, a_i)) \tag{3.14}$$

then randomly select an action in the set

$$A_{selected} = \{a_i \in A(s) | \hat{M} - 2\epsilon \leq \hat{V}(f(s, a_i)\} \tag{3.15}$$

where $\epsilon \in [0, 1]$ is the exploration parameter.

If $\epsilon = 0$, the agent will, according to its current knowledge, select the best possible move. If $\epsilon = 1$, the agent will play completely randomly. If $\epsilon \in (0, 1)$, the agent should play somewhat randomly, but avoid making any critical mistakes. As the training goes on the agent will get better at distinguishing good and bad moves and it should play less and less randomly. If I did not specify the exploration in an experiment setup, it is set to $\epsilon = 0.01$.

If we assume the approximation error for $\hat{V}$ is less than ϵ,

$$\left|\hat{V}(s) - V(s)\right| \leq \epsilon \qquad \forall s \in S \tag{3.16}$$

then the best action will be in $A_{selected}$. Proof:

Let a^* be the best action in state s,

$$a^* = \underset{a_i \in A(s)}{\operatorname{argmax}} V(f(s, a_i)) \tag{3.17}$$

and M the true value for the successor state,

$$M = V(f(s, a^*)) \tag{3.18}$$

then

$$\begin{cases} \hat{M} \leq M + \epsilon \\ M - \epsilon \leq \hat{V}(f(s, a^*)) \end{cases} \tag{3.19}$$

$$\iff \begin{cases} \hat{M} - \epsilon \leq M \\ M \leq \hat{V}(f(s, a^*)) + \epsilon \end{cases} \tag{3.20}$$

$$\implies \hat{M} - 2\epsilon \leq \hat{V}(f(s, a^*)) \iff a^* \in A_{selected}. \quad \square \tag{3.21}$$

3.7 The Opponent

To measure the performance of the agents, it is necessary to use a static reference point. I have previously designed a 5-in-a-row agent which uses heuristics for decision making. It creates a ranking for each cell of the board based on how good opportunities the players have and selects the cells with the highest ranking. It is fast, systematic and plays the game at the level of an intermediate player. I will refer to this agent as QuickBot.

Let $O(c, \text{player})$ be the set of all opportunities (possible ways to get 5 in a row) for the player using the cell c and let $n(o)$ be the number in a row it has in this position already. The occupied cells get ranking 0 and the ranking of an empty cell c is

$$\text{ranking}(c, player) = 1.5 \cdot \sum_{o \in O(c, player)} f(n(o, \text{player})) + \sum_{o \in O(c, \text{opponent})} f(n(o, \text{opponent})), \tag{3.22}$$

where $f(n)$ is a scalar function representing the ranking of having n in a row. I have been using

$$f(n) = \begin{cases} 0 & n = 0 \\ 5^n & n > 0 \end{cases} \tag{3.23}$$

This could probably be fine tuned through evolutionary learning, but I have just hand tuned it. The 1.5-factor is added to account for the advantage of playing the next move.

3.8 Training Process

The training is performed as a sequence of training iterations. A training iteration is described in Figure 3.2 and consists of 3 steps:

1. **Training Games.** Each iteration starts with N (usually set to 50) practice games. After each game the end result is used to assign a new output value to the states in the game. This is done by iterating back from the final state using the training formulas for the different algorithms. The new state/output pairs are saved for training the neural network.

2. **Network Update.** The network is trained for five epochs using the data generated in the previous step.

3. **Evaluation Games.** The agent, using it's optimal policy ($\epsilon = 0$), plays 10 evaluation games against QuickBot starting with 1 random move. This is only done to track the progress and the data is not used for training.

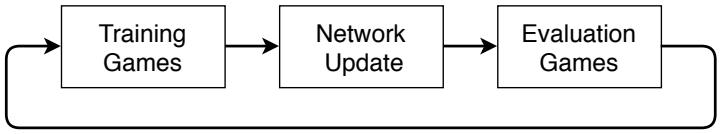

Figure 3.2: A training iteration consists of training games, neural network update and evaluation games.

Most agents have been trained for 1000 iterations of 50 games. Throughout the training the score and length of each game is saved and this can be used to analyse the learning process.

3.9 Tournament

For comparing the performance of the different agents I implemented a tournament with 2000 games between each pair of agents. The first 4 moves are randomized to avoid the agents playing the same game over and over and to test their ability to handle unfamiliar situations.

3.10 Implementation

The algorithms were implemented as similarly as possible. They shared most of the code; only the decision making and calculation of new outputs were done separately. They analysed the board using convolutional neural networks of identical design except the output size, output activation and loss function. The neural networks were implemented in Keras using the TensorFlow backend. The training was done on a GPU server and the SuperMike2 super computer at Louisiana State University.

The input of the network was divided to 2 channels, just like two colors in an image. The first channel represented the board positions of the player who placed the most recent move. The second channel represented the board positions for the opponent. Both channels consisted of binary 11 by 11 arrays where a cell is 1 if the player has a marker in this position and 0 otherwise. A similar input representation was used in AlphaGo [19] and it's successors.

To make decisions in the game, each possible action in the current state was evaluated by analysing the successor state using the neural network and this data was saved for generating new training data. When a game had finished, new training data was generated by iterating backwards through the game using the update formulas presented in Section 3.1 to 3.5. To get eight times more training data, the input is rotated and reflected according to the symmetries of the game. This is the only knowledge about the game, apart from the rules, used by the agents.

3.11 Neural Networks

I have tried many network architectures and these are specified for the different experiments. Here, I will describe the properties they all have in common.

The structure of the board makes the game naturally suitable for convolutional neural networks. We can tell the quality of a state by looking at local patterns on the board; having 5 in a row means the same thing all over the board. A convolutional network naturally has an "intuition" for this property. Using a fully connected network would be like looking at a reordered version of the board where you would have to learn the relationship between the different cells and this game would be very hard for humans to learn. Therefore, convolutional neural networks were used throughout the project.

TDBot used neural networks with a single value as output and the output activation function tanh since $V(s) \in [-1, 1]$. The networks were trained using the Mean Squared Error (MSE) loss function,

$$\text{MSE}(V(s), \hat{V}(s)) = \left(\hat{V}(s) - V(s)\right)^2. \tag{3.24}$$

where $\hat{V}(s)$ is the target value and $V(s)$ is the current estimate from the neural network for state s. This is common practice for regression tasks [15].

The distributional algorithms used neural networks with three outputs processed through a softmax activation function since they represent three probabilities from the same distribution. The network used categorical cross entropy loss,

$$\text{CrossEntropy}(g(s), \hat{g}(s)) = -\sum_i \hat{g}(s)_i \log(g(s)_i) \tag{3.25}$$

where $g(s)_i$ is the i'th component of $g(s)$, $\hat{g}(s)$ is the target value and $g(s)$ is the current output of the network. This is common practice when using softmax [15].

All layers except the output used ReLu activation and uniform He initialization. According to [4] He-initialization is the best choice for layers with ReLu activation. If the activation is tanh, sigmoid or softmax, Glorot-activation is a good choice. This has been used consistently throughout my experiments. All networks were trained using an Adam optimizer with the common default parameters $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$.

