Getting to Know the Captain's Mistress with Reinforcement Learning

Thor Bagge, 20114756
Kent Grigo, 20114876

Master's Thesis, Computer Science
June 2016
Advisor: Kasper G. Larsen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Reinforcement learning has been used to beat professional human players at Checkers, Chess, and Go. Reinforcement learning is a machine-learning method that uses rewards to train an agent in order to achieve some goal. The aim of this thesis is to develop a reinforcement-learning agent that learns how to play Connect Four by training against itself. The agent uses a neural network to evaluate board positions and trains the network using the TD(λ) algorithm. Furthermore, the agent is combined with the search methods minimax and Monte-Carlo Tree Search (MCTS). Configuring the hyperparameters of each algorithm turns out to be the most time-consuming process. The results of our experiments indicate that the number of states in Connect Four is not large enough to warrant a reinforcement-learning approach compared to MCTS. However, we find that increasing the complexity of the game makes it more difficult for MCTS to compete with reinforcement learning. This suggests that reinforcement-learning methods can be used as a powerful tool when solving problems that have a large state space.

Acknowledgements

We want to thank our advisor Kasper G. Larsen for discussing problems that we had while writing this thesis.

We want to thank Allan Madsen, Amanda T. L. Sandegaard, Janus B. Kristensen, Kasper G. Larsen, Laila Grahl-Madsen, Lasse B. Kristensen, Mathias V. Pedersen, Michael Nielsen, and Rolf Bagge for detailed feedback on previous drafts. A special thanks goes to Lasse B. Kristensen who vigilantly caught errors in our math and helped discuss the properties of partial differentiation that led to the proofs in the appendix.

Kent Grigo
I want to thank Thor Bagge with whom I had the pleasure to write this thesis. His detailed understanding of the concepts and his splendid ability to convey helped keeping me up-to-date.

I want to thank my family and friends for their interest in my ways of computer science.

I want to thank Amanda T. L. Sandegaard for her loving support.

Kent Grigo,
Aarhus, June 21, 2016

Thor Bagge
My deepest gratitude goes to Kent Grigo for not only his work, but also his companionship. If not for his immaculate attention to detail and inspiring work ethic, this thesis would only be a shadow of what it is today.

I want to thank my family for their unconditional love and support.

Finally, I want to thank of all of my friends for always being there for me whenever I need them.

Thor Bagge,
Aarhus, June 21, 2016

Contents

Chapter 1

Introduction

1.1 Connect Four

"An amusing story is that the game is [called the Captain's Mistress] from the fact that Captain Cook hid away in his cabin for hours playing this game, his crew supposing that he was locked away with a mistress. Whether true or not, it certainly gives the game a charming name" (Cyningstan, 2016). Today, The Captain's Mistress is more commonly known as Connect Four. Therefore, we will refer to it as Connect Four for the rest of the thesis.

Connect Four is a two-player, turn-based game that is played on an empty grid with 6 rows and 7 columns. Each player takes turns dropping a piece in a free column, and the piece is placed in the lowest free row of that column. As a result, the board always fills up from the bottom row. The objective is to have a sequence of four pieces: horizontally, vertically, or diagonally. Whoever completes such a sequence first wins the game. The game ends in a draw if neither player completes a sequence before the board is filled. Physical pieces are usually colored disks but we denote the players and the pieces as ⓧ and ⓞ, where ⓧ always starts.

In order to address moves and positions in Connect Four, we introduce a notation inspired by the algebraic notation used in Chess. Each column is denoted by the letters *a* through *g*, where *a* is the first column and *g* is the last column. The rows are denoted by the numbers *1* through *6*, where *1* is the bottom row and *6* is the top row. Figure 1.1 shows the notation. For example, dropping a piece in the first column of the bottom row will be denoted *a1* or dropping a piece in the middle column of the fourth row will be denoted *d4*. Even though a piece can only be placed in the first free row of a column, we will explicitly state the row for the sake of clarity.

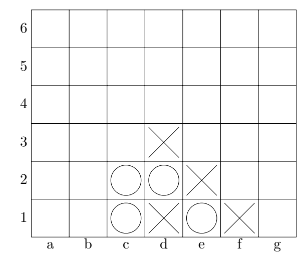


Figure 1.1: Example of a position in Connect Four.

A valid board position is called a *state* and figure 1.1 shows an example of a valid state. Looking at the state, we conclude that ⓧ is about to make a move since there are an even number of pieces and ⓧ always starts. In this state, ⓧ has three pieces in a row and needs a fourth to complete a sequence and win the game. We refer to three such pieces as a threat if there is an open space where the sequence can be completed. ⓧ would prefer to put his piece in *c4* to complete the sequence, but a piece has to be placed in *c3* first.

1.2 Motivation for Reinforcement Learning

Allis (1988) weakly solved Connect Four using a knowledge-based approach where he presented an algorithm with a winning strategy for the starting player from the beginning of the game. Since the solution is weak, the algorithm may not make optimal plays when the opponent makes non-optimal moves. This solution required expert knowledge about the game and the strategies applied to high-level play.

To strongly solve a game means to provide an algorithm that can produce perfect moves from any game position. The number of possible game positions is $4.5 \cdot 10^{12}$ for Connect Four (Edelkamp and Kissmann, 2008). Even with so many possible states, Connect Four has been strongly solved by brute force (Tromp, 2015). However, this result only applies to the standard 6×7 board. Increasing the board size will greatly increase the running time of a brute-force method and quickly render it infeasible.

Our motivation for using reinforcement learning is two-fold. First, we can create an algorithm without knowing anything about the game other than its rules. Second, a reinforcement-learning approach can be applied to variations of the game.

1.3 Machine Learning

In the most general sense, machine learning is learning from data to make predictions or decisions without being explicitly programmed to do so. The field of machine learning is broadly divided into three types of tasks based on the feedback given to the learner: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning is training on a set of labeled data. The learner has to generalize from training and predict the label of new unseen data. The signal given during training is the correct label for each data point. Unsupervised learning is training on a set of unlabeled data. The goal of unsupervised learning can be discovering patterns. Alternatively, the goal could be to give meaning to new data by clustering it with old data. Reinforcement learning is training by interacting with an environment. The learner interacts with the environment to achieve some goal, being rewarded along the way based on performance. The goal of the learner is then to maximize the reward (Abu-Mostafa et al., 2012).

1.3.1 Reinforcement Learning

This section presents reinforcement learning informally, and chapter 3 gives a more formal presentation. In reinforcement learning, the task is deciding which actions to take in order to achieve some goal. The learner and decision maker is called the *agent* and what it interacts with is called the *environment*. The environment is composed of everything outside the agent, and the state contains all the information of the environment that the agent needs to make decisions. The agent is not told what actions to take but is instead given a numerical reward by the environment based on which action is chosen. This means that the agent must try different actions in order to figure out which rewards they give through a process of trial and error. The agent chooses an action based on its evaluation of the environment's state.

Since "many researchers seemed to believe that they were studying reinforcement learning when they were actually studying supervised learning" (Sutton and Barto, 1998), we address the differences between these two paradigms. In reinforcement learning, the agent is not told whether an action is correct. In fact, there might not be a correct action. Instead the agent is given a reward signaling how good the action was. The agent must learn to control the environment by influencing it based on these rewards. This contrasts with supervised learning, where the feedback concerns whether the given answer is correct. The result is that the learner must change its behavior to match the instructions given by the environment.

The agent selects actions based on a policy, which is a mapping from states to actions. Whenever the agent encounters a state, the policy determines which action to take. Thus, the agent interacts with the environment according to its current policy. The ideal scenario is finding an *optimal policy*, i.e., a policy that selects the most rewarding action from every state.

When rewarding the agent, we must not reward a subgoal, such as getting a sequence of three pieces in Connect Four, because the agent may prioritize completing these subgoals rather than completing the ultimate goal. The reward is our way of telling the agent *what* to achieve and not *how* to achieve it. There are other aspects of rewarding an agent. Sometimes, a task should be solved with as few actions as possible. This can be achieved by punishing the agent with a negative reward for every action.

Most reinforcement learning methods are based on estimating value functions. These are functions of states that estimate how good that state is for the agent. These functions are defined with respect to some policy and the function approximates the expected return if the agent starts in a given state and follows the policy from that state.

Some algorithms search through the policy space without using a value function. Such algorithms are called evolutionary algorithms and are based on the ideas of biological evolution. This means that evolutionary algorithms do not interact with the environment, which is the opposite of reinforcement learning. Therefore, evolutionary algorithms are not included in reinforcement learning (Sutton and Barto, 1998). Evolutionary algorithms have advantages when dealing with problems where the agent cannot sense the state of the environment. Evolutionary algorithms play according to a set of policies many times. Only the outcome of a play is taken into account when evaluating a given policy. This means that bad moves and moves that were not made are evaluated equally to the good moves. After assessing the current set of policies, a new set of policies is chosen based on a combination of mutation, reproduction, and natural selection. This cycle of policy evaluation and selection continues until a sufficiently good policy is found.

One caveat of reinforcement learning, and machine learning in general, is that there are often many *hyperparameters* to set. Most parameters are learned by the algorithm through training, but hyperparameters are values that we have to set beforehand. The hyperparameters change how the algorithms work, e.g., by controlling how much we adjust our value function estimation after observing a reward. The problem with hyperparameters is that there are only suggestions for how to choose them. This means that finding good values for all hyperparameters becomes a time-consuming process of trial and error. Section 7.1 optimizes the hyperparameters.

1.4 Structure

We present related work on reinforcement learning in chapter 2. We present the concepts that are needed for this thesis independently and self-contained: reinforcement learning in chapter 3, neural networks in chapter 4, and search methods in chapter 5. We combine these concepts and further extensions to create a game-playing agent in chapter 6. We optimize the agent and its extensions in chapter 7.

Chapter 2

Related Work

Connect Four was weakly solved using a knowledge-based approach by Allis (1988). A program, referred to as VICTOR, uses a set of high-level strategic rules and search algorithms to evaluate board positions. The result is that if ⓧ plays optimally, ⓧ can win if the first piece is placed in the middle column. If ⓧ starts in any other column, the game can be forced into a draw by ⓞ. By using a database of approximately half a million board positions, VICTOR was able to play in real-time against opponents on a full 6×7 board. These results required expert knowledge on how to play Connect Four in order to derive the necessary strategic rules. As future work, Allis (1988) suggests to automatically derive strategic rules instead of implementing rules suggested by a human expert.

Numerous attempts have been made at creating game-learning programs based on reinforcement learning. Tesauro (1995)'s TD-Gammon is a neural network trained to be an evaluation function for Backgammon. Given a board position, the network estimates the probability of winning from that position. The neural network was trained using the TD(λ) algorithm with $\lambda = 0$ by self-play. Chapter 3 describes TD(λ) and self-play, and chapter 4 describes neural networks. This result was especially significant because the program was able to achieve a high level of play without the assistance of a knowledgeable teacher. It was noted that the program in many cases evaluated board positions in a different way than human players would. It even turned out that some of the plays made by TD-Gammon were superior to the strategies of the top human players. Part of the program's success was attributed to the stochastic nature of Backgammon. There is a problem of training on only a small number of positions during self-play, but the high influence of the die forces the network to experience a larger state space. Another factor was that the outcome of Backgammon is a real-valued function with continuity such that small changes to board positions are reflected by small changes in the outcome of the game. This factor helps the network during training since small adjustments to weights are apparent in the terminal outcome of the game. In contrast, games like Chess and Connect Four have discrete outcomes, so small changes in play are harder to measure in the final outcome. This means that it will be harder to measure any difference in performance and thus presumably harder to learn.

Go has been the ultimate challenge for machine-learning algorithms for a long time, as the high branching factor defeats the conventional search approaches, which we will describe in chapter 5. A TD-learning approach to Go was made by Schraudolph et al. (1994) where the authors trained the network by self-play, against knowledgeable teachers, and by letting the network observe high-level games. Bootstrapping the learning algorithm by self-play was noted to be sluggish as thousands of games were required to obtain a low level of play. The authors mentioned that letting the network observe recorded games between human players or random move generators helped speed up the learning process of the network in the early stages. Furthermore, it was noted that while playing against a conventional Go program could help the network, it turned out that training too long against the same opponent could cause the network to overfit and thus end up hindering the network's performance. To greatly reduce the complexity, the authors exploited the fact that Go positions retain their properties with respect to the eight-fold symmetry of the game board. This reduction in the number of states significantly reduced the number of training games required to beat conventional Go programs.

In more recent times, the program AlphaGo by Silver et al. (2016) has managed to reach and possibly even exceed professional levels of play in Go. Their approach was based on two neural networks: a policy network and a value network combined with a variation of Monte-Carlo Tree Search (MCTS). Section 5.2 describes MCTS in more detail. The policy network is used to quickly perform game simulations and provide move suggestions fast. The value network is used to compute the probability of winning from a given state. Both networks were trained using a combination of supervised and reinforcement learning. To avoid overfitting against a single policy during self-play training, an opponent was randomly selected from a pool of previous iterations of the policy network. AlphaGo defeated European Go champion Fan Hui 5-0 and later defeated world champion Lee Sedol 4-1.

A comparison between hand-tuned weights and TD-learned weights was made by Schaeffer et al. (2001). The authors compared the performance of the hand-tuned weights of the Checkers program Chinook with weights learned by training the program using the TD-learning algorithm TDLeaf. The weights of the original Chinook program were hand-tuned over 5 years, assigning weights to knowledge-based features of Checkers. Results indicated that self-play learning was sufficient to achieve the same level of play as the hand-tuned version of Chinook. This result was partly attributed to the relatively low number of parameters to fit. The authors also noted that training the program specifically to play White or Black did not have a statistically significant impact on the performance of the program when playing the other color. All results were obtained from machine-versus-machine games, and Schaeffer et al. (2001) comments that situations that occur less frequently during self-play were valued differently to the hand-tuned weights. As humans have a unique way of playing the game, the authors concluded that playing against humans would be necessary to complete the training of the program.

Chapter 3

Reinforcement Learning

3.1 Agent and Environment

In reinforcement learning, the learner and decision maker is called the agent and what it interacts with is called the environment. The agent chooses an action to perform on the environment and the environment returns a reward, which the agent tries to maximize. These interactions happen at discrete time steps $t \in \mathbb{N}$. For a given time step t, the agent is presented with state $s_t \in \mathcal{S}$ from the environment, where $\mathcal{S}$ is the set of states. For a given state s_t, the agent chooses an action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of actions available in state s_t. This leads to a state s_{t+1} and a reward $r_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is the set of rewards. Figure 3.1 shows this interaction between the agent and the environment. These interactions can be *episodic* or *continuous*. Episodic tasks are broken into *episodes* of interactions that start in some initial state and end in a terminal state. Continuous tasks also start in an initial state but can potentially go on forever. We will only investigate episodic tasks.

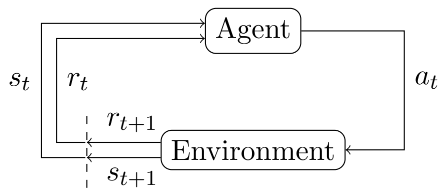


Figure 3.1: Interaction between agent and environment in reinforcement learning (Sutton and Barto, 1998).

A state of the environment consists of any information that the agent needs to make its decisions. The information contained in a state can be low-level, such as the pieces placed in a board game, or it can be high-level, such as whether a complex strategic setup is present.

Another important factor is where the line between agent and environment is drawn. We define the environment as everything that is outside of the agent's control. This means that things that are physically part of the agent could be made a part of the environment if the agent cannot control them. For instance, consider a person who is deciding whether he should get something to eat. This decision involves several factors like: "How hungry am I?", "What is the distance to the nearest food?", and "How expensive is the food?". Even though hunger is something that comes from inside the agent's own body, it would be a part of the environment since the agent is not directly in control of hunger. When the problem at hand is playing board games, the distinction between environment and agent is often easy to make, but it can be an important factor to keep in mind for some problems.

The questions above are examples of *control problems*, where an agent tries to determine what action to make in a given state. Another type of problems that arise in reinforcement learning are *prediction problems*. Framing the examples above as prediction problems, the questions could be: "When will I be hungry?" for time scheduling or "What is the probability of becoming hungry in this time interval?" for likelihood estimation. Section 3.4 describes these types of problems.

3.2 Markov Property

If the current state contains all the information needed to choose an action, it is said to have the *Markov property*. For example, states that capture the position of the pieces on the board in Connect Four have the Markov property since the states capture all that has happened in the past except for the order of moves. But the order is not needed to determine what is possible in the future. The order says something about the opponent's mindset and what his strategy might be, but the agent is not better off knowing this information, since the opponent's possibilities stay the same. This stands in contrast to Poker where the mindset and gestures of the opponents should be taken into consideration, because not all the information about the game is available, which makes bluffing a big part of the game.

Formally, a reinforcement-learning problem has the Markov property if and only if

$$\begin{aligned}&\Pr\left[s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right]\\ =&\Pr\left[s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\right],\end{aligned}$$

for all $s', r, s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0$.

A reinforcement-learning problem that has the Markov property is called a *Markov decision process* (MDP). If the state and action space of an MDP is finite, then it is referred to as a finite MDP. A finite MDP is defined by its state and action sets, its transition probabilities, and the expected rewards. The transition probabilities of a finite MDP are defined as

$$\mathcal{P}^a_{ss'} = \Pr\left[s_{t+1} = s' \mid s_t = s, a_t = a\right],$$

where $\mathcal{P}^a_{ss'}$ is the probability of the next state being s', given that the current state is s and the action a is made. Similarly, the expected rewards are defined as

$$\mathcal{R}^a_{ss'} = E\left[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right],$$

where $\mathcal{R}^a_{ss'}$ is the expected reward when the agent is in state s, performs action a, and transitions to state s'.

