Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Henrik Bruåsdal

Deep Reinforcement Learning Using Monte-Carlo Tree Search for Hex and Othello

Master's thesis in Computer science
Supervisor: Keith Downing
January 2020

NTNU
Norwegian University of Science and Technology

Henrik Bruåsdal

Deep Reinforcement Learning Using Monte-Carlo Tree Search for Hex and Othello

Master's thesis in Computer science
Supervisor: Keith Downing
January 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

When Deepmind's AlphaGo computer program beat the human professional Go player Fan Hui in 2015, it was a major breakthrough in AI game playing. Go had proved resilient to techniques that had long since beaten humans in games like chess. Through a novel combination of deep neural networks, reinforcement learning and Monte Carlo tree search, Go was finally mastered. Soon after came AlphaGo Zero, which accomplished even better results while learning completely from self-play, and AlphaZero, which generalized it to other games.

This work contains a thorough description of these systems and the work in the field which led up to them. It details my own implementation of this approach as applied to the games Hex and Othello. Using this implementation, the role rollouts play in the algorithm has been investigated. These were a core part of earlier work in the field and still used in AlphaGo, but then absent from AlphaGo Zero and AlphaZero. Several experiments have been conducted to gain empirical data on whether rollouts can still be a beneficial part of this novel combination of techniques, and how these rollouts should be performed.

Though there were some indications in the data that rollouts provide little or no benefit, the results were ultimately mostly inconclusive. Some weaknesses in the setup have been identified and some new questions have been raised. But the work has resulted in a functional system that could be used to further investigate the issue and produce more conclusive data or insight into new questions.

Preface

This work is the result of a master's program at the Department of Computer Science at the Norwegian University of Science and Technology. It was supervised by Keith Downing and I would like to thank him for his assistance. I would also like to thank IDI for providing the computational resources required to run my experiments. Finally I must thank my mother, without whose encouragement this document might never have seen the light of day.

Henrik Bruåsdal
Trondheim, January 26, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Section 1.1 shows the background and motivating forces for the research by providing a brief history of some game-playing AIs. The specific goals of the research and how it will be conducted is shown in section 1.2. Section 1.3 gives a brief overview of the document structure.

1.1 Background and Motivation

Game playing has long been a focal point for artificial intelligence research and with time, artificial intelligences (AIs) have surpassed human players in many games. Perhaps the most famous example of this is IBM's Deep Blue beating the then world champion Garry Kasparov at chess in 1997. It used a technique which has proved successful in a number of games: alpha-beta search along with an evaluation function hand-crafted by domain experts [Campbell et al., 2002]. Attempts to apply the same technique to the game of Go proved unsuccessful however, due to its huge search space and the difficulty of crafting a good evaluation function [Gelly et al., 2012]. The first breakthrough came with the introduction of the Monte Carlo tree search algorithm in 2006, based on guiding a search by playing many full simulations until the end of the game, so-called rollouts [Coulom, 2006]. With this, the level of human professional players was reached for a smaller-scale version of Go by 2008 [Gelly and Silver, 2008], with further improvements in the following years; however, a similar success at full-scale Go seemed out of reach. At the same time there was another revolution in the field of AI with the resurgence of deep learning [Krizhevsky et al., 2012]. In 2015, a novel combination of Monte Carlo tree search and deep neural networks was used in Alpha Go, with training based on both supervised learning and reinforcement learning. This was tremendously successful and finally allowed

a computer system to best a human professional Go player [Silver et al., 2016]. Soon after came AlphaGo Zero, now relying on a simpler algorithm that learns the game from scratch entirely through self-play, with even more impressive results [Silver et al., 2017]. Finally, there was AlphaZero, which generalized the technique to the games of chess and shogi, beating the existing state-of-the-art alpha-beta systems and indicating that this can be a feasible framework for general game solving [Silver et al., 2018]. My thesis will take a closer look at these systems.

1.2 Goals and Research Questions

Goal Investigate the use of a flexible combination of deep neural networks, reinforcement learning and Monte Carlo tree search for the games Hex and Othello.

The basis for this combination will be the AlphaGo, AlphaGo Zero and AlphaZero systems. I will follow an experimental research methodology, designing and implementing a system with inspiration from these, along with a set of experiments applying variations of this system to Hex and Othello. These should produce empirical results which enable me to analyze and discuss the merits of the general technique and variations of it. These games are chosen as benchmarks partially because they are relatively simple games that allow for easy and performant implementations. There is also prior art on using standard Monte Carlo tree search for these games, as seen in section 3.3. An explicit goal of the system is flexibility, configurability and modularity, enabling me to easily create a wide range of variations and experiments. Expanding the system to include other games should be as simple as possible and not require any modification of the existing system, only a new implementation for the game logic. The system should not require any knowledge of a game outside of the rules, enabling it to learn completely from scratch.

Research question 1 Is it beneficial to use both value network evaluation and rollout evaluation in Monte Carlo tree search?

While rollouts were used in AlphaGo, AlphaGo Zero and AlphaZero dropped them entirely. The advantage of this is faster computation and no need for a rollout policy, but it makes value estimates less accurate, especially in the early parts of a game. I would like to investigate whether rollouts can provide a performance benefit in my system, both when performed all the time as in AlphaGo or when only performed a portion of the time.

Research question 2 Is it preferable to do few rollouts with a policy network or many with a simpler, faster policy?

The rollout policy used in AlphaGo was relatively simple and fast compared to its policy network. This allowed for performing many rollouts. An interesting question is whether it could be beneficial to instead use a more sophisticated but slower rollout policy and thus perform fewer but more accurate rollouts. If given the same time per move, which of these approaches produce better players? I would like to investigate this question, using the policy network which is trained by the system for my rollout policy.

1.3 Thesis Structure

Chapter 2 will provide the reader with relevant background theory for the related work and my own research.
Chapter 3 will expand on section 1.1 to further introduce the reader to the relevant research through a literature review.
Chapter 4 will explain the system built as a part of this work in detail and provide justification for choices made.
Chapter 5 shows how the system is used to answer the research questions by providing the experimental design, parameters and results.
Chapter 6 contains an evaluation of the results, a discussion of the work as a whole and potential future work.

Chapter 2

Background Theory

This chapter provides the background theory necessary for the reader to understand the fields the research touches upon. As some understanding of the games used in this work can be useful, this section will first familiarize the reader with Hex and Othello. It will go on to explain the Monte Carlo tree search algorithm, convolutional neural networks and reinforcement learning in general, all topics relevant both in my own work and for much of the prior art presented in chapter 3.

2.1 Hex

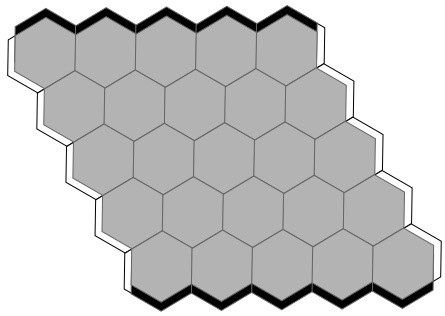
Hex is a two-player zero-sum finite deterministic perfect information strategy game. It is played on a rhombus-shaped board consisting of an NxN grid of hexagons. 11x11 is generally the standard size. The rules are very simple. The players, commonly denoted black and white, take turns placing pieces of their respective colors on unoccupied tiles. The objective for each player is to produce a connected path of pieces in their color between two opposite edges of the board, commonly the top and bottom for black, the left and right for white. The game ends when a player achieves this. [Browne, 2000, p. 1-4]

Figure 2.1 shows a 5x5 board before play has started (a) and after white has won (b). Note the unbroken chain of white pieces between the two whites sides in b.

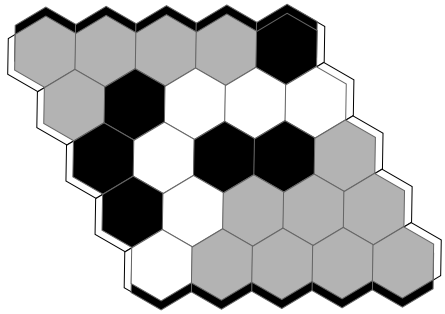
A notable property of Hex is that a game can never end in a draw. A filled board will always contain a connected path for one of the players and not the other [Gale, 1979]. Additionally, Hex is a first-player-win game, meaning that the first player can always guarantee a win with perfect play. This can be shown *reductio ad absurdum* using Nash's strategy-stealing argument and relies on the

fact that an extra piece is never a disadvantage in Hex [Nash, 1952]. To weaken this advantage and produce more fair games, a swap rule is often used: The second player has the option of swapping colors after the first player has made their move, taking that move as their own [Arneson et al., 2010].

Despite its simplicity, the computational complexity of Hex is large, due to its board size and the fact that every open position constitutes a legal move. Browne [2000] gives an upper bound on the number of valid board positions of roughly 2.38×10^{56}. The branching factor is around 100, situated between Go (250) and Chess (40), and far higher than Checkers (2.8) [Allis, 1994].



(a) An empty 5x5 Hex board.



(b) A 5x5 Hex board after 12 moves where white has won by connecting the left and right sides.

Figure 2.1

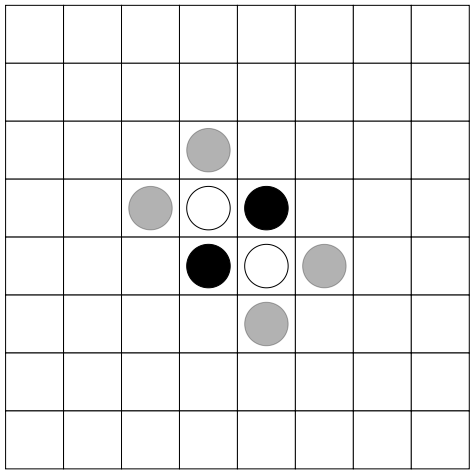
2.2 Othello

Similarly to Hex, Othello is a two-player zero-sum finite deterministic perfect information strategy game. It is also a board game where pieces are placed and then not moved. But the similarities end here. In Othello, the objective is to capture the opponent's pieces and end up with more pieces than them at the end of the game. The game is played on a square board with an even number of tiles in each direction, generally 8x8. Initially there are four pieces on the board, placed in the four middle tiles. White has the north-west and south-east tile, black the north-east and south-west one. This initial configuration can be seen in fig. 2.2a. Black plays the first move. A piece can only be placed on a tile if there is a horizontal, vertical or diagonal line between the new piece and another of that player's pieces, with a contiguous line of the opponent's pieces between them. When the piece is placed, the opponent's pieces between them are captured and take on the color of the current player. Such a capture is shown in fig. 2.2, which showcases black's first move and the allowed moves for both black and white. If there are no valid moves for a player, they are forced to pass and the opponent plays instead. Passing is not legal if there are valid moves. The game ends when both players must pass or the board is full. Whoever has the most pieces at the end of the game wins. Draws are possible and occur when the players have the same number of pieces. [Landau, 1985]

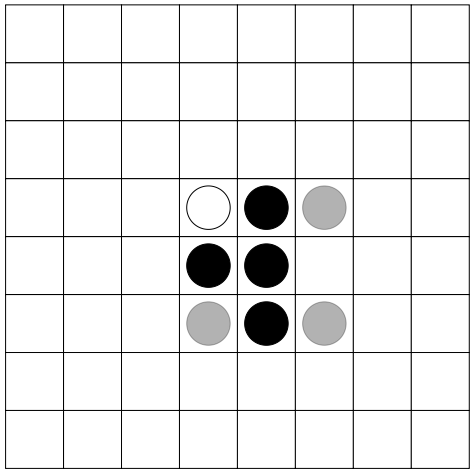
Because the rules for where a piece can be placed are fairly limiting (although also partially because of the smaller board size), 8x8 Othello has a far lower branching factor than 11x11 Hex, estimated at around 10 by Allis [1994]. He gives an upper bound on the number of states of about 10^{28}.

2.3 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a relatively recent tree search algorithm that has proved to be very powerful for problems where the search space is too large for an exhaustive search and too complex to reduce to a manageable size using heuristics. It was introduced as late as 2006 and first applied to the game of Go [Coulom, 2006]. As the name suggests, it is a Monte Carlo method, meaning it relies on repeated sampling for its quantitative results. Specifically, it uses Monte Carlo rollouts, playing out from a state S to a final state F without branching or backup. The value of S can be estimated as the mean value of the outcomes of these simulations. If the problem is deterministic, the playouts generally use a non-deterministic policy to introduce stochasticity. The policy can be simple, for example, uniformly choosing among the valid actions. While many such rollouts might then be needed for a good estimate, the complexity can be far lower than an exhaustive search.



(a) An initial 8x8 Othello board. The potential moves for black are shown in gray.



(b) A 8x8 Othello board after black has played their first move. The potential moves for white are shown in gray.

Figure 2.2

MCTS uses this to guide a search through the state tree by evaluating leaf nodes using Monte Carlo rollouts and backpropagating information from the rollouts. The root node in the tree corresponds to the current state of the game. Its child nodes are the potential next states in the game, with edges representing actions taken. Each node contains a visit count and each edge contains a visit count and a value. The search starts in the root node and proceeds recursively as follows: If the state already exists in the tree, select an action using the *tree policy* (fig. 2.3a). This policy will generally be based on the node statistics. If the node is not fully expanded, expand it by adding at least one of its children to the tree (fig. 2.3b). Then perform a Monte Carlo rollout from it or one of its children using the *default policy* (fig. 2.3c), also known as the rollout policy This policy is generally non-deterministic and not based on node statistics as the nodes visited in the rollouts are usually not added to the tree due to memory limitations. [Gelly and Silver, 2011]

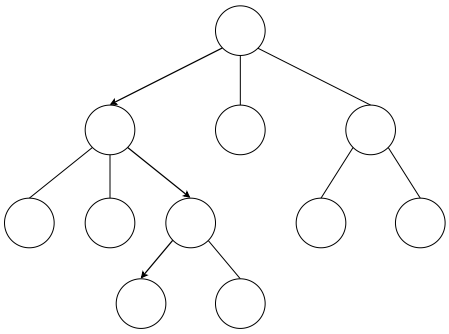
The information from the rollout will then be propagated back up the tree (fig. 2.3d). The visit count of the nodes and edges along the visited path will be incremented and the action values will be updated based on the outcome. Then a new search starts from the root node.

This is repeated a number of times, iteratively refining the value estimates, which will guide the tree policy to more promising regions of the search space. As the number of simulations grow to infinity, the value estimates will approach the optimum. [Kocsis and Szepesvári, 2006]

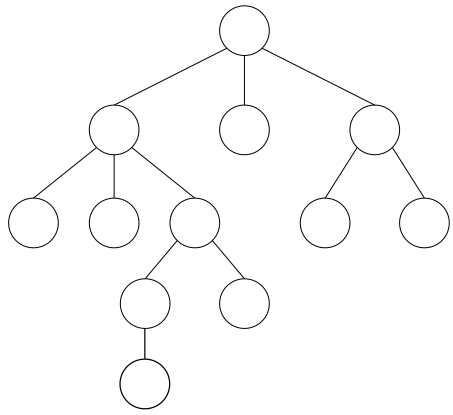
Instead of running simulations all the way to a final state, it is possible to truncate the search at a lower depth and use an evaluation function to estimate the value. This is dependent on being able to construct a function which both gives an accurate estimate that does not increase the estimation bias too much, and is reasonably fast. In that case, it can speed up the search and reduce its variance. [Gelly and Silver, 2011]

After a certain number of simulations, when the estimates are deemed accurate enough or there is no time left for additional searches, an actual move from state S can be made based on the obtained information. This is commonly done by choosing the action with the highest visit count, as it is less prone to outliers than the maximum action value [Enzenberger et al., 2010]. The subtree rooted at the next node can then be retained so that the gathered statistics can be used as initial values in the search for the next move.

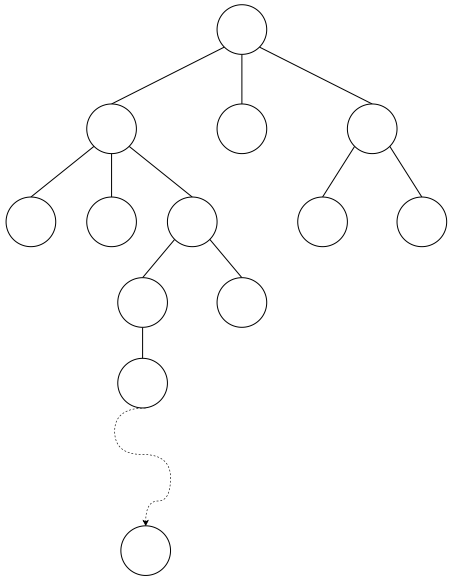
The policies used can have a large effect on the performance of MCTS. A completely greedy tree policy will tend to be too biased and avoid good performing moves if they happen to receive a poor evaluation early in the search when the estimates are still very uncertain. Some element of exploration is necessary to avoid this problem. The *Upper Confidence bounds applied to Trees* algorithm (UCT) does this by increasing the estimated value of actions based on the un-



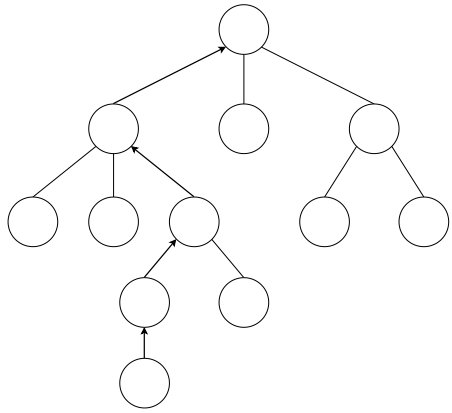
(a) **Selection**: The existing states of the tree are searched until a node that is not fully expanded is reached. The edges (actions) to follow are chosen by the tree policy.