Chapter 4

Experiments

I have tried many different hyper parameter values and network architectures, in total, more than 100 different setups. In the following sections, I will describe some of these experiments and how the results of one experiment leads to the construction of the next. For each experiment, there is a figure of the training data showing the score, the length of the training games and the loss function. Moreover, there is a graph of the evaluation data displaying the average score and length of the evaluation games. Section 3.8 describes how this data is generated. I will mostly discuss the evaluation results since this is a better indicator of the performance of the algorithms, but the training data of the experiments is included for completeness.

4.1 Initial Network Design

After some initial experiments, I found a network architecture that learned to play the game quite well.

Setup

All agents practiced against QuickBot for 50 000 games with 50 games per iteration, updating the neural network after every 50th game. The network architecture is presented in Table 4.1. First, a 5x5 convolution is used to analyse the raw board. This helps the network see the patterns of length 5 which are important for the game. Next, the data is processed through three layers of 3x3 convolutions. Before calculating the outputs through two dense layers, a 1x1 convolution reduces the dimensions to avoid having too many parameters in the dense layers. This technique was also used in AlphaGoZero [20]. All convolutional layers have batch normalization before the activation function.

Type of layer	Output channels/units	Padding	Parameters
5x5 convolution	64 channels	No padding	3 500
3x3 convolution	64 channels	1x1 padding	37 000
3x3 convolution	64 channels	1x1 padding	37 000
3x3 convolution	64 channels	1x1 padding	37 000
1x1 convolution	16 channels	1x1 padding	1 100
Dense	64 units		50 000
Dense	1 or 3		$\sim$ 100

Table 4.1: The initial network architecture. The input is at the top of the table and the output layer is in the bottom.

Results

The data from the training games is presented in Figure 4.2 and the data from the evaluation game is shown in Figure 4.1.

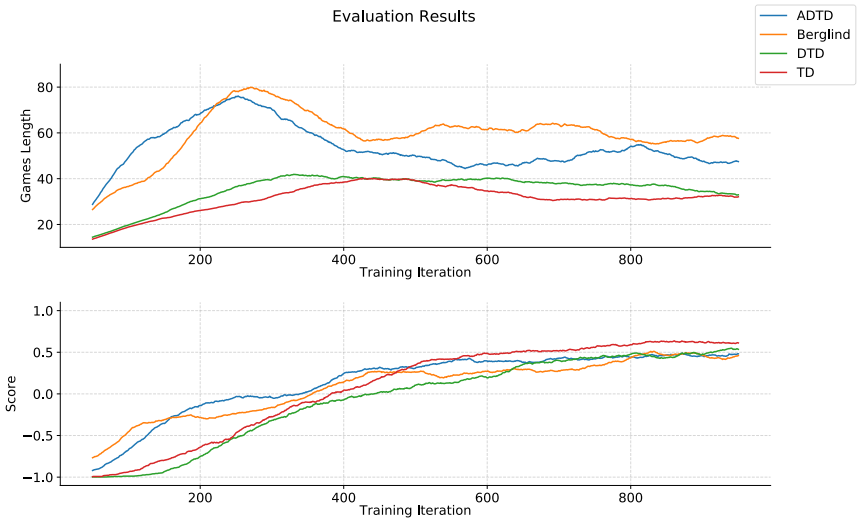

Figure 4.1: Evaluation results for the different algorithms using the initial network architecture. This is the results of the evaluation games played at the end of each training iteration using the current best policy ($\epsilon = 0$). The graphs have been smoothed using a running average over 100 iterations. The same amount of smoothing has been applied to all graphs displaying evaluation results.

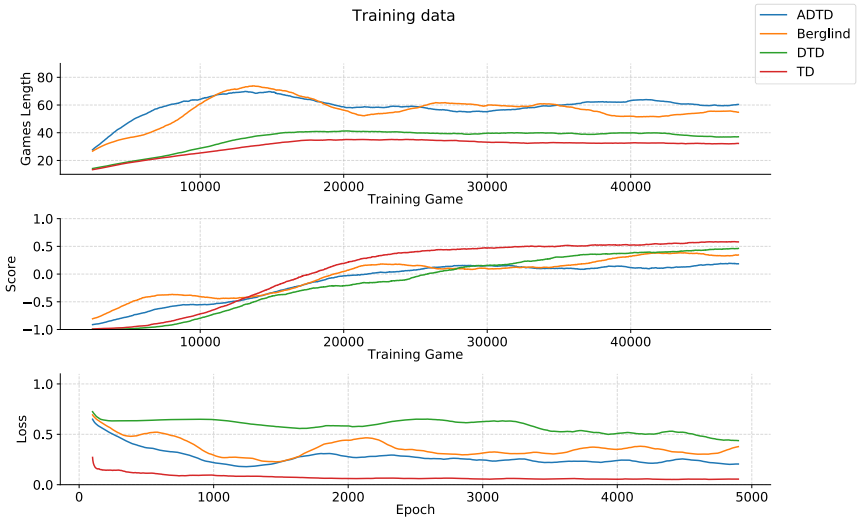

Figure 4.2: Training data for the different algorithms using the initial network architecture. The score and game length presented here are from the training games played to generate training data. These game are played using a suboptimal policy ($\epsilon = 0.01$). The length and score graphs have been smoothed using a running average over 5000 games and the loss function is smoothed over 200 epochs. The same amount smoothing has been applied to all graphs displaying training data

A tournament played between all algorithms using this network and QuickBot resulted in the scores in Table 4.2.

TDBot	2652
ADTDBot	535
DTDBot	21
BerglindBot	-716
QuickBot	-2492

Table 4.2: Total score from a tournament of 2000 games played between each pair of players.

Discussion

From the tournament (Table 4.2) and the evaluation games (Figure 4.1) it is clear that TDBot performs the best with this network architecture reaching the highest score in the evaluation games and the tournament, however BerglindBot and ADTDBot learn faster in the beginning. The use of optimality as an adaptive learning rate appears to speeds up the beginning of the learning process.

There are some interesting patterns in the evaluation result (Figure 4.1). The length of the games seems to be negatively correlated to the score. For example, TDBot has the highest score in the end, but played the shortest games and BerglindBot played long games, but had the lowest score. Furthermore, peaks in the game length often correspond to valleys in the score. This can be seen in the evaluation result for ADTDBot and BerglindBot. It indicates that they are developing a too defensive strategy during some parts of their training.

4.2 Larger Networks

To improve the result, I tried increasing the size of the network by making it deeper (400 000 parameters) or wider (600 000 parameters), but neither seemed to improve the performance. The evaluation results are shown in Figure 4.3. To avoid making the graph too cluttered, only ADTDBot and TDBot are included, but the other agents showed similar results. In general, the distributional algorithms seems to improve slightly with a larger network, but not enough to outperform the initial TDBot.