For the rest of this thesis, we will assume that the reinforcement-learning problems are finite MDPs because that is the case for most games and, especially, Connect Four.

3.3 Policies and Value Functions

The agent's actions are determined by its *policy*. A policy can either be *deterministic* or *stochastic*. A deterministic policy is a mapping $\pi(s) = a$ from all possible states $s \in \mathcal{S}$ to the action $a \in \mathcal{A}(s)$ that the agent should make in state s. A stochastic policy $\pi(s, a)$ maps all pairs of possible states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}(s)$ to the probability of making action a when in state s. Section 3.5 gives concrete examples of policies.

The expected reward can be re-defined to fit a specific policy. The expected reward from state s given a policy π is

$$E_\pi\left[r_{t+1} \mid s_t = s\right] = \sum_{a \in \mathcal{A}(s)} \pi(s, a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'}.$$

All the reinforcement-learning algorithms that we will consider are based on estimating *value functions*. A value function maps states to the state's expected return, or it maps state-action pairs to the expected return when performing that action from that state. The return from some time step t is defined as

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1},$$

where T is the number of time steps until the end of the episode and γ is the *discount rate*. The discount rate $\gamma \in [0, 1]$ decreases the value of rewards in the future. The point of decreasing the future rewards is to motivate the learning agent to be as fast as possible. If the discount rate is close to 0, the agent will be focused on obtaining immediate rewards without thinking far into the future. If the discount rate is close to 1, the agent will be more farsighted and will forego a reward now if it leads to a higher reward in the long run. The discount rate can thus be used to alter the behavior of the agent.

With the definition of return, we can formally define value functions. Since the value of a state depends on the policy of the agent, its state-value function is defined as the expected return when starting in state s and following policy π

$$V^\pi(s) = E_\pi \left[R_t \mid s_t = s \right] = E_\pi \left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s \right],$$

where t is some time step. Similarly, the action-value function is defined as the expected reward when performing action a in state s and thereafter following policy π

$$\begin{aligned} Q^\pi(s, a) &= E_\pi \left[R_t \mid s_t = s, a_t = a \right] \\ &= E_\pi \left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right], \end{aligned} \tag{3.1}$$

where t is some time step. If the state and action space of our problem is sufficiently small, our value function can be kept as a table that maps states or state-action pairs to their expected return. However, for big state spaces, it is necessary to maintain parameterized functions that compute the expected return for each state or state-action pair.

An important property of value functions is that they satisfy the following *Bellman equation*

$$\begin{aligned}
V^\pi(s) &= E_\pi\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s\right] \\
&= E_\pi\left[r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k r_{t+k+2} \mid s_t = s\right] \\
&= \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma E_\pi\left[\sum_{k=0}^{T} \gamma^k r_{t+k+2} \mid s_{t+1} = s'\right]\right) \\
&= \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma V^\pi(s')\right).
\end{aligned} \tag{3.2}$$

Intuitively, this equation says that the value of state s is the expected reward from each of its possible successor states, multiplied by the probability of transitioning to those states. The value function is the unique solution to its Bellman equation (Sutton and Barto, 1998).

Notice that there is a dependency between value function and policy. This leads to the idea of *generalized policy iteration* (GPI), which has two simultaneous, interacting processes: *policy evaluation* that makes the value function consistent with respect to the current policy and *policy improvement* that makes the policy greedy with respect to the current value function. These two processes of GPI work together on finding a joint solution and the processes can only stabilize if we find a policy that is greedy with respect to its own value function. For now, let us assume that our policy is deterministic. Given a policy π and a value function V^π, policy improvement chooses a new policy π' that is defined as

$$\begin{aligned}
\pi'(s) &= \arg\max_{a \in \mathcal{A}(s)} Q^\pi(s,a) \\
&= \arg\max_{a \in \mathcal{A}(s)} E_\pi\left[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\right] \\
&= \arg\max_{a \in \mathcal{A}(s)} \sum_{s' \in S} \mathcal{P}_{ss'}^a \left(\mathcal{R}_{ss'}^a + \gamma V^\pi(s')\right),
\end{aligned}$$

which selects the action that maximizes the value function in each state. We want to be sure that the new policy π' is an improvement over π, i.e., we want to make sure that $V^\pi(s) \leq V^{\pi'}(s)$ for all $s \in S$. We have chosen $\pi'(s)$

such that for all $s \in S$,

$$V^{\pi}(s) \leq Q^{\pi}(s, \pi'(s)), \tag{3.3}$$

as we are choosing the action that maximizes $V^{\pi}(s)$. Let $s \in S$ be some state. By repeatedly applying equation (3.3) and equation (3.1), we get that

$$\begin{aligned}
V^{\pi}(s) &\leq Q^{\pi}(s, \pi'(s)) \\
&= E_{\pi'}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s\right] \\
&\leq E_{\pi'}\left[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) \mid s_t = s\right] \\
&= E_{\pi'}\left[r_{t+1} + \gamma E_{\pi'}\left[r_{t+2} + \gamma V^{\pi}(s_{t+2})\right] \mid s_t = s\right] \\
&= E_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}(s_{t+2}) \mid s_t = s\right] \\
&\leq E_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 Q^{\pi}(s_{t+2}, \pi'(s_{t+2})) \mid s_t = s\right] \\
&\vdots \\
&= E_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots\right] \\
&= V^{\pi'}(s).
\end{aligned}$$

This shows that $V^{\pi}(s) \leq V^{\pi'}(s)$ for all $s \in S$, i.e., it shows that the new policy π' is actually an improvement.

When GPI stabilizes, we have found the optimal value function V^* and an optimal policy π^*. To see why, let us look at the Bellman equation for the optimal value function, also called the *Bellman optimality equation.* The optimal value function must also satisfy equation (3.2). The intuition behind this equation is that the value of a state following an optimal policy must equal the expected return for the best action from that state. The Bellman optimality equation is given by

$$\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}(s)} Q^{\pi^*}(s, a) \\
&= \max_{a \in \mathcal{A}(s)} E_{\pi^*}\left[R_t \mid s_t = s, a_t = a\right] \\
&= \max_{a \in \mathcal{A}(s)} E_{\pi^*}\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right] \\
&= \max_{a \in \mathcal{A}(s)} E_{\pi^*}\left[r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a\right] \\
&= \max_{a \in \mathcal{A}(s)} E_{\pi^*}\left[r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right] \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s' \in S} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^*(s_{t+1})\right).
\end{aligned}$$

The optimal value function is expressed in terms of itself without referring to a specific policy. If both GPI processes have stabilized, it must be because the optimal value function has been found. The policy found by policy improvement must be an optimal policy as it is greedy with regard to the optimal value function, but there could be many optimal policies because the best action for a state might not be unique.

While the above sounds motivating, it is not realistic to find the optimal value function and an optimal policy in practice. However, the methods we present in the following chapters are based on this idea of GPI and uses estimations to do policy evaluation.

3.4 Prediction and Control Problems

Prediction problems are about trying to estimate a state-value function $V^\pi(s)$ where s is a given state. A prediction algorithm's objective is to predict the outcome of a given state. Control problems are about trying to estimate an action-value function $Q^\pi(s,a)$ where s is a given state and a is a given action. A control algorithm's objective is to control the environment by finding the best actions for some given states. In both cases, if we have a good value function, it can be used to solve the task. For prediction problems, we can predict the outcome of state s by applying the state-value function to s. For control problems, we can find the best action for a state s by looking ahead at each possible action a and applying the action-value function to each state-action pair.

We want to develop methods for policy evaluation, i.e., we want to make the value function consistent with regard to the current policy. Let us assume for simplicity that we are in the tabular case, meaning that there are few enough states that we can keep our value function as a table with one entry for each state. We are going to improve our value function by interacting with the environment and by adjusting our estimates based on these interactions. A natural way of estimating the value of each state is to go through episodes of interaction and adjust the value of each state to be closer to the return seen after visiting that state. This is known as a *Monte-Carlo method*. That is, we want to make the following update to the value of each state s_t visited after finishing an episode

$$V^\pi(s_t) := V^\pi(s_t) + \alpha\left(R_t - V^\pi(s_t)\right).$$

This update is read as: The estimate is updated from the prior estimate towards a *target*, which is the newly seen return, with a *learning rate*

$\alpha \in [0, 1]$. We are only interested in α values inside this bound because an α value smaller than 0 or greater than 1 will update the estimate away from the target. An α value of 1 means that the algorithm will ignore the old estimate and set it to the target. An α value of 0 means that the algorithm will ignore the target and keep the old estimate.

The first intuition might be to set α to 1 because we want to get the best estimate as fast as possible. But we cannot fully rely on the return if the input is noisy. It is not wrong to depend on a noisy input because it does give an insight into the valuation of a state, but we need a combination of estimates from the same state to get the best estimate. In a game, the input will be noisy if the agent or the opponent does not play optimally. Therefore, α should be set in the interval $[0, 1]$ and decrease as the agent gets a good estimate of the states.

A problem with the Monte-Carlo method is that we cannot update the value of an estimate before the end of an episode, as we do not know the return before this. Temporal-difference methods (TD methods) provide an alternative that addresses this issue. The simplest TD method is TD(0), which makes updates after every time step according to the update rule

$$V^{\pi}(s_t) := V^{\pi}(s_t) + \alpha \left(r_{t+1} + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t) \right).$$

That is, the target for TD(0) is $r_{t+1} + \gamma V^{\pi}(s_{t+1})$, i.e., the observed reward after making an action and the estimated value of the resulting state. This means that we can make an update to our value function after every observed reward instead of after each episode. Notice that the value function applied to the next state is part of the target for TD(0). In general, TD methods are based on the idea of *bootstrapping*, i.e., using the current value function to form a target for the next update. According to Sutton and Barto (1998), TD methods will converge to an optimal policy if the learning rate decreases properly.

An example that might shed some light on the difference between Monte-Carlo methods and TD methods is trying to estimate the time to get home (Sutton and Barto, 1998). Initially, you estimate that it takes 30 *min* to get home from the office but, when you get to your car, 5 *min* have already passed. The estimate of the total time is changed to 35 *min*. When you get to the highway, the weather is good and there is little traffic. This decreases the estimate to 25 *min*. When you get off the highway, you get stuck behind a tractor with no possibility of by-passing it. This increases the estimate to 32 *min*.

Changing your estimate along the way without seeing the final result is exactly the benefit of using TD methods. After traveling part of the way, we can adjust our estimate by combining what we have seen with our current estimate of how long the rest of the journey will take. This contrasts with Monte-Carlo methods, where we cannot make adjustments until the journey is complete and we have seen the total travel time. The two methods also differ in how the adjustments are made. For TD methods, the estimate for each part of the road is adjusted toward the observed travel time from that point to the next and the estimated time for the rest of the journey. For Monte-Carlo methods, the estimate of every part of the road is adjusted toward the observed travel time from that point until the end of the journey. This also means that a delay at the end of the journey is not reflected in the travel time of the parts before the delay when using TD-learning, as updates have already been made. Section 3.6 introduces eligibility traces, which gives a middle ground between Monte-Carlo methods and TD-learning.

For many real-world problems, the state space is too big to keep our value function as a table. For these problems, the value function can be approximated by a parameterized function where updates are made by changing the values of the function's parameters. Section 3.7 describes the TD(λ) algorithm, a generalization of TD(0) with eligibility traces and without the assumption of a small state space.

3.5 Exploration versus Exploitation

A recurring problem in reinforcement learning is the trade-off between *exploration* and *exploitation*. The agent does not know the reward for making an action in a state, and the agent cannot explore the entire state space if it gets too big. At some point, the agent has to exploit the knowledge that it has obtained from exploring, even though the agent does not have a complete picture of the valuation of the states. For instance, consider a person who wants to earn as much money as possible by playing slot machines but he can only play a limited number of times. There are many slot machines, each with a random reward from a probability distribution specific to that machine. The problem is that he does not know which machine provides the biggest payout. This means that he has to try the machines and figure out which machine is the best, but it is necessary to try each machine many times since the reward is random. He will never get to *exploit* the knowledge that he obtains if he wastes all of his limited tries *exploring*. This is precisely the issue of balancing exploration and exploitation. We will investigate two policies that handle this problem: ε-greedy and softmax.

An ε-greedy policy chooses a random action with a probability of ε, otherwise the policy chooses an action greedily based on a value function, i.e., it will choose the highest-valued action in the given state. When the policy makes a random selection, all actions are chosen uniformly at random regardless of their estimated value. This is beneficial when our value function is randomly initialized as it is not accurate at this point. However, once the agent gets a better estimate for each action, it is preferable to explore only the actions that look good.

A softmax policy prioritizes high-valued states by choosing every action randomly but with a probability proportional to the action's estimated valuation so that actions with high estimates are more likely to be chosen. Formally, we define

$$p(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a \in \mathcal{A}(s)} e^{Q(s,a)/\tau}},$$

where $p(s,a)$ is the probability of choosing action a in state s and τ is the *temperature*. The temperature determines how much the value of an action affects the probability of choosing the action. As $\tau \to \infty$, softmax tends towards a uniform random choice. As $\tau \to 0^+$, softmax tends towards the greedy choice.

We call ε and τ the *exploration rates*. We want a high-valued exploration rate to explore the unknown state space in the beginning, but we also want to exploit our knowledge at some point. This can be done by decreasing the exploration rate over time by some decay factor. The literature gives no optimal initial value or decay factor for the exploration rate. Section 7.2 compares ε-greedy and softmax policies and looks for a good initial value. Section 7.4 looks for a good decay factor.

3.6 Eligibility Traces

As mentioned in section 3.4, Monte-Carlo methods run through an entire episode to propagate the evaluation backwards to the visited states. The problem is that the states are not updated while playing, so it will take a long time for the agent to learn if the episodes are long. This is different from temporal-difference learning (TD learning) where the given states are updated when visited. However, this raises a new problem. The fact that the previous states might result in a bad state is not reflected on them. Therefore, eligibility traces are introduced. The intuition behind eligibility traces is that they keep track of how influential previous states have been.

These traces give a middle ground between Monte-Carlo methods and TD learning by remembering the states that were visited during the episode. For each visited state, the update for that state is propagated backwards with some decay factor λ. The update is decayed because the states that were visited a long time ago are presumably not as influential on the current result as the recently visited states. Section 3.4 describes the updates made by TD(0) algorithm. TD(0) is a special case of the more general TD(λ) algorithm. Here, we describe TD(λ) and eligibility traces in the tabular case. Section 3.7 describes how to extend the algorithm to larger state spaces by using a parameterized function. When encountering a new state, TD(λ) makes updates to its value function for all states s according to

$$V^{\pi}(s) = V^{\pi}(s) + \alpha\left(r_{t+1} + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right) e_t(s), \tag{3.4}$$

where $e_t(s)$ is the eligibility trace for state s and is initialized to be zero.

There are two common ways of defining eligibility traces. One way is *additive traces*, which are defined as

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) + 1, & \text{if } s = s_t \\ \gamma\lambda e_{t-1}(s), & \text{otherwise.} \end{cases}$$

That is, the trace of a state is always decayed by a factor $\gamma\lambda$, but it is increased by 1 if it is visited in that time step.

A problem with additive eligibility traces is that cycles can increase the eligibility trace of the cycled states arbitrarily, resulting in those states being weighted too high. This problem can be solved by setting the trace of a state to 1 after each visit instead of adding 1. These alternative type of traces are called *replacing traces*. Replacing traces are formally defined as

$$e_t(s) = \begin{cases} 1, & \text{if } s = s_t \\ \gamma\lambda e_{t-1}(s), & \text{otherwise.} \end{cases}$$

For non-cyclic problems, these two approaches are equivalent.

The value of λ controls how fast the eligibility trace of a state decays after visiting it. Choosing $\lambda = 1$ makes TD(λ) learning equivalent to Monte-Carlo methods, except that learning still takes place after each time step as opposed to after an episode finishes. This is because each reward is only decayed by γ when going back one step, as is the case for the Monte-Carlo method that we saw earlier. Choosing $\lambda = 0$ gives the TD(0) algorithm that does not update previous states after visiting them. Choosing $0 < \lambda < 1$ gives a middle ground between Monte-Carlo methods and TD learning.

Sutton and Barto (1998) note that the traces of states that were visited a long time ago will become near-zero. Therefore, it will not make sense to follow the trace all the way back to the beginning since the impact will be too small compared to the time that it will take. However, this is only a problem for long episodes. Since episodes of Connect Four are at most 42 steps long, short-circuiting the computation will not make much of a difference. But this might be a problem for other games such as Backgammon and Chess. For such games, we might have to set traces to 0 when they drop below some threshold.

3.7 TD(λ)

In the previous sections, we have presented TD methods and eligibility traces in the tabular case. In this section, we are going to extend those ideas to bigger state spaces where the value function is approximated by a parameterized function. TD(λ) is a reinforcement-learning prediction algorithm, e.g., TD(λ) can be used to predict the likelihood of winning from a given state in a game. The idea behind the algorithm is to start in some initial state and repeatedly make actions according to a policy derived from the current value function. After making an action, the resulting reward and next state is used to update the value function. This process is repeated until a terminal state is reached. The algorithm can be seen as an extreme form of GPI: Making an update to the value function after seeing a reward corresponds to policy evaluation since we are making the value function more consistent with regard to our policy. The next action is chosen by a policy according to the value function, thus an iteration of policy improvement is made implicitly.