(b) **Expansion**: One or more (possibly all) child states are added to the tree.



(c) **Rollout**: A Monte Carlo rollout is performed using the default policy from a newly added node until the end of the game.



(d) **Backpropagation**: The outcome of the rollout is propagated back through the tree, incrementing the visit count and action values along the path that was searched.

Figure 2.3: The four phases of MCTS.

certainty of their value. Specifically, it adds the exploration bias $c\sqrt{\frac{\log N(s)}{N(s,a)}}$ to the estimated value, where $N(s)$ is the visit count for state s and $N(s,a)$ is the number of times action a has been taken in state s [Kocsis and Szepesvári, 2006]. This makes it more likely to choose rarely taken actions, even those that seem poor if the node is visited enough, since the bonus grows without bound for actions that aren't visited. At the same time, it decreases fast for actions that are taken more often, so their value will mostly be based on the rollouts.

Improving the default policy can also be very beneficial to the performance of MCTS. Incorporating domain knowledge has been shown to be an effective way to do this [Gelly and Silver, 2011].

2.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural network that have gained a large amount of popularity in recent years and have been shown to perform exceptionally well on data with a grid-like structure, such as for image classification [Krizhevsky et al., 2012]. The distinguishing feature of a convolutional network is the replacement of the matrix multiplication used in a regular neural network layer with the convolution operator. That is, the output of a layer in the neural network is calculated by shifting a kernel of weights across the input matrix, taking the weighted sum of the overlapping part of the input. This can be seen in fig. 2.4, where each of the outputs is the sum of a 3x3 square from the input, weighted by the corresponding value in the 3x3 kernel. The input can also be 3-dimensional, in which case the convolution is done separately for each layer and the results summed. The size of the shift is called the *stride* and is 1 in the example. The behavior at the edge is implementation specific, but might, for example, be done by zero padding. The example shown has no padding, leading to a smaller output than input. As in a regular neural network, each layer is generally followed by a non-linear activation function. Krizhevsky et al. [2012] showed that Rectified Linear Units (ReLUs), defined as $f(x) = \max(0, x)$, make a CNN train far faster than with more traditional activation functions such as tanh or sigmoid.

CNNs take advantage of sparse weights, shared parameters and equivariance. The first is accomplished by using a kernel which is smaller than the input. This is useful because a small kernel can still detect semantically meaningful features while requiring far less computation. It allows early layers to focus on simple, localized features like edges in an image, while later layers consider abstract combinations of these from a larger portion of the input. Shared parameters are done by using the same kernel at all locations of the image. Thus, there is no need to learn a separate set of parameters at each location, which greatly reduces the

number of parameters needed. Equivariance means that if a feature in the input is shifted, its representation in the output will shift in a corresponding manner. This is a natural consequence of parameter sharing, as the kernel will detect the same features at any location of the input. [Goodfellow et al., 2016, p. 326-335]

One of the greatest successes of CNNs is their ability to learn representations from raw data, in contrast with earlier methods which required hand-crafted feature extractors created by domain experts. As previously mentioned, the early convolutional layers tend to detect simple features such as edges in certain orientations, making them feature detectors in their own right. Each layer can contain a stack of kernels, each detecting a different feature at a similar abstraction level. Many such layers are composed, each transforming its input into a representation at a higher, more semantically useful abstraction level, highlighting important details and ignoring noise. After being transformed trough many such layers, the final representation will be at a level where it can be used to classify the input, for example. It turns out that the depth of the network has a large impact on how effectively it can do this. However, deeper networks are generally harder to train. He et al. [2016] found that introducing residual connections in the network could mitigate this. These are skip connections where the input to a layer is added directly to the output of itself or a later layer, as seen in fig. 2.5.

2.5 Reinforcement Learning

Consider the problem of an agent interacting with an environment. The environment is in some state s. At each time-step t it can perform an action a, which takes it to some state s' depending on s and a. Transitioning to s' grants it some reward r. This is shown in fig. 2.6. It can be formally stated as a Markov decision process:

S A set of states.

A_s A set of actions in each state.

$P(s'|s,a)$ A transition model giving the probability of going to state s' when performing action a in state s.

$R(s)$ A reward function giving the reward for going to state s.

We define a policy $\pi(s)$ as a function deciding which action a the agent should perform in state s. Reinforcement learning concerns itself with learning an optimal such policy by interacting with the environment and observing the rewards. The optimal policy π^* is the one which maximizes the expected total reward:

$$\pi^* = \arg\max_{\pi} E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right] \tag{2.1}$$

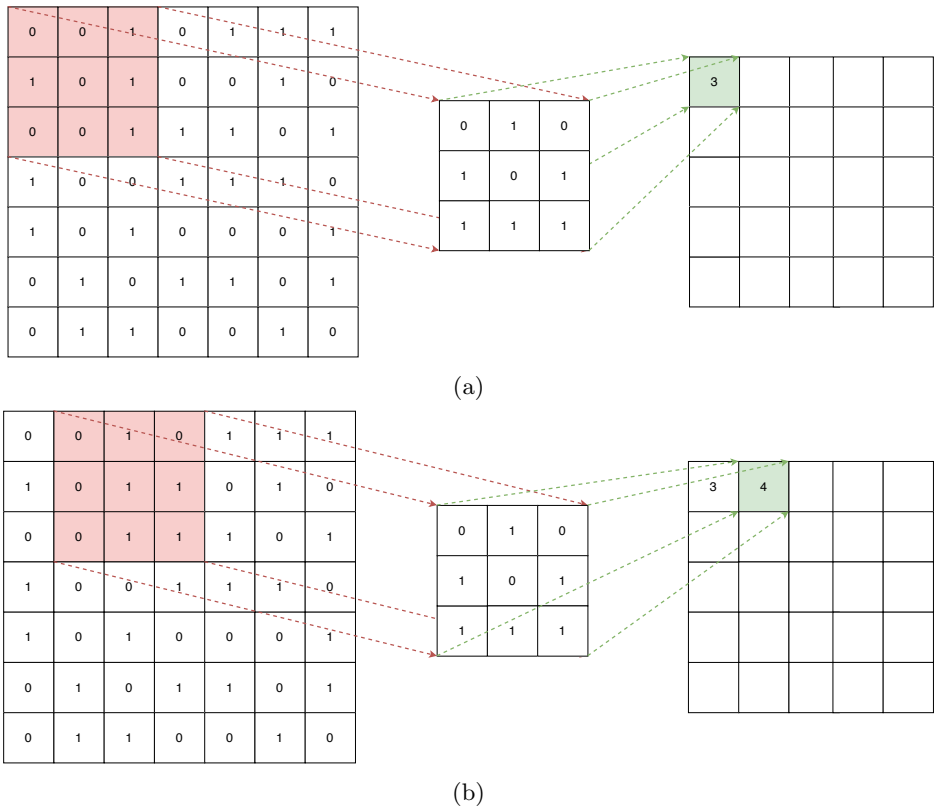


Figure 2.4: Part of a convolution between a 7x7 matrix with a 3x3 kernel with stride 1 and no padding, resulting in a 5x5 matrix. The kernel slides across the matrix and each element in the result is the sum of an element-wise multiplication between the kernel and the underlying 3x3 selection from the matrix.

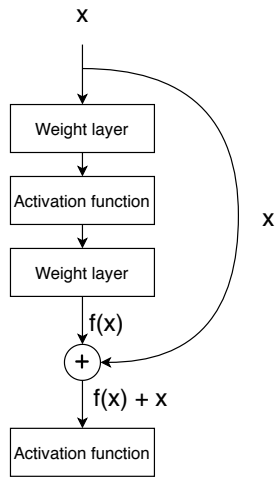


Figure 2.5: A residual block with a connection that skips two weight layers before being added to the pre-activation output of the second layer.

γ is the discount rate, a number between 0 and 1 which makes future rewards less valuable than immediate ones. [Russell and Norvig, 2010, p. 647, 830]

The fact that the policy is learned while gathering rewards leads to the *exploitation-exploration* trade-off. The agent must find an appropriate balance between exploring the environment to gather information about the rewards and exploiting its current knowledge to maximize its reward. A simply policy to ensure exploration is to be ϵ-greedy: Select the option that looks optimal most of the time, but select a random action for a proportion ϵ of the time. A better exploration policy would be one that prioritizes actions that are poorly explored and deprioritizes actions which seem to have a low value. [Russell and Norvig, 2010, p. 839-840]

We can define a state-value function $V^\pi(s)$ and a state-action-value function $Q^\pi(s, a)$ to be, respectively, the expected reward of following policy π from state s and the expected reward of following policy π from state s after performing action a. If Q^{π^*} is known, π^* is simple: Pick the action with the maximum Q-value. Notably this does not require knowledge of P and R, so learning Q is a so-called *model-free* method. For the optimal policy, Q obeys the Bellman equation:

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a') \tag{2.2}$$

Informally, the value of doing a in s is the immediate reward in s and the sum

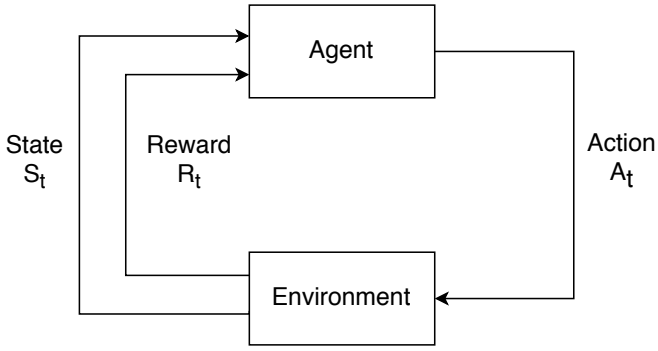


Figure 2.6: An agent affects the environment through actions, producing new states and rewards.

of the values of choosing the best actions in the next states you might end up in, weighted by the chance of actually ending up in them. This equation can be used to learn Q in an iterative fashion. Starting out with an estimate for Q, when action a is performed in state s leading to state s', it can be updated as

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha(R(s) + \gamma \max_{a'} Q_t(s',a') - Q_t(s,a)), \qquad (2.3)$$

where α is the learning rate. This adjusts our estimate closer to the equilibrium that must hold for the correct value function. This method is called Q-learning. If α is decreasing it is guaranteed to converge. Because it assumes that the optimal action is taken in the next state, regardless of what policy is actually used, it is an *off-policy* method. [Russell and Norvig, 2010, p. 833, 843-844]

Real-world reinforcement learning problems tend to have a very large state-action space. This can make a tabular approach where every state-action pair is enumerated and their values estimated separately completely infeasible. Not only is the space required to store the function in such a way very large, but the complete lack of generalization means that there is no information available about state-action pairs that have not been explored. Even if visiting them all is realistic, it is clearly inefficient, as real-world problems tend to have similarities between states that could be exploited. One way to do this is function approximation. The approximation could be a linear function or a non-linear function such as a neural network. In the case of a deep neural network the technique is called *deep reinforcement learning*. Various things can be estimated, such as the state value, the state-action value or a probability distribution representing the optimal policy. In the second case, a neural network could take both the state and action as input and output a single value or take just the state and output a value for each action, or there could be one network per action with one output.

The last of these has the advantage that modifying the network for one action avoids affecting the output for other actions [Lin, 1993]. The first is efficient, as it avoids having to evaluate it multiple times per state [Mnih et al., 2015]. When the function approximation is an estimate for Q, it can be trained using stochastic gradient descent. If the approximation of Q is $\hat{Q}_\theta$, where θ is the function parameters, and we define the error to be the mean squared error between the approximation and the target Q-value as given by the Bellman equation, we get the following parameter update:

$$\Delta\theta_i = \alpha \left[R(s) + \gamma \max_{a'} \hat{Q}_\theta(s,a) - \hat{Q}_\theta(s,a) \right] \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i} \tag{2.4}$$

We can see that this is essentially the same as Q-learning. [Russell and Norvig, 2010, p. 845-847]

One technique used to improve the training of function approximators for reinforcement learning is experience replay [Lin, 1993]. After each choice, the agent's experience, consisting of s, a, $R(s)$ and s', is stored in so-called replay memory. Instead of training on the most recent experience, training examples are sampled from the replay memory, at random or prioritized in some way. This has multiple advantages. An experience can be presented to the algorithm multiple times instead of simply being thrown away, which is especially valuable if the experience is rare or involves negative rewards. Thus, it keeps the agent from forgetting transitions it has not experienced for a while and helps avoid revisiting bad experiences. It is also more efficient to learn from random samples than strongly correlated consecutive samples, and averaging over previous experiences created with different parameters helps avoid oscillations or divergence [Mnih et al., 2015].

Chapter 3

Related work

This chapter provides a thorough review of both earlier work and the state-of-the-art in fields relevant for the project and aims to motivate my own research. It also contains a justification for the choice of literature. The sections on deep reinforcement learning and Monte Carlo tree search both cover multiple research articles. The remaining sections are more in-depth and only cover a single article each, as I regard these as the most important for my own work. The chapter ends with a summary of how the research work motivates my own work.

3.1 Selection of Literature

The main method used for finding literature to review has been snowballing, meaning using a few key articles as a starting point and then recursively consulting their references for other relevant articles to include. This was a natural choice, because there are three clear starting points: the DeepMind articles mentioned previously, Silver et al. [2016], Silver et al. [2017] and Silver et al. [2018]. These are where the specific combination of techniques I am investigating were introduced and they contain a large number of references, so the assumption was that they were likely to directly or indirectly reference most relevant articles.

One of their direct references is Mnih et al. [2015], which, while not utilizing MCTS, introduced an alternative, novel combination of deep learning and reinforcement learning with state-of-the-art results. It's included in my review to increase the breadth. Lundgaard and McKee [2006] was not found through the snowballing process, but is a recommended read in the Artificial Intelligence Programming course. As an earlier attempt at combining neural networks with reinforcement learning to video games, it serves as a good introduction to Mnih et al. [2015].

There were of course many referenced articles relevant to MCTS. Chief among these were Coulom [2006], Kocsis and Szepesvári [2006] and Wang and Gelly [2007], which introduced MCTS itself, UCT and their combined application to Go respectively. As the original articles for these central ideas, they were a natural part of the review. A followup to Wang and Gelly [2007], Gelly and Silver [2007], is reviewed to show how the technique quickly developed further and reached state-of-the-art levels in Go. Maddison et al. [2014] describes a much later development where deep neural networks are used in combination with MCTS. This article concludes the pre-Alpha Go timeline and thus sets the stage for a review of the three main articles. All these articles were directly referenced by those three.

Hingston and Masek [2007] and Arneson et al. [2010] show how MCTS was applied to games other than Go. Because these apply to Othello and Hex respectively, they also serve as a description of early prior art for my games of interest, which is mainly why these articles in particular are used. As the DeepMind articles naturally focused on prior art for Go, these were not found through snowballing, but through searching Google Scholar for "monte carlo tree search othello" and "monte carlo tree search hex". The latter search also produced Anthony et al. [2017], which was included to show an alternative approach inspired by AlphaGo and because it makes use of uniformly random rollouts.

3.2 Deep Reinforcement Learning

Many games are ill-suited for supervised learning, as it is hard to define what a correct move looks like. Many games also provide some kind of score or reward during gameplay, which makes reinforcement learning an obvious choice. However, the traditionally successful reinforcement learning techniques are model-less algorithms that enumerate the entire state or state-action space, which can be intractable in many cases. Combining these techniques with neural networks as a function approximator can provide sufficient generalization to make them applicable. One such attempt was Lundgaard and McKee [2006], which applied it to the game of Tetris. They noted that the raw state space of Tetris is on the order of 7.87×10^{61}, though many of these states are functionally equivalent. Some pieces are rotationally or horizontally symmetric, rearranging columns can produce an equivalent board and pieces below the top layer may not affect where a piece should be placed. While also doing experiments with the raw state, they devised a higher-level representation of states and actions. The states consist of 7 heuristic features which reduce the state space to about 40,000. The higher level actions are things such as "Pick the action that *minimizes holes*/*minimizes column height*/*clears a line*" and other such heuristics. They implemented two different neural network agents, both ϵ-greedy, one based on raw state-action

pairs, the other on high-level pairs. The neural networks were very narrow and shallow, containing only a single hidden layer with 10 sigmoid activation neurons and an output layer with linear activations. They took a state representation as input and had one output per action. Training was done with stochastic gradient descent as shown in section 2.5. Future actions were heavily discounted, with $\gamma = 0.1$. The low-level agent showed little sign of learning and did not surpass a random agent. The high-level agent learned, but only reached a level comparable to an agent always attempting to clear lines. It did however score more points per cleared line, suggesting slightly smarter play. It won 29.75% of games against a very capable agent based on brute-force search and a hand-tuned evaluation function. Lundgaard and McKee [2006] remark that it appears to perform well compared to a human player, though it is important to note that a human is limited by their reflexes.