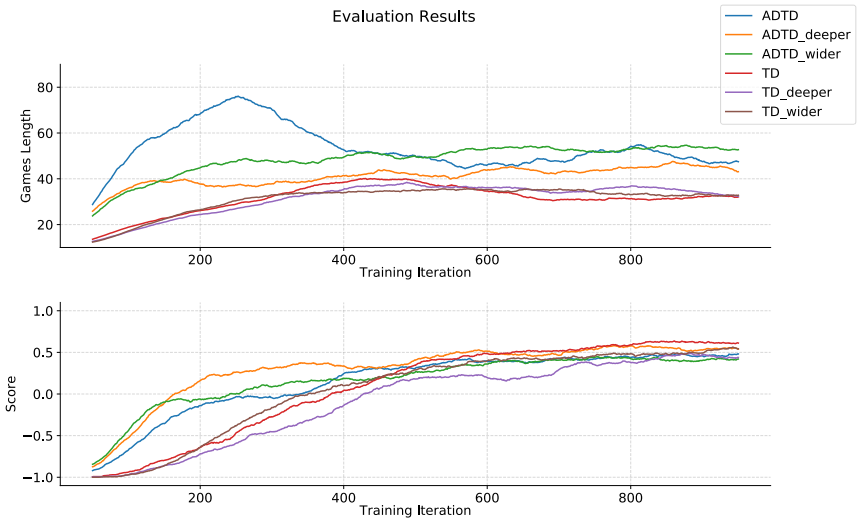

Figure 4.3: Evaluation result of ADTDBot and TDBot with three different network designs.

4.3 Batch Size

In the previous experiments, the neural network was trained using a batch size of 32 for the Adam optimizer . In this experiment, I tried increasing it. A larger batch size is beneficial for practical purposes since it makes it possible

to take advantage of the parallelization of a GPU and significantly speed up the computation. However, it often results in worse generalization and slower convergence [5]. Larger batch size lead to fewer gradient decent steps per epoch, but the estimated gradients are more accurate. This makes it harder to follow the curvature of a highly nonlinear surface, but locally the more precise gradient estimate improves the accuracy.

It would be best to try a range of batch sizes for all algorithms, but I didn't have the time or the computational resources to run all those experiments. Therefore, I only tried training ADTDBot with the batch sizes 32, 64, 128, 256 and 512 and assumed there would be similar results with the other algorithms.

Result

The result is displayed in Figure 4.4 and 4.5. The data for batch size 32 is the same as in Section 4.1. The training time with batch size 64 was 16h and with batch size 512 this was reduced to 10.5 hours. However, several models were being trained at the same time on the same processor, so these numbers are not very accurate. Nevertheless, it is clear that a larger batch size significantly decreases the training time.

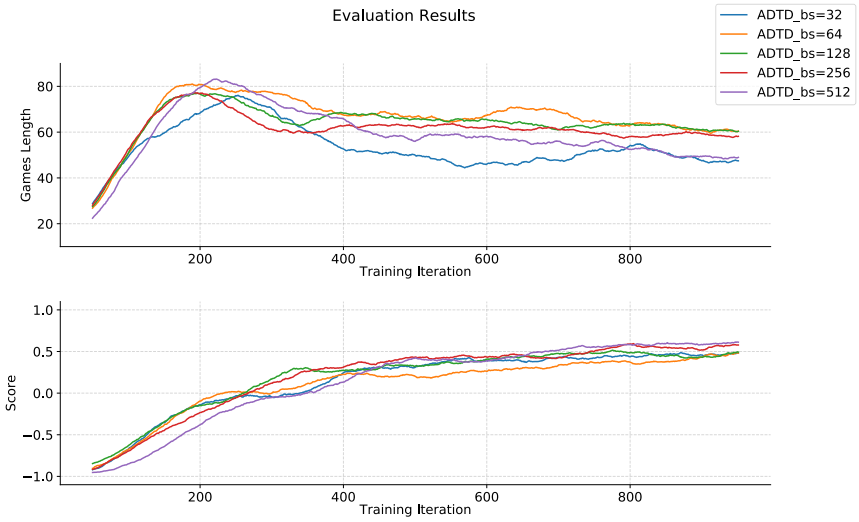

Figure 4.4: Evaluation result of ADTDBot with batch size ranging from 32 to 512.

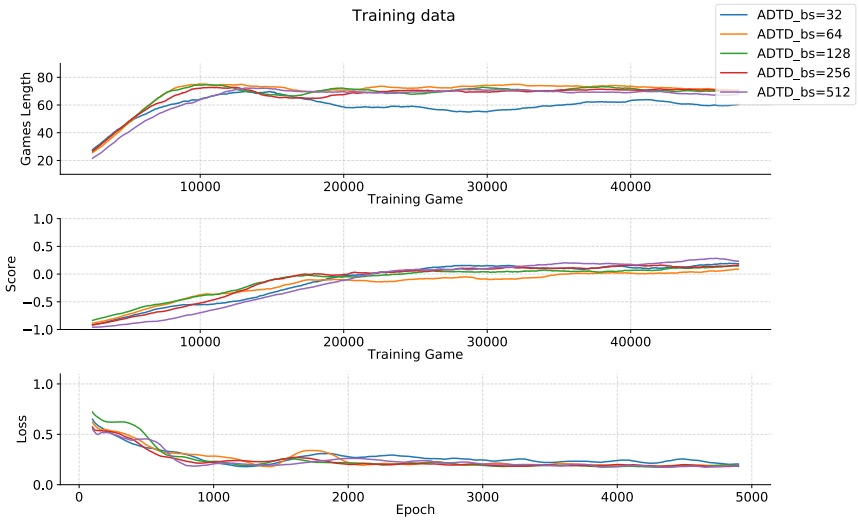

Figure 4.5: Training result of ADTDBot with batch size ranging from 32 to 512.

Discussion

Surprisingly, a larger batch size improves the results. It learns slower in the beginning but achieves a better result in the end. With lower batch size, the training seems slightly unstable; the game length and score are fluctuating during the training, while a larger batch size leads to a more smooth and steady progress (compare the blue and purple curves in Figure 4.4). Since the network with larger batch size trains faster (in the sense of computational time), reaches a better end result and shows more stable behavior, I will use a batch size of 512 for the remaining experiments.

4.4 Residual Network

The residual network architecture (ResNet) was introduced in 2015 and showed state of the art performance in several image classification tasks [7]. AlphaGoZero and AlphaZero used a simplified ResNet Architecture and it showed greatly improved performance compared to a similar sequential network [20][18]. With these impressive results in mind, I decided to try using a residual network architecture.

ResNet

A ResNet is comprised of blocks of two convolutional layers with a skip connection which adds the input of the block to the output. This reduces problems with vanishing or exploding gradients and makes it possible to train deeper

networks. I used blocks similar to those used for AlphaZero. These blocks are described in Table 4.3 and Figure 4.6. The network design is specified in Table 4.4. AlphaGoZero and AlphaZero used L2 weight normalization, but when I tried this, it prevented the network from learning properly, so I removed the weight normalization.

Type of layer	**Output channels/units**	**Padding**	**Parameters**
3x3 convolution	64 channels	1x1 padding	37000
Batch normalization	64 channels		256
ReLU	64 channels		0
3x3 convolution	64 channels	1x1 padding	37000
Batch normalization	64 channels		256
Skip Connection	64 channels		0
ReLU	64 channels		0

Table 4.3: A residual block. The input is processed through two 3x3 convolutions. The output of these convolutions is added to the input. Finally, this sum is processed through a ReLu nonlinearity.

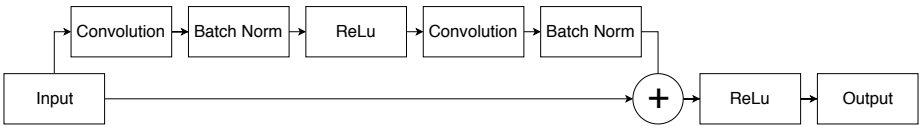

Figure 4.6: A residual block. The input is processed through two 3x3 convolutions. The output of these convolutions is added to the input. Finally, this sum is processed through a ReLu nonlinearity.