Since the value function is approximated by a parameterized function, TD(λ) makes updates to these parameters in order to adjust the value function. If we let θ be the vector whose components are the parameters of our value function, the update made by TD(λ) to these parameters is

$$\theta_{t+1} = \theta_t + \alpha(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t))e_t, \tag{3.5}$$

where e_t is a vector of eligibility traces for each parameter of the value function. For each step, e_t is updated according to

$$e_t = \gamma\lambda e_{t-1} + \nabla_\theta V_t(s_t),$$

where $\nabla_\theta V_t(s_t)$ is the gradient of $V_t(s_t)$, i.e., the vector of partial derivatives of the value function with regard to its parameters. Notice that the eligibility traces are decayed by a factor of $\gamma\lambda$ after each update as before.

```
function TD-lambda($\alpha$, $\gamma$, $\lambda$)
  initialize $\theta$ arbitrarily
  for each episode
    initialize $s$
    $e := 0$
    while $s$ is non-terminal
      choose $a$ from $s$ using policy derived from $V$
      take action $a$, observe reward $r$ and next state $s'$
      $e := \gamma\lambda e + \nabla_\theta V(s)$
      $\theta := \theta + \alpha(r + \gamma V(s') - V(s))e$
      $s := s'$
```

Figure 3.2: Pseudocode for TD(λ) (Sutton and Barto, 1998).

The update made by equation (3.5) closely resembles the update made to the value function in the tabular case made by equation (3.4). The big differences are that the eligibility traces are now over function parameters rather than states and updates are made to the parameters rather than table entries. However, the update made to the value function is essentially the same. For each parameter, its eligibility trace tells us how a small change to that parameter will affect the output of the function. Intuitively, equation (3.5) says that the parameters of the value function are changed such that its output for state s_t is moved towards the target $r_{t+1} + \gamma V_t(s_{t+1})$. This is the same target used for updates in both TD(0) and TD(λ) in the tabular case. The values of previous states are also adjusted toward the new target if $\lambda > 0$. The downside of having to use a parameterized function is that updates made to one state may affect the value of other states.

Figure 3.2 shows the complete pseudocode for TD(λ). The only requirement for the value function is that we can find its partial derivatives. Chapter 4 describes neural networks, which can be used as value functions since they have an efficient algorithm for finding partial derivatives. Section 6.3 describes how neural networks are used with TD(λ) to create a game-playing agent.

3.8 Model

Some reinforcement-learning agents use a model of the environment to look ahead and improve their decision making. A model is anything that can be

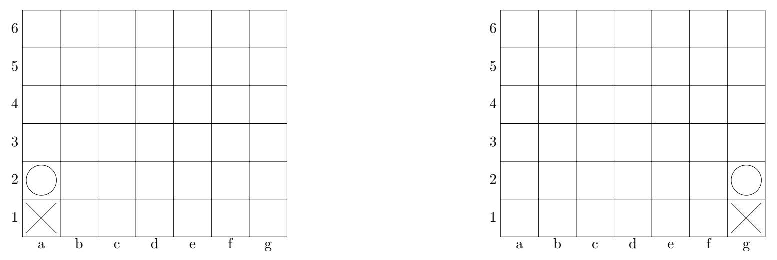


Figure 3.3: Example of reflection symmetry in Connect Four.

used to predict how the environment will respond to actions. For games, a model is often used by a *search method* that looks through many different possible actions and gets a statistical insight into the quality of the possible actions. For deterministic games, it is possible to get a perfect model that can accurately predict how an action will affect the environment. This allows the agent to look ahead and simulate many possible plays using the model before making a decision. However, the state space may be too large for a search method to give a good insight into the state space by itself, so a combination with another heuristic may be necessary, this heuristic could be the reinforcement-learning agent. Thus, the search method can use the reinforcement-learning agent to guide its search and, in turn, the agent can improve its approximation of a given state-value or state-action pair using the search method. Chapter 5 describes search methods further and section 6.7 describes how they are incorporated into reinforcement-learning agents.

A model represents the environment, and sometimes the environment might have setups that are the same up to some transformation. Rotational symmetry and reflection symmetry are often applicable in games. Tic-tac-toe is an example of rotational symmetry, since it does not make a difference which of the four corners the first player marks. Connect Four is an example of reflection symmetry, since it does not make a difference whether the first player places a piece in the outer-most field to the left or to the right. Figure 3.3 shows an example of reflection symmetry in Connect Four. Exploiting symmetry reduces the size of the model with a factor depending on the order of symmetry. In the two examples, the factor will be four and two respectively. Reducing the size of the model will reduce the space consumption and therefore the time consumption. The time consumption will also decrease because we have a smaller state space that we have to explore.

3.9 Self-Play in Games

When we say that we want to *train* a game-playing agent, we want to generate episodes that the agent can learn from using an algorithm such as TD(λ). There are many ways to train. We can train the agent by letting it play against humans but that will take a long time per game. We can train the agent by letting it play against other computer programs but that requires that such programs are available. We could make a random opponent, or an opponent that uses a search method such as minimax or Monte-Carlo Tree Search, which we will return to in chapter 5. However, these opponents do not work well for complex games, which we will show in section 7.8, and they do not adapt, which could lead to overfitting.

We want to use an opponent that is fast, scales well to complex games, and changes over time to avoid overfitting. But that is exactly the benefits of our reinforcement-learning agent. So, we will let it play against itself, which we will refer to as *self-play*. However, this introduces new problems because the process will be sluggish in the beginning according to Schraudolph et al. (1994) since the agent "must bootstrap itself out of ignorance without the benefit of exposure to skilled opponents." To address this issue, we could use supervised learning on sampled training data, or we could let the agent train against the random player or one of the search-method players in the beginning of the training session, but that is out of the scope of this thesis. Besides, according to Schaeffer et al. (2001), there is no evidence that playing against a well-trained opponent is required to reach a competitive level.

If the purpose of an agent is to play against human players, the agent should also train against humans because humans will make moves that an agent might not consider. Therefore, the agent might not encounter these moves during training that will be common in play against a human (Schaeffer et al., 2001; Tesauro, 1995).

Chapter 4

Neural Networks

The brain consists of many neurons that are connected. Neurons communicate by sending electrical impulses to other neurons, which will in turn send the charge to other neurons depending on the charge of the impulse. A neural network is inspired by the brain and uses a directed network of neurons to approximate functions. In a neural network, a neuron receives input and computes an *activation* by applying an *activation function* to a weighting of its inputs. The neuron then sends this activation to all the neurons that it is connected to within the network. How the activation is computed depends on the type of neuron.

An infinite number of different types of neurons exist, but we will only look at *perceptrons* and *sigmoid neurons*. We focus on the perceptron first because it is useful for explaining the basic concepts of neurons. But the perceptron has problems that the sigmoid neuron solves. The sigmoid neuron also has the properties required for explaining the theory behind gradient descent and backpropagation. Section 4.4.1 describes gradient descent, and section 4.4.3 describes backpropagation.

4.1 Perceptrons

A perceptron weighs a number of real-valued inputs and gives a binary output. The activation function of a perceptron checks if the weighted input is less than or equal to some threshold t and outputs 0 if that is the case, otherwise it outputs 1. Formally, the activation function of the perceptron is defined as

$$f(x_1, \dots, x_n) = \begin{cases} 0, & \text{if } \sum_{i=1}^{n} w_i x_i \leq t \\ 1, & \text{otherwise,} \end{cases} \tag{4.1}$$

where i is the index of all neurons connected to this neuron, $w_i \in \mathbb{R}$ is the weight of input i, $x_i \in \mathbb{R}$ is the value of input i, $\sum_i w_i x_i$ is what we refer to as the weighted input, and $t \in \mathbb{R}$ determines when this neuron will activate, i.e., output 0 or 1.

The weighted input in the activation function of the perceptron can be written as $w \cdot x$ where w is a vector of the weights for the inputs leading into the perceptron and x is a vector of those inputs. Another common rewriting of the activation function is to use a *bias* instead of the current threshold. This is done by choosing bias $b = -t$ and rewriting equation (4.1) into

$$f(x) = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{otherwise.} \end{cases}$$

It might not be immediately obvious why introducing a bias is better than having a threshold, but when dealing with other types of neurons the bias allows for simpler notation. Intuitively, the bias can be thought of as how easy it is to get a neuron to activate. The neuron will activate with smaller inputs if the bias is high, and the neuron will require higher inputs to activate if the bias is low.

Figure 4.1 depicts a perceptron with two inputs, x_1 and x_2. To better understand how such a perceptron computes a function, let us try to assign weights to the inputs in order to make the perceptron compute the logical *NAND* function. To that end, we will set $w_1 = w_2 = -2$ and $b = 3$. We check that the perceptron correctly computes the *NAND* function of its inputs. If both inputs are 1, then

$$w_1 x_1 + w_2 x_2 + b = (-2) \cdot 1 + (-2) \cdot 1 + 3 = -1 \leq 0.$$

So, the perceptron will output 0. If any inputs are different from 1, then $w_1 x_1 + w_2 x_2 \geq -2$, which means that $w_1 x_1 + w_2 x_2 + b > 0$ and the perceptron outputs 1 as required.

Since a perceptron can compute the *NAND* function, we can conclude that a network of perceptrons has the ability to compute any logical function. In fact, a neural network can approximate any continuous function to arbitrary accuracy (Hornik et al., 1989).

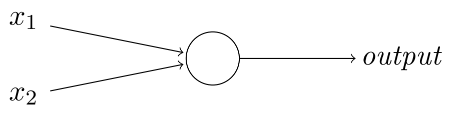


Figure 4.1: A perceptron with two inputs.

In the above example, we manually selected the weights and bias such that the perceptron computed exactly what we wanted. The point of a neural network is to automatically learn values for the weights and biases of the neurons in the network such that the network learns how to compute some desired function. In a supervised setting, this is done using *training examples* that consist of input and the desired output. To learn values for the weights and biases, we want to use a learning algorithm that makes small changes to the network's weights and biases in order to nudge the values towards computing the output that we want. In other words, given a training example, we change the weights and biases of the network such that its output is closer to the desired output for that input. Section 4.4 explains the process of training a neural network in the supervised setting. A problem with the perceptron is that a small change to its weights or bias either does not affect the output of the perceptron at all or flips it completely. This will make it difficult to figure out what changes to make in order to improve the network, as a small change that fixes the output for one training example might flip it for another training example. The next section introduces the sigmoid neuron, which addresses this problem.

4.2 Sigmoid Neurons

In the previous section, it was stated that figure 4.1 showed a perceptron with two inputs. In fact, the figure could be seen as any type of neuron, and it applies equally well to a sigmoid neuron. The sigmoid neuron has two major differences compared to the perceptron: it produces output between 0 and 1 instead of binary values and it uses a different activation function. The activation function of a sigmoid neuron is the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

When writing the activation of a neuron with weights w, input x, and bias b, we will use the notation

$$a = \sigma(w \cdot x + b),$$

where a is the activation. We also introduce the weighted input $z = w \cdot x + b$. The weighted input turns out to be important when doing backpropagation, which we will describe in section 4.4.3. The activation can now be written as

$$a = \sigma(z).$$

The sigmoid function behaves similarly to the activation function of a perceptron for large inputs. At the limits, $e^{-z} \to 0$ as $z \to \infty$ and therefore $\frac{1}{1+e^{-z}} \to 1$, so the sigmoid neuron approximately outputs 1. Similarly, $e^{-z} \to \infty$ as $z \to -\infty$ and therefore $\frac{1}{1+e^{-z}} \to 0$, so the sigmoid neuron approximately outputs 0. This tells us that a sigmoid neuron outputs roughly the same thing as a perceptron at its limits. However, when inputs are smaller, the sigmoid neuron produces outputs between 0 and 1.

To understand how the function behaves for smaller input values, it helps to look at the shape of the function. Figure 4.2 shows the sigmoid function for input values between -5 and 5. The smoothness of the function means that small changes to the weights and bias of a sigmoid neuron will result in a small change in its output. In fact, if we make small changes Δw_i to the weights and Δb to the bias of a neuron, the change in its output is

$$\Delta a = \Delta\sigma(w \cdot x + b) \approx \sum_i \frac{\partial \sigma(w \cdot x + b)}{\partial w_i} \Delta w_i + \frac{\partial \sigma(w \cdot x + b)}{\partial b} \Delta b. \tag{4.2}$$

This is a key component of gradient descent, which we explain in section 4.4.1.

We have described perceptron and sigmoid neurons in isolation. The remainder of this chapter explains how these neurons are connected to form a neural network and subsequently how this network can be trained.

4.3 Networks of Neurons

A neural network consist of three different types of layers: input, hidden, and output. Each layer consists of a set of neurons connected to other neurons.

We will discuss how to represent a neural network with regard to the following examples: recognizing numbers from 0 to 9 based on the MNIST data set (LeCun et al., 1998) and the book of Nielsen (2015), and playing Connect Four. The MNIST data set is a collection of images of hand-written numbers and corresponding labels. Each image consists of $28 \times 28 = 784$ gray levels, one for each pixel. For each image, the corresponding label tells us which number the image represents.

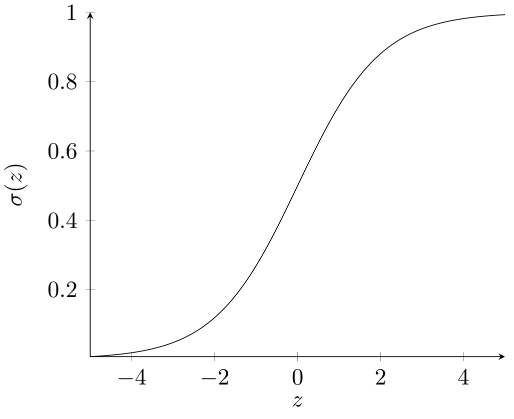


Figure 4.2: Plot of the sigmoid function for input values between -5 and 5.

There is one input layer that accepts some representation of a given state. For recognizing numbers, this layer may consist of 784 neurons to represent the 28×28 pixels of each picture in the MNIST data set. For playing Connect Four, this layer may consist of 42 neurons to represent the 6×7 squares of the board.

The neurons in the input layer are a different type of neuron that we have not addressed yet. These *input neurons* directly output whatever they are given as input, in other words, they have no bias and their activation function is the identity function.

There is one output layer. The activation of the neurons in the output layer represents an evaluation of a given state. For recognizing numbers, this layer may consist of 10 neurons to represent the estimated probability that the given picture shows one of the 10 different numbers. For playing Connect Four, this layer may consist of 1 neuron to represent the probability of winning in the given state of the game.

When designing a neural network, the number of neurons in the input and output layers are determined mostly by the problem. The number of inputs depends on the available information and the number of outputs depends on what we want to compute. However, it is not always trivial to find the best number of input and output neurons. For recognizing numbers, it seems plausible that having an input neuron for each pixel and an output neuron for each possible number is a good approach, but there is no a priori reason why this is the case. The output layer could have consisted of $\lceil \log_2(10) \rceil = 4$ neurons but this encoding performs worse in practice

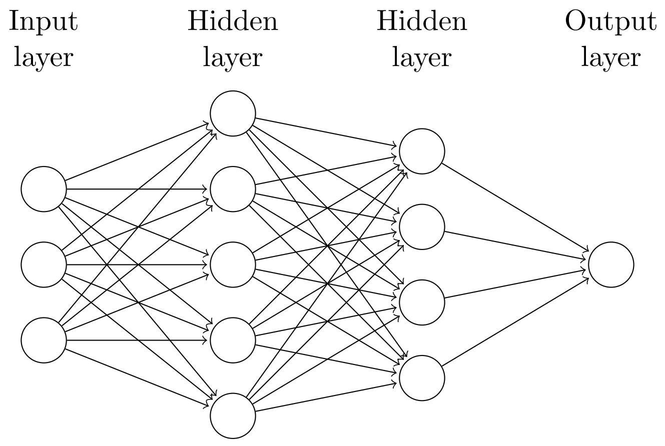


Figure 4.3: Neural network with two hidden layers.

(Nielsen, 2015). Unfortunately, there is no mechanical way of determining the best number of neurons to use.

The hidden layers are a little different from the input and output layers. There may be an arbitrary amount of hidden layers and each layer can have any number of hidden neurons. Increasing the number of hidden neurons and number of hidden layers increases the potential complexity of the network. This allows the network to compute more complicated functions or more accurately estimate a simpler function, but it also increases the number of weights and biases to train. Thus there is a trade-off between having a network that can be trained efficiently while being complex enough for the problem one is solving. A neural network with one hidden layer is called *shallow* and a network with more layers is called *deep.* Figure 4.3 shows a neural network with two hidden layers.

There are different kinds of neural networks, e.g., *feed-forward*, *convolutional*, and *recurrent.* For a feed-forward network, the neurons in a layer l are fully connected to the neurons in the next layer $l+1$ in the network. For a convolutional network, the neurons in a layer l are partially connected to the neurons in the next layer $l+1$ in the network. For a recurrent network, the neurons may be connected to any other neuron, even creating loops. Note that figure 4.3 is a feed-forward network because all of its layers are fully connected to the next layer in the network. A feed-forward network is easier to setup because we only have to decide the number of layers and the number of neurons per layer. This is in contrast to both convolutional and recurrent networks where connections are not fixed to a particular pattern.

Recurrent networks are more similar to how the brain works but the networks are less used than the other two variants (Nielsen, 2015). In addition, recurrent networks require more complex training algorithms. Recurrent networks are especially useful when dealing with problems that require data segmentation such as handwriting or speech recognition (Graves et al., 2009). Because feed-forward networks are fully connected, they may be slower to train and evaluate than sparsely connected convolutional networks. However, making a sparsely connected convolutional network requires insight into which of the connections in the network that are useful. This additional insight can be used to encode features of the input directly in the network by only connecting selected neurons. For the rest of this thesis, we will use feed-forward networks with sigmoid neurons since they are simpler to set up than convolutional networks and are the most commonly used variant.