Though the Tetris system used a neural network, it arguably does not qualify as deep reinforcement learning, given the size of the network. This was however not the case for Mnih et al. [2015], which combined reinforcement learning and a significantly larger neural network with convolutions in a technique called deep Q-network (DQN). The resulting agent could play a number of Atari games with great success while incorporating no domain knowledge.

The network input was based on the raw pixel values from the games, although downscaled, cropped and converted to grayscale. Because the games were not necessarily fully observable from the current frame, the four last frames were stacked. The network's hidden layers consisted of 3 convolutional layers and a fully-connected layer, all with rectifier non-linearities. As in Lundgaard and McKee [2006], the network had a separate output for the expected reward of each action. Similarly, the agent used an ϵ-greedy policy with off-policy training where the predicted rewards were adjusted towards the Q-value by gradient descent. Instead of training directly on each sample, experience replay was used, with a mini-batch being sampled after each move. The same network architecture, hyperparameters and algorithm were used for all games. Because the game scores have different scales, during training positive rewards were changed to 1 and negative rewards to -1. The discounting was significantly less harsh than in Lundgaard and McKee [2006], with $\gamma = 0.99$. To have stable targets that discourage divergence or oscillations, the network used to generate the Q-value targets was only updated every 10,000 iterations. For the same reason, error terms were clipped between -1 and 1. To increase the number of games that could be played, an action was only picked every 4th frame, then repeated on the following 3.

DQN was tested on 49 different Atari games. On 43 of these it outperformed any existing reinforcement learning agent, including those incorporating game-specific domain knowledge. On 29 of them it performed comparably to a human professional games tester. It outperformed a random agent on all but one game,

Montezuma's Revenge, which requires long-term planning to a much greater extent than many of the others. The network was found to react similarly to visually similar states, as well as to states with a similar expected reward, showing good generalization. Testing showed that using a single linear layer would be a significant detriment to performance. Experience replay and spacing out updates of the target network also played a large part in its success, although to a lesser degree.

3.3 Monte Carlo Tree Search

The term *Monte Carlo tree search* was coined by Coulom [2006], though the general idea of combining Monte Carlo evaluation with tree search was not new and had previously been attempted with Go. He applied a version of the technique to Go with the Crazy Stone program, with competitive results. However, there are some notable differences between MCTS as described by Coulom [2006] and the approach that ended up being most popular for Go, UCT [Kocsis and Szepesvári, 2006].

He notes that some advantages of his scheme compared to earlier work is that any new information is immediately backpropagated to the root and that it can provably converge to the optimum. It does not make use of progressive pruning of the search tree, which can cut off potentially good moves that have yet to be properly explored. He prefers an approach for the tree search policy where the probability of exploring bad moves tend to zero without ever reaching it, and says that inspiration for this can be found in the fields of discrete stochastic optimization and n-armed bandit problems. He recognizes that both assume stationary distributions, while the probability distribution in a tree search is continually changing, and also identifies some problems with their optimization target. Kocsis and Szepesvári [2006] would later prove that despite this, the UCB1 metric as used for n-armed bandits will guarantee convergence to the optimum in MCTS.

Coulom's [2006] proposed solution is to for each move store both an estimate of its value μ and the uncertainty of its estimate, in the form of its variance σ^2. Then each move i is selected with probability proportional to

$$\exp(-2.4\frac{\mu_0 - \mu_i}{\sqrt{2(\sigma_0^2 + \sigma_i^2}}) + \epsilon_i\,, \tag{3.1}$$

where $\mu_0 > \mu_1 > ... > \mu_N$ and e_i is a heuristic which is constant for a given move. The formula is chosen to approximate the probability of that move being the best assuming a Gaussian distribution. The author remarks that it is similar to the Boltzmann distribution used in many n-armed bandit problems. Kocsis

and Szepesvári [2006] later found the Boltzmann distribution to be significantly weaker than UCB1 as a strategy in MCTS.

He makes a distinction between internal and external nodes based on the number of visits. External nodes have had few visits. Rollouts are performed from these nodes with a domain-specific heuristic policy. Their value for μ is the average of their simulation values, while their variance σ^2 is a heuristic designed to match their large uncertainty. Internal nodes are those from which the tree policy is used. Their value and variance are based on a relatively complicated backup procedure. Coulom [2006] remarks that backing up the average will result in an underestimate and the max will give an overestimate, and proceeds to show empirically that a mixing operator is the most accurate. This is roughly a linear combination of the average and the "robust max", backpropagating the value of the move with the highest number of visits.

Crazy Stone was evaluated against Indigo, which is based on an earlier Monte Carlo method with pruning [Bouzy, 2004], and GNU Go, which uses state-of-the-art search methods that precede the use of Monte Carlo for Go [Silver et al., 2016]. It was found to outperform Indigo, but not GNU Go, against which it only won 28% of games despite spending 10 times as long on computation. Coulom [2006] remarks that Crazy Stone seems to be worse at deep tactics, but has a better global understanding of the game.

To solve the problem of finding a balance between exploiting the currently best move and exploring moves that look less promising, Kocsis and Szepesvári [2006] looked to multi-armed bandit problems. These are problems that can be described as a number of gambling machines, each with an arm where pulling it grants a reward drawn from that arm's distribution. The objective is to maximize the total reward when pulling arms in sequence. This resembles the move selection in MCTS, as you need to choose between exploiting an arm that has given a good outcome so far and exploring arms that might ultimately turn out to be better. This resemblance makes the UCB1 algorithm a promising choice. It is simple and bounds the growth of the regret, the loss caused by not always pulling the best arm, as $O(\ln n)$. When transferred to the MCTS domain, the formula is as described in section 2.3. UCB1 assumes a static distribution, which is not the case in MCTS. Nevertheless, Kocsis and Szepesvári [2006] proved that when UCB1 is applied to MCTS (called UCT), the probability of choosing an incorrect action at the root converges to zero. It does so at a rate that is polynomial in the number of simulations. They also showed it to be effective experimentally.

The idea of using UCT to guide the tree search in MCTS was quickly adopted for Go. Wang and Gelly [2007] used it to develop MoGo. The tree search is as in Crazy Horse, but replacing stochastic optimization with UCT. They used a slightly modified version of UCB1 which takes into account the variance of the estimate. Their rollout policy was a domain-specific heuristic based on local pat-

terns that are matched against positions on the board around the location of the last move. They experimented with pruning of branches and found some improvement from it. Interestingly, they remark "[...] sophisticated pruning techniques are undoubtedly necessary to improve the level of Computer-Go programs," which would turn out to not be the case. One problem they note with UCT is that upon encountering a new state there is no information to choose an action, nor is there information about which order to explore them. Thus, all actions should in theory be explored once before applying UCT, but they found improvements from starting exploitation early if a move seemed particularly promising after one simulation. They also attempted to order the moves by the value of the same moves in an earlier position, with the rationale that a move which is good in one position might also be good in a later position. This somewhat resembles the RAVE algorithm, which will be covered later. They modified the algorithm to run on multiple processors with mutexes to lock access to the tree, again with noticeable improvement. With these techniques, MoGo became the leading computer Go player at the time.

MoGo was further improved in Gelly and Silver [2007]. They created a value function consisting of a linear combination of binary features, where the features are local patterns which are matched against all positions of the board. The value function was trained using temporal difference learning from games of self-play. This value function outperformed the default policy of the previous MoGo when played directly against each other. Despite this, they found that for Monte Carlo simulations, policies based on the value function performed significantly worse than the original policy. They also note that injecting an appropriate amount of nondeterminism improved the performance, but that it did not close the gap. Instead, they attempted to use the value function to introduce prior knowledge into UTC. When a new state-action pair is encountered during the tree search, the value estimate can be initialized with a heuristic, and the visit count can be initialized as the number of simulations normally required to achieve an estimate of that accuracy. They compared different heuristics and found that the value function was the most effective and that the equivalent experience of the estimate was about 50 simulations.

To solve the problem of UCT requiring many samples of each action in a state to produce an estimate with low variance, Gelly and Silver [2007] introduced the UCT_{RAVE} algorithm. This is based on the all-moves-as-first heuristic, which is the idea that instead of averaging over the simulations where an action is selected in the current state, one can average over the simulations where the action is selected in the current or any later states. The reasoning behind this is that an action which is a good choice now is likely to be so later as well. This lets the algorithm generalize the estimate between related states, causing more effective learning. Because this is a biased estimate, it is calculated such

that the RAVE estimate will dominate initially, but it will converge to the UCT estimate. They found this technique to improve performance equivalent to several thousand extra simulations. Combining this with the prior knowledge from the learned value function and the handcrafted default policy enabled MoGo to win 69% of matches against GNU Go in 9x9 Go. It achieved strong master level play and was the first program to beat a professional human player for this board size [Gelly and Silver, 2008].

MCTS was soon also used for other games than Go. Hingston and Masek [2007] attempted to use it for Othello. They used UCT and a learned evaluation function based on weighted piece counts to guide their default policy. Interestingly, they experimented with an evolutionary algorithm to adjust the piece weights, without great success. They were not able to develop a stronger player for Othello than earlier minimax approaches. Arneson et al. [2010] developed MoHex, a Hex player using RAVE and no UCT exploration. The tree search is augmented by domain knowledge. Specifically, moves that are provably inferior are pruned (avoiding the earlier mentioned problem of pruning potentially superior moves) and cells on the board which can be proved to not affect the outcome are filled in. A solver runs parallel to the tree search which gives perfect play when it is able to solve the position. They found that the program had weak opening play, theorizing it could be because there is less information to guide the search early on. This caused them to add an opening book. The default policy was uniformly random, except for a single pattern which aims to maintain a connection if the opponent is attempting to break it. MoHex went undefeated in the 2009 Computer Olympiad, but was similar in strength to an earlier program when omitting the solver and opening book.

Maddison et al. [2014] experimented with training deep convolutional networks for Go as an alternative to the relatively simple and shallow evaluation functions used up to that point. This took the form of a 12-layer policy network trained using stochastic gradient descent to predict actions from a data set of human expert moves. The inputs were a number of 19x19 feature planes. They consisted of a mix of representations directly from the state of the board and the rules of the game, a heuristic and the rank of the expert playing the move. The data set was augmented with random rotations and reflections. This network achieved a 55% prediction accuracy, compared to 35% and 39% for earlier state-of-the-art move prediction, and 44% for an independent result around the same time by Clark and Storkey [2015]. Using the CNN directly as a policy by greedily selecting the recommended move won 97% of games against GNU Go, was about equal to MoGo using 100,000 simulations per move and about equal to the state-of-the-art program Pachi with 10,000 simulations per move. They tested the use of the network's output as prior knowledge in UCT MCTS with RAVE and rollouts based on simple patterns. While this was not compared against other

programs, it won 86.7% of games against the raw network when using 100,000 simulations. This showed the promise of using convolutional networks for move evaluation in MCTS, leading directly into the work done for AlphaGo.

3.4 AlphaGo

Silver et al. [2016] introduced a new approach to computer Go, resulting in a program which was substantially better than any previous attempt and capable of beating a human professional player. They called it AlphaGo. The program used a novel combination of deep learning, Monte Carlo tree search and reinforcement learning. This differentiates it from the most successful approaches up until that point, which were based on MCTS along with shallow policies using linear combinations of features trained to predict human expert moves. These only achieved strong amateur play in 19x19 Go.

While Silver et al. [2016] did make use of supervised training using human moves, they took advantage of the rise of deep learning to construct much more capable approximators based on convolutional neural nets. They use two different network architectures. One is a policy network, which takes the state as an input and outputs a probability distribution over the legal moves in that state. This is based on the work in Maddison et al. [2014]. The other is a value network, which again takes the state as an input, but outputs a single value predicting the winner of the game.

The policy network is a 13-layer CNN with rectifier non-linearities. The input is 19x19x48, corresponding to the 19x19 Go board with 48 feature maps. Similarly the output is a 19x19 probability distribution over the positions on the board, achieved with a softmax output layer. The training of the network is initially supervised using stochastic gradient descent to predict human expert moves. Taking advantage of Go's symmetry, the training data is augmented with rotations and reflections of each position. In addition, a simpler but faster rollout policy based on a linear softmax of patterns is trained with the same data set, achieving only roughly half the accuracy, but being three orders of magnitude faster. The policy network is trained further using reinforcement learning based on self-play. Games are played between the current policy network and a previous iteration of the network selected at random to prevent overfitting. Actions are sampled from the output probability distribution. The games are played in parallel until termination and the final outcomes are used as rewards. The parameters are adjusted with policy gradient updates using the REINFORCE algorithm [Williams, 1992]. After training, this network was by itself better than Pachi, the best performing open-source Go program. It also beat a version of itself only trained with supervised learning by a similar margin.

The value network is a 15-layer CNN with a similar architecture to the policy

network, with the exception that the two last layers are fully connected. The final output is one tanh neuron. It takes the same input, except an extra value for the color of the current player (needed because the other feature maps were represented as "current player" and "opponent", not "black" and "white"). The value network was trained by stochastic gradient descent to minimize the mean squared error between the predicted and actual outcome on self-play games by the policy network. Each training example was sampled from a separate game to obtain uncorrelated positions. The resulting network was more accurate at evaluating positions than Monte Carlo rollouts using the fast rollout policy. Despite being 15,000 times more efficient, it was nearly as accurate as rollouts using the policy network.

During gameplay, the value and policy networks are used in MCTS to guide and truncate the search. In addition to the action value and visit count, each edge of the search tree stores a prior probability. This probability is the policy network's estimated value for that action in the given state, and is calculated when the node is expanded. During the tree search, actions are chosen to maximize the sum of the action value and a bonus similar to UCT called PUCT [Rosin, 2011], $c_{\text{PUCT}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$, where c_{PUCT} is an exploration constant. This encourages choosing actions with a high action value and prior probability, as well as exploring actions with a low visit count. Silver et al. [2016] found that using a policy network trained purely by supervised learning performed better at this stage, theorizing that it's caused by humans picking more diverse moves. When reaching a leaf node, two different evaluations are performed: A full Monte Carlo rollout with the fast rollout policy, as well as a value network evaluation. The final evaluation is a weighted average of these. It was found that weighing them equally performed the best, although using only the value network still produced results superior to any existing program.

Because evaluating the networks is computationally expensive, this is done asynchronously on GPUs by adding them to a queue, while the search simultaneously runs in multiple threads on CPUs. To discourage multiple threads from searching the same branches, a virtual loss mechanism is used where during the search, $N(s,a)$ is increased by a constant amount and the value estimate is decreased by the same amount. The updates during backpropagation are done without locks. Leaf nodes are only expanded after reaching a dynamic visit threshold. This is adjusted on-the-fly such that the positions are evaluated by the policy network at the same rate as they are added to the queue. Both the policy and value network evaluate positions one at a time.

AlphaGo does not use any heuristics such as all-moves-as-first or rapid action value estimation, as this was shown to not be beneficial, nor extra knowledge such as opening books. In addition to the features used in the fast rollout policy, which won't be covered, and the implicit knowledge in the training examples,

AlphaGo makes use of the following domain knowledge:

- Explicit knowledge of the game rules is used during MCTS.
- The symmetry of the board is used to do data set augmentation and a random reflection/rotation before network evaluation.
- The input features are mostly direct representations of the board or handcrafted features which are relatively direct results of the game rules (one is partly a heuristic and two are based on a search).

When evaluating AlphaGo, Silver et al. [2016] found it to win practically every match against state-of-the-art commercial and open-source Go programs, and being substantially better even when playing with a handicap. Remarkably it also beat the European Go champion Fan Hui, marking the first time a computer had beat a human professional player at full-sized Go. They did this without requiring the handcrafted evaluation functions that had been used in many other games, but seemed intractable to create in Go. Thus they showed action selection and position evaluation based on deep learning combined with MCTS to be a highly effective technique for playing the game of Go.

3.5 AlphaGo Zero

Despite using reinforcement learning to train its policy network, AlphaGo still required supervised learning based on human experience for the initial training. This makes it hard to transfer the technique to other domains where such data sets are not available or practical. It could also constrain the program by biasing it towards the way humans play. Silver et al. [2017] introduced AlphaGo Zero, which in contrast to AlphaGo starts from scratch and learns purely by self-play reinforcement learning. This led to a superhuman player with substantially better results than AlphaGo.