Type of layer	Output channels/units	Padding	Parameters
5x5 convolution	64 channels	No padding	3 500
Batch Normalization	64 channels		256
ReLu	64 channels		0
ResBlock	64 channels	1x1 padding	74 000
ResBlock	64 channels	1x1 padding	74 000
ResBlock	64 channels	1x1 padding	74 000
ResBlock	64 channels	1x1 padding	74 000
ResBlock	64 channels	1x1 padding	74 000
1x1 convolution	1 channel	No padding	65
Batch Normalization	1 channel		4
ReLu	1 channels		0
Dense	64 units		8000
Dense	1 or 3		~ 100

Table 4.4: The residual architecture used for these experiments. It has a total of 380 000 parameters.

With $\gamma = 1$ for all batch normalization layers the network initially outputs very large values. For scalar TD-learning, the output was often very near -1 or 1. With such an initialization the untrained agent would be almost certain some states lead to a win and other states lead to a loss. As the agent trains, these errors would spread to the training data and make the training unstable. To solve this issue I set $\gamma = 0.1$ for the first batch normalization layer and this resulted in reasonable starting outputs around -0.1 to 0.1 for TDBot.

Result

The data from the training games is presented in Figure 4.8 and the data from the evaluation games is shown in Figure 4.7.

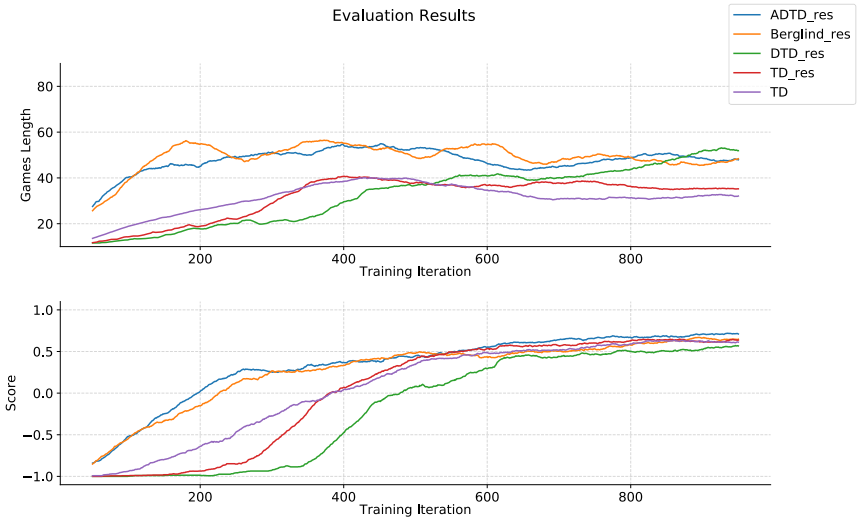

Figure 4.7: Evaluation results for the different algorithms using the residual network architecture. I included the previous best player, TDBot for reference.

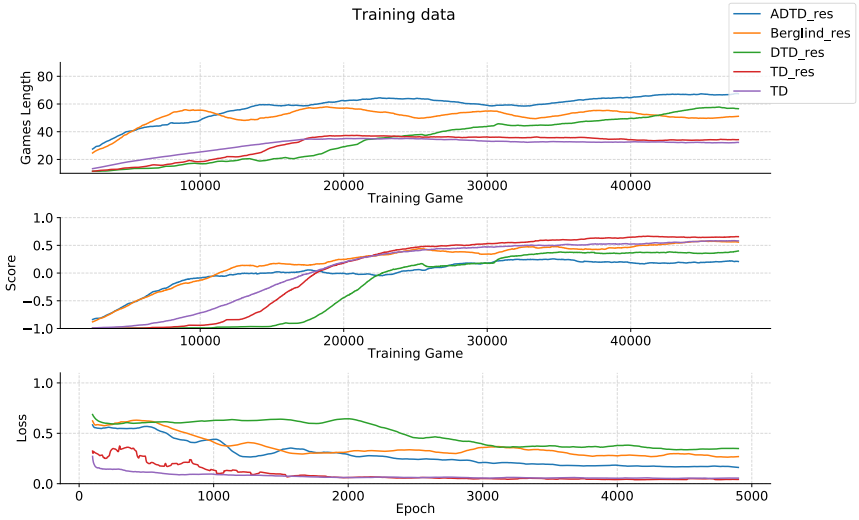

Figure 4.8: Training data for the different algorithms using the residual network architecture. I included the previous best player, TDBot for reference.

A tournament played between all algorithms using this network, TDBot using the initial convolutional network and QuickBot resulted in the scores in

Table 4.5.

ADTDBot ResNet	1714
TDBot	1661
TDBot ResNet	1548
BerglindBot ResNet	38
DTDBot ResNet	−302
QuickBot	−4659

Table 4.5: Total score from a tournament of 2000 games played between each pair of players.

Discussion

ADTDBot using a residual network outperforms TDBot. In Figure 4.7, we can see that it learns faster and reaches a higher end score. It also achieves the highest score in the tournament. Quite surprisingly, TDBot does not seem to benefit from the ResNet, but the performance is quite similar. If we compare Figure 4.7 and 4.1, it seems like a ResNet overall gives a better result.

Looking at Figure 4.7, it seems like BerglindBot is performing similarly to TDBot, but in the tournament, it performs quite poorly. Unlike the other algorithms, it is only learning from the actions selected in the game. Therefore, it learns directly from the strategy of QuickBot and does not adapt well to other opponents.

4.5 Improved Residual Networks

The residual network in the previous section is based on a paper from 2015 [7]. Since then there has been plenty of research attempting to understand and improve the residual architecture. I tried applying a couple of these changes to my network architecture.

Set Up

Initializing the γ-parameter of the final batch normalization of each block to zero and $\gamma = 1$ for all other batch normalization layers will cause each block to start out as an identity function letting the signal pass unchanged through the network. Goyal et al claims that such an initialization ease optimization in the beginning of training and improves all models but is especially helpful for large mini batch training [6].

According to He et al, by reordering the layers of a residual block and keeping the skip connection free of any non-linearities, the network gets easier to train and generalizes better [8]. The new ResBlock is described in Figure 4.9. He et al creatively named their new network architecture ResNet2.

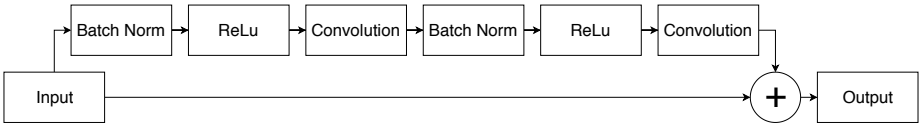

Figure 4.9: The new residual block.

I separately applied these modifications to two residual networks and trained ADTDBot for 50 000 games using these networks.

Result & discussion

The results are presented in Figure 4.10 and 4.11. Unfortunately, these changes seem to decrease the performance of the network and I decided to keep the previous ResNet architecture instead.