A neural network transforms an input vector into an output vector by a process called *feed-forward*. This process works by sending the input vector as input to the input layer. For each neuron, the result of its activation is then sent to every neuron in the next layer. The activation function of the neurons in the next layer are then computed for their respective weighted input. Feed-forward is repeated for the remaining layers until the output layer is reached, where the result of the activation functions of the neurons in that layer is the output of the network.

So, what is a neural network actually computing through this process? To understand this, we first look at what a single sigmoid neuron in a network is computing. Say we are looking at the j'th neuron in layer l. The neuron's activation can be written as

$$a_j^l = \sigma(z_j^l) = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right),$$

where w_{jk}^l is the weight from neuron k in layer $l-1$ to neuron j in layer l, a_k^{l-1} is the activation of neuron k in layer $l-1$, b_j^l is the bias of neuron j in layer l, the sum is over all neurons k in layer $l-1$, and z_j^l is the weighted input of neuron j in layer l.

To avoid an index nightmare, we introduce a matrix-based notation. Let w^l be the weight matrix of layer l, i.e., the j'th row and k'th column is w_{jk}^l. Let b^l be the bias vector of layer l, and let a^l be the activations for layer l. We can then write the activation of layer l as

$$a^l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l), \tag{4.3}$$

where σ applied to a vector means that σ is applied to each element of the vector, i.e., $\sigma(z^l)_j = \sigma(z^l_j)$. This gives a simple way of thinking about the activation of a layer: multiply the weights by the activation of the previous layer, add the bias, and apply the activation function. This way of writing the activation also makes it simpler to reason about the network in its entirety. If we let L be the output layer of a network with a single hidden layer, the activation of the output layer can be written as

$$a^L = \sigma(w^L a^{L-1} + b^L).$$

By applying equation (4.3), we get

$$a^L = \sigma(w^L a^{L-1} + b^L) = \sigma\left(w^L \sigma(w^{L-1} a^{L-2} + b^{L-1}) + b^L\right). \tag{4.4}$$

As the network only has one hidden layer, $L-2$ is the input layer and a^{L-2} is the input to the network. If the network had more hidden layers, we would expand the activation further by using equation (4.3). Equation (4.4) shows that the output of a neural network is just a function of the weights and biases of the network.

4.4 Training a Neural Network

In this section, we want to figure out how to make adjustments to the weights and biases of equation (4.4) in order to make the network compute a desired function. From equation (4.2), the change in the network's output is a well-approximated linear function of these adjustments as long as these adjustments are small. Because of this, we can look at a particular input where the network is not computing the right output, and then make a small adjustment such that the network's output becomes closer to what we want. This paradigm of looking at *training examples* and adjusting the network towards a correct output is what is called *supervised learning.*

We denote the input by x and the correct output by $y(x)$. The goal of training is to make the network output $y(x)$ for all training examples x. In order to evaluate a network for a set of training examples, we introduce a *cost function.* An example of a cost function is the *quadratic cost*:

$$C = \frac{1}{2n} \sum_x ||y(x) - a^L(x, w, b)||^2,$$

where $a^L(x, w, b)$ is the output of the network on input x, w is the weights of the network, b is the biases, and n is the number of training examples.

Up until now we have left the input, weights, and biases as implicit input to both the activations and weighted inputs of the network. We will continue to do so when there is no confusion about which training example, weights, and biases we are talking about.

The cost function is a function of the output activation of the network. It is small when the network outputs $a^L(x) \approx y(x)$ for all x. On the other hand, the cost function is large if $a^L(x)$ is not close to $y(x)$ for a large number of training examples x. Therefore, if we can find weights and biases such that the cost function is as small as possible, the network is computing $a^L(x)$ as close to $y(x)$ as possible for all examples x.

4.4.1 Gradient Descent

One technique for minimizing a function is *gradient descent*. Let us say that we are trying to minimize the cost function of a neural network by changing its weights and biases. The idea of gradient descent is to figure out how the cost function behaves when we make a small change to any of the weights or biases. Once we know how the cost function behaves, we make a small change to each weight and bias such that the cost function decreases. We keep making these small changes for some set amount of time or until the cost is sufficiently small. Let $w = w_1, \ldots, w_r$ be the weights of a neural network and $b = b_1, \ldots, b_t$ be the biases. As long as we are only making small changes to the weights and biases of a network, the change in the cost function is

$$\Delta C \approx \sum_{i=1}^{r} \frac{\partial C}{\partial w_i} \Delta w_i + \sum_{j=1}^{t} \frac{\partial C}{\partial b_j} \Delta b_j. \tag{4.5}$$

We want to rewrite this in terms of the *gradient* of the cost function. The gradient is defined as

$$\nabla C = \left(\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_r}, \frac{\partial C}{\partial b_1}, \ldots, \frac{\partial C}{\partial b_t} \right)^T. \tag{4.6}$$

Furthermore, we write the vector of changes in weights and biases as

$$\Delta(w, b) = (\Delta w_1, \ldots, \Delta w_r, \Delta b_1, \ldots, \Delta b_t)^T.$$

We can then rewrite equation (4.5) into

$$\Delta C \approx \nabla C \cdot \Delta(w, b). \tag{4.7}$$

Now, we choose the small positive hyperparameter μ such that

$$\Delta(w, b) = -\mu \nabla C \tag{4.8}$$

is small enough for equation (4.7) to be a good approximation. The update rules for making changes to a single weight or bias in the network are

$$w_i := w_i - \mu \frac{\partial C}{\partial w_i}$$

$$b_i := b_i - \mu \frac{\partial C}{\partial b_i}.$$

Applying these update rules for all weights and biases is equivalent to the update defined by equation (4.8). We can write the change in the cost function as

$$\Delta C \approx \nabla C \cdot \Delta(w, b) = \nabla C(-\mu \nabla C) = -\mu ||\nabla C||^2. \tag{4.9}$$

This tells us that C will decrease if we make changes to the weights and biases of the network in accordance with equation (4.8). Note that this only holds when equation (4.7) is a good approximation, but this is the case if μ is chosen small enough. One concern is that if μ is chosen too small, the changes made to weights and biases will be too small for gradient descent to make noticeable progress.

4.4.2 Stochastic Gradient Descent

One issue with gradient descent is that the quadratic cost function is an average over the cost of each training example. We can write the cost as

$$C = \frac{1}{2n} \sum_x ||y(x) - a^L(x)||^2 = \frac{1}{n} \sum_x C_x,$$

where $C_x = \frac{1}{2}||y(x) - a^L(x)||^2$ is the cost of training example x. Computing the gradient ∇C would require us to compute the gradient ∇C_x for all training examples x since $\nabla C = \frac{1}{n} \sum_x \nabla C_x$. In practice, this takes a long time if the number of training examples is large. Instead, the technique known as *stochastic gradient descent* is used. Stochastic gradient descent works by picking a *mini-batch* of m training examples $X_1, \ldots, X_m$. If m is large enough, we expect that

$$\frac{1}{m} \sum_{i=1}^{m} \nabla C_{X_i} \approx \frac{1}{n} \sum_x \nabla C_x = \nabla C.$$

This gives us new update rules for the weights and biases of our network

$$w_i := w_i - \frac{\mu}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial w_i} \tag{4.10}$$

$$b_i := b_i - \frac{\mu}{m}\sum_{j=1}^{m}\frac{\partial C_{X_j}}{\partial b_i}. \tag{4.11}$$

After making these updates, pick a new mini-batch of m training examples and repeat the process until all training examples have been used. This is referred to as an *epoch* of training. Like μ, it is important to choose a good value for m, i.e., it must be large enough to ensure good approximations and small enough so that the batches can be processed quickly.

In order to train using stochastic gradient descent, what remains is to find an efficient way of computing the partial derivatives from equation (4.10) and equation (4.11). This is exactly what the *backpropagation* algorithm does.

4.4.3 Backpropagation

In the sections above, we only worked with the quadratic cost function, but we can use any cost function for backpropagation that satisfies the following two assumptions. First, it must be possible to write the cost as an average of individual training examples, i.e.,

$$C = \frac{1}{n}\sum_{x}^{n} C_x,$$

We already saw that this assumption holds for the quadratic cost. This is important because the backpropagation algorithm computes the partial derivatives for individual training examples. We can recover the partial derivative for all training examples by averaging over the sum of the derivative for each training example. Second, we must be able to write the cost as a differentiable function of the output activations of the network, i.e., the components of a^L. This assumption holds for the quadratic cost, where the cost of a single training example x is

$$C_x = \frac{1}{2}||y(x) - a^L(x)||^2 = \frac{1}{2}\sum_j (y(x)_j - a^L(x)_j)^2.$$

The sum is over the neurons in the output layer. This assumption is necessary because we compute $\frac{\partial C_x}{\partial a^L(x)_j}$ as part of the backpropagation algorithm.

We will now describe how the backpropagation algorithm works. Backpropagation computes the partial derivatives of the cost function with regard to a fixed training example x. Backpropagation computes these partial derivatives by computing the errors δ^l_j for every neuron j in every layer l. The error is defined as

$$\delta^l_j = \frac{\partial C_x}{\partial z^l_j}. \tag{4.12}$$

The error tells us how the cost function changes when making small adjustments to the weighted input of neuron j in layer l. The error is then related to the gradient from equation (4.6). The name backpropagation comes from the fact that the error is computed at the output layer and then propagated back to the previous layers. The error at the output layer, L, is given by

$$\delta^L_j = \frac{\partial C_x}{\partial z^L_j} = \frac{\partial C_x}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial C_x}{\partial a^L_j}\frac{\partial \sigma(z^L_j)}{\partial z^L_j} = \frac{\partial C_x}{\partial a^L_j}\sigma'\left(z^L_j\right). \tag{4.13}$$

From our second assumption, we know how to compute $\frac{\partial C_x}{\partial a^L_j}$. Furthermore, the sigmoid function has the attribute that its derivative is $\sigma'(z) = \sigma(z)(1-\sigma(z))$, which we can also compute. This means that everything in equation (4.13) can be computed from the input, output, and weighted input that we already know how to compute by the feed-forward process. Once again, we can avoid the cumbersome index notation by writing equation (4.13) in a vector-based form

$$\delta^L = \nabla_a C_x \odot \sigma'\left(z^L\right), \tag{4.14}$$

where $\nabla_a C_x$ is the vector of partial derivatives $\frac{\partial C_x}{\partial a^L_j}$ of the cost function with regard to the activation of the neurons in the output layer and $\odot$ is the Hadamard product, i.e., the element-wise product.

The error can be propagated back through the network using

$$\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l). \tag{4.15}$$

Equation (4.14) combined with equation (4.15) allows us to compute the error δ^l for any layer l.

Finally, we want to relate the error δ_j^l of a neuron to the partial derivatives of the cost function with regard to our fixed training example x. For a specific bias, we have

$$\frac{\partial C_x}{\partial b_j^l} = \delta_j^l \tag{4.16}$$

and, for a specific weight, we have

$$\frac{\partial C_x}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l, \tag{4.17}$$

all of which we already know how to compute.

Before we explain how to combine these four equations to form the full backpropagation algorithm, let us look at their proofs. All of the equations can be proven by using the chain rule and the definitions that are already stated in this chapter. We will only prove equation (4.15) as the proofs of the three other equations are similar. However, we present the three other proofs in the appendix for completeness.

What we want to prove is

$$\delta^l = \Big((w^{l+1})^T \delta^{l+1}\Big) \odot \sigma'(z^l). \tag{4.15}$$

Starting with the definition of the error from equation (4.12), we get

$$\delta_j^l = \frac{\partial C_x}{\partial z_j^l} \overset{(1)}{=} \sum_k \frac{\partial C_x}{z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \overset{(2)}{=} \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \overset{(3)}{=} \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l),$$

where (1) is application of the chain rule with regard to the weighted input of the neurons in the next layer $l+1$. (2) is substituting equation (4.12), the definition of the error. (3) comes from the fact that we can write

$$z_k^{l+1} = \sum_i w_{ki}^{l+1} a_i^l + b_k^{l+1} = \sum_i w_{ki}^{l+1} \sigma(z_i^l) + b_k^{l+1} \tag{4.18}$$

by using the definition of the weighted input and the activation. This gives us that

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l}\left(\sum_i w_{ki}^{l+1}\sigma(z_i^l) + b_k^{l+1}\right) = w_{kj}^{l+1}\sigma'(z_j^l) \tag{4.19}$$

by substitution of equation (4.18) and differentiation. We substitute equation (4.19) into the appropriate term. This shows that

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l),$$

which is equation (4.15) in component form.

Combining equation (4.3) and equation (4.14) – (4.17), we can define the full backpropagation algorithm for computing the gradient for equation (4.6):

- **Input:** a set of training examples $X_1, \dots, X_m$.
  For each training example:
  1. Complete a forward pass through the network to compute the weighted input z^l and the activation a^l for each layer by repeatedly applying equation (4.3).
  2. Find the error δ^L at the output layer by equation (4.14).
  3. Complete a backwards pass through the network to compute the error δ^l for each layer by repeatedly applying equation (4.15).
  4. Compute the gradient ∇C_{X_i} according to equation (4.16) and equation (4.17).
- **Output:** the gradient $\nabla C = \frac{1}{m} \sum_{i=1}^m \nabla C_{X_i}$.

The components of the gradient's output are exactly the partial derivatives needed to perform stochastic gradient descent. Combining the backpropagation algorithm with stochastic gradient descent thus gives us a way of training neural networks using a set of training examples.

Chapter 5

Search Methods

In the context of playing games, search methods are algorithms that search through the state space of the game to select moves. We describe search methods for discrete, two-player games with alternating moves. One approach to finding good moves is to look ahead by trying every possible move to figure out what will happen. For games with few possible moves, brute-force methods such as this can be effective, but more intelligent search methods are required for games with large state spaces and many possible moves. This chapter describes the search methods: minimax and Monte-Carlo Tree Search (MCTS). Both search methods build up a game tree to look ahead in the game and determine a move based on this tree. A game tree is a tree where nodes correspond to game states and child nodes correspond to the states resulting from taking some legal action from the parent node's state. MCTS builds an asymmetric game tree by randomly sampling playthroughs while minimax looks at all possible moves of both players, building a complete game tree to a certain depth. Section 6.7 describes how a learning agent can be integrated with these search methods to improve its performance.

5.1 Minimax

We call the current player the *maximizing player* because he wants to maximize the outcome of his moves, and we call the opponent the *minimizing player* because he wants to minimize the maximizing player's outcome, since this will maximize the minimizing player's own outcome in a zero-sum game. For a given state of the game, the idea of the minimax algorithm is to evaluate all of the maximizing player's moves, then evaluate all of the minimizing player's counter-moves, then the maximizing player's counter-moves, and so forth. Minimax repeats this process until a terminal state or a specified depth has been reached. It is assumed that the maximizing player will make the move with the highest value and that the minimizing player will make the move with the lowest value, i.e., the move that is rated worst for the maximizing player. In other words, the algorithm assumes optimal play from both players. This means that the best value found by minimax is a guaranteed lower bound for the maximizing player.

The value of a move is a measure of how good the resulting state is for the maximizing player. If the resulting state is terminal, a value is assigned to the node based on the result of the game. If the resulting state is not terminal, minimax can use a heuristic function in order to determine the state's value. A simple heuristic is to randomly choose a move if all the best moves are ones that result in non-terminal states. Using this heuristic and given sufficient search depth, minimax will make moves that guarantee a win and avoid a loss if it is possible. With a good heuristic function, we can assess moves that do not result in terminal states and make informed decisions even when none of the best moves result in terminal states.

Figure 5.1 shows the pseudocode for the minimax algorithm in its basic form. The initial call for minimax and the maximizing player is `minimax(root, depth, TRUE)`, where `root` is a node with the current state of the game. This pseudocode only returns the highest value that the maximizing player is guaranteed and would have to be modified to also return the move corresponding to that value.

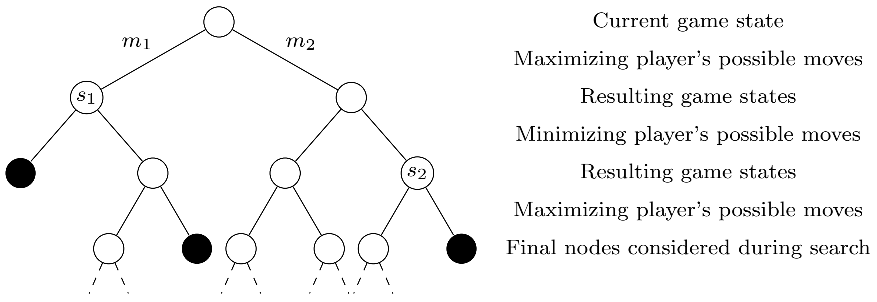
To select the best moves, we want to search as deep into the game tree as possible. The complexity of doing a minimax search can be described in terms of the branching factor of the game and the depth of the search. The branching factor of a game is the number of possible moves that lead to new states for each player. As an example, the branching factor of Connect Four is at most 7 since there are 7 possible moves corresponding to the 7 free columns at the start of the game. For most games, the branching factor decreases as the game goes on, allowing deeper searches later in the game when there are fewer possible moves. With branching factor b and search depth d, the cost of doing minimax search is $O(b^d)$.