Despite the improvement in results, AlphaGo Zero is much simpler than its predecessor in several aspects. It uses far fewer feature planes, 17 instead of 48. These planes only represent the stones of the current player and the opponent for the current and last 7 states, as well the color of the current player. This is thus a pure representation of the board state, instead of using features which are implicit from the game's rules or heuristic in nature. Additionally there is only a single CNN, which fulfills the role of both the policy and value network. Finally, it uses pure truncated search with no rollouts, only value estimation by the network. This gets rid of the rollout policy, which required domain knowledge to construct and supervised learning to train.

The most important difference between AlphaGo and AlphaGo Zero, and what enables using pure reinforcement learning, is the introduction of look ahead

search using MCTS during the training phase. This search is mostly similar to the search AlphaGo performed during gameplay. The notable differences are the lack of rollouts and that nodes are always expanded. Positions are passed to the network for estimation in mini-batches of eight instead of one by one. In the early parts of the game, actions are sampled proportionally based on their visit counts, while later the most visited action is picked. Dirichlet noise is added to the probabilities in the root node to encourage additional exploration. Self-play games using this technique are played continuously by the best-performing network so far. Training examples consisting of the state, outcome and action probabilities (proportional to visit counts in the root) for each position are stored after each game in a replay buffer.

In parallel, the network's parameters are adjusted using stochastic gradient descent to minimize the difference between the predicted and actual winner and the suggested probabilities and the search probabilities. The training examples are sampled from the replay buffer. Because the search will tend to create better probabilities than the raw network, adjusting the network towards these probabilities will make it a stronger player. Along with making it a better predictor of the outcome, this will in turn improve the search and lead to stronger play when creating new self-play games. This is essentially a policy iteration procedure, alternating between improvement and evaluation of the policy and continuously improving it. To ensure that the training leads to an actual improvement, the new network is always evaluated against the best network so far, and it only replaces it if it wins by a margin greater than 55%.

As previously mentioned there is a single network responsible for both the action policy and the position evaluation. This is a CNN with a number of shared layers and two separate output heads. The network is far deeper than the two used in AlphaGo, consisting of 79 shared layers in the full variant, and a 2-layer policy head and 3-layer value head. The network uses residual blocks, but with the addition of batch normalization after each convolution. The value head is similar to the last layers of the value network from AlphaGo, but with batch normalization after the last convolutional layer. The policy head consists of a convolutional layer with batch normalization, followed by a fully connected layer instead of another convolutional layer. It was shown that combining the two networks into one reduced the accuracy of move prediction, but it also reduced the value error and substantially improved the overall performance. Using residual blocks improved both metrics. To isolate the effect of pure reinforcement learning, Silver et al. [2017] also compared the network to one with the same architecture but trained on human expert moves. This network was better at predicting human moves, but worse at predicting game outcomes and substantially worse overall. They suggest that this could mean AlphaGo Zero learns moves that are qualitatively different to how humans play the game.

Using the described techniques, AlphaGo Zero won 100-0 against the published version of AlphaGo. It was substantially better than an unpublished, improved version which beat the world champion Lee Sedol in 2016. It also won 89-11 against another unpublished version similar to AlphaGo Zero, but with Alpha Go's features, rollouts and initial supervised learning, which in online games beat the best human players by 60-0. During training they found that the program first rediscovered many of the strategies used by human players, and then later discovered entirely new ones. With this they showed that a program based on reinforcement learning could teach itself Go from scratch to superhuman performance, with little domain knowledge other than the rules of the game. It was even evident that biasing the program with human knowledge using supervised learning, rollout policies and hand-crafted features could constraint it and be detrimental to performance.

3.6 AlphaZero

AlphaGo Zero achieved performance better than any existing Go program using a very general pure reinforcement learning approach. This naturally raises the question of whether this approach can be adapted to other domains with the same success. With AlphaZero, Silver et al. [2018] attempted this in chess and shogi. The result was yet again a notable improvement on the state-of-the-art in these games.

Because few parts of AlphaGo Zero relied on domain knowledge, the modifications needed to use it for chess and shogi were limited. Go cannot end in a draw, something which was exploited to optimize and estimate the probability of winning. Because this is not the case in general, and for chess and shogi specifically, AlphaZero instead uses the expected outcome. These games are also not symmetrical like Go, so no data set augmentation or random reflections or rotations were performed. The Dirichlet noise added in the root node was scaled in inverse proportion to the typical number of legal moves in a position of that game. The main change is in the representation of the state which is passed to the network. These are obviously game dependent, but similar to in AlphaGo they represent the positions of pieces on the board and the color of the current player, as well as features for special, unobserved rules. Though this is not a necessary change, AlphaZero also behaves differently during network training. While its predecessor tested each updated network against the best performing network so far, this algorithm uses a single network which is continuously updated and always used for new self-play games. To explore the effect of these changes compared to AlphaGo, AlphaZero was trained on Go as well.

It is remarked that intuitively a convolutional architecture seems much better suited to Go than chess and shogi, due to its rules being invariant to translation

and based on adjacencies between positions. The rules of chess and shogi depend on the position on the board and the types of pieces and there are complex interactions over greater distances. Note that both Hex and Othello are more similar to Go in these respects. Despite the differences AlphaZero uses essentially the same network architecture as AlphaGo with great success. The policy heads are game dependent, but fairly similar. Unlike AlphaGo Zero, but similar to AlphaGo, the output layer for chess and shogi uses a convolution. Their outputs are represented as two parts, which piece to move and which position to move it to.

In addition to the network architecture, the hyperparameters and algorithm settings were general enough to be reused for all games, except for the learning rate and the previously mentioned exploration noise.

Training was done using 5,000 first-generation tensor processing units (TPUs) for self-play and 16 second-generation TPUs for optimization. Using this hardware they generated 44 million, 24 million and 140 million training games for chess, shogi and Go respectively and trained each network for 700,000 iterations in mini-batches of 4,096 positions.

AlphaZero was tested against Stockfish, Elmo and AlphaGo for chess, shogi and Go respectively. For chess, it won 155 games and lost 6 out of 1000. Common human moves were discovered during training and AlphaZero was able to beat Stockfish from all of these openings. For shogi, AlphaZero won 91.2% of its games. For Go, it won 61%, despite only generating 1/8th of AlphaGo Zero's data due to not exploiting symmetry. AlphaZero's performance is noteworthy, as chess programs using regular MCTS have been weaker than those using alpha-beta search, and alpha-beta search with neural networks has been weaker than with handcrafted evaluation functions. Silver et al. [2018] theorize that while alpha-beta search will propagate the larger errors a neural network may have compared to have liner evaluation function, they will be averaged out using MCTS in AlphaZero. Additionally, AlphaZero searches three orders of magnitude fewer moves per second than its opponents which use alpha-beta search and are heavily dependent on domain knowledge. These results show that a pure reinforcement learning technique originally designed for Go with no domain knowledge other than the rules of the game can be easily adapted to other games. Not only that, but it proved superior even where handcrafted evaluation functions have been a viable approach to superhuman performance, which was never the case in Go.

3.7 Expert Iteration

Anthony et al. [2017] introduced the Expert Iteration (ExIt) algorithm. It drew inspiration from AlphaGo and though it was an independent discovery, it is very similar to AlphaGo Zero and AlphaZero ("Alpha(Go) Zero" for convenience). It

is of particular interest here because it was used to successfully learn the game of Hex through pure reinforcement learning, achieving very impressive results.

Though the end result is similar, they approach the problem from a different angle with different terminology. They argue first that when playing a board game, a human player uses their intuition to select interesting avenues of play to consider more deeply. This reasoning then improves their intuition, which further improves their analysis. An algorithm with no look ahead is akin to a human only using their intuition to play, while using a neural network as an intuition to guide a tree search and using this to improve the neural network is more reminiscent of human play. This is of course the essence of Alpha(Go) Zero, and, it turns out, ExIt. They also draw parallels to imitation learning, where an apprentice policy is trained to imitate the behavior of an expert policy. In this case the apprentice is a neural network and the expert is a tree search. The expert uses the apprentice to guide its search, such that when the apprentice is improved by imitating the expert, the expert is also improved, which they call expert improvement. This in turn gives the apprentice a better target to learn from, so it can be viewed as a case of iterative imitation learning.

One major difference between Alpha(Go) Zero and ExIt is how they sample positions. Anthony et al. [2017] chose to only sample a single position from each game to avoid correlated positions, similar to the approach used in the original AlphaGo to train the value network. Additionally, instead of playing full games with the expert policy, they use the apprentice policy directly, which runs much faster. The expert policy is only used to predict a better action for the single position chosen from each game. This is equivalent to the DAGGER algorithm [Ross et al., 2011]. It does not produce as accurate predictions as expert self-play however, as the statistics will not be carried over from move to move. The positions will also likely be different from those the expert policy would end up in.

Some other differences are that ExIt uses early stopping instead of L2 regularization and that the loss for value estimation is cross-entropy instead of mean-squared error. Value estimates are averaged with rollout estimates as in the original AlphaGo, although the rollout policy is uniformly random. The neural network is a conventional and much shallower 14-layer CNN. They use a continually improving network as in AlphaZero, but unlike AlphaGo Zero. ExIt's MCTS uses the RAVE [Gelly and Silver, 2007] algorithm. MCTS is performed synchronously, threads merely search from another position while waiting for neural network evaluations. Dirichlet noise is not used, instead their version of UCT guarantees that all moves will be tried at least once regardless of the prior probabilities.

Before performing expert improvement, they warm start the apprentice policy by training it using a pure imitation learning procedure where the expert policy

is a MCTS which does not utilize the apprentice policy. Of note is that they compare two training targets, one where the apprentice only tries to imitate the expert's chosen move, another where it tries to imitate the expert's probability distribution of moves. The former is similar to how AlphaGo initially learned to imitate human expert moves. They found that while they achieve a similar prediction accuracy, the latter leads to a better player. The end result was about comparable to the MCTS expert. A combination of a warm started apprentice policy network and MCTS won 97% of games against the pure MCTS expert.

They then went on to train the policy network further with ExIt, comparing it with REINFORCE as used to train the policy network in AlphaGo. They found that ExIt's learning is faster and more stable and leads to a stronger policy. They continued training until they had built up a large dataset of around 550,000 positions and then used this to train a combined policy and value network. ExIt does not provide a way to do this directly. Their solution is to use the trained policy network to get a Monte Carlo estimate of the value of each position. This is another difference from Alpha(Go) Zero, where the value estimates are from self-play by the expert. Anthony et al. [2017] say this was not feasible with their computational resources. The combined network was further improved with ExIt. A comparison shows a substantial improvement from adding value estimates, to the point where the combined network apprentice outperforms the policy network expert.

ExIt was compared against MoHex, the best publicly available Hex program, which as described earlier is highly dependent on domain knowledge. In comparison, ExIt learns from scratch utilizing no information except the rules of the game (although it does make use of the RAVE algorithm). ExIt won 75.3% of games with the same number of MCTS iterations and 59.3% with 1/10th of the iterations. These results show the validity of this approach in Hex, as Alpha(Go) Zero did for Go, chess and shogi, even if many of the details are different. Notably it showed that letting the apprentice do more of the work can still lead to good results, while using vastly less computing resources. With the available resources, a solution more similar to Alpha(Go) Zero would not have been feasible, according to the authors. It is unclear, however, how such a solution would compare if the necessary resources were available, although it is likely that relying more on the expert would give it an edge.

3.8 Motivation

In this review, we first saw that reinforcement learning and deep learning can be combined to create very successful game-playing agents. We then saw the history of the development of Monte Carlo tree search, from the original idea, through to the more familiar UCT variant and then on with further refinements. Utilizing

these techniques, a much greater success than previously was over time achieved in Go. We also saw how they could be applied to other games. They were eventually combined with deep learning, but without the crucial reinforcement learning aspect. That came with AlphaGo, which could even beat professional players, but also required the use of supervised learning. AlphaGo Zero got rid of this and could learn from scratch, with a substantial increase in performance. But AlphaGo Zero lacked rollouts, which had been one of the central concepts in MCTS until then and present in all the previous systems I have reviewed. This was also missing from AlphaZero, which generalized the technique to chess and shogi. It was present in ExIt, an similar but alternative approach applied to Hex. I found it fascinating that what seemed like a core part of the algorithm could be fully replaced by a direct, shallow evaluation by deep neural networks, with state-of-the-art superhuman results. Clearly AlphaGo had shown that the two techniques could be combined, but it required a domain-specific rollout algorithm and supervised learning and still performed worse than the rollout-less algorithms. This raised the question in my mind of whether there could be some benefit to retaining the rollouts, but without compromising the ability to learn from scratch. It led me to implement a system similar to AlphaZero so I could investigate the idea.

Chapter 4

System Architecture

Based on the theory and research presented in chapter 2 and chapter 3, I have designed and implemented a system for playing two-player zero-sum finite deterministic perfect information games with a combination of deep neural networks, reinforcement learning and Monte Carlo tree search. The system is intended to be flexible and modular and parts of the system can be swapped out without major effort. Game-specific code is kept separate from the general implementation of the algorithm. The goal is to be able to easily create agents for additional games by writing code purely for the game logic and without touching any existing code. The system is implemented in Python. There are four principal, independent parts of the system, the game manager (section 4.1), game net (section 4.2), tree search (section 4.3) and training algorithm (section 4.4).

4.1 Game Manager

The logic for a game is encapsulated in a *game manager*. A game manager knows how to represent states and actions of the game. A state always contains the current player and can contain additional game-specific data, such as a board representation. Actions are encoded as integers between 0 and the number of possible moves in the game. Using these representations and the rules of the game it can:

- Construct the initial game state.
- Generate the legal actions in a state.
- Generate the next state given a state and an action.
- Check if the end of the game is reached in a state.

- Determine the outcome of a final state.

The generic game manager is defined purely as an interface with a method for each of these, with any actual behavior found in concrete, game-specific managers. Game managers are used to isolate the game logic from the generic MCTS algorithm as described in section 4.3.

Because the same state can be found in many branches of the MCTS game tree, the methods will potentially be called with the same arguments again and again. To take advantage of this, game managers contain a cache that ensures computation is reused. Because the system is built such that each process only contains a single game manager, this cache is essentially global within each process and greatly decreases the computational needs of the game logic.

For Hex, the game manager is relatively simple. The board state is represented as a 2-dimensional grid where each square is either empty (-1) or filled by black (1) or white (0). The initial state is an empty grid. This grid representation is an NxN array equivalent to shifting the rows of the board into a square. The initial state is thus such an array filled with only -1's. Black's goal is to achieve a connected path between the top and bottom side, while white must do the same between the left and the right. The 6 neighbors of each square are the squares to the east, west, south, north, south-west and north-east. Actions are 2D-coordinates of the square to occupy, with the origin in the upper left corner, x increasing to the right and y increasing downwards. They are stored as a single integer on the form $N \cdot y + x$. The legal actions are simply those that fill an empty square. The next state is one where that square is filled by the current player. To determine a winner, a depth first search is first run from any of the left squares filled by black, traversing to any neighboring squares filled by black. If this search reaches the right side, black has won. If not an equivalent search is run from top to bottom to check if white has won. If neither player has won, the game is not yet over, since Hex cannot end in a draw.

Othello uses the same representation of an NxN array with elements consisting of -1, 1 or 0 for an empty square, a black piece and a white piece respectively. Similarly, the action representation is the same, with one caveat. Pass is a legal action in Othello when no other actions are possible and its representation is N^2, one more than placing a piece in the lower right corner. The algorithm for finding legal actions is as follows:

1. For each position of the current player
    (a) For each cardinal direction
        i. If there is a path from the position in the cardinal direction consisting of squares occupied by the other player and ending in an empty square, add the action of placing a piece there as a legal action.

2. If no legal actions were found, add the pass move as a legal action.

The algorithm for generating a child state is as follows:

1. If it's the pass move, simply reuse the state with the other player.
2. For each cardinal direction
   (a) If there is a path from the position in the cardinal direction consisting of squares occupied by the other player and ending in an empty square, flip all enemy pieces in the path into pieces of the current player.

The trivial optimization of directly encoding the found direction of the path into the action itself when calculating legal actions is invalid, because an action might flip pieces in multiple directions. The algorithm could still be modified to encode these multiple directions, which would give a small constant-factor speed-up, but at the cost of additional complexity, so it wasn't implemented (it turns out that this is not a bottleneck in any case).

To determine whether the game is over, first the legal actions of the current state is checked. If the only legal action is the pass move, the legal actions of the next state is checked. If it too only allows for the pass move, the game is over. The outcome can then be determined by simply counting the number of pieces for each player.

4.2 Game Net

Similarly to the systems described in section 3.4, section 3.5, section 3.6 and section 3.7, my system uses deep neural networks to estimate the state value (value network) and probability distribution for the policy (policy network). This is fully encapsulated in a *game net*, which contains such a neural network. It is responsible for taking the state representation used by the game manager and converting it to a representation which can be passed to this network for evaluation. Finally it needs to identify illegal moves in the policy output and zero out (mask) the probabilities of these so that only legal moves can be chosen. In addition to these game-specific behaviors, defined as a generic interface, a game net has some default functionality that does not need to be reimplemented for each game. This includes:

- Saving and loading from a file to allow checkpointing.
- Moving the net to a different hardware device (related to the use of GPUs).