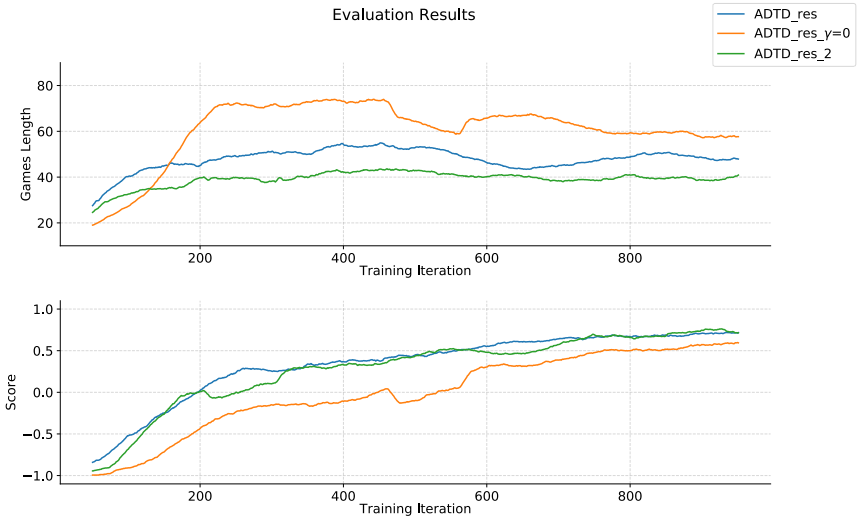

Figure 4.10: Evaluation results for ADTDBot using the different residual networks.

Figure 4.11: Training data for ADTDBot using the different residual networks.

4.6 Self Play

Although the results with the first ResNet (Section 4.4), were quite satisfactory, there were some problems. The agents tended to develop a preference for a single cell on the board, often the center, and as soon as they have this cell, they would think they are almost guaranteed to win. From there on, a policy with $\epsilon > 0$ would almost be random since the player is certain it will win anyway. Furthermore, the agent developed a strong strategy against QuickBot, but could act quite erratically when it played against a different opponent, for example when playing against itself. I think this is because its exploration is limited by the playing of QuickBot; the agent was trained to play against QuickBot, not to play the game in general. To solve these problems, I let the agents train by playing against themselves. Since ADTDBot and TDBot have shown the most promising results I focused on these algorithms.

First Experiment

First, I tried using the simple initial hyperparameters and network architecture (shown in Table 4.1). However, even though this setup worked fairly well when playing against QuickBot, it did not work well for self play. The amount of exploration ($\epsilon = 0.01$) was not sufficient for the agents to discover all winning configurations. They ended up frequently losing games immediately when playing against QuickBot because they didn't realize they were about to lose.

More Exploration

To improve upon the previous result, I tried using more exploration, $\epsilon = 0.05$ or $\epsilon = 0.1$, and the ResNet design from Table 4.4. TDBot showed promising results, but was learning slowly, so I trained it for another 1000 iterations. These results are shown in Figure 4.12 and 4.13. In Figure 4.12, we can see that ADTD learns just as well through self play as it did while training against QuickBot. However it is no longer specialized on playing against Quick Bot and is likely to show better performance against other opponents. To put this to test, I ran a tournament between these agents and QuickBot.

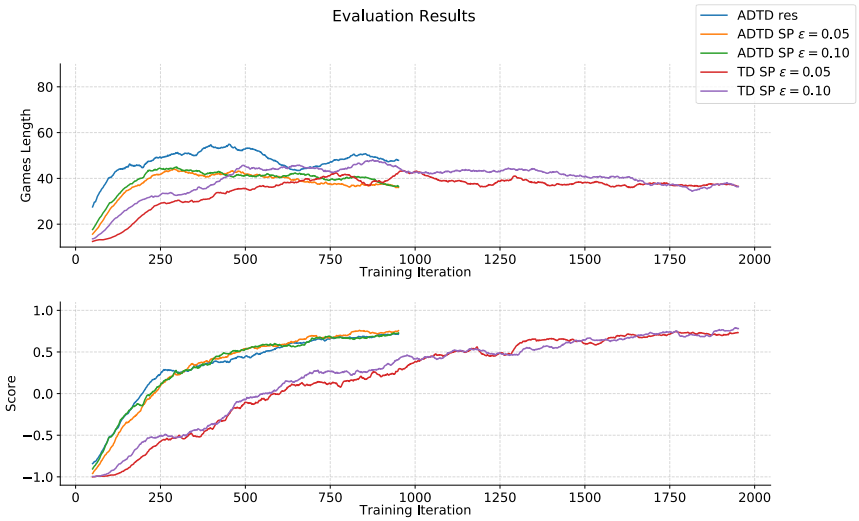

Figure 4.12: Evaluation result of TDBot and ADTDBot using a ResNet, self play and exploration $\epsilon = 0.05$ and $\epsilon = 0.1$. I let TDBot train for an extra 1000 iterations since it was still improving at the end of the first 1000 iterations.

Figure 4.13: Training data of TDBot and ADTDBot using a ResNet, self play and exploration $\epsilon = 0.05$ and $\epsilon = 0.1$. I let TDBot train for an extra 50 000 games since it was still improving at the end of the first 50 000 games. Note that the score graph only shows random noise since the agents are playing against themselves.

TDBot Self Play	$\epsilon = 0.05$	2112
TDBot Self Play	$\epsilon = 0.10$	1946
ADTDBot Self Play	$\epsilon = 0.05$	1290
ADTDBot Self Play	$\epsilon = 0.10$	680
ADTDBot ResNet	$\epsilon = 0.01$	−243
QuickBot		−5785

Table 4.6: Total score from a tournament with 2000 games played between each pair of players.

The results from the tournament are shown in Table 4.6. TDBot performed the best, but it trained twice as long as ADTDBot. Both algorithms benefited from self play and greatly outperformed the ADTDBot who practiced against QuickBot.

Improving TDBot

In all experiments so far, TDBot has used a learning rate of $\alpha = 0.3$. This was based on some early experiments training against QuickBot using a small neural network. For consistency, I have kept the same α since then, but seeing that

TDBot learns a lot slower than ADTDBot, it could be worth trying to increase α.

In this experiment, I tried learning rates ranging from the old value of 0.3 to 0.9 with exploration $\epsilon = 0.1$. The results are shown in Figure 4.14 and 4.15.

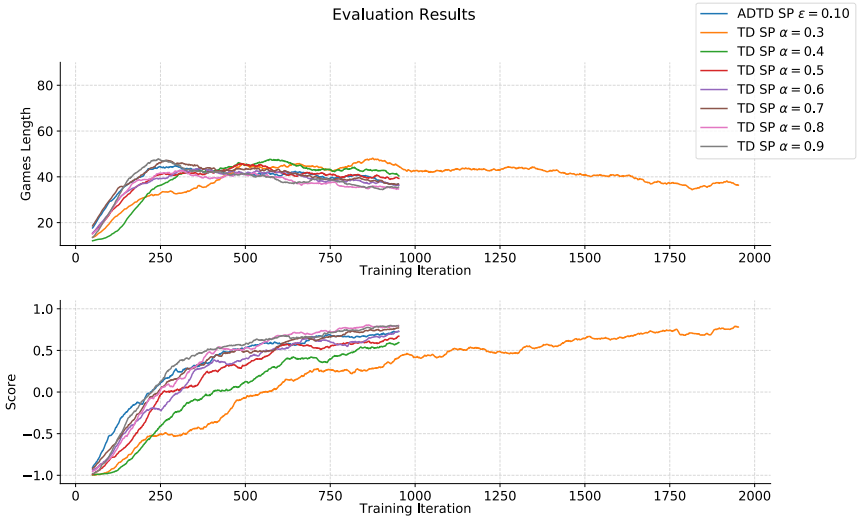

Figure 4.14: Evaluation result for TDBot with learning rate from $\alpha = 0.3$ to $\alpha = 0.9$. Note that the data for $\epsilon = 0.3$ is the same as in Figure 4.12.