Figure 5.2 shows an example of a minimax search tree with depth $d = 3$ for a game with branching factor $b = 2$. The game rules are such that a player can only win on their own turn, and there are only two outcomes: winning and losing. Once the search reaches a terminal node or a depth of three, the search stops. The filled nodes indicate nodes with terminal state and the dashed lines indicate where the search would continue if the

```
function minimax(node, depth, maximizingPlayer)
  if depth = 0 or node is a terminal node
    return the heuristic value of node

  if maximizingPlayer
    bestValue := -∞
    for each child of node
      v := minimax(child, depth-1, FALSE)
      bestValue := max(bestValue, v)
    return bestValue
  else (* minimizing player *)
    bestValue := ∞
    for each child of node
      v := minimax(child, depth-1, TRUE)
      bestValue := min(bestValue, v)
    return bestValue
```

Figure 5.1: Pseudocode for the basic minimax algorithm.



m_1
m_2
s_1
s_2
Current game state
Maximizing player's possible moves
Resulting game states
Minimizing player's possible moves
Resulting game states
Maximizing player's possible moves
Final nodes considered during search

Figure 5.2: Example of a game tree for minimax with search depth three and two possible moves per player. Black nodes represent terminal states.

search depth was increased. The leaf nodes that are non-terminal are where a heuristic function would be required in order to evaluate the game state. But we can determine which of the moves m_1 or m_2 that is the best move for the maximizing player even without a heuristic function. Since a player can only win on their own turn, we can see that making move m_1 moves the game to a state where the minimizing player has a winning move. Thus, we can conclude that the maximizing player has to make the move m_2 to avoid losing the game immediately.

From the game tree in figure 5.2, we can gain another insight: sometimes the search in a branch can be stopped early without affecting the final result. When considering the minimizing player's possible moves at s_1, we would like to prune the rest of the search tree when finding a winning move for the minimizing player. We can safely prune the tree because there is no better move than a winning move, so we know that the search cannot find any nodes with lower value for the maximizing player than what is already discovered. For more complex games, a winning move might not always be optimal, e.g., a move that is not an immediate win can secure more points for the maximizing player than a move that immediately ends the game. This leads to the idea of using alpha-beta pruning to prune the search tree when it is guaranteed that it will not affect the final result.

Figure 5.3 shows the pseudocode for the alpha-beta pruning version of minimax. The initial call for minimax with alpha-beta pruning and the maximizing player is `minimax(root, depth, -∞, ∞, TRUE)`. The intuition behind the α and β values is that α is the maximum lower bound of node values and β is the minimum upper bound. We only increase α and decrease β while searching through child nodes. At any node, if β is smaller than α, the remaining unvisited children will be pruned.

Let us return to the example minimax game tree in figure 5.2. We assign a value of ∞ to winning terminal states and value $-\infty$ to losing terminal states. If child nodes are visited from left to right, alpha-beta pruning would prune the right child of s_1 as β would be $-\infty$ after visiting the left child. However, notice that we are unable to prune the left child of s_2 because of the order of child nodes, as it will already have been visited when we discover a winning state in the right child. This is assuming that it is pruned as a result of the heuristic values found at the leaf nodes. If we visited the right child of s_2 first, the left child would be pruned.

The above example shows how the order in which child nodes are traversed is an important factor in the running time of minimax with alpha-beta pruning. If child nodes are searched through in the worst possible order, i.e., the best value is discovered last, alpha-beta pruning reduces to

```
function minimax(node, depth, $\alpha$, $\beta$, maximizingPlayer)
  if depth = 0 or node is a terminal node
    return the heuristic value of node

  if maximizingPlayer
    bestValue := $-\infty$
    for each child of node
      v := minimax(child, depth-1, $\alpha$, $\beta$, FALSE)
      bestValue := max(bestValue, v)
      $\alpha$ := max($\alpha$, bestValue)
      if $\beta \leq \alpha$
        break
    return bestValue
  else (* minimizing player *)
    bestValue := $\infty$
    for each child of node
      v := minimax(child, depth-1, $\alpha$, $\beta$, TRUE)
      bestValue := min(bestValue, v)
      $\beta$ := min($\beta$, bestValue)
      if $\beta \leq \alpha$
        break
    return bestValue
```

Figure 5.3: Pseudocode for the minimax algorithm with alpha-beta pruning.

normal minimax search. In this case, the complexity is still $O(b^d)$. However, the complexity of the algorithm drops to $O(\sqrt{b^d})$, assuming d is even, if the nodes are searched through in the optimal order, i.e., the best node is considered first (Russell and Norvig, 2003). This means that ordering the nodes correctly reduces the effective branching factor from b to $\sqrt{b}$, allowing the search to look twice as deep in the same amount of time.

5.2 Monte-Carlo Tree Search

MCTS is a family of algorithms that build an asymmetric game tree incrementally by performing simulations. In this section, we describe the Upper Confidence Bounds for Trees (UCT) algorithm, which is an instance of MCTS.

For each iteration of the algorithm, the *tree policy* selects a promising node of the tree. A promising node is either one that has been visited often and has a high estimated value, or one that has been visited infrequently. This is to ensure that all nodes are eventually selected even if their estimated value is low. If the selected node is terminal, it is returned directly. If it is non-terminal, a new child node that corresponds to an unexplored action is added to the tree and is returned. Once a node has been returned by the tree policy, a game is simulated until termination from the board state of the returned node, and the tree is updated according to the result of this simulation. The moves made during a simulation are chosen according to a *default policy*. In the simplest case, the default policy makes random moves.

One of the benefits of MCTS is that little knowledge about the game is required in order to apply the algorithm. Minimax search requires that non-terminal nodes are given a heuristic value, which means that some knowledge about game states is necessary. In contrast, MCTS only requires that we can assign values to terminal states and simulate the rules of the game. Another property of MCTS is that it can be interrupted at any time, whereas minimax search is started with a certain target depth and must run until this search depth has been reached.

Figure 5.4 shows the pseudocode for UCT. Each node v in the tree has four values associated with it: $s(v)$ is the state of the node, $N(v)$ is the number of times the node has been visited, $Q(v)$ is the sum of simulation rewards in this node and its children, $a(v)$ is the action that generates this node's state from its parent node's state, and $f(s(v), a)$ is the function that returns the state resulting from performing action a in the state $s(v)$. A new node is initialized with $Q(v) = 0$ and $N(v) = 0$.

```
function UCTSearch($s_0$)
  initialize root node $v_0$ with state $s_0$
  while within computational budget
    $v_l$ := treePolicy($v_0$)
    $\Delta$ := defaultPolicy($s(v_l)$)
    backup($v_l$, $\Delta$)
  return a(bestChild($v_0$, 0))

function treePolicy($v$)
  while $v$ is non-terminal
    if $v$ not fully expanded
      return expand($v$)
    else
      $v$ := bestChild($v, C_p$)
  return $v$

function expand($v$)
  choose $a$ from untried actions from $A(s(v))$
  add new child $v'$ to $v$
    where $s(v') := f(s(v), a)$
          $a(v') := a$
  return $v'$

function bestChild($v, c$)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

function defaultPolicy($s$)
  while s is non-terminal
    choose $a$ from $A(s)$ uniformly at random
    $s := f(s, a)$
  return reward of state $s$

function backup($v, \Delta$)
  while $v$ is not NULL
    $N(v) := N(v) + 1$
    $Q(v) := Q(v) + \Delta$
    $\Delta := -\Delta$
    $v$ := parent of $v$
```

Figure 5.4: The UCT algorithm pseudocode with two-player backup (Browne et al., 2012).

`backup` is applicable to two-player, zero-sum games with alternating moves, e.g., Connect Four or Chess. This assumption for `backup` is necessary because we need to adapt the reward Δ for every change of turn to the player in question. Since the game is zero-sum, we can negate Δ.

`treePolicy` uses `bestChild` to select promising child nodes. The following two terms of the equation in `bestChild` attempt to balance exploration and exploitation in the tree search.

$$\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

The first term $\frac{Q(v')}{N(v')}$ is the average return of playouts that have passed through this node and, as such, it is an approximation of the node's game-theoretic value. The second term $c\sqrt{\frac{2 \ln N(v)}{N(v')}}$ ensures that all child nodes are eventually explored. Whenever a node is visited, the value of this term decreases for that node. However, when a node's sibling is visited, the value of this term increases. This ensures that the search continues to visit all nodes at some point. The value of c controls the amount of exploration. Higher values of c increase the amount of exploration while lower values encourage less exploration and more exploitation of current knowledge. C_p is a hyperparameter that is chosen based on the problem domain.

Figure 5.5 shows an example of a game tree built by UCT for Connect Four. In this game, a player has two possible actions in each state. The selected node represents a non-terminal state, which has not yet been fully expanded. A new child corresponding to a random action is added and returned by the tree policy. The default policy then runs a simulation from the newly returned node, which results in a win for the player who made the last move. The new node is updated with the result of the simulation and this result is backed up to the root of the tree. This results in the game tree shown on the right side.

In this instance of MCTS, the action returned by the search is the action of the child with the highest game-theoretic value. Another way of choosing an action is to return the action of the most visited child. The most visited child must have been chosen often by the tree policy, thus it must have a high estimated value, and a child that has been visited many times should have a stable estimated value. Alternatively, the two methods can be combined, allowing the search to terminate only if the child with the highest game-theoretic value is also the most visited child, but this means that the search method may have to exceed its budget.

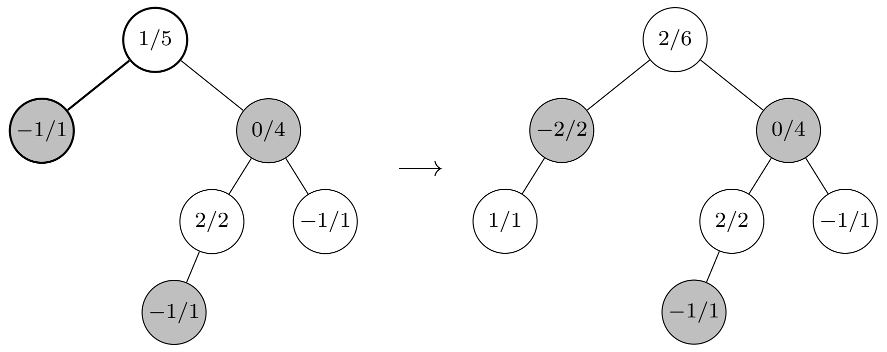


Figure 5.5: Example of a game tree built by UCT. All nodes v are annotated with the current value of $Q(v)$ and $N(v)$, written as $Q(v)/N(v)$. Gray nodes indicate states where the opponent makes a move. White nodes indicate states where the searching player makes a move. The bold nodes indicate the children selected by the tree policy in the current iteration.

Chapter 6

Putting It All Together

6.1 Connect Four as a Reinforcement-Learning Problem

This section formalizes Connect Four as a reinforcement-learning control problem. We need to define the states, actions, transitions, action-value function, and rewards for the problem. The set of possible states $\mathcal{S}$ is the set of all legal board positions. Terminal states are the states where some player has won or where the board is filled without a winner, resulting in a draw. For a state s, the set of possible actions $\mathcal{A}(s)$ is the set of columns where there exists an unoccupied row. For example, the state s_a shown in figure 6.1 has the possible actions $\mathcal{A}(s_a) = \{a1, b2, e2, f2, g1\}$. The agent is only rewarded upon reaching a terminal state where the reward is 1 for a win, 0 for a draw, and 0 for a loss. This definition of Connect Four has the Markov property, since the current board position and some action completely defines the next state and reward. We consider the opponent to be a part of the environment as it is not something our agent can control.

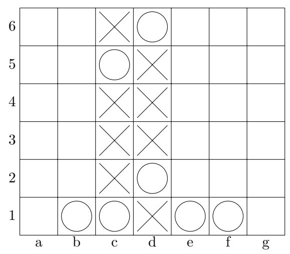


Figure 6.1: Board state s_a.

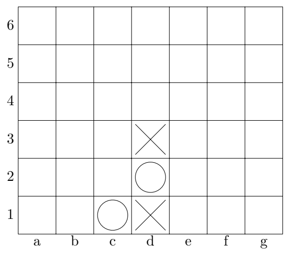
Since Connect Four is finite and has the Markov property, we have a finite Markov decision process.

Our action-value function maps state-action pairs (s, a) to the estimated probability that the agent will win by performing action a in state s. The interpretation is that if $Q(s, a) \approx 1$, the agent is confident that it will win from state s by performing action a. Conversely, if $Q(s, a) \approx 0$, the agent is confident that it will lose from state s by performing action a.

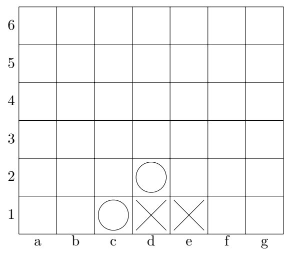
Since Connect Four is deterministic, we know that performing some action a in some state s will always lead to the same state s'. This means that we can incorporate the notion of *afterstates* described by Sutton and Barto (1998). The next section describes how we can incorporate afterstates and why they are interesting.

6.2 Reducing the State Space through Afterstates

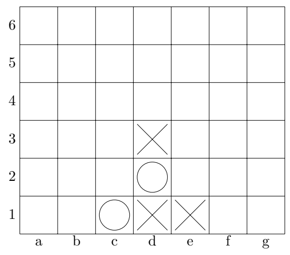
When dealing with a control problem, the agent usually wants to approximate an action-value function $Q(s, a)$ that gives the value of performing action a in state s. For some problems, the immediate result of an action is already known. For example, consider the Connect Four board shown in figure 6.2a. In this case, the agent plays $\otimes$. The state tells us that it is the agent's turn to move as there is an even number of pieces. Let us call the state in figure 6.2a s_1. If our agent chooses to make the move *e1*, the resulting board will be s'_1, which is shown in figure 6.2c. The learning agent can then make an update to its action-value function after the opponent has made his move and the learning agent gets a reward from the environment. The idea of afterstates is that we only estimate the value of states instead of estimating the value of every action from every state. We can do this because being in state s_1 and performing action *e1* always results in entering state s'_1. Instead of estimating the pair s_1 and *e1*, we estimate the value of state s'_1 and make use of the fact that it is the result of performing action *e1* in state s_1. It is not immediately clear why this reduces our state space as we still have to estimate the value of seven afterstates for each state as opposed to seven state-action pairs. Looking at one more example illustrates why afterstates are actually an improvement. Say we instead find ourselves in the state s_2, which is shown in figure 6.2b, and choose to make the move *d3*. This action will result in the state s'_1, which is shown in figure 6.2c, even though the previous state and action were different from before. Because the two state-action pairs result in the same state, we only have to estimate $V(s'_1)$ instead of estimating the values of both $Q(s_1, e1)$ and $Q(s_2, d3)$.



(a) Board state s_1, ⓧ's turn to move.



(b) Board state s_2, ⓧ's turn to move.



(c) Board state s'_1, result of move *e1* from s_1 or move *d3* from s_2.

Figure 6.2: Example board states.

The result is that the state space of the problem is reduced with the branching factor for each state.

6.3 Combining Reinforcement Learning and Neural Networks

Chapter 3 describes reinforcement learning and chapter 4 describes neural networks. We want to combine these concepts in order to create an agent that learns from playing Connect Four. The basic idea is that we will train a neural network to be the value function of our learning agent. In order to choose which action to take for a state, we evaluate the possible actions and use the neural network to find the approximate value of each action. We then choose an action based on the values returned by the network. As actions are chosen and the environment responds, we can use the rewards and resulting states to improve the approximation done by our network. In section 6.7, it is also described how we can incorporate the search methods described in chapter 5 to improve the decision making of our learning agent.

We will use TD(λ) to train our agent. The parameters θ of our value function are the weights and biases of the neural network. Recall that the updates made to these parameters in TD(λ) are

$$\theta_{t+1} = \theta_t + \alpha(r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t))e_t,$$

where e_t is the vector of eligibility traces for each parameter

$$e_t = \gamma\lambda e_{t-1} + \nabla_\theta V_t(s_t).$$

We need to be able to compute $\nabla_\theta V_t(s_t)$, which is the vector of the partial derivatives of the output of the network with regard to its weights and biases.

Recall also that the backpropagation algorithm for neural networks allows us to compute the partial derivatives of a cost function with regard to the weights and biases of the networks. Fortunately, there is a small change to the backpropagation algorithm that makes it compute what we need. We change the definition of the error at each neuron to

$$\delta_j^l = \frac{\partial a^L}{\partial z_j^l},$$

i.e., we define the error of a neuron to be the partial derivative of the output activation of the network with regard to the weighted input of that neuron. Since the activation at the output layer is defined as

$$a^L = \sigma\left(z^L\right),$$

this new error can easily be computed at the output layer by

$$\delta^L = \frac{\partial a^L}{\partial z^L} = \sigma'\left(z^L\right).$$

The rest of the backpropagation algorithm remains unchanged. Through proofs identical to those for the original backpropagation algorithm, it can be shown that

$$\frac{\partial a^L}{\partial b_j^l} = \delta_j^l$$
$$\frac{\partial a^L}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$

for our new definition of error. These are exactly the values that we need to compute in order to run TD(λ).

A potential problem with Connect Four is the discrete outcome of each game, but it is not obvious whether this is truly a problem in practice. Since our neural network approximates the value function by a continuous function, it may not find the optimal value function. However, the probability of winning from each state might still be closely approximated by a continuous function.

6.4 Board Encoding

As described in chapter 4, a neural network takes as input a vector of numbers, which feeds through the network to produce an output. Therefore, we need to translate board positions into vectors that the network can take as input. The choice of board encoding has an impact on the performance of the network since only features present in the encoding can affect the output of the network. The board encodings described in this section assume a standard Connect Four board of size 6×7, but they can extend to boards of any size.

The first board encoding uses 42 entries to encode the pieces placed on the board using one number per square. Let us call it the *naive board encoding.* A square occupied by an agent's piece is given a value of 1. A square occupied by an opponent's piece is given a value of -1. A square that is not occupied is given a value of 0. In addition, a value is used to indicate whose turn it is to move, i.e., the value is set to 1 if it is the agent's turn to move and -1 if it is the opponent's turn to move.