- Evaluating a single state by using the specialized methods to change the representation, forwarding it through the neural network, zeroing out the illegal probabilities and then renormalizing them.
- Converting a distribution over actions from MCTS to a representation that can be compared with the policy network's output.
- Training the network, see section 4.4 for details about this.

There is also a default implementation of masking that simply consults the game's manager for legal actions and masks out the illegal ones.

Some assumptions are made about the neural network wrapped by a game net. The policy output is flattened, softmaxed, masked and renormalized before being treated as a policy distribution in the following way: The integer value i of action a can be treated as an index into the final output p, so that $P(a) = p_i$. This means that the initial output should be unnormalized and regardless of shape, the total number of elements must equal the number of actions. The value network output should also be unnormalized. A tanh is applied for normalization between -1 and 1. This value is interpreted as relative to the current player, with 1 meaning a 100% probability of the current player winning and -1 a 100% probability of losing. The value is negated for the min player and scaled to a range between 0 and 1. The reason for this is that MCTS expects a value between 0 and 1 from the perspective of the max player.

The game net interface is flexible and implementations can use any kind of network architecture (e.g. fully connected or convolutional) with both split and combined policy and value networks. It could in principle be done using any neural network library. My implementations are done in PyTorch [Paszke et al., 2017].

Because the game net also handles game logic to some extent, it could be argued that it should be a part of the game manager. However, separating them has some clear advantages. It gives a clean break between the pure representation of the game and the game as represented by the neural network. This allows for reusing the same game manager with multiple different game nets, which is very flexible and can be used to easily compare different network architectures. It also allows for running MCTS without a policy and value network, for example by substituting in estimators which give zero value to every state and uniform probability to all actions and relying purely on rollouts. This is useful for testing. Many other configurations are also possible, such as estimating just the value or just the policy. As mentioned, an important goal of the system is modularity, precisely because it allows for such flexibility.

I have implemented game nets based on a combined policy and value convolutional network. For both games the input to the network is a stack of 3 2D-grids of the same size as the board, where the first one represents the pieces

of the current player, the second one represents the pieces of the other player and the third one is a constant representing whether the current player is black or white. Specifically the first two grids have value 0 for empty tiles and 1 for tiles where the respective player has a piece, while the third grid has value 1 for all tiles if the current player is black and 0 if it is white. Using the current player as the first grid and the opponent as the second rather than e.g. the black player and then the white player is intended to improve generalization. It means that two equivalent but flipped (black pieces turned into white and vice versa) board states look the same to the network. Together with the interpretation of the value network output, this means that the network could completely ignore who's actually playing. Still, the third grid ensures that if relevant the network can still tell who's playing and take into account e.g. first-player advantage (in some games you would also need to know the current player to determine which moves are legal or preferable).

For Hex, the input grid is rotated 90 degrees when white is playing. This is intended to improve generalization. The idea is that rather than the network having to learn to connect different sides depending on the player, it consistently sees the target of the current player as connecting the north and south sides. This means that the output will also be rotated for the white player, so the rotation is reversed after forwarding through the neural network. No such transformations are done for Othello.

The complexity of the network itself is highly configurable. The general architecture consists of a single convolutional block, then a configurable number of residual blocks, a policy head and a value head. Figure 4.1 depicts this architecture.

A convolutional block consists of:

1. A convolutional layer with a configurable kernel size, padding and number of filters.
2. A batch normalization layer.
3. A rectified activation.

A residual block consists of:

1. A convolutional block with a kernel size of 3, padding of 1 and a configurable number of filters.
2. A convolutional layer with a kernel size of 3, padding of 1 and a configurable number of filters.
3. A batch normalization layer.
4. A skip connection from the block input.

5. A rectified activation.

The policy head consists of:

1. A convolutional block with a kernel size of 3, padding of 1 and a configurable number of filters.
2. A linear, fully connected layer with a configurable number of outputs.
3. A softmax activation.

The value head consists of:

1. A convolutional block with a kernel size of 1, no padding and 1 filter.
2. A linear, fully connected layer with a configurable number of outputs.
3. A rectified activation.
4. A linear, fully connected layer with 1 output.
5. A tanh activation.

4.3 Monte Carlo Tree Search

Similarly to the other parts of the system, the MCTS algorithm is confined to its own module and designed to be flexible. It is completely generic and game-independent, so that no modification or specialization is required for a new game. It does this by delegating any game specific logic such as finding child states and determining if the game is over to game managers, and the evaluation of states to game nets. See fig. 4.2 for examples of this. Specifically, one must pass a game manager and a *state evaluator* for the algorithm to use. The latter is an interface for a function which given a state returns an estimate of its value and a probability distribution over the legal actions in that state. This is typically based on a game net, but as mentioned in the previous section, it does not necessarily have to be. The number of simulations per move can also be customized.

The algorithm can be passed a *rollout policy*, which simply needs to be a function returning a legal action when passed a state. This policy is used to choose actions while performing a rollout to the end of the game. The game manager generates the next state for each action and determines when the game is over. The use of the game manager and rollout policy is shown in fig. 4.3. There is an adjustable parameter for the probability of performing a rollout. When a rollout is performed, the backpropagated value is the average of the state evaluation and the outcome of the rollout.

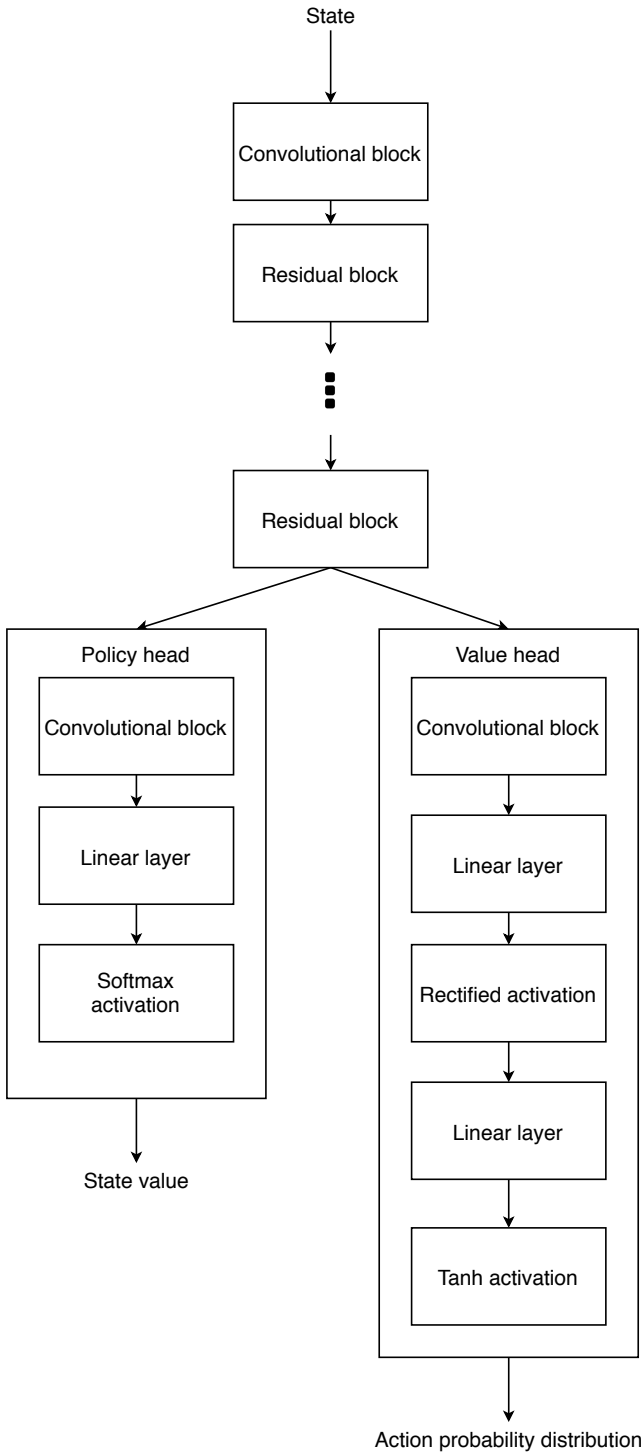
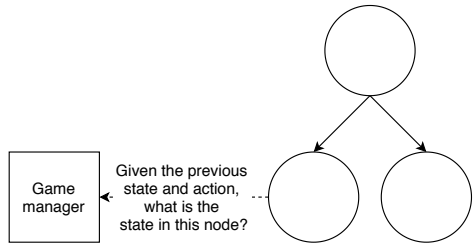
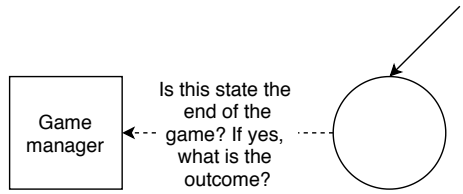


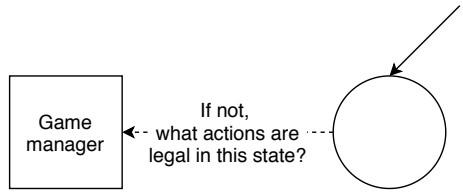
Figure 4.1: The network architecture, showing the split heads of the network and their corresponding outputs.



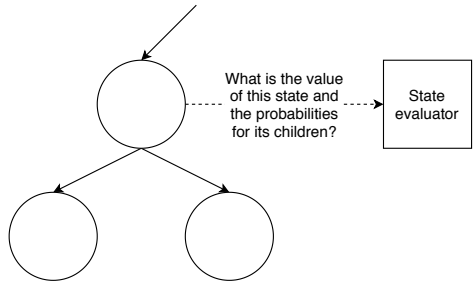
(a) As the node is visited, the game manager is queried for the current state, given the state of the previous node and the action taken by the tree policy.



(b) Now that the state is known, the game manager is used to check if the end of the game has been reached. In that case the outcome is also known by the game manager and can be backpropagated directly.



(c) If it wasn't the end of the game, there are actions to be taken and the node should have child nodes. The game manager is asked for these and they are all added.



(d) Finally the state evaluator is used to set probabilities for the child nodes and get a value estimate that can be backpropagated.

Figure 4.2: The process of expanding a node and how the game manager and state evaluator are used during it. Rollouts take place during expansion, but are shown in fig. 4.3.

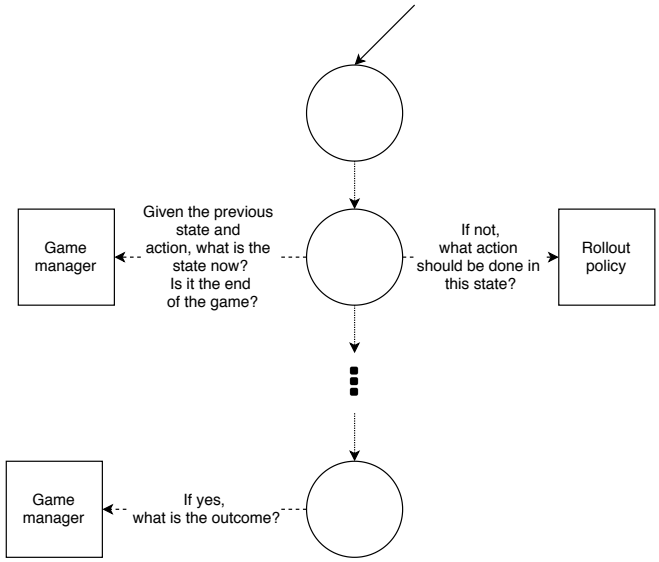


Figure 4.3: The process of performing a rollout and how the game manager and rollout policy are used during it.

Tree nodes handled by the algorithm store the sum of their value estimates E, their visit count N and the prior probability P of the action leading to them. When expanded they also contain a mapping from legal actions to child nodes. As an optimization, states are not stored in nodes, but rather computed once a node is actually visited.

The tree policy is based on the same PUCT measure used in the AlphaGo, AlphaGo Zero and Alpha Zero papers, meaning the exploration bonus for action a in state s, $u_{s,a}$, is $c_{\text{PUCT}} P_{s,a} \frac{\sqrt{N_s}}{1+N_{s,a}}$ [Rosin, 2011]. Independent of the player, a value estimate close to 1 is an indication that player 0 is winning, while an estimate close 0 is an indication that player 1 is winning. Thus, player 0 is the max player and player 1 is the min player. The total value for a in s is then $\frac{E_{s,a}}{N_{s,a}} + u_{s,a}$ for player 0, and $\frac{E_{s,a}}{N_{s,a}} - u_{s,a}$ for player 1. When a node is unvisited, its value estimate is estimated as a loss for the previous player. For example, if the current player in an unvisited node is 1, the value estimate is 0, because this is a loss for player 0, the current player in the parent node. This somewhat offsets the high exploration bonus of unvisited nodes and can delay visits to them.

A measure that instead improves exploration is that Dirichlet noise with a configurable α is optionally added in the root node each time MCTS is used to find the next move. The noise is mixed with the probability distribution from

the state evaluator using a configurable mixing factor d:

$$P_{s,a} = (1 - d)P_{n_{s,a}} + d\text{Dir}(\alpha)_i \,, \tag{4.1}$$

where P_n is the network estimate, a is the action leading to the i-th child node and $\text{Dir}(\alpha)$ is a symmetric Dirichlet distribution with the same number of parameters as the number of child nodes, all equal to α. Note that when a move is chosen and played in the actual game, the corresponding node of the game tree is set as the new root. This means that any searches that were previously done in this subtree are reused, essentially giving us free simulations. But this means that the Dirichlet noise generally has the biggest impact at the beginning of a game, because the probabilities are part of the exploration bonus, which decreases with the number of visits. This might be beneficial, because in many games there are more moves to choose between in the beginning and mistakes are less costly, so focusing on exploration then is more beneficial than at the end of the game. If however we end up in a branch that is not well explored, the Dirichlet noise will have a larger impact, which fits with the intuition that even late in the game exploration is important if you end up in an unexpected situation.

When the tree simulations are done, an actual move to play is chosen. This is done differently depending on how many moves have been played. After a certain number of moves, which is configurable, the move with the max number of visits in the root node is chosen. Before this, a move is sampled from the distribution over the number of visits. This is consistent with the earlier explanation: In earlier parts of the game a mistake is likely less costly, so you can afford sampling a bad move then, while in the endgame you probably want to pick the one move you think is best. The motivation for sampling is to increase diversity between self-play games, since they are used as training targets. Many similar (or even equal), strongly correlated games are less useful targets than diverse games that explore a multitude of possibilities, which should lead to a more robust generalization. This is especially true when there are many moves that seem good and we want to avoid only greedily playing the one that seems best. However it also encourages sometimes trying moves that seem poor and thus checking if they actually increase the chance of a loss.

The algorithm proceeds as follows for a complete self-play game:

1. Initialize the root node using the game manager.
2. While the game is not over (as determined by the game manager):
   (a) If unexpanded, expand the root node using the game manager and evaluate it using the state evaluator to get P for its children.
   (b) Mix Dirichlet noise with parameter a with the child nodes' probabilities.

(c) For each simulation:
   i. Search from the root to a leaf (unexpanded) node using the tree policy ($\arg\max_a q_{s,a}$ for the max player, $\arg\min_a q_{s,a}$ for the min player).
   ii. If the leaf node is at the end of the game:
      A. Use the actual outcome as determined by the game manager as the value estimate.
   iii. Else:
      A. Expand the leaf node using the game manager, adding all subsequent states as child nodes, and evaluate it using the state evaluator to get an estimate of its value and P for its children.
      B. Optionally and with probability p_r perform a rollout to the end of the game using the rollout policy and average the result with the estimated value.
   iv. Backpropagate the value up to the root, incrementing visit counts and adding to E for each visited node, including the leaf itself.

(d) If the number of moves played is below the sampling threshold:
   i. Sample an action proportional to the visit counts in the root node.

(e) Else:
   i. Pick the action with the highest visit count in the root.

(f) Execute the action using the game manager and set the root node to the corresponding child node, reusing the subtree.

The algorithm is single-threaded and does synchronous evaluation. This is far slower than the procedure utilizing parallel search threads and batching used in e.g. Silver et al. [2018], because the process is blocked on state evaluation and cannot do useful work. It is however also easier and quicker to implement and debug, partially because of the inherent complexity of multi-threaded code, but also because of implementation details in the Python runtime. Specifically, it contains a global-interpreter lock which by default blocks Python threads in the same process from executing in parallel [Python contributors, 2019b]. A partial mitigation of this problem will be presented in the next section.

4.4 Training

Similarly to the MCTS algorithm, the training algorithm is game-agnostic. It requires passing a game net and game manager, which contain all the necessary game-specific logic.

The algorithm initializes a replay buffer to store training games. This is a first-in-first-out queue of a finite size and thus it evicts the oldest games first if it reaches its max size. Storing all prior games would be a problem regarding memory consumption, but the main reason to avoid it is that older games were played using older, less accurate versions of the network. It might therefore be better to evict these rather than continuing to use them as training targets.