Figure 4.15: Training data for TDBot with learning rate from $\alpha = 0.3$ to $\alpha = 0.9$. Note that the data for $\epsilon = 0.3$ is the same as in Figure 4.13.

It is clear that the previous learning rate of 0.3 was far too low. A learning rate of 0.8 or 0.9 seems to be a good choice. For my final experiment, I will use a learning rate of 0.8 since training with a lower learning rate is more stable and likely to be better later in the training.

4.7 Final Experiment

The previous experiments with self play have shown really promising results. As a final experiment, I ran a similar setup with all four algorithms for 200 000 games to see how they converge and how far they can go.

Setup

Just like in the previous experiments, I used a residual network and self play. To give the network larger data sets for training, 100 instead of 50 games were played per iteration. The agents all trained for 2000 iterations, a total of 200 000 games. Since the tournament result in Table 4.6 indicate that $\epsilon = 0.05$ is a good amount of exploration, I used this for all algorithms. In accordance to the result in the previous experiment, TDBot and DTD used the learning rate $\alpha = 0.8$.

Result

The result is presented in Figure 4.16 and 4.17. BerglindBot achieved an average score of 0.81 at the end of training, ADTDBot reached 0.9 , TDBot 0.92, and DTDBot 0.95.

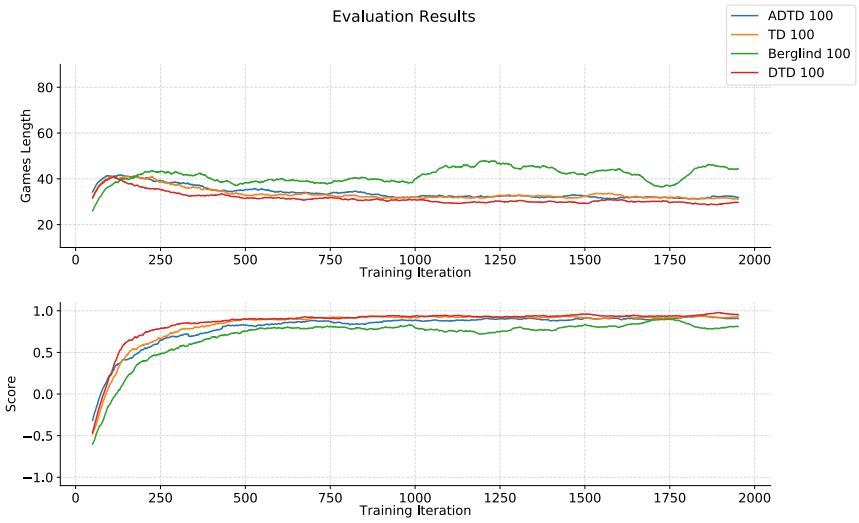

Figure 4.16: Evaluation result with all four algorithms using self play, ResNet and 100 games per iteration. Note that one iteration contains twice as many games as in the previous experiments and the graph contains twice as many iterations. This results in 4 times longer training.

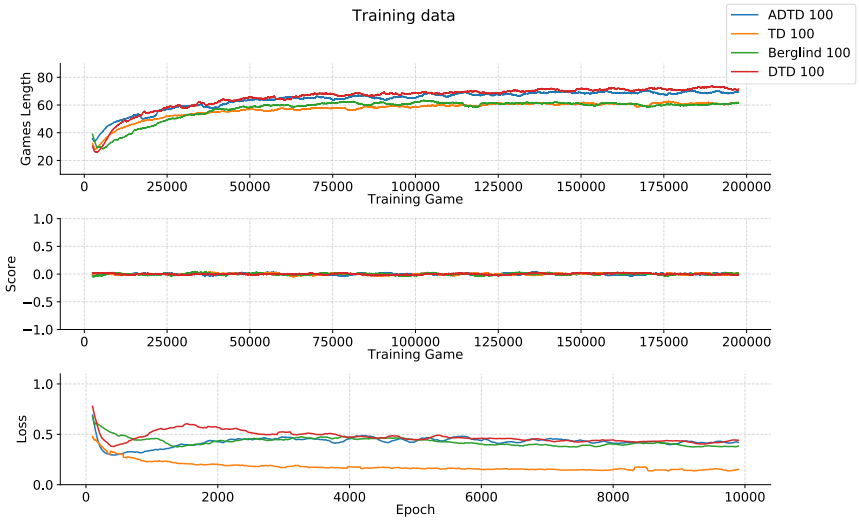

Figure 4.17: Evaluation result with all four algorithms using self play, ResNet and 100 games per iteration. Note that this experiment had 4 times longer training.

The result from a tournament played between these agents and QuickBot is shown in Table 4.7.

TDBot Final	2416
DTDBot Final	2173
ADTDBot Final	972
BerglindBot Final	705
QuickBot	-6266

Table 4.7: Total score from a tournament with 2000 games played between each pair of players. Each game started with four random moves.

To get a comparison between the players in an environment more similar to the training games, I ran a tournament with a single random move in the beginning of each game. Since there are less possible starting positions, I only ran it for 400 games between each pair of players to avoid wasting computations on repeated games.

DTDBot Final	655
TDBot Final	533
ADTDBot Final	272
BerglindBot Final	-29
QuickBot	-1431

Table 4.8: Total score from a tournament with 400 games between each pair of players. Each game started with one random move.

Discussion

Training through self play using larger datasets clearly improved the results: all agents reach a higher evaluation score (see Figure 4.16), almost winning every game against QuickBot at the end training. This improvement should generalize to other opponents since they did not train against QuickBot. The larger data sets let the neural networks see a more complete picture of the game and learn more complex patterns. It reduces the risk of the network overfitting on the training data and it allows the agents to explore more options before settling on a strategy.

All algorithms learn the game quite well. However, BerglindBot shows the lowest performance. Since it learns from the actions selected in the game and not the best actions its performance suffers from the high exploration ($\epsilon = 0.05$) used in this experiment.

ADTDBot learns faster in the beginning, but does not converge as well as the algorithms with constant learning rate. In the end, DTDBot and TDBot show the strongest performance. DTDBot learns faster than TDBot. This might be because it is learning the more complex distribution instead of a scalar and thereby extracts more information from a single training game. It could also be because of the different neural network output layer and loss function. It has the highest score in the evaluation games (Figure 4.16) and got the highest score in the tournament with one random move (Table 4.8), but gets outperformed by TDBot in the tournament with four random moves(Table 4.7). This might be because the discount factor of TDBot makes the value function smaller in the beginning of the game and increases the randomness of the first move, making the training games more similar to the tournament.

Chapter 5

Discussions

Deep reinforcement learning is immensely complex and one has to be careful when drawing conclusions from these experiments. The results comes from the interaction of a deep neural network, the adaptive Adam optimizer and the reinforcement learning algorithms. This interplay of three components, all very complex on their own, makes it difficult to understand what is happening. For example, it is very hard to explain why TDBot achieved the best results with the initial network (see Figure 4.1), but ADTDBot worked the best when using a ResNet (see Figure 4.7). It is surprising that DTDBot performed well in the final experiment (Figure 4.16) after performing quite poorly in the previous ResNet experiment (Figure 4.7). Nonetheless, I will try to draw some conclusions.