There are some potential problems with the naive encoding. A piece in a square has to be given the same weight for the agent and the opponent. This puts an unnecessary restriction on our network as there is no a priori reason for why this should be the case. Another restriction made by this board encoding is that a blank square cannot be weighted in the evaluation of a state since blank squares are given an entry of zero in the input vector.

In the light of these potential problems, we draw inspiration from the encoding used in TD-Gammon as described by Sutton and Barto (1998) in their case study. The resulting encoding uses 126 bits to encode the squares on the board, i.e., it has three bits per square on the board and two bits to indicate turn. For the three bits corresponding to a square, the first bit is set if the agent has a piece in the square. The second bit is set if the opponent has a piece in the square, and the third bit is set if the square is empty. In this way, blank squares can also be given a weight and possession of a square can be weighted differently for the agent and the opponent.

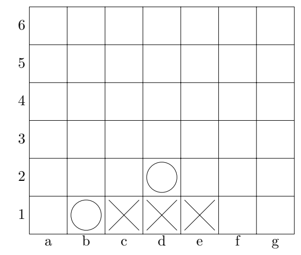


Figure 6.3: The square *f1* is a threat to ⓞ.

The two additional bits indicate turn in a binary fashion. Let us call this the *extended board encoding.*

Finally, we experiment with a board encoding that has a set of handcrafted features in its input in addition to the input encoded by the extended board encoding. We look at the feature that encodes all threats on the board. This feature uses an additional 84 bits compared to the extended board encoding, resulting in 212 bits in total. This gives us one more bit per square on the board per player. A bit is set if the square that it corresponds to is blank and the player that it corresponds to will win if they place a piece in that square. For example, in the state shown in figure 6.3, the bit corresponding to square *f1* and player ⓧ would be set since the square is currently blank and ⓧ wins by dropping a piece in that square. We call this the *feature board encoding.* When including these handcrafted features, it is important to keep in mind that increasing the number of inputs to the network also increases the number of weights and biases, thus increasing the complexity of the network. This means that we also increase the number of training games that are required to appropriately tune the additional weights and biases. Therefore, we have to consider the trade-off between including more input information and increasing the complexity of the network. On top of this, one of the main motivations for using reinforcement learning was that we did not have to know about the game beforehand. However, if we want to construct more interesting features, we have to develop a deeper understanding of the game, which goes against our goal of automatically learning the important features of the game.

We compare the different board encodings in section 7.5.

6.5 Benchmarking

In order to evaluate the progress of an agent's training, some form of benchmark is required. One way to evaluate an agent is to make the agent play against an opponent player between training epochs and keep track of the record against this *benchmark opponent*. We present three benchmark opponents: a random player, a minimax player, and an MCTS player.

A random player chooses a move uniformly at random from the set of legal moves. This opponent should be easily beaten as it will rarely make sensible plays. Nevertheless, it works as a sanity check to make sure that progress is made during training.

A minimax player uses the minimax algorithm described in section 5.1 to look for moves. If no good move is found, a move is selected uniformly at random from the set of possible moves. This opponent plays like a random opponent at first, but will block immediate threats and foresee future threats with appropriate search depth. The benefit of minimax is that its strength can be scaled by increasing the search depth, but this has the downside of slowing benchmarking down to the point where it becomes the bottleneck of training if the search depth is too high. We found that a search depth of five provided a good measurement of the agent without taking too much time compared to the training. Performance against a minimax opponent also seemed to be a good indication of the difficulty of playing against the agent as a human.

An MCTS player uses the MCTS algorithm described in section 5.2 to look for moves. As with the minimax player, this benchmark opponent's strength can be scaled by varying its search time. This opponent outperforms our agent with just $10\,ms$ to search for a move on a standard board. Because of this, the MCTS player is only used for benchmarking when the size of the board is increased or when the learning agent is also allowed to use search methods to make its moves. We experiment with increasing the size of the board in section 7.8.

Another way to benchmark an agent is to make it play against human players. Using humans as a benchmark opponent during training is a tedious process as many games are needed to accurately assess the agent's play and humans usually make moves slowly compared to computer opponents. So, with a lot of training, benchmarking against humans becomes a time-consuming process.

6.6 Addressing the Problems of Self-Play

Section 3.9 introduced self-play, but Tesauro (1995) and Schraudolph et al. (1994) noted that deterministic games need noise injection in order to make self-play feasible. Since Connect Four is deterministic, we need to ensure that the agent does not play the same games over and over, thus not learning.

One way to address noise injection is exploratory starts. The idea is to randomize the starting state of each self-play game, i.e., each self-play game can be started after a number of random moves has been played by both sides. In this way, the number of random initial pieces can be adjusted as a parameter for the amount of noise. One potential problem is that adding too many initial pieces can make the game meaningless since one side might already have won because of a stronger opening. Over time, the number of games decided by the random opening should be even for both sides, but this might result in wasted training games.

Another way to ensure varied play is to keep the policies, i.e., the action selection, exploratory during training, which is described in section 3.5. For both the ε-greedy policy and the softmax policy, we want to keep the exploration rates ε and τ high enough such that we keep exploring new moves. As the agent becomes better at estimating state values, the exploration rate should decrease to improve these estimates.

The exploration rate, i.e., ε or τ, should decrease gradually as training progresses such that the agent eventually converges towards an optimal value function. However, lowering the exploration rate too much will make the learning stagnate and produce overfitting during self-play. To avoid stagnation, we maintain some level of exploration during training at all times. In section 7.2, we compare the noise-injection methods outlined above and find initial values for the exploration rate and the number of initial pieces.

6.7 Incorporating Search Methods

Chapter 5 describes two search methods: minimax and Monte-Carlo Tree Search (MCTS). This section explains how these search methods can be incorporated into our learning agent in order to improve its performance.

Without search methods, an action is selected by letting the agent evaluate the resulting state of each possible action. Then a policy chooses one of the actions based on their estimated value. As an alternative, we can utilize a search method to find an action using the agent to improve the search.

6.7.1 Minimax search

An agent can enhance minimax with alpha-beta pruning in two ways. First, minimax needs a heuristic function to evaluate non-terminal states. We can use the agent's value function to estimate these heuristic values. Second, we can evaluate the states of the children of each node and order them by the value that they are estimated to have according to the agent's value function. As a result, the better the agent's value function is at evaluating board positions, the more efficiently we can prune our search tree and the more accurately we can evaluate non-terminal positions.

When playing against a human, the agent is able to use minimax with search depth 9 and alpha-beta pruning, but increasing the search depth makes the agent slow to play against as it takes about 15 seconds to make a move at the beginning of the game on a standard laptop. The search depth can be increased as the game goes on because the state space that we search through is reduced significantly as more pieces are placed on the board.

6.7.2 Monte-Carlo Tree Search

When using MCTS, the agent can be used to try to improve the two policies: the tree policy that selects nodes and the default policy that is used to select actions during simulations.

The easiest policy to modify is the default policy. Instead of choosing actions uniformly at random during a simulation, the agent can be used to find the best action according to its value function. This should provide a great improvement in accuracy over random simulations, but it comes at the cost of slower simulations. A way to improve the speed is to only allow the network to consider some of the legal moves from each state (Drake and Uurtamo, 2007). This increases the speed of each simulation by reducing the branching factor and ensures that the simulations are still randomized such that the same simulation is not run every time. We compared an MCTS player that uses random simulation to an MCTS player that uses the agent's value function to make simulation. Using random simulations turned out to be better because the player was able to make more simulations, which made up for the randomness.

With regard to the tree policy, several extensions have been suggested and experimented with in the literature. These extensions include the evaluation of a heuristic function when selecting the best child. That is, the extensions modify the equation

$$\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

such that it also includes a term with an estimation from some heuristic function. Recall that $Q(v)$ is the sum of rewards from playouts that went through node v and $N(v)$ is the number of visits to node v during search. In Silver et al. (2016), the results of simulations are combined with the evaluation of a trained value function using a mixing parameter λ that determines the importance of the simulations and the function evaluation. A different approach by Gelly and Silver (2011) is to initialize the values $Q(v)$ and $N(v)$ according to a heuristic evaluation function $H(s, a)$ and a heuristic confidence function $C(s, a)$. New tree nodes are initialized with $Q(v) = H(s(v), a(v))$ and $N(v) = C(s(v), a(v))$. The confidence is equivalent to how many simulations would be required in order to achieve a value of similar accuracy to the heuristic value. We experiment with initializing $Q(v)$ with the agent's value function and choosing a corresponding $N(v)$ in section 7.7.

Chapter 7

Experiments

This chapter describes the experiments that were done in order to optimize the hyperparameters for TD(λ) and verify some of the hypotheses from the previous chapters. One issue with these experiments is that they depend on each other. We chose the order of these experiments such that the parameters that we estimate to have the biggest impact are optimized first.

We evaluate agents in two ways: The agent can play against a search method, which we will refer to as a *benchmark*. The agent can play against another agent, which we will refer to as a *tournament game*. Section 7.3 explains tournaments and tournament games.

When an agent is benchmarked or plays in a tournament, we want the agent to play greedily with regard to its value function and we do not want the agent to change its value function. Making the agent play with these settings makes it deterministic, which will make the two players repeat the same game if the opponent is also deterministic. To keep the games from repeating, we use a softmax policy instead of a greedy policy. The reason for using softmax rather than ε-greedy is that the randomness in softmax is dependent on the evaluations of the value function, whereas the exploratory moves in an ε-greedy policy are chosen uniformly at random. In the following sections, all benchmark games are played with learning disabled and with a greedy policy since we benchmark against minimax and MCTS, which are both non-deterministic. When we benchmark against minimax, we give it a search depth of five and the heuristic function that returns 0 for non-terminal states, 1 for winning states, and -1 for losing states. All tournament games are played with learning disabled but with a softmax policy with a temperature of 0.1.

The experiments involve training and doing benchmarks with many different agents and parameter settings. Determining how many games to train and how many benchmark games to run between each epoch is a trade-off between time spent and accuracy. More training yields a better result in most cases, but it is more time consuming. Another concern is the mutual dependency between hyperparameters and the number of training games. Hyperparameters are optimized for a certain number of training games, and the number of training games required to reach good performance depends on the hyperparameters found. We choose to run each experiment with 10,000 training games for 10 epochs, resulting in 100,000 training games, unless otherwise stated. This number of training games allowed the settings that we found during our initial experiments to reach peak performance against a minimax benchmark opponent. We later experiment with additional training games in section 7.3 and section 7.9. During benchmark games, we are trying to assess the agent's win rate against its benchmark opponent. Since the benchmark games are independent, the chance of being a constant factor away from the true win rate decreases exponentially with more benchmark games by a Chernoff bound. The benchmark games are independent because we disable learning during benchmarks. Because of this, we have settled on 1,000 benchmark games after each epoch. All experiments will use these settings and train with self-play unless otherwise stated.

We have to decide on which board encoding to use and whether to switch sides during training, until we run those experiments in section 7.5 and section 7.6. With regard to board encoding, we will use the feature board encoding described in section 6.4 for all agents. With regard to switching sides, we let the agent start in benchmark games. For all other games, we make the players switch starting positions between each game. The reason for letting the agent start in benchmark games is that we know from Allis (1988) that the starting player is always able to win. Thus, we know that our agent is not playing optimally if it is unable to win every game.

7.1 Different Hyperparameters

To compare the different training setups, we need to have the best parameters for each scenario. To find the best hyperparameters, we use a combination of grid search and random search. In grid search, we define a set of values for each parameter and then test all combinations of those values.

For example, with two hyperparameters γ and λ, we could define

$$\begin{aligned}\gamma &\in \{0.25, 0.5, 0.75, 1\}\\ \lambda &\in \{0, 0.5, 1\}.\end{aligned}$$

A grid search tries all combinations of γ and λ from the two sets.

One requirement for a reasonable grid search is that these sets of possible values must be defined beforehand, i.e., we need a way to figure out what values might work. We use random search over the different parameters to help find initial values. The random search simply creates combinations of uniformly distributed random values on the intervals of each hyperparameter (Bergstra and Bengio, 2012).

Both random search and grid search produce a lot of training and benchmark data. To avoid manually looking at the data for every combination of parameters, we sort and filter the results by benchmark performance. Combinations of parameters that do not improve their benchmark performance are removed and the results are sorted. In this way, we only have to look at the best performing combinations even if thousands of combinations were tried during the search. Due to the random nature of network initialization and early training, we train and benchmark each set of parameters three times and average over the results.

Benchmark performance can be measured in several ways. The best performing combination can be the one whose benchmark win rate is: the highest; most increasing; the most stable, i.e., does not suddenly fall and then increase again; or a weighted combination of the above. We have chosen the settings that give the highest win rate because that is our ultimate goal and we want to evaluate the networks on how well they are doing, not on how they are achieving it.

The hyperparameters to determine consist of the exploration rate, learning rate, discount rate, and decay rate. Exploration rate is ε when using an ε-greedy policy and the temperature τ when using a softmax policy. A good value for the exploration rate will be determined in section 7.2. Learning rate is the α value for the reinforcement-learning algorithm. Discount rate γ is the amount of decay on rewards in the reinforcement-learning algorithm. Decay rate λ is the trace-decay parameter that controls how much rewards affect previous states. This means that there are four hyperparameters to set for TD(λ). To reduce the number of parameters that we vary at the same time, we exploit that the exploration rate is sufficiently independent of the other hyperparameters and optimize it after the others. Furthermore, we keep α constant while varying γ and λ and vice versa.

Parameter	TD(λ)
Learning rate (α)	0.1
Discount rate (γ)	1
Decay rate (λ)	0

Table 7.1: Best hyperparameter settings for TD(λ).

All parameters were evaluated by doing self-play training and benchmarking against a minimax opponent. The initial grid searches had 10 epochs with 1,000 training games per epoch and 100 benchmark games after each epoch to allow rapid evaluation of many parameters. The best parameters found from these searches were then used for further searches with 10 epochs of 10,000 training games and 1,000 benchmark games after each epoch. Training was done with an ε-greedy policy and a randomly initialized board with up to 6 pieces each game. Randomly initializing the board is discussed in section 7.2 along with the exploration rate.

We trained networks with one hidden layer and 100 hidden neurons. Table 7.1 shows the best combination of parameters found for TD(λ). Section 3.6 introduced eligibility traces, but it turns out that using $\lambda = 0$ performed best. Setting the decay rate to 0 is equivalent to disabling eligibility traces and only making one-step updates. This is not surprising, since "in off-line applications in which data can be generated cheaply, perhaps from an inexpensive simulation, then it often does not pay to use eligibility traces" (Sutton and Barto, 1998). It is worth noting that for other games, other decay rates are used, e.g., Schaeffer et al. (2001) used $\lambda = 0.95$ for their TD approach to Checkers.

With these settings, TD(λ) achieved an average win rate of 78.5% in benchmark games after the final epoch of training. The experiments in the following sections use these settings for training unless otherwise stated. Note that these settings were found without changing parameters during training. Section 7.4 describes experiments where we let some of these parameters decay over time.

7.2 Different Noise-Injection Methods

Since learning stagnates during self-play in deterministic games (Sutton and Barto, 1998; Schraudolph et al., 1994; Tesauro, 1995), we introduce nondeterminism through noise injection. We inject noise in two ways: exploration rate and exploratory starts by randomly placing initial pieces.

Exploration Rate. The exploration rate is set for two policies: ε-greedy and softmax. ε-greedy has a probability of ε to choose a random move or otherwise choose a move greedily. Softmax chooses every move randomly but softmax is more likely to choose the best estimated moves. The temperature τ determines how greedy the selection is.

Exploratory Starts. Exploratory starts perform noise injection by starting the game at a random state instead of always starting with an empty board. We initialize the board by placing up to n random initial pieces, letting the game proceed from the resulting state.

What we want to determine is how these two methods can be most effectively combined to train networks. Benchmarks are done against minimax. All settings are run three times and results are averaged.

We want to vary both the exploration rate and the number of random pieces n used during exploratory starts. For ε-greedy policies, we try ε values of 0, 0.25, 0.5, 0.75, and 1. For softmax, we try τ values of 0.1, 1, 10, 100, 500, and 1000. Finally, we try n values of 0, 5, and 10.

Table 7.2 shows the results of the most interesting settings. With an ε-greedy policy, the first result indicates that $\varepsilon = 0$ with 0 initial pieces cannot do any learning because the agent does not explore at all. However, if we initialize the board with 5 or 10 initial pieces, some learning can take place even with $\varepsilon = 0$, i.e., with a policy that is greedy with regard to its network. It is also noteworthy that with $\varepsilon = 0.5$ or even $\varepsilon = 1$, a lot of meaningful learning can take place. In fact, $\varepsilon = 0.5$ and no initial pieces achieved the highest win rate out of all the ε-greedy settings that were tested. The performance of agents with high values of ε or τ seem unaffected by the number of initial pieces as their benchmarks with 0, 5, and 10 initial pieces are within 3% of each other.

With a softmax policy, it turns out that $\tau = 0.1$ performed best out of all settings, regardless of the number of initial pieces. In fact, these were the only softmax settings to achieve a win rate above 70% in benchmarks. As opposed to the values tested for ε-greedy, all softmax settings seemed to have meaningful learning. The worst performing softmax settings were $\tau = 100$ with 5 initial pieces, which achieved a benchmark win rate of 63%.

A natural conclusion from these results is that the softmax values for τ might have been set too high. It would seem plausible that lower values of τ could produce more interesting results as the lowest value performed best in all cases. As a follow-up, we ran the same experiment again for softmax with τ values of 0.05, 0.01, and 0.005.