The primary bottleneck in the system is the generation of training examples using MCTS self-play. The problem is that network evaluation is relatively slow and because it runs in each tree simulation, it takes up the majority of the runtime. One mitigation is to run the network on a GPU, which is specialized for matrix operations and provides a large speed-up, especially for convolutional networks. This is however not enough, as it does not solve the problem of everything being blocked waiting for network evaluation. As mentioned in the previous section no parallelization is used within a single game. The solution has been to instead introduce parallelization at a higher level of the training process. Specifically, the optimization of the neural network and the generation of training targets through self-play are run in parallel, and the generation itself consists of multiple games of self-play executing in parallel. The parallelization is based on separate processes, not threads within a single process, with the optimization occurring in the main process and each self-play game occurring in its own separate process. This can be seen in fig. 4.4. This ensures that another game can run concurrently even though no progress is done in a game during network evaluation. In an environment where the number of cores equal or exceed the number of games, they can all run simultaneously. A downside of processes is a greater overhead than threads. Communication between processes in Python does not generally use shared memory, but instead requires serialization of objects and message passing [Python contributors, 2019a]. For one this means that passing training examples back to the main process is much slower than if it ran within the same process or thread. This overhead also precludes the use of batched network evaluation. This method can trivially be adapted to running multiple single-threaded self-play games in parallel by batching between games instead of search threads within a game, but requires a separate process to do network evaluation. This was attempted but later abandoned, because the overhead from passing states from each self-play process to an evaluation process and then passing results back trumped the performance benefit of batching. This means that as you increase the number of processes, the GPU soon becomes a bottleneck. This is partially because the processes clamor for the use of the GPU without coordination and partially because each process uses a separate chunk of the limited memory of the GPU. To make matters worse, the main process also requires use of a GPU to speed up the optimization process. For this reason the system is designed such that the main process can use a separate GPU from the

other processes, partially easing the pressure on their GPU. Another measure that is taken to decrease GPU usage is to cache all state evaluation, since, as explained previously, a state might be present in multiple locations in the game tree. They can also repeat between games. This cache is purged whenever the neural network is updated.

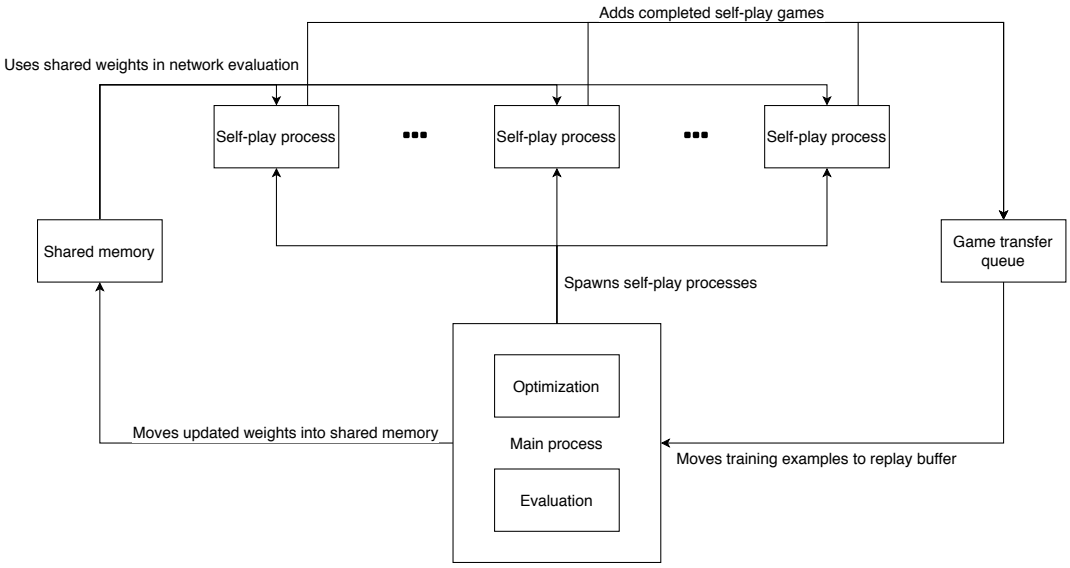


Figure 4.4: An overview of the responsibilities of the different processes and how they communicate between each other.

Each process runs a certain configurable number of self-play games using MCTS. They receive regular weight updates to their game net from the main process. Before they receive their first weight update, they use a network that provides a uniform evaluation (a state evaluation of 0.5 and equal probability of all legal actions). After each game, they transfers the state, distribution of visits in the root node and the final outcome for each move of the game to the main process. Thus the improved policy of MCTS and the actual winner of the game will be used as targets for the neural network, as explained in section 3.5. Because we want the training targets to be from the perspective of the current player, the outcome is negated for the min player (the game net reverses this transformation when used as an evaluator, as mentioned previously). The main process receives the examples and uses the game net to transform the states into the game-specific representation before inserting them into the replay buffer. A configurable number of examples are sampled uniformly random from the replay

buffer and passed to the game net's generic training implementation mentioned previously, which forwards them through the network. It then calculates the loss as the sum of the MSE between the expected outcome v and actual outcome z and the cross-entropy between the network distribution p and MCTS distribution π:

$$l = (z - v)^2 - \pi \log p \tag{4.2}$$

This is averaged over the batch of training examples. Additionally, L2 regularization is calculated for the network weights and added to the loss to improve generalization. The loss is then used to optimize the network. The optimization technique used is stochastic gradient descent with momentum, where the weight update Δw depends both on the gradients and on the previous weight updates:

$$\Delta w_{t+1} = \alpha \nabla l + m \Delta w_t \,, \tag{4.3}$$

where α is the learning rate and m is a constant between 0 and 1 adjusting the strength of the momentum.

As in Alpha Zero, but unlike AlphaGo Zero, the network is improved continuously [Silver et al., 2017, 2018]. However, the updated weights are not continuously transferred to the self-play processes. This is done at a configurable interval. This is partially because even if the network improves over time, the change from iteration to iteration might be more erratic. Training for several iterations before using the network might therefore provide better and more stable targets. This is also done for performance reasons. PyTorch provides a mechanism to share memory directly between processes and thus avoiding the overhead of serializing and passing weights to each process independently [PyTorch contributors, 2019]. Despite this, it would be too slow to update the shared weights at every iteration of training.

In addition, there is both checkpointing to disk and evaluation at configurable intervals. The former is done to enable post-hoc analysis of the training process and trained models, as well as to enable restarting the training in case something goes wrong or further training is desired. The latter allows for monitoring the progress. To evaluate the network, it is compared against both the previous network and MCTS using a random rollout policy. The comparisons use the MCTS-enhanced policy, not the direct network output. This is slower, but more realistic. A configurable number of games is played between the agents with the starting player alternating. In the games between the networks, actions are chosen ϵ-greedily to introduce stochasticity, meaning that $\epsilon \cdot 100\%$ of the time a move is sampled based on the visit distribution and otherwise the most visited move is chosen. This is done because the outcome when two networks play against each other would otherwise be completely deterministic (Dirichlet noise and early move sampling is not used during actual game play) and not give a particularly useful indication of performance (the win rate would always be either 0%, 50% or

100%). Plays against the agent using random rollouts are inherently stochastic, so the action with the most visits is always played. Evaluation is about as much work as self-play, so it's relatively slow, and no network optimization takes place while evaluation is running. Because the network is adjusted continuously without checking if the new network outperforms the old one, evaluation could be moved to a separate process. But as long as it uses the same GPU as the optimization, the speed-up would likely not be large. Additionally, because the self-play is slow, optimization is already very fast compared to the rate at which new training data is produced.

The following is pseudocode for the main process:

1. Initialize a replay buffer.
2. Move the game net to a GPU.
3. Create a copy of the game net on another GPU and put the network weights into shared memory.
4. Spawn multiple self-play processes and provide them with the copied game net and a queue to transfer training examples.
5. While the self-play processes are still running:
   (a) Get any new training examples from the queue, convert the states to the network-specific representation and put them into the replay buffer.
   (b) Sample a batch from the replay buffer.
   (c) Forward the batch through the network.
   (d) Calculate the policy and value loss, backpropagate and adjust the weights using the gradients and L2 regularization.
   (e) If the correct number of training iterations has passed, move the new weights into shared memory so they are available to the self-play processes.
   (f) If the correct number of training iterations has passed, save the network weights and the rest of the training parameters to disk.
   (g) If the correct number of training iterations has passed, evaluate the network as described previously. Cache both the current and previous network to reduce the number of GPU evaluations.

The following is pseudocode for the self-play processes:

1. Initialize a uniform state evaluator to use until the first weight update is received.

2. For the number of games to play:
   (a) If still using the uniform evaluator and the weights have been updated, switch to the neural network evaluator. Initialize a cache to reduce the number of GPU evaluations.
   (b) If the weights have been updated, reinitialize the cache.
   (c) Play a self-play game using MCTS, recording each state-action pair.
   (d) Add each state-action pair along with the final outcome of the game to the queue to transfer the game to the main process.

4.5 Overview

As the preceding sections have shown, I have designed and implemented a flexible framework for game-playing based on deep learning, Monte Carlo tree search and reinforcement learning. General and game-specific code is cleanly separated, so that even though game logic has only been implemented for Hex and Othello, adding additional games would like little effort. The overall architecture of the system can be seen in fig. 4.5. We can clearly see the modularity of the system, with four different parts of the system, separated but connected so that each fulfill their own function as part of a whole. The game manager encapsulates the game logic, providing the MCTS module with what it needs to construct and search the game tree. This module also receives evaluations of states and actions from the game net. MCTS is then used by the training algorithm to run many separate instances of self-play games in parallel, creating training examples that can then be used to optimize the game net. This is a system ready to run experiments that can shed light on my research questions.

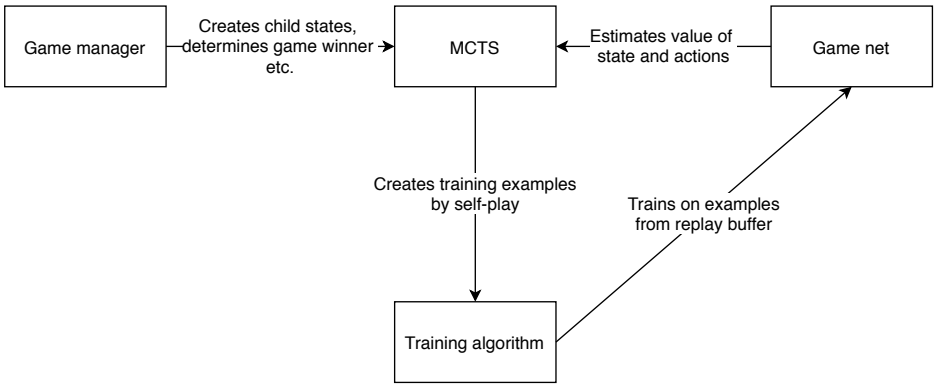


Figure 4.5: The system architecture with the four parts of the system and their relation to each other.

Chapter 5

Experiments and Results

To investigate the research questions posed in section 1.2, empirical data is needed. This chapter covers the experimental parts of the project, with information about the planned experiments, the parameters used for running the system and the resulting data.

5.1 Experimental Plan

Experiment 1 Run the training process without rollouts. Afterwards, use each checkpointed network as a state evaluator in MCTS and compare it to an agent with a random rollout policy and to an agent using the network from the previous checkpoint.

The goal of experiment 1 is to ensure that the system is working correctly and that there is a smooth increase in performance during training.

Experiment 2 Using the final network trained in experiment 1 as a state evaluator, compare agents that have a 0%, 20%, ..., 80% and 100% chance of doing a rollout with a random rollout policy on each expansion. Compare each agent against all others.

Experiment 2 aims to get an indication of the effect rollouts have on the performance and thus to answer research question 1.

Experiment 3 Using the final network trained in experiment 1, compare an agent with a rollout policy of choosing the best movie suggested by the policy network to one that uses a random rollout policy. Do rollouts on every expansion. Balance the number of simulations such that the time per

move is approximately the same. For comparison, run it both with and without also using the network as a state evaluator for both agents.

This is to investigate the effectiveness of a sophisticated but slow rollout policy and thus to answer research question 2.

Comparisons between agents are done as previously described in section 4.4. The only caveat is that for experiment 3, caches are not shared between agents and games. Caching gives a lower time per simulation and thus more simulations per move, meaning that the second player would have an advantage and later games would involve more simulations. To have a fair comparison and independent games this must be avoided.

5.2 Experimental Setup

The hyperparameters used are shown in table 5.1, table 5.2 and table 5.3 for experiment 1, 2 and 3 respectively. Because the networks trained in experiment 1 are reused, network parameters and board sizes are naturally the same in the other experiments and have not been repeated in those tables. Experiment 1 is run 20 times. Experiment 2 and 3 are run once for each trained model, i.e. also 20 times.

Evaluation is used during training to keep an eye on the progress, but with an intentionally low number of games to decrease the performance impact. Evaluation is rerun after training using the checkpointed networks with more games to increase the precision of the results. For this reason, there are two numbers given for the evaluation games in table 5.1.

Table 5.4 shows the hardware used to run the experiments. This is IDI's Malvik server. Because it has two GPUs, training and optimization are set to run on separate ones. The number of self-play processes are set to the maximum allowed by the GPU's memory. Note that this is a shared server without a queue system, so the access to these resources is not exclusive.

Max games in replay buffer	5,000
Games per process	1,000
Evaluation games during training	20
Evaluation games after training	40
Self-play processes	25
Evaluation interval	10,000
Checkpoint interval	10,000
Network transfer interval	1,000
Simulations	50
Simulations for random opponent	100
Batch size	1,024
Learning rate	0.01
Momentum	0.9
Regularization constant	0.001
Residual blocks	3
Filters	128
Value head hidden units	128
ϵ for move selection	0.05
c_{PUCT}	1.25
Dirichlet factor	0.25
Board size	6x6
Sample move cutoff	10
Dirichlet α (Hex)	0.33
Dirichlet α (Othello)	1

Table 5.1: Experiment 1 parameters.

Simulations	50
Games per agent pair	40

Table 5.2: Experiment 2 parameters. Board size and network parameters are as in table 5.1.

Games	240
Seconds per move	1

Table 5.3: Experiment 3 parameters. Board size and network parameters are as in table 5.1.

CPU	2x Intel Xeon 6132
GPU	2x Nvidia Tesla V100
RAM	768GB

Table 5.4: Specifications of the machine the experiments are run on.

5.3 Experimental Results

The results of experiment 1 are shown in fig. 5.2 as plots of the win ratios during training for each model. The results are shown separately for each model as there isn't a suitable way to combine them while still judging how each training progresses and being able to compare them. Recall that the training finishes as soon as the self-play processes do, so though they're based on the same number of self-play games, there is some variation in the total number of iterations for each model. The average number of iterations is about 320,000 for both games. Of note in the data is that we see a clear progression for every single model across both games, but with some instability that will be expanded upon in section 6.1 along with the other results.

Note that the win ratio is not calculated as $\frac{n_{\text{wins}}}{n_{\text{games}}}$, but $\frac{n_{\text{wins}}+0.5n_{\text{draws}}}{n_{\text{games}}}$ (though these are equivalent for Hex), meaning draws are counted as half a win. This allows the draws to be included in the plot, without cluttering it with another pair of lines, and lets Hex and Othello be treated the same. This calculation is also used in fig. 5.3. They are however kept separate in table 5.5 in order to calculate confidence intervals.

Figure 5.3 shows the results of experiment 2 as the win ratios for all possible pairs of rollout probabilities. Because this creates a large number of values (36 for each model), only the means across all models are shown. Note the surprising result of win ratios increasing towards the right, indicating that higher rollout probabilities decrease performance.

Table 5.5 shows the results of experiment 3 in the form of aggregate win, draw and loss ratios for policy network rollouts against random rollouts. The numbers are given with 95% confidence intervals and separately for each game and by whether the network was also used as a state evaluator. The full tables are not shown to avoid clutter, but are available in appendix A. Recall that draws are not possible in Hex, but they have been included simply to keep the tables the same. Table 5.6 shows the average number of rollouts and simulations done per move during experiment 3. We can see by the first table that policy network rollouts consistently don't perform as well as random rollouts. There are considerable differences based on the game and whether a state evaluator was used. The second table shows that the vast majority of simulations don't

contain a rollout and that including state evaluations has a large impact on the number of simulations and rollouts.