5.1 Comparison of the Algorithms

In the end, it is clear that the algorithms with constant learning rate achieved the strongest results. However, throughout the experiments, ADTD has shown the most consistent performance and seems to learn more quickly in the beginning of the training. When I switched from training the agents against QuickBot to self play, the constant learning rate algorithms needed a different learning rate, while ADTD and BerglindBot showed good performance without needing any retuning. ADTD has consistantly been one of the best algorithms while DTD worked quite poorly in most of the experiments, to suddenly rise and perform very well in the final experiment. This might indicate that it was not using the best learning rate in the first experiments, or perhaps that the algorithm needs the stable conditions of self play to thrive.

Scalar TD consistently achieved strong results, it was easier to implement and it seemed less sensitive to the capacity of the neural network. The discount factor (γ) can be useful for encouraging winning more quickly making it less likely for the player to make mistakes. However, with the distributional algorithms, it is possible to modify the ranking function $g(s)$ (Equation 3.5) after training to customize the behaviour of the agent, for example making it play

more aggressively or defensively. This flexibility can be really useful for some applications.

BerglindBot showed weak generalization when training against QuickBot and reached the lowest score when using self play. This is a strong indicator of the benefits of generating training data using the best successor states (the maximization in Equation 3.3 and 3.9), instead of the states actually visited during the game.

5.2 Quality of the Results

The final results shown in Section 4.7 are very strong. All agents except BerglindBot reached an average score above 0.9 when playing against QuickBot. Most people play at a level similar to that of QuickBot and they would probably be greatly outperformed by the fully trained neural networks. Nonetheless, there are some simple changes that probably would improve the results, but I did not have the time for any more experiments.

The final experiment showed that training with larger datasets improved the results. It would probably be beneficial to go even further and use 200 games per iteration, but then it would take even longer to execute the training. The choice to simply use constant learning rates might have limited the performance. By gradually decreasing the learning rate of temporal difference learning, you can help the algorithms converge. A carefully tuned schedule decreasing the learning rate is likely to improve the result, but creating such a schedule can be quite laborious.

I designed the exploration method (see Section 3.6) to automatically decrease the randomization as the training advance. However, this might not have been sufficient for the best training result. By scheduling the exploration to start out with a large value, maybe $\epsilon = 0.5$ and gradually decrease, perhaps to $\epsilon = 0.01$, you can make the agent explore a wide range of options in the beginning of the training and later on use its knowledge to learn from games played at a very high level. Additionally, my exploration scheme might put too much trust in the neural network. It could be good to combine it with the classic epsilon greedy strategy: selecting completely random actions with probability ϵ, questioning the prediction of the neural network and perhaps learning that it was incorrect or confirming what it predicted.

5.3 Problems with Optimality as Adaptive Learning Rate

There are several problems when using optimality as an adaptive learning rate. The results indicate that it works quite well in the beginning of the training, often learning faster than with a fixed learning rate, but later on, it does not converge as nicely as an algorithm with a fixed learning rate. Theoretically, the adaptive algorithms are based on a false assumption. There are also some

practical problems: the adaptive algorithms are learning too "carefully", they are biased towards underestimation and the learning rate tend to increase over time.

Theoretically, the update formula using optimality is based on the assumption that the prediction for the successor state, $g(f(s, a))$ and the optimality of the action are independent, but this is clearly false since the prediction is in the formula for the optimality. However, the theoretical correctness of the optimality as a learning rate is not neccessarily a problem. Many machine learning algorithms are based on such assumptions and work fine. It's hard to avoid these paradoxes when trying to create knowledge out of nothing by learning new, better approximations using old approximations.

The idea of the optimality is "careful" learning: only learn what you know to be true (probabilistically speaking). If you are uncertain, then just change the value a little bit. However, the results indicate that it might still be better to learn something that could be wrong, than not to learn anything. For example, if we have a state with probabilities (0.9 ,0 ,0.1) with 80 successor states all with the probabilities (0.1, 0 , 0.9) it is clear that the value for the first state way too optimistic, but the optimality for any option would be really low, about 0.1, so the prediction for the state would approximately from shift from $(0.9, 0, 0.1)$ to $(0.8, 0, 0.2)$. Whereas with a fixed learning rate of 0.8 it would shift from $(0.9, 0, 0.1)$ to $(0.26, 0., 0.74)$. In a sense, the result with optimality is correct because any of those 80 successor states has a 10 percent chance of winning, so we can't know for certain that any of them is an optimal choice, but intuitively, the fixed learning rate gives a reasonable result.

As the example in the previous paragraph indicates, the adaptive algorithms struggle to learn to recognize a losing situation because it will lead to a low learning rate since it is very hard to show that there is no way out of the bad situation. On the other hand, a winning situation will lead to high learning rate and the information will propagate to the preceding state. With the change of perspective due to the alternating nature of the game, this will lead to underestimation. All basic TD algorithms have an overestimation bias due to the maximization in the Bellman Equation 2.1 and in this context it turns into an underestimation bias. This effect is amplified by the adaptive learning rate and this might be the cause of the lower end result of ADTD.

With the optimality as a learning rate we are avoiding to learn incorrect values by using the certainty as a learning rate. As the neural network is being trained it will learn to make more and more certain predictions; the optimality will increase. It is common practice to decrease the learning rate during the training process to help the reinforcement learning algorithm converge. Here, we get the opposite behavior and that might be problematic. The algorithm will continue to change it's predictions until it has solved the game and all states have hard labels, but it probably does not have the capacity to do this and might just collapse. I am just speculating about the behaviour here and I have not really seen any such collapse in my results. Perhaps the adaptive learning rate of the Adam optimizer helped stabilize the process.

Chapter 6

Conclusions

This project aimed to investigate the possible advantages of distributional temporal difference learning (DTD) and an adaptive learning rate in place of the classic scalar temporal difference learning for game playing using deep neural networks. All methods I tried performed well and developed strong strategies, but there were significant differences in their performance.

Distributional TD

My experiments indicate that DTD can achieve strong results, but it is more sensitive to tuning and the neural network capacity. The classic scalar TD works well and is easier to implement and tune than DTD. If one wants quick working result, then scalar TD is a good choice. For the best result and to gain more information, it can be worth experimenting with DTD.

Adaptive Learning Rate

Using the action's optimality as an adaptive learning rate leads to more consistent results without parameter tuning. It learns more quickly in the beginning, but does not converge as nicely as DTD with well tuned constant learning rate and has lower performance in the end. More research is needed to evaluate the usefulness of the optimality measure and an adaptive learning rate.

Learning From the Best Successor State

According to my results, it is very beneficial to learn from the best possible successor state, instead of simply using the successor state visited in the game. It allows the algorithm to explore options even though they were not actually played and helps the it learn more quickly and reach a better end result.

Residual Network

A residual network architecture helps improve the performance and enables the use of deeper networks. However, the output of the untrained network can get very large and cause poor conditions for the reinforcement learning. This can be solved by careful initialization of the residual network.

Self Play

Self play helps stabilizing the learning process and lets the agents freely explore the game from the perspective of both players. It leads to an overall stronger strategy and better generalization against new opponents.