Policy	Exploration rate	Initial pieces	Benchmark win rate
ε-greedy	0	0	8.0%
ε-greedy	0	5	60.3%
ε-greedy	0	10	61.9%
ε-greedy	0.5	0	80.5%
ε-greedy	0.5	5	74.7%
ε-greedy	0.5	10	75.3%
ε-greedy	1	0	66.3%
ε-greedy	1	5	67.9%
ε-greedy	1	10	66.7%
Softmax	0.1	0	75.5%
Softmax	0.1	5	80.5%
Softmax	0.1	10	82.6%
Softmax	1000	0	68.7%
Softmax	1000	5	66.6%
Softmax	1000	10	69.1%

Table 7.2: Partial noise-injection experiment results. The benchmark win rate shows the percentage of games won against minimax in the 1,000 benchmark games after the final epoch of training.

Table 7.3 shows the results from this follow-up experiment. All of these temperatures performed worse than $\tau = 0.1$, regardless of the number of initial pieces. The best performing settings were $\tau = 0.05$ with 5 initial pieces, which achieved a benchmark win rate of 80.7%. The lower temperatures show the same behavior as low values of ε for the ε-greedy policies.

We can conclude the following about noise injection: For both ε-greedy and softmax policies, it is sufficient to have exploration and noise implemented by the policy. We observed that there was little change in benchmark by randomly initializing the board when using a high exploration rate. However, adding initial pieces was beneficial for the policies with a low exploration rate. The conclusion is that we need some amount of noise to keep the training games from stagnating, which is also concluded by Sutton and Barto (1998); Schraudolph et al. (1994); Tesauro (1995). Whether that noise comes from exploratory starts or the policy of the agent does not seem to matter. Since we plan to decay the exploration rate over time, it makes sense to keep exploratory starts in order to ensure that learning does not stagnate once the exploration rate becomes low.

Temperature	Initial pieces	Benchmark win rate
0.05	0	76.0%
0.05	5	80.7%
0.05	10	78.5%
0.01	0	66.0%
0.01	5	62.6%
0.01	10	71.7%
0.005	0	50.6%
0.005	5	63.6%
0.005	10	75.0%

Table 7.3: Results from the follow-up noise-injection experiment with softmax policies. The benchmark win rate shows the percentage of games won against minimax in the 1,000 benchmark games after the final epoch of training.

From now on, we use softmax with a temperature of 0.1 and 10 initial pieces since these settings produced the best results. We will use these settings for the experiment in section 7.3, but afterwards we are going to decay the exploration rate over time, which might require a new initial value.

7.3 Different Architectures

The hidden layers of the network architecture can be varied in two different ways: the number of neurons per hidden layer and the number of hidden layers. More of both gives the ability to approximate more complex functions. However, more neurons and layers also increases the number of training games required to achieve peak performance, and more complexity is not necessarily better. So, we look for a golden mean.

In order to determine what kind of architecture performs best, we train several different architectures and compare their performance. We do this by making all of the trained networks play against each other in a round-robin tournament, i.e., every network plays against every other network. For each architecture, we train three networks and select the network with the highest benchmark to represent that architecture in the tournament. Each matchup in the tournament will consist of 10,000 games between the two contestants.

As mentioned in chapter 4, there are no precise rules for how many hidden neurons or how many hidden layers a network should have. Some general recommendations and rules of thumb by Bengio (2012), and Sheela and Deepa (2013) do exist:

- Increasing the number of hidden neurons usually does not hurt generalization.
- Using the same size for all hidden layers generally works better than varying the amount of neurons per layer.
- Having only one hidden layer should be sufficient, but more layers can be beneficial in some cases.
- The number of hidden neurons should be somewhere between the number of input neurons and number of output neurons.
- The number of hidden neurons should be smaller than two times the number of input neurons.

What we can conclude from these rules of thumb is that the architecture mostly comes down to the problem at hand, but the guidelines can help us choose some sensible candidates.

Table 7.4 shows the results of the round-robin tournament. An entry in the table lists the win rate of the architecture in that row against the architecture in that column. For each architecture, only the number of hidden neurons in each hidden layer is listed since all of the networks have the same number of inputs and outputs. The champion of this tournament is the shallow network with 100 hidden neurons, but the other shallow networks perform similarly. The results show that the shallow networks perform better than the deeper networks. However, it could be the case that the simpler architectures have an advantage due to the number of training games not being high enough. We will attempt to test this hypothesis by running a second tournament. All of the settings for the second tournament are kept the same as for the previous tournament except the number of training epochs is increased to 20, increasing the number of training games to 200,000. The results of this tournament can be seen in table 7.5. The additional training games allow the more complex networks to perform better than before, and we see that the shallow network with 500 hidden neurons has a win rate of more than 50% against all other networks. The shallow network with 100 hidden neurons, which championed the first tournament, still wins all of its matches against the deeper networks but loses against both of the more

Architecture	500	250 × 250	250	100 × 100 × 100	100 × 100	100
100	51.8%	70.1%	51.6%	60.3%	54.0%	-
100 × 100	48.0%	67.1%	47.8%	56.2%	-	
100 × 100 × 100	43.7%	56.7%	43.2%	-		
250	50.3%	67.9%	-			
250 × 250	31.8%	-				
500	-					

Table 7.4: Results from the round-robin tournament to optimize the network architecture. An entry lists the win rate of the architecture in that row against the architecture in that column. $x \times y$ represents a network architecture with x neurons in the first hidden layer and y neurons in the second hidden layer.

Architecture	500	250 × 250	250	100 × 100 × 100	100 × 100	100
100	46.1%	55.2%	49.3%	57.5%	51.5%	-
100 × 100	43.7%	54.5%	49.1%	59.2%	-	
100 × 100 × 100	37.3%	54.5%	40.8%	-		
250	45.1%	55.0%	-			
250 × 250	40.2%	-				
500	-					

Table 7.5: Results from the second round-robin tournament to test the effect of more training games.

complex shallow networks. The tendency seems to be that increasing the number of training games allows the more complex networks to outperform the simpler ones.

To verify whether this is the case, we run a third tournament with 50 epochs of training for each network, increasing the number of training games to 500,000. Since the shallow network with 500 hidden neurons won the last tournament, we include some more complex architectures in this tournament. We exclude the network with three layers and the shallow network with 250 hidden neurons from this tournament but keep the shallow network with 100 hidden neurons because it won the first tournament. The results of the third tournament can be seen in table 7.6.

One thing to note about this tournament is that there was a huge difference between the time spent training for each network architecture. The previous tournaments both finished within less than a day. In the third tournament, the simpler architectures also finished all of their training runs within a day. However, the most complex architectures, i.e., the 500 × 500

Architecture	500×500	1,000	750	500	250×250	100×100	100
100	67.0%	57.6%	48.2%	57.4%	58.8%	57.6%	-
100×100	59.4%	51.9%	43.4%	49.3%	51.7%	-	
250×250	57.1%	49.1%	40.7%	48.1%	-		
500	60.5%	49.4%	42.9%	-			
750	66.0%	56.9%	-				
1,000	57.5%	-					
500×500	-						

Table 7.6: Results from the third round-robin tournament with additional training games.

network and the shallow network with 1,000 hidden neurons, took almost five days to complete 50 epochs of training on a standard laptop.

As for the tournament itself, the shallow networks still perform better than the deeper networks in general. The most surprising result is that the shallow network with 500 hidden neurons performs worst of all the trained shallow networks. Furthermore, the network with 750 hidden neurons performs better than both the simpler and the more complex networks. The data suggests a couple of things: First, it suggests that the 1,000 hidden neuron network could benefit from additional training. Second, it suggests that the network with 500 hidden neurons might have begun overfitting with this number of training games since its performance relative to the other networks suddenly drops. It seems plausible that the shallow network with just 100 hidden neurons performs better now because its simpler architecture is less prone to overfitting. We could keep running more tournaments with additional training games, but we stop at this point and make our conclusions based on these three tournaments due to time constraints.

We see that increasing the number of training games will allow the more complex architectures to perform better as we would expect. However, it seems that making the architecture too complex can also result in overfitting, which suggests that a balance is needed between architecture complexity and number of training games. If evaluation and training time did not play a role, we would choose one of the more complex architectures for the rest of our experiments. However, when we incorporate search methods in section 7.7, the evaluation time of our neural network is going to be important. This is due to the fact that the time spent evaluating the network cuts away time that we could have spent on searching through possible moves. Furthermore, it is significant whether training a network takes a couple of hours or several days. Because of this, we chose to continue using the shallow network with

100 hidden neurons for further experiments as it struck a good balance between performance and evaluation time. It also performed best after 100,000 training games, which is the amount of games we have been using thus far.

7.4 Different Parameter Decay Factors

In section 7.1 and section 7.2, we found 0.1 to be a good value for the learning rate and the exploration rate. These rates were kept static over the 10 epochs that we trained the agents, but Sutton and Barto (1998) note that these rates should decrease over time for the value function to converge. In the beginning, we want to explore to get reasonable parameters because the value function is initialized with random weights and biases. However, the amount of exploration should decrease as the agent gets a better understanding of the game. To decrease the rates over time, we introduce new hyperparameters that decrease the value of the learning rate and the exploration rate by multiplying the hyperparameter to the rates after some time. We call these new hyperparameters *decay factors*. For instance, when we apply a decay factor of 0.9 twice to the initial value of 0.1, we get $0.1 \cdot 0.9^2 = 0.081$.

We can apply the decay factors in two ways: We can apply the decay factors after every epoch or we can integrate a metric for stagnation such that we can apply the decay factors every time the performance declines. We will apply the decay factor after every epoch since we need many epochs for a metric for stagnation to work. Besides, stagnation requires a heuristic for when it occurs and how often it should happen before applying the decay factor. Finding such a heuristic is out of the scope of this thesis.

We start by testing both optimal rates with different decay factors such that the settings are

- Initial exploration rates: 0.1
- Initial learning rates: 0.1
- Exploration-rate decay factors: 0.8, 0.85, 0.9, 0.95, and 1
- Learning-rate decay factors: 0.8, 0.85, 0.9, 0.95, and 1

We choose 0.8 as the smallest decay factor since this gives rates of $0.1 \cdot 0.8^{10} = 0.011$ after 10 epochs, i.e., a lower decay factor will presumably drop the rates too far below the values found in section 7.1 and section 7.2 to be any good.

Initial value		Decay factor		
Exploration rate	Learning rate	Exploration rate	Learning rate	Benchmark win rate
0.1	0.1	1.00	0.80	81.8%
0.1	0.1	0.90	0.85	81.7%
0.1	0.1	0.90	0.90	81.1%
0.1	0.1	0.95	0.90	80.8%
0.1	0.1	1.00	1.00	79.9%

Table 7.7: Results from the best static initial learning and exploration rates combined with decay factors. The benchmark win rate shows the percentage of games won against minimax in the 1,000 benchmark games after the final epoch of training.

Table 7.7 shows the results of the five best performing settings. Introducing decay factors gives an increased performance, but it is better to decrease the learning rate than the exploration rate. Since the lowest decay factor was better for the learning rate, we tried the lower factor 0.7 while increasing the initial value to 0.2. Since some decay is good for the exploration rate, we also tried to raise the initial exploration rate to 0.2. To keep the amount of combinations that we have to test low, we only test

- Initial exploration rates: 0.1 and 0.2
- Initial learning rates: 0.1 and 0.2
- Exploration-rate decay factors: 0.8, 0.9, and 1
- Learning-rate decay factors: 0.7, 0.8, and 0.9

Table 7.8 shows the results of the follow-up experiment, again only showing the top 5 best performing settings. A higher learning rate and lower decay factor were consistently better. The exploration rate is hesitant about the initial value and about its decay factor but with a preference towards 0.8. Because the learning rate prefers higher initial value and lower decay factor, we try another experiment with the following settings.

- Initial exploration rates: 0.2 and 0.3
- Initial learning rates: 0.2 and 0.3
- Exploration-rate decay factors: 0.6, 0.7, and 0.8
- Learning-rate decay factors: 0.5, 0.6, and 0.7

Initial value		Decay factor		
Exploration rate	Learning rate	Exploration rate	Learning rate	Benchmark win rate
0.2	0.2	0.80	0.70	83.8%
0.1	0.2	0.80	0.70	83.6%
0.2	0.2	0.90	0.80	83.3%
0.2	0.2	0.80	0.80	82.4%
0.1	0.2	0.90	0.70	82.2%

Table 7.8: Results from changing the initial learning and exploration rates, and the decay factors. The benchmark win rate shows the percentage of games won against minimax in the 1,000 benchmark games after the final epoch of training.

Initial value		Decay factor		
Exploration rate	Learning rate	Exploration rate	Learning rate	Benchmark win rate
0.3	0.3	0.80	0.60	84.7%
0.3	0.2	0.80	0.70	84.7%
0.2	0.2	0.60	0.50	84.1%
0.3	0.3	0.60	0.50	83.4%
0.3	0.3	0.80	0.70	83.4%

Table 7.9: Results from further changing the initial learning and exploration rates, and the decay factors. The benchmark win rate shows the percentage of games won against minimax in the 1,000 benchmark games after the final epoch of training.

We remove some combinations to keep the amount of combinations that we have to test low.

Table 7.9 shows the results of the second follow-up experiment, again only showing the top 5 best performing settings. For both the learning and exploration rate, an initial value of 0.3 and a higher decay factor, or an initial value of 0.2 and a lower decay factor seem to be the best contenders. Since neither of them are in the extremes of our test data, we will not continue searching for better values.

We have two settings that perform equally well so we choose the first settings from table 7.9. This gives us the hyperparameters seen in table 7.10.

Note that the decay factors have been found for 10 epochs so the learning and exploration rates might become too small if the number of epochs is increased or decreased.

Hyperparameter	Value
Discount rate	1
Decay rate	0
Initial pieces	10
Hidden-layer architecture	100
Initial exploration rate	0.3
Initial learning rate	0.3
Exploration-rate decay factor	0.8
Learning-rate decay factor	0.6

Table 7.10: Final hyperparameters determined by our experiments.

7.5 Different Board Encodings

In this experiment, we want to compare the three board encodings described in section 6.4. We will give some concrete examples of bad moves that were made by the agent when it was trained with the naive or the extended board encoding. Finally, we compare the three board encodings by training one network with each encoding and playing a round-robin tournament between the resulting agents.

The described examples use agents that play greedily with regard to their value function. A network trained using the naive board encoding will sometimes miss immediate threats made by its opponent. Take for example the game state shown in figure 7.1. This example is taken from a game played against an agent with the naive board encoding after 100,000 self-play games. The agent is playing as ⓞ and the move *f1* has just been made by ⓧ. The only viable move for ⓞ is *g1* in order block the immediate threat made by ⓧ. Instead of blocking the threat, the move made by our agent is *c2*. This results in our agent losing the game as ⓧ makes the move *g1* and completes a sequence. Situations such as the one described above were encountered several times during play, resulting in early losses for the agent. One could argue that maybe more training would allow the agent to not make such mistakes, but the agent was able to avoid these simple mistakes more reliably by switching to the extended board encoding.

Using the extended board encoding, our agent more reliably blocks immediate threats, though they are still missed on occasion. However, another problem encountered with both encodings is the inability to determine that a move opens up a winning move for the opponent. As an example, consider the situation depicted in figure 7.2. Here, the agent is playing ⓧ and

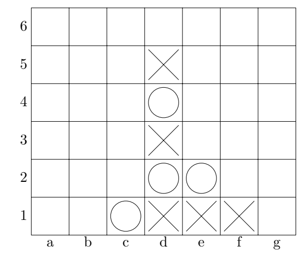


Figure 7.1: The agent is playing ⓞ. The agent's next move is *c2* instead of blocking the threat at *g1*.

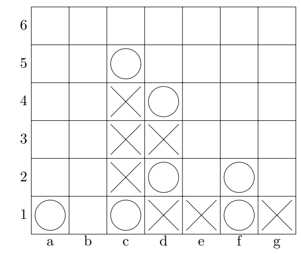


Figure 7.2: The agent is playing ⓧ. The agent's next move is *e2*, which allows ⓞ to win by playing *e3*.

is about to make a move. The agent plays *e2*, even though ⓞ then wins by playing *e3*. While more training improves the agent's ability to block threats, the agent still makes moves that lead to immediate losses.

The feature board encoding attempts to address both of these problems by including threats as a handcrafted feature. In order to compare the three encoding methods, networks were trained using each encoding and subsequently played against each other in a round-robin tournament. Once again, we train three networks for each setup and choose the one with the highest benchmark as representative for the tournament. The results of these matches can be seen in table 7.11.

	Feature	Extended	Naive
Naive	24.9%	46.1%	-
Extended	28.6%	-	
Feature	-		

Table 7.11: Results of the round-robin tournament between networks with different board encodings.

	Feature	Extended	Naive
Naive	25.5%	40.2%	-
Extended	36.1%	-	
Feature	-		

Table 7.12: Results of the second round-robin tournament between networks with different board encodings.

The feature board encoding is a clear winner of the tournament, handily beating both the naive and the extended encoding. The results also show that the extended board encoding is not a huge improvement over the naive board encoding. One thing to keep in mind for this experiment is that the simpler board encodings have far fewer weights to tune. The naive board encoding has 4,400 weights, the extended board encoding has 13,000, and the feature board encoding has 21,400. Reducing the number of training games should benefit the naive board encoding, while increasing the number of training games should benefit the feature board encoding. To verify whether this is the case, we run an additional tournament with 10,000 training games.

Table 7.12 shows the results of this follow-up tournament. The extended board encoding performs better against both the feature board encoding and the naive board encoding. In order for the naive board encoding to win against the others, we had to decrease the number of training games all they way down to 1,000 in total. With such a low number of training games, the random initialization of each network plays a huge role because the parameters are barely adjusted. This means that the results vary depending on how each network is initialized, making the results unreliable. What we have to conclude is that the feature board encoding simply outperforms both alternatives even with a low number of training games. We continue to use this board encoding for the remaining experiments.