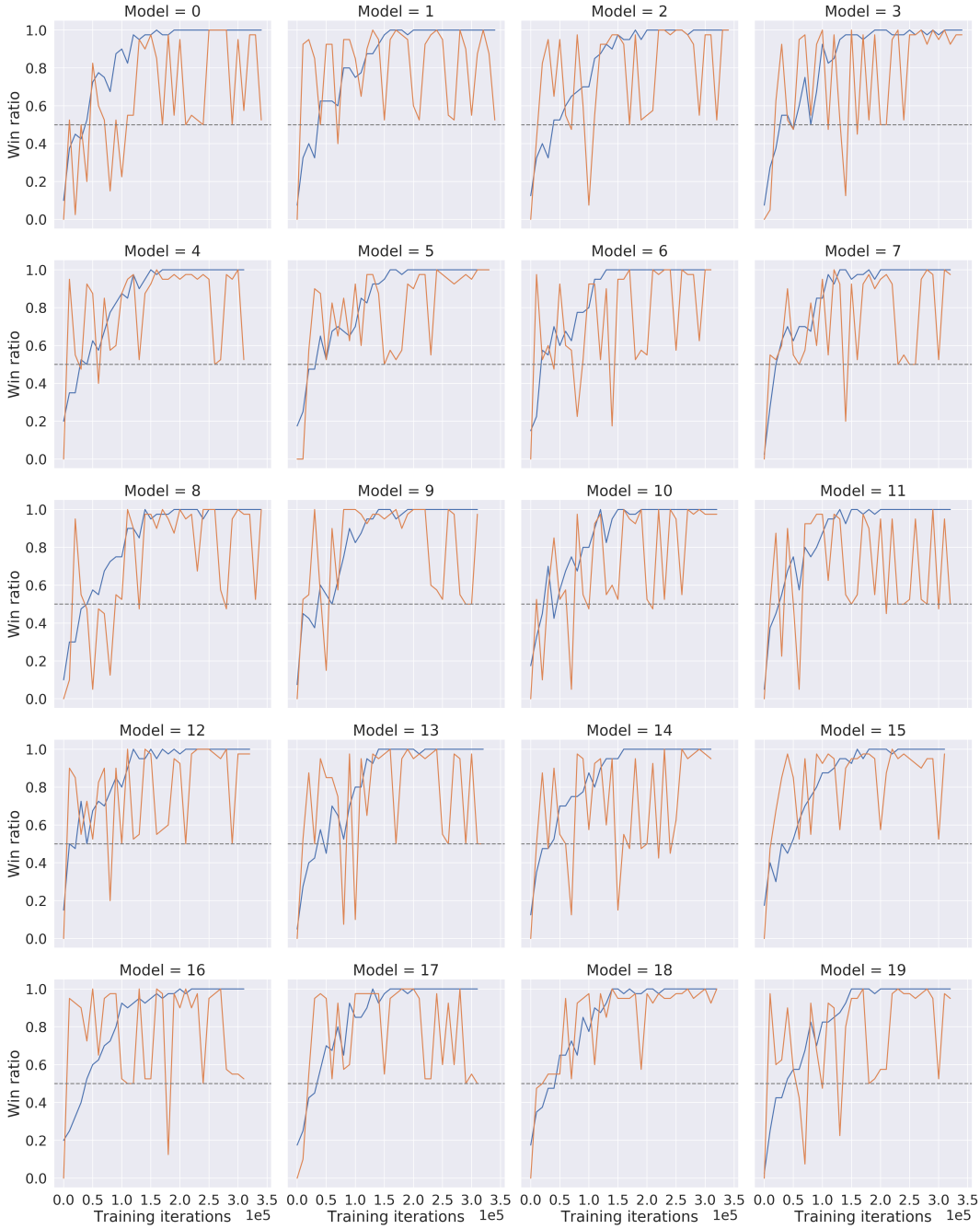


Figure 5.1: The training process for all 20 models in **Hex**, showing how the win ratio evolves over the course of training. Results against both an agent based on random rollouts and on the network from the previous iteration are shown. The dashed line indicates a win ratio of 0.5. Legend: Random rollouts (blue), previous network (red)

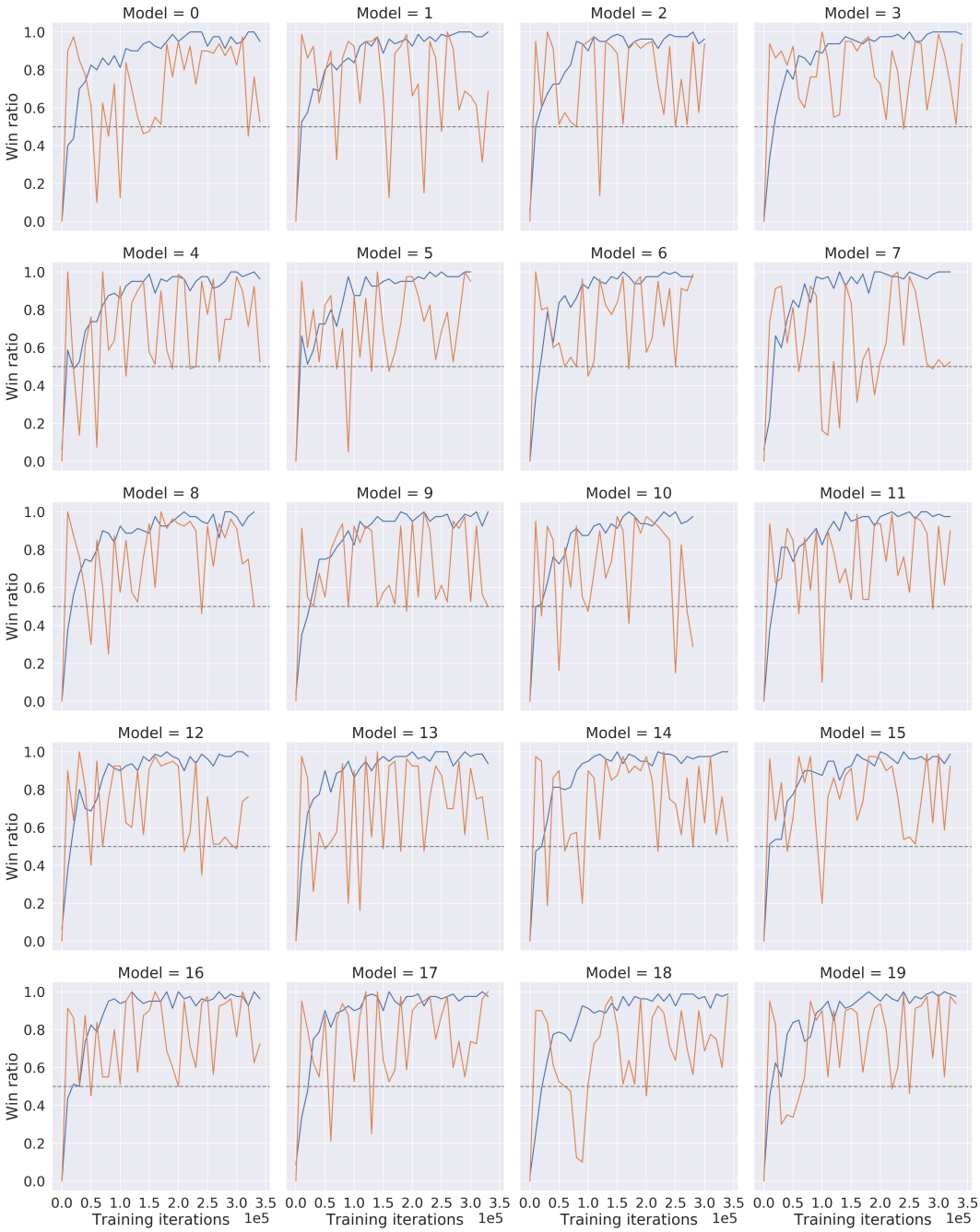
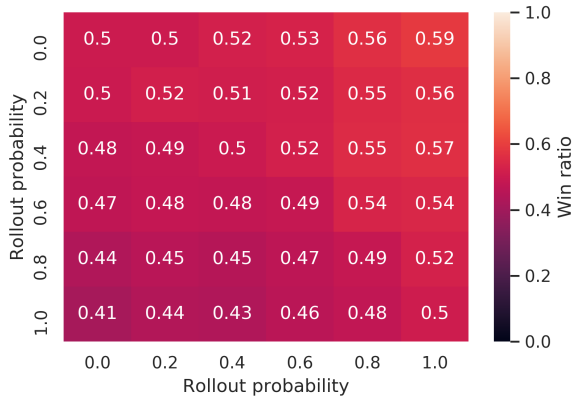
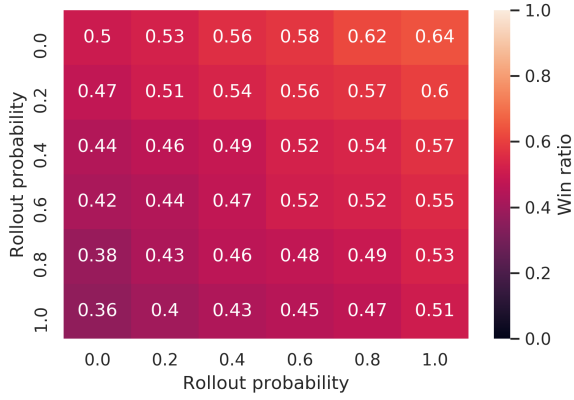


Figure 5.2: The training process for all 20 models in **Othello**, showing how the win ratio evolves over the course of training. Results against both an agent based on random rollouts and on the network from the previous iteration are shown. The dashed line indicates a win ratio of 0.5. Legend: Random rollouts (blue), previous network (red)



(a) Othello



(b) Hex

Figure 5.3: A comparison of win ratios when agents with different rollout probabilities play against each other. The ratio is calculated as $\frac{n_{\text{wins}}+0.5n_{\text{draws}}}{n_{\text{games}}}$, where n_{wins}, n_{draws} and n_{games} are the number of wins, draws and games for the agent on the y-axis against the agent on the x-axis. The results shown are the means across all models.

win	0.28 ± 0.01
draw	0.27 ± 0.01
loss	0.45 ± 0.01

(a) Othello with state evaluator.

win	0.40 ± 0.01
draw	0.00 ± 0.00
loss	0.60 ± 0.01

(b) Hex with state evaluator.

win	0.42 ± 0.01
draw	0.05 ± 0.01
loss	0.53 ± 0.01

(c) Othello without state evaluator.

win	0.32 ± 0.01
draw	0.00 ± 0.00
loss	0.68 ± 0.01

(d) Hex without state evaluator.

Table 5.5: The win, draw and loss ratio for policy network rollouts against random rollouts across all models with 95% confidence intervals, both with and without also using the network as a state evaluator, calculated as $\frac{n_{\text{wins}}}{n_{\text{games}}}$, $\frac{n_{\text{draws}}}{n_{\text{games}}}$ and $\frac{n_{\text{losses}}}{n_{\text{games}}}$. See appendix A for the full data and calculation.

	Rollouts	Simulations
Random rollouts	117	19490
Policy network rollouts	64	19147

(a) Othello with state evaluator.

	Rollouts	Simulations
Random rollouts	157	3146
Policy network rollouts	84	2999

(b) Hex with state evaluator.

	Rollouts	Simulations
Random rollouts	344	19874
Policy network rollouts	59	16637

(c) Othello without state evaluator.

	Rollouts	Simulations
Random rollouts	665	3332
Policy network rollouts	110	1854

(d) Hex without state evaluator.

Table 5.6: The average number of rollouts and simulations per move for the two agents in the different configurations of experiment 3.

Chapter 6

Evaluation and Conclusion

This chapter contains an evaluation and discussion of the results provided in the previous chapter as well as concluding remarks for the work as a whole. It also provides some ideas for potential improvements on the work.

6.1 Evaluation

Looking at the plots for experiment 1 in fig. 5.2, the results seem very promising. First and foremost, all the models trained in Hex reach a 100% win percentage against the opponent using random rollouts. Note that although this opponent uses an extremely basic policy, it's allowed to perform twice as many simulations. Most of the models seem to eventually be stable at the maximum level, although there are some that fluctuate down to 90% late in the process, most notably model 3. All the models feature small fluctuations earlier in the process, but there is nevertheless a clear and relatively steady progress that indicates learning is taking place. The average difference between subsequent values, ignoring those that are stable at 1.0, is about 0.046. If we look at the results against the previous version of the network, we see something far less stable, but this is also to be expected. The improvement from network to network will depend on random factors such as e.g. what new training examples it has received and the current landscape of the loss function. Of course ideally this value would stay at 1.0, indicating a crushing defeat of the previous network each time, but what we at least want to see is for it to stay above 0.5 (shown by the dashed line), because this indicates some improvement. This is also the case most of the time, with any exceptions of a notable size occurring before the midpoint of the training process in all cases. The average ratio is about 0.75. This seems to confirm that there is an improvement over time with learning taking place. It's however interesting

to note how in most of the plots, despite staying above 0.5, it tends to go back and forth between high values close to 1.0 and lower ones close to 0.5 in a fairly steady manner. One possibility is that this is a learning rate problem, where too large steps are being taken in the loss landscape. Another reason might be that them playing ϵ-greedily is simply causing a noticeable variance in the result, although the high number of games should have alleviated this somewhat.

Looking at Othello, we see some of the same tendencies, but none of the models perform as well against the random opponent as was the case for Hex. While all but one model (18) reach 1.0 at some point, the performance is not as stable and continues to fluctuate at a high level for all of them, mostly above 0.9. We still see a clear progression from the early stages though, where they improve quite quickly, even more so than Hex. The average difference between subsequent values, ignoring those that are stable at 1.0, is about 0.031. If we look at the other line we can see that the value mostly stays above 0.5 in later stages, with models 1, 7, and 10 as notable exceptions, though it is quite unstable. This does indicate that some improvement is taking place still, so it is possible that they haven't properly plateaued and if trained for longer could perhaps consistently reach 1.0 against the random opponent. However it might also be that this requires improved parameter tuning. The average ratio is about 0.72. Based on these observations it seems clear that learning is taking place to some extent for Othello as well.

It's possible that the performance being worse for Othello is an indication that the system has more trouble learning this game than Hex. Othello is more complex in some ways, as an example not taking a chance to flip many pieces can easily lead to a missed opportunity with large consequences, while in Hex you can perform the same move on the next turn unless your opponent placed their piece in that exact place. It also needs to learn not only some notion of good and bad moves, but also a more complicated notion of illegal ones than in Hex. But it could just be that the random rollouts are relatively more effective in Othello. After all, there's a much larger tree to explore in Hex, due to its far higher branching factor, so more rollouts are likely necessary for a good estimate than in Othello.

It's worth pointing out that there is a large number of hyperparameters that could be adjusted. I have already mentioned the learning rate, but all of them have some impact on the result, some more than others. Many of these are likely to be far from optimal, as tuning a slow system with many parameters is challenging. If properly set. both the progress during training and the final result would likely look better. I have also been prevented from increasing parameters such as network size and the number of simulations, which likely have a big impact on the performance, and board size, which would bring the games closer to the size that is usually played. These have a large effect on the time it takes to train

the system. Note how AlphaZero used 41/42-layer networks, 800 simulations per move and full game boards and had hyperparameters tuned by Bayesian optimization, which is completely unachievable in my system [Silver et al., 2018].

Another difference from AlphaZero worth noting is the ratio between the number of self-play games and training iterations. By distributing self-play over 5,000 TPUs, AlphaZero generated about 63 (chess), 34 (shogi) and 200 (Go) games per iteration [Silver et al., 2018]. In contrast, my system generated only $25\,\text{processes} \cdot 1000\frac{\text{games}}{\text{process}} = 25\,000$ games, but trained for an average of 230,000 iterations, giving a ratio of 0.11. Even taking into account that their mini-batches were 4 times as large as mine, this means they produced 77 times as many games per iteration. This is despite the fact that in my system training halts while doing evaluation. It's a definite possibility that this extremely high number of iterations compared to games causes the network to continuously overfit the current contents of the replay buffer. This would strongly limit the system's ability to produce well-rounded agents that generalize well. It's however hard to judge if this is the case based on the available data.

The results for experiment 2 in fig. 5.3 can initially seem quite surprising. Looking at each row of the heat map, we can consistently see an improvement in the win ratio as we move to the right. This seems to indicate that the more rollouts are done, the smaller the chance of winning the game. Doing no rollouts at all when playing against an agent that always does rollouts gives a win ratio of 0.59 and 0.64 for Othello and Hex respectively. This seems like a clear advantage. We can see that each time we increase the rollout ratio by 20 percentage points, it seems to fairly consistently decrease the win ratio by one to a few percentage points, with the decrease being larger for Hex. This cannot be caused by the extra rollouts taking longer and thus reducing the time available, because a constant number of simulations were used (arguably an unfair advantage for anyone doing more rollouts). It can seem like the rollout itself is detrimental to the performance. But why would that be?

The problem might stem from the fact that the policy is entirely random. With a constant number of simulations, a rollout is essentially a free opportunity to look at an end state that can be reached from the current state. But the quality of this information does depend on how likely you are of actually ending up in this end state. The outcome of a rollout through a path that is extremely unlikely to occur in actual gameplay is not particularly helpful in guiding later search, and a random rollout might very well take such a path. That is not to say that a random rollout is fundamentally detrimental. We know that random rollouts have the potential to be useful, because random samples are the cornerstone of Monte Carlo methods and MCTS does converge using them. But this depends on taking a large number of random samples, while one might get away with doing much fewer simulations using a smarter policy, or alternatively doing the

same number with a much better result. This is why all the systems reviewed in section 3.3 uses domain-specific heuristics to some extent. Note that there is a trade-off here. A more accurate policy might also be more computationally expensive, but the number of simulations is also important, and realistically you tend to be constrained by time and not by a constant number of simulations.

In any case the key is to recognize the lower value of a random outcome compared to an informed one. This can cause a problem if you're getting both an informed outcome and a random one, which is precisely the case in this experiment. If proper learning has taken place, the value head of the game net will output an estimate of the value of the state that is certainly not random, but informed to some extent. This means it is likely to provide a better estimate of the real value than the outcome of the random rollout. But my system ignores this asymmetry and does a simple unweighted average of the two values, thus treating them as being of equal value. This average is then likely to be further away from the real value of the state than the network estimate. Essentially the rollouts can sabotage what could potentially be a good estimate, and this is a plausible explanation for why my results show rollouts lowering the win ratio. But this also points to how rollouts could be helpful. In a state where the network estimate is too low for example, a random rollout with outcome 1 could in a weighted average nudge the total value closer to the real estimate. The key would be to find a weight that accurately reflects the quality of the information provided by a random rollout. But note that as previously mentioned, random rollouts require many simulations to achieve a good approximation. With few simulations the influence of rollouts on the sum could be inaccurate and prone to random fluctuation. As a side note, this might actually explain why the decrease in performance is greater in Hex. Following the same argument as for experiment 1, Hex has a greater branching factor and state space, possible leading to more inaccurate estimates for the same number of rollouts. With a lower weight the influence of each rollout is smaller, so the influence on the total value would also be quite small, and these inaccuracies might not matter much. However that also means the rollouts might not prove to be advantageous even with an appropriate weighting. Recall that my system was only doing 50 simulations in this experiment, which might be too low for correctly weighted rollouts to have a noticeable positive effect. It's interesting to note that an unweighted average was used in Silver et al. [2016], which could indicate their network estimates and heuristic rollouts were of a similar quality, although this seems surprising.

Judging by table 5.5, it seems like random rollouts are in fact outperforming rollouts using the policy network, regardless of game and whether a state evaluator is also used. The win ratio for the policy network is significantly below 0.5 for Hex in both cases. For Othello we can see a large difference in the win and draw ratio between the two configurations, but the loss ratio is significantly

higher than the win ratio in both cases. But clearly the policy network should make much smarter moves during rollouts than if they are just picked at random, so why are we seeing this result? My hypothesis is that this is mainly based on the number of rollouts each agent is able to perform per move, which is why I have included this in table 5.6. We can see that for both games, about twice the number of rollouts are done per move with the state evaluator and about six times more without it. This is because it is much slower to evaluate the neural network than it is to simply find the legal moves and pick one of them at random. Thus, even though the policy network makes better choices and provides more value per rollout, it cannot do nearly as many, reducing the total value of them. Doing a higher number of less accurate rollouts might ultimately give more accurate estimates near the root, giving a better understanding of what move to perform. The results seem to indicate that this is the case. It is also possible that the agent using a random policy is benefiting from the stochasticity of the rollouts. This will increase exploration compared to the fully deterministic policy network rollouts. This is not inherently an advantage, but very well could be one with such a high number of rollouts, especially if there is insufficient exploration, as noted later.

The differences between Hex and Othello and running with and without a state evaluator are interesting. In the case of Hex, we see that while the policy network rollouts are worse in both configurations, the results are slightly more even with a state evaluator. Intuitively there is some sense in this. Both agents are using the same state evaluator, so this helps even out the differences between them, which is most advantageous for the weaker agent. There is also a much larger difference in the number of rollouts performed by the agents when not using the state evaluator, because this evaluation consumes part of the time budget. This naturally strengthens the random agent relative to the network-based one. We can see the same tendency of evening for Othello. In that case the state evaluator evens out the result by leading to far more draws. But while the agent using policy network rollouts does lose less often using the state evaluator, it also wins less often because games are ending in draws instead. If we score draws as half wins, we get an average score of 0.415 with the state evaluator and 0.445 without, so the latter is arguably better, as opposed to in Hex. Perhaps this could indicate that the network for Othello is biased in some way, with the random rollouts being able to compensate for it and the policy network rollouts being unable to, since it's based on the same network. Interestingly this runs contrary to what you would expect based on the increased number of random rollouts without a state evaluator.

It is worth remarking on the large difference between the number of rollouts and the total number of simulations. In this experiment rollouts were performed on every expansion. This means that a simulation without a rollout can only

occur without an expansion, i.e. if the tree search ends in a true leaf node, one representing the end of the game. Such simulations only need to compute the final outcome of the game, which is fast in itself, but extremely fast due to caching if that final state has been encountered previously. If the tree search consistently ends in such nodes we would therefore expect a large number of simulations and a much lower number of rollouts, which is in fact what the data shows. But this means that the value estimates will be entirely dominated by the values backpropagated from actual outcomes as opposed to rollouts. Initially this might sound like a good thing, because actual outcomes are clearly more accurate than ones simulated by a rollout. This might however indicate a problem with lack of exploration. We are not interested in backing up an actual outcome again and again from the same node when we instead could be exploring other branches. The close ancestors of that node will already have an estimate close to it and will not change much by propagating back the same value yet again. In fact if MCTS only explores a few branches until the end of the game and then subsequently mostly searches these, the lack of exploration will tend to make values near the root very inaccurate. If this is the case, it is likely beneficial to increase the exploration constant c_{PUCT}. But note that these averages are based on all moves, including those close to the end of games. In those cases many or most branches will quickly lead to final states, which brings up the average. Based on this it cannot be concluded that there is a lack of exploration.

As a side note, some informal testing actually shows that the difference in computation time between the two policies is much larger than one would expect from the number seen, about two orders of magnitude. I believe there to be a couple of reasons for this discrepancy. One is that caching is still used within each agent (though, like mentioned previously, not between games or agents). Network evaluations benefit immensely from caching, becoming about three orders of magnitude faster, while finding legal states is only about one order of magnitude faster when cached. This means that if states are often repeated between rollouts, we can expect the difference between network-based rollouts and random rollouts to be far less than two orders of magnitude. The fact that the policy network rollouts are deterministic will also make repeated states more likely, increasing the benefit of caching. There is also overhead from e.g. the tree search, expansion and backpropagation that spends some of the limited time budget.

6.2 Conclusion

When I first heard about AlphaGo, I found it massively impressive. Here was a game, Go, that had eluded attempts to master it using conventional game-playing techniques and that few expected computers to beat human professionals at for a long time. Then almost out of the blue came AlphaGo, a novel combination of

reinforcement learning, deep learning and Monte Carlo tree search, and beat one of the champions of the game. Then came AlphaGo Zero, the undisputed world champion of Go, learning the game completely from scratch, and I was in awe. When I saw news of AlphaZero, showing the technique could be generalized to other games, I knew this was something I wanted to look deeper into. I read Silver et al. [2016], Silver et al. [2017], Silver et al. [2018] and found them extremely inspiring. This led me to my research goal of investigating this novel combination.

As seen in chapter 3, I then used these papers as a springboard into the relevant fields of research, attempting to gain a thorough understanding of both past work and the state-of-the-art. I looked into alternative techniques such as DQN, which clearly showed the power of deep convolutional function approximators for reinforcement learning. I looked at the history of Monte Carlo tree search from its early conception by Coulom [2006], to the addition of UCT by Kocsis and Szepesvári [2006] and on through various improvements that led to the first revolution in Go AIs. But these could still only match human professionals for smaller-scale boards. I looked at attempts at applying MCTS to other games in the form of Hingston and Masek [2007] and Arneson et al. [2010], which showed the technique could be used more generally. I found Maddison et al. [2014], coming a couple of years after convolutional neural networks were popularized by Krizhevsky et al. [2012] and serving as a sort of predecessor to AlphaGo and its successors by combining MCTS and deep CNNs. Finally I looked into Anthony et al. [2017], an independent work inspired by AlphaGo and similar to AlphaZero, but providing a somewhat alternative approach.

At this point, I had a research goal and a good understanding of the field, but despite the process being underway, it was still not clear to me exactly which questions I should research. But I went back to the three main papers and thought about the various things that set these systems apart from earlier work. As I talked about in section 3.8, what struck me was how rollouts were a central part of all the previous work in MCTS and even played an important role in AlphaGo, while being completely absent from its successors. This made me think about whether rollouts could still be beneficial in these systems. If they were, could it be an idea to use the same network trained as part of the algorithm to perform rollouts, instead of a faster but simpler policy? These two questions then became my research goals.

Because my project is based on an experimental methodology, researching these questions meant building a system and designing experiments that would allow me to get empirical data. I set out to build a configurable, flexible and modular framework that would allow me to apply this general combination of MCTS, deep learning and reinforcement learning to various games without mixing game-specific details with the core algorithm. This has been accomplished. As seen in chapter 4, it is a system with four principle components. To start learning

a new game from scratch, all that is needed is to implement a game manager representing the rules of the game and a game net taking the states of the game and feeding them through a neural network. Neither MCTS nor the overall training algorithm has any knowledge of what game they are working on. This has made it extremely easy to apply the system to Hex and Othello. The system is similar to AlphaZero in many ways. One key difference is in how it parallelizes and distributes training. Another is that while AlphaZero has no concept of rollouts, my system allows a very flexible application of them, configuring how often and exactly how to perform them. This was key to investigating my research goals.

One problem I have touched upon multiple times throughout this document is the speed of the system. A lot of time was spent working on this, perhaps even too much. But while it is still an issue in the final system, it is massively improved from earlier iterations of it. The key solutions were caching and distributing self-play over multiple processes and training over multiple GPUs, which have all provided massive speed-ups. I was also greatly helped by the access to hardware provided by the department (but note the massive disparity between this hardware and what used for Silver et al. [2018]). Without these improvements it would have been difficult to perform the experiments I have, and it has taught me a lot about scaling machine learning systems.

Based on my research goals I designed three experiments that I hoped could help shed light on them. As seen in section 5.1, these were meant to confirm whether learning was taking place in the system, whether performing random rollouts more often would lead to better results and whether rollouts based on the policy network would outperform random rollouts. These experiments were performed using the system and the results have been presented in section 5.3 and thoroughly discussed in section 6.1. What can these results tell us about the system and the research questions?

Experiment 1 indicates that the system is to some extent able to learn to play both Hex and Othello from scratch and consistently beat random rollout MCTS with twice the number of simulations. This is not a hugely surprising result. As we have seen from both Silver et al. [2018] and Anthony et al. [2017], the general technique seems to generalize well to games other than Go, including Hex. It does serve as a confirmation that the system works and can be used for further experiments. Because the resulting agents have not been compared to a known baseline such as a previously published algorithm, it is difficult to say exactly how well the agents have learned to play. There are some instabilities in the result that could point to a suboptimal hyperparameters and additionally parameters such as network size, board size and simulations per move have been kept low to limit the runtime. This should be kept in mind, as my other results don't necessarily generalize to a well-tuned and more complex configuration. By

comparison with Silver et al. [2018], there is also suspicion that the number of self-play games are too low compared to the number of training iterations, but it is not clear from the results whether this is actually a problem.

Experiment 2 seemed at first to suggest that the answer to my first research question was no, rollouts are in fact detrimental to the performance of the algorithm. But as discussed in the evaluation, the answer is not so clear-cut. It is in hindsight to be expected for random rollouts to decrease performance when combined with a low number of simulations and an unweighted average. This does not give us a reason to conclude that this combination of network evaluation and random rollout should be avoided in general. It does however hint at the relatively low value of a random rollout, which might mean that any potential benefit is small, but this is again not clear from these results. Experiment 2 has not given us a clear answer to the research question.

When it comes to the second research question, the corresponding experiment is slightly more helpful. The results of experiment 3 indicate that using the policy network as a rollout policy does not improve on random rollouts, likely because it is too slow and therefore results in a far lower number of rollouts, perhaps also because it leads to reduced exploration. But there are caveats which mean we cannot conclude this with certainty. For example, a network that is trained with better tuning of hyperparameters or simply for longer could result in better results. Note that a larger network would be even slower, so that is less likely to be helpful. It is also not clear whether an improvement on random rollouts would be an improvement on not performing rollouts at all. This is of course related to research question 1. But instead of doing a rollout, which requires many evaluations of the network, it is not inconceivable that you are better off simply performing more simulations, which only require a maximum of one evaluation each.

Experiment 3 also highlighted a potential problem with lack of exploration due to many searches ending in final states. The data is not conclusive due to the caveats mentioned in the evaluation, but if this is the case, it does affect all three experiments. It is hard to say exactly how this issue would influence the results, but it could possibly give an advantage to random rollouts, as they are inherently more explorative.

These results do raise the question of whether other kinds of rollout policies could be more useful. A random policy is completely uninformed, while a deep policy network is extremely slow. These are large disadvantages which might make them unsuitable as rollout policies. If they are, perhaps there are other policies which could be a better fit. But the big advantage of these policies is that they still allow you to learn from scratch without additional domain knowledge. Perhaps using a domain-specific rollout policy, such as in Silver et al. [2016], could be beneficial to create a better player, but the loss of generality would make this

a less interesting result.

In the end, the contributions of this work aren't major. We still don't know whether it was optimal for Silver et al. [2017, 2018] to not include rollouts, this technique that was still an important aspect of Silver et al. [2016] and had been so central to the previous work with MCTS. There are some indications that random rollouts might not provide much benefit and better indications that policy network rollouts are worse. The results are nevertheless too inconclusive to say this for sure. But the work has resulted in a functional system that could be used to investigate the issue of rollouts further, either with improved experiments that could give more conclusive data or alternative ones that could provide insight into new questions that have been raised.

6.3 Future Work

Some changes could be made to the system to speed it up further. One possibility is to implement threaded MCTS in e.g. C++. This would allow searches in parallel within each self-play game and batched network evaluation without the overhead of processes. If the system could be distributed across more GPUs, this would also be helpful in increasing the ratio between self-play games and training iterations. It could for example be modified to run on NTNU's Idun cluster. With a more computationally performant system, it would then be easier to test various hyperparameters to properly tune the system. This could also allow for larger networks, bigger boards, more simulations and training the networks for longer.

To properly assess how well the system learns, the resulting agents could be compared to those of prior work in the field, e.g. MoHex. This could involve straight comparisons of win ratios, but also a deeper analysis of playstyles to identify strengths and weaknesses.

To get more conclusive data on the first research question, the number of simulations could be increased and various weightings could be tested in experiment 2 to see any of them made random rollouts beneficial. For the second research question, experiment 3 could be repeated with a better trained network and a fine-tuning of parameters such as the exploration constant. An agent without rollouts could also be included in the comparison. Improving these experiments would be far more doable if the speed of the system was improved beforehand.

If an alternative, novel rollout policy that both preserves generality and improves performance could be formulated, this would be a large contribution to the field. This might require e.g. a deep study of prior art to see if there has been related work in general policies that could be applicable.

Bibliography

Allis, L. (1994). *Searching for solutions in games and artificial intelligence*. PhD thesis, Maastricht University.

Anthony, T., Tian, Z., and Barber, D. (2017). Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370.

Arneson, B., Hayward, R. B., and Henderson, P. (2010). Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258.

Bouzy, B. (2004). Associating shallow and selective global tree search with monte carlo for 9× 9 go. In *International Conference on Computers and Games*, pages 67–80. Springer.

Browne, C. (2000). *Hex Strategy: Making the Right Connections*. Ak Peters Series. Taylor & Francis.

Campbell, M., Hoane Jr, A. J., and Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1-2):57–83.

Clark, C. and Storkey, A. (2015). Training deep convolutional neural networks to play go. In *International conference on machine learning*, pages 1766–1774.

Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer.

Enzenberger, M., Muller, M., Arneson, B., and Segal, R. (2010). Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270.

Gale, D. (1979). The game of hex and the brouwer fixed-point theorem. *The American Mathematical Monthly*, 86(10):818–827.

Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., and Teytaud, O. (2012). The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113.

Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM.

Gelly, S. and Silver, D. (2008). Achieving master level play in 9 x 9 computer go. In *AAAI*, volume 8, pages 1537–1540.

Gelly, S. and Silver, D. (2011). Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hingston, P. and Masek, M. (2007). Experiments with monte carlo othello. In *2007 IEEE Congress on Evolutionary Computation*, pages 4059–4064. IEEE.

Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Landau, T. (1985). Othello: Brief & basic. *US Othello Association*, 920:22980–23425.

Lin, L.-J. (1993). Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Lundgaard, N. and McKee, B. (2006). Reinforcement learning and neural networks for tetris. Technical report, University of Oklahoma.

Maddison, C. J., Huang, A., Sutskever, I., and Silver, D. (2014). Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

Nash, J. (1952). Some games and machines for playing them. Technical Report D-1164, Rand Corp.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

Python contributors (2019a). Pipes and queues. `https://docs.