6.1 Future Directions

In my experiments, the algorithms with constant learning rate reached the best end results, but I still think some of the new concepts are worth investigating further. Here, I will describe some ideas for moving further with this research.

Adaptive Learning Rate

I find the idea of adapting learning rate very appealing. Hyperparameter tuning is quite a tedious process and being able to adapt to new environments is one of the essential traits of intelligence. Although the optimality did not quite cut it, I think it could be a part of the puzzle. It showed strong performance in the beginning of the learning process, but later on got outperformed by the algorithms using a constant learning rate. Perhaps a solution combining the optimality measure, adaptive learning rate decay and a method for dealing with the uncertainty of the neural networks estimation could provide an effective adaptive learning rate.

Learning from Several Successor States

In temporal difference learning, we want to learn from the best successor state. The most successful algorithms in this project selects the successor state with the highest expected reward. However, when the choice of the best successor state is very uncertain it might be better to learn from several successor states. I believe the optimality could be useful as a measure of certainty and somehow be used to generate a weighted sum of the old estimate and several successor states to create the new estimate.

Using Optimality for Exploration

When exploring new strategies through training games, the expected reward might not be the best measure for selecting actions. The goal is to discover the optimal action and I think the optimality can be useful to select actions.

It should give a similar result to the expected reward, but encourage a more aggressive strategy and focus on trying to win unless it is quite certain it is impossible to win.

Q-learning

All algorithms used in this project can be modified to Q-learning. The Q-value is the expected reward given a selected action, $Q(s, a) = V(f(s, a))$. You can set up a neural network to output all Q-values, or corresponding distributions, for the possible actions in s, to get estimates for all successor states using a single network evaluation. This would speed up the algorithms significantly and make the execution time independent of the game's branching factor (size of the action space $A(s)$). For 5-in-a-row this would be a reduced execution time by a factor of almost 100. However, you would need to find a way for the network to encode the actions and avoid doing illegal actions. Q-learning would change the way the network generalize its predictions and it might lead to slower learning and a different end result, but it is impossible to know without trying it out. A comparison between the performance of Q-learning and the scalar TD-learning used in this project would be interesting.

Combining Algorithms

The distributional algorithms does not have a discount factor and this might be the reason why the scalar algorithm performed the best in many experiments. A distributional algorithm could be combined with scalar TD learning by using a two-headed neural network, similar to that used for AlphaZero [18], learning both the distribution and the discounted reward. Then the distribution could be used to make long term decisions, for smart exploration and perhaps an adaptive learning rate. The discounted reward could be used to promote winning as fast as possible, reducing the risk of unexpected problems due to inaccurate predictions. This combined algorithm might learn more quickly since it will generate twice as much training data each game. According to Silver et al. [20], the two headed network architecture used for AlphaGoZero and AlphaZero was beneficial due to improved computational efficiency, but more importantly the regularizing affect of the dual objectives. If the information in the distribution and the discounted reward is different enough, this network design should show similar benefits.

Other Applications

The exact same algorithms could be used for many other games. It would be interesting to see if the algorithms works just as well for more complex games like Chess, Othello and Go or a simpler game like Connect Four and whether the hyperparameters would need to be retuned for different games. These algorithms could also be used for single agent Markov Decision Processes with a discrete reward at the final state.

Bibliography

[1] Marc G. Bellemare, Will Dabney, and Rémi Munos. "A Distributional Perspective on Reinforcement Learning". In: *CoRR* abs/1707.06887 (2017). arXiv: 1707.06887. URL: http://arxiv.org/abs/1707.06887.

[2] *Exploration vs. Exploitation - Learning the Optimal Reinforcement Learning Policy*. Oct. 2018. URL: https://deeplizard.com/learn/video/mo96Nqlo1L8 (visited on 12/12/2019).

[3] Gary. *Applications of Reinforcement Learning in Real World*. Aug. 2018. URL: https://towardsdatascience.com/applications-of-reinforcement-learning-in-real-world-1a94955bcd12 (visited on 12/12/2019).

[4] Daniel Godoy. *Hyper-parameters in Action! Part II - Weight Initializers*. Dec. 2018. URL: https://towardsdatascience.com/hyper-parameters-in-action-part-ii-weight-initializers-35aee1a28404 (visited on 12/12/2019).

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016, pp. 26, 276.

[6] Priya Goyal et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *CoRR* abs/1706.02677 (2017). arXiv: 1706.02677. URL: http://arxiv.org/abs/1706.02677.

[7] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *arXiv preprint arXiv:1512.03385* (2015).

[8] Kaiming He et al. "Identity Mappings in Deep Residual Networks". In: *CoRR* abs/1603.05027 (2016). arXiv: 1603.05027. URL: http://arxiv.org/abs/1603.05027.

[9] Alex Irpan. *Deep Reinforcement Learning Doesn't Work Yet*. https://www.alexirpan.com/2018/02/14/rl-hard.html. 2018. (Visited on 12/12/2019).

[10] M.P.H. Huntjens L.V. Ailis H.J. van den Herik. "Gomoku Solved by New Search Techniques". In: *AAAI Technical Report FS-93-02* (1993). URL: https://www.aaai.org/Papers/Symposia/Fall/1993/FS-93-02/FS93-02-001.pdf.

[11] *Markov Decision Processes (MDPs) - Structuring a Reinforcement Learning Problem*. Sept. 2018. URL: https://deeplizard.com/learn/video/my207WNoeyA (visited on 12/12/2019).

[12] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[13] *Policies and Value Functions - Good Actions for a Reinforcement Learning Agent*. Sept. 2018. URL: https://deeplizard.com/learn/video/eMxOGwbdqKY (visited on 12/12/2019).

[14] *Q-Learning Explained - A Reinforcement Learning Technique*. Oct. 2018. URL: https://deeplizard.com/learn/video/qhRNvCVVJaA (visited on 12/12/2019).

[15] Stacey Ronaghan. *Deep Learning: Which Loss and Activation Functions should I use?* Aug. 2019. URL: https://towardsdatascience.com/deep-learning-which-loss-and-activation-functions-should-i-use-ac02f1c56aa8 (visited on 12/12/2019).

[16] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009, pp. 163–164. ISBN: 9780136042594.

[17] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM J. Res. Dev.* 3.3 (July 1959), pp. 210–229. ISSN: 0018-8646. DOI: 10.1147/rd.33.0210. URL: http://dx.doi.org/10.1147/rd.33.0210.

[18] D. et al. Silver. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." In: *TODO* (2017).

[19] D. et al. Silver. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489.

[20] D. et al. Silver. "Mastering the game of go without human knowledge". In: *Nature* 550 (2017), pp. 354–359.

[21] David Silver, Richard S. Sutton, and Martin Müller. "Temporal-difference search in computer Go". In: *Machine Learning* 87.2 (May 2012), pp. 183–219. ISSN: 1573-0565. DOI: 10.1007/s10994-012-5280-0. URL: https://doi.org/10.1007/s10994-012-5280-0.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018, pp. 64, 119–120. URL: http://incompleteideas.net/book/the-book-2nd.html.

[23] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018, pp. 8–10. URL: http://gameaibook.org.

Master's Theses in Mathematical Sciences 2019:E66
ISSN 1404-6342

LUTFMA-3370-2019

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/