7.6 Different Sides

In this experiment, we want to figure out how agents are affected by always playing first or alternating between playing first and second. We train two agents: player A always plays first and player B alternates between playing first and second. We will compare these agents based on how well they play when they (1) always play first, (2) always play second, and (3) alternate between playing first and second.

Starting player	Win rate of player A	Benchmark win rate
Player A	58.5%	85.8%
Player B	38.7%	86.2%
Alternating	48.4%	-

Table 7.13: Results of training the agents on different sides. Starting player indicates the starting player for the tournament games between player A and player B.

Table 7.13 shows the results from comparing player A and player B. We see that playing first gives an advantage and that player B is better at playing first with 2.8%. We see that player B plays 1.6% better than player A when competing in an alternation tournament, and that player B benchmarks marginally better with 0.4% more. A reason for this could be that an agent trained on both sides is able to generalize better, as it has explored a more diverse state space. The difference in benchmark win rates is not significant, but the results of the tournament games indicate that it is beneficial to train an agent on both sides.

7.7 Different Search Methods

We want to compare the two search methods minimax with alpha-beta pruning and MCTS. Chapter 5 described both search methods. As noted in section 6.7, we can integrate the search methods into our agent by using the neural network as a heuristic while searching. For minimax, the neural network is used to estimate the value of non-terminal states. For MCTS, the neural network is used to initialize the estimated value of each node. Two agents play against each other using the same network but with different search methods. Since the MCTS search method introduces non-determinism, we can make both agents greedy with regard to the moves found by their search.

In order to compare the two search methods, we need to give each agent an equivalent search budget. In the minimax algorithm, the search budget can be adjusted by changing the search depth. In the MCTS algorithm, the search budget can be adjusted by terminating the search after any amount of time. Thus, we can adjust the search time of MCTS such that it approximately matches the time spent in the minimax algorithm. For a specific search depth of minimax, we measure the average time spent making a move and set the search time of MCTS to this value.

Search budget		Wins		
Minimax	MCTS	Minimax	MCTS	Draws
3	4 *ms*	6718	2511	771
5	30 *ms*	5703	3345	952
7	230 *ms*	3845	5324	831

Table 7.14: Results of comparing an agent using minimax search with an agent using MCTS. Search budget is the search depth for minimax and search time for MCTS.

We try minimax with search depths of 3, 5, and 7. These searches take on average 4 *ms*, 30 *ms*, and 230 *ms* to find a move, respectively. The results of this comparison can be seen in table 7.14. Since these agents now use search methods, it is more common for the games to last until the board is full or almost full. As a result, these games result in draws more frequently than any of the other experiments. Because of this, we list the number of games won instead of a win percentage. These results indicate that with a low search budget, minimax performs better than MCTS. However, as the search budget is increased, MCTS performs better. This is likely due to the low accuracy of the random simulations in MCTS. An increased search budget leads to more simulations, which improve the accuracy of the estimations. This increased accuracy is more beneficial than the increase in search depth that a higher search budget gives to minimax.

7.8 Different Board Sizes

We observe that MCTS with a search budget of 10 *ms* outperforms the agent on the standard 6×7 board. Therefore, we want to see whether MCTS keeps outperforming the agent when we change the board size. We increase only the width since this directly increases the branching factor. Table 7.15 shows that the agent only wins 10.2% of the matches against MCTS on a standard board. But they are equal on a board of size 6×25, and the agent wins 97.5% of the matches when the board size is changed to 6×50. However, the increased board size increases the size of the input layer of the agent's neural network. This increases the number of weights, which makes backpropagation and evaluation of the network slower.

Changing the size of the board means that the state space will be bigger so the hyperparameters of the agent should be reconfigured. In particular, the learning rate and exploration rate should be reconfigured because the

Board size	Win rate
6×7	10.2%
6×25	47.8%
6×50	97.5%

Table 7.15: Results of benchmarking an agent against MCTS with a search budget of 10 *ms* on boards with different sizes.

agent has to explore more of the state space to have an understanding of it. Furthermore, we should expect the agent to require more training games in order to reach peak performance. This is because of the increased number of inputs and the larger state space.

7.9 Using What We Learned

In the previous sections, we have optimized the reinforcement-learning agent's hyperparameters, policy, and board encoding. In this section, we are going to test the effects of training an agent for 100 epochs, i.e., for 1,000,000 training games. The biggest concern with increasing the number of epochs is that the decay factors that we found in section 7.4 are optimized for 10 epochs. This means that the learning rate and exploration rate will become smaller than anything we tested in that experiment. However, we are still training with 10 initial pieces, which should help prevent stagnation or overfitting even with a low exploration rate as we saw in section 7.2.

After training, we benchmark the agent against minimax with a search depth of five and MCTS with 10 *ms* search time. Against minimax, the agent achieves a 86.5% win rate, which is only a 0.3% increase compared to the highest benchmark seen with 100,000 training games. This increase is not significant, so we conclude that the additional training has not helped the agent against minimax. Against MCTS, the benchmark is increased from 10.2% to 21.0%, which is a significant increase in performance. One explanation for the increased performance is that the agent is able to more consistently block threats and not make moves that open up winning moves for the opponent through more training. Since the minimax opponent only attempts to win by making moves that are guaranteed to work, it does not set up threats that could be easily blocked. MCTS sets up threats more frequently, as it makes any move that looks promising after its simulations.

Finally, we examine the effect of further increasing the number of epochs to 200. The increased number of training games does not improve the agent's benchmark against minimax as it now scores 85.4%, which is a 1.1% decrease compared to last time. The agent's benchmark against MCTS is now 17.1%, which is a 3.9% decrease compared to last time. This suggests that the agent is overfitting during training because the exploration rate and learning rate have decayed too much. Overfitting seems plausible because the low learning rate allows the agent to fine-tune its value function. Even though exploratory starts help to reduce overfitting, they evidently do not prevent it. Also, exploratory starts only randomize the opening of the game, but the low exploration rate means that the remainder of the game is played out by the agent with little noise injection.

Chapter 8

Conclusion

In this thesis, we combined a reinforcement-learning agent with a neural network and a search method to determine the agent's ability to play Connect Four. Allis (1988) weakly solved the game with a knowledge-based approached that required getting insight into the game and strategies, and Tromp (2015) strongly solved the game by brute force. We have presented a reinforcement-learning approach that requires an implementation of the game but no strategic knowledge.

We looked at agents trained by self-play with the reinforcement-learning algorithm TD(λ) using neural networks as value functions. We determined hyperparameters for TD(λ), the agent's policy, and the neural network. It is worth noting that hyperparameter optimization is dominated by guidelines and rules of thumb. This means that finding good hyperparameters becomes a time-consuming process of trial and error. Table 8.1 shows the best hyperparameters that we found through our experiments.

Hyperparameter	Value
Discount rate	1
Decay rate	0
Initial pieces	10
Hidden-layer architecture	100
Initial exploration rate	0.3
Initial learning rate	0.3
Exploration-rate decay factor	0.8
Learning-rate decay factor	0.6

Table 8.1: Final hyperparameters determined by our experiments.

We compared different encoding methods for transforming board positions into neural network vector inputs. The feature board encoder performed far better than the simpler alternatives. The feature board encoder used five bits per square on the board and two additional bits: three bits per square to indicate if a player has a piece on that square, two bits per square to indicate if a player has a threat on that square, and two bits to indicate whose turn it is to make a move. This result was significant because it told us that incorporating knowledge about the game into the agent can help tremendously. Our motivation for using reinforcement learning was primarily that we wanted to play Connect Four without having insight in the game. But our experience with the feature board encoder suggests that knowledge about the game still plays an important role.

Even though TD(λ) can use eligibility traces to update previously visited states, it was found better to disable these traces and only make one-step updates. As noted by Sutton and Barto (1998), this is often the case for applications where episodes can be generated cheaply, e.g., through simulations. However, it is not always the case that disabling eligibility traces is superior for reinforcement-learning approaches to game playing. Schaeffer et al. (2001) showed an example where they used $\lambda = 0.95$, i.e., previous states were highly affected by updates made to future states.

We looked at using the search methods minimax and Monte-Carlo Tree Search (MCTS) with a reinforcement-learning agent. With a low search budget, minimax performed better than MCTS, but MCTS performed better when the agent was allowed to use about $200\,ms$ or more per move. In general, we found that while Connect Four has a large state space, it is not large enough for reinforcement learning to be necessary compared to MCTS. A likely explanation is that Connect Four has a low branching factor with at most seven possible moves each turn. However, when we increase the number of columns in the Connect Four board, reinforcement learning performs better than MCTS. The search time of MCTS can be increased to make it stronger, but we believe that combining reinforcement learning with MCTS becomes superior when the task is sufficiently complex. This was also the approach by Silver et al. (2016) when building AlphaGo, which was introduced in chapter 2.

We experimented with different neural-network architectures by varying the number of hidden layers and the number of neurons in each hidden layer. In general, we found that shallow networks, i.e., networks with one hidden layer, performed better than deeper networks with the same amount of training. For the shallow networks, increasing the number of hidden neurons also increased the number of training games required to reach peak performance. With sufficient training, we found that the shallow networks with more neurons also performed better. However, the increase in evaluation time for the more complex networks was not worth their improved accuracy when we incorporated search methods. This is due to the fact that the time spent evaluating the network cuts away time that we could have spent on searching through possible moves.

We tested the effect of training an agent when it only plays first or when it alternates between playing first and second. We observed that playing both sides during training had a positive impact on the agent's performance. This increased performance was seen both when the agent played only on one side and when the agent alternated between playing both sides. We hypothesized that this is due to the increased diversity of the state space explored by an agent alternating between both sides.

An observation made by several authors is that self-play has a tendency to stagnate in deterministic games (Sutton and Barto, 1998; Schraudolph et al., 1994; Tesauro, 1995). To avoid stagnation, noise injection is required to keep learners from getting stuck in a small part of the state space. We experimented with noise injection in two ways: through the policy of the agent and through exploratory starts. We found that policies with a high exploration rate were sufficient to keep the learning process from stagnating. However, as one wants the exploration rate to decrease over time, exploratory starts are beneficial to keep learning from stagnating when the exploration rate becomes low. We also found that combining a low exploration rate with exploratory starts generally performed better than using a high exploration rate.

Chapter 9

Future Work

In the introduction, we mentioned the three machine-learning paradigms: supervised learning, unsupervised learning, and reinforcement learning. We exclusively used reinforcement learning and self-play training in this thesis. An alternative approach is to use supervised learning to learn from sample games. A combination of the two methods allows the agent to learn from sample games and then train against itself. This avoids that the agent has to "bootstrap itself out of ignorance" (Schraudolph et al., 1994).

Our current approach uses a neural network as a value function that outputs the estimated chance of winning from a board position. An alternative approach could be to have seven outputs from the neural network, one for each possible move. The network would then be trained to output a score for each move, where the output corresponds to the expected win rate for that move. In this alternative approach, we evaluate the network only once when selecting moves, possibly making it a viable substitute for the random simulations in MCTS. The concern is that making the network approximate seven values makes it less precise.

As mentioned in chapter 4, there are several types of neural networks. We only experimented with feed-forward networks that are fully connected. Alternatively, we could train recurrent networks or convolutional networks to be value functions. In particular, deep convolutional networks have recently been used to great success by Silver et al. (2016) in AlphaGo.

One of the motivations for using reinforcement learning was that we could avoid having to learn the strategies of Connect Four. However, the experiments showed the benefits of the feature board encoding. Therefore, it would be natural to encode some of the strategic rules from Allis (1988)'s knowledge-based approach into the network's input.

In section 7.8, we experimented with a variation of Connect Four with a larger board. The methods that we have used in this thesis can be applied to any game as long as the game can be formalized as a finite MDP and simulated efficiently. Therefore, we could train an agent to play an entirely different game.

Bibliography

Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from Data*. AMLBook, 2nd edition, 2012.

Victor Allis. A Knowledge-Based Approach to Connect-Four. The Game is Solved: White Wins. Master's thesis, Vrije Universiteit, 1988.

Yoshua Bengio. Practical Recommendations for Gradient-based Training of Deep Architectures. *Computing Research Repository*, abs/1206.5533: 1–33, 2012.

James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

Cyningstan. Captain's mistress.
http://www.cyningstan.com/game/271/captains-mistress, 2016.

Peter Drake and Steve Uurtamo. Move Ordering vs Heavy Playouts: Where Should Heuristics be Applied in Monte Carlo Go. *Proceedings of the 3rd North American Game-On Conference*, pages 35–42, 2007.

Stefan Edelkamp and Peter Kissmann. Symbolic Classification of General Two-Player Games. In *KI 2008: Advances in Artificial Intelligence*, pages 185–192, 2008.

Sylvain Gelly and David Silver. Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A Novel Connectionist System for Unconstrained Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 855–868, 2009.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2:359–366, 1989.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, draft edition, 2015.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2nd edition, 2003.

Jonathan Schaeffer, Markian Hlynka, and Vili Jussila. Temporal Difference Learning Applied to a High-Performance Game-Playing Program. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, pages 529–534, 2001.

Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal Difference Learning of Position Evaluation in the Game of Go. In *Advances in Neural Information Processing Systems 6*, pages 817–824, 1994.

K. G. Sheela and Subramaniam N. Deepa. Review on Methods to Fix Number of Hidden Neurons in Neural Networks. *Mathematical Problems in Engineering*, 2013:1–11, 2013.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1st edition, 1998.

Gerald Tesauro. Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

John Tromp. John's Connect Four Playground. http://tromp.github.io/c4/c4.html, 2015.

Appendix

Backpropagation uses the four equations

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{9.1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{9.2}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{9.3}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{9.4}$$

We proved equation (9.2) in section 4.4.3. We prove the other three equations here.

Remember that the definition of the error δ_j^l is

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}, \tag{9.5}$$

the definition of the activation a_j^l is

$$a_j^l = \sigma(z_j^l), \tag{9.6}$$

the definition of the weighted input z_j^l is

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l, \tag{9.7}$$

and the definition of the chain rule for higher dimensions is

$$\frac{\partial y}{\partial x} = \sum_i \frac{\partial y}{\partial z_i} \frac{\partial z_i}{\partial x}. \tag{9.8}$$

Equation (9.1) To prove equation (9.1), we look at its component form.

$$\begin{aligned}
\delta_j^L &\overset{\text{equation (9.5)}}{=} \frac{\partial C}{\partial z_j^L} \\
&\overset{\text{equation (9.8)}}{=} \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \\
&\overset{\text{equation (9.6)}}{=} \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial \sigma(z_k^L)}{\partial z_j^L} \\
&\overset{\text{unfolding sum}}{=} \frac{\partial C}{\partial a_0^L} \frac{\partial \sigma(z_0^L)}{\partial z_j^L} + \cdots + \frac{\partial C}{\partial a_j^L} \frac{\partial \sigma(z_j^L)}{\partial z_j^L} + \cdots + \frac{\partial C}{\partial a_n^L} \frac{\partial \sigma(z_n^L)}{\partial z_j^L} \\
&\overset{\text{differentiation}}{=} \frac{\partial C}{\partial a_j^L} \frac{\partial \sigma(z_j^L)}{\partial z_j^L} \\
&\overset{\text{differentiation}}{=} \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)
\end{aligned}$$

Equation (9.3)

$$\begin{aligned}
\frac{\partial C}{\partial b_j^l} &\overset{\text{equation (9.8)}}{=} \sum_k \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial b_j^l} \\
&\overset{\text{unfolding sum}}{=} \frac{\partial C}{\partial z_0^l} \frac{\partial z_0^l}{\partial b_j^l} + \cdots + \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} + \cdots + \frac{\partial C}{\partial z_n^l} \frac{\partial z_n^l}{\partial b_j^l} \\
&\overset{\text{differentiation}}{=} \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} \\
&\overset{\text{equation (9.7)}}{=} \frac{\partial C}{\partial z_j^l} \frac{\partial \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right)}{\partial b_j^l} \\
&\overset{\text{differentiation}}{=} \frac{\partial C}{\partial z_j^l} \\
&\overset{\text{equation (9.5)}}{=} \delta_j^l
\end{aligned}$$

Equation (9.4)

$$
\begin{aligned}
\frac{\partial C}{\partial w^l_{jk}} &\overset{\text{equation (9.8)}}{=} \sum_i \frac{\partial C}{\partial z^l_i} \frac{\partial z^l_i}{\partial w^l_{jk}} \\
&\overset{\text{unfolding sum}}{=} \left(\frac{\partial C}{\partial z^l_0} \frac{\partial z^l_0}{\partial w^l_{jk}} + \cdots + \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}} + \cdots + \frac{\partial C}{\partial z^l_n} \frac{\partial z^l_n}{\partial w^l_{jk}} \right) \\
&\overset{\text{differentiation}}{=} \frac{\partial C}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}} \\
&\overset{\text{equation (9.5)}}{=} \delta^l_j \frac{\partial z^l_j}{\partial w^l_{jk}} \\
&\overset{\text{equation (9.7)}}{=} \delta^l_j \frac{\partial \left(\sum_i w^l_{ji} a^{l-1}_i + b^l_j \right)}{\partial w^l_{jk}} \\
&\overset{\text{unfolding sum}}{=} \delta^l_j \frac{\partial \left(w^l_{j0} a^{l-1}_0 + \cdots + w^l_{jk} a^{l-1}_k + \cdots + w^l_{jm} a^{l-1}_m + b^l_j \right)}{\partial w^l_{jk}} \\
&\overset{\text{differentiation}}{=} \delta^l_j a^{l-1}_k
\end{aligned}
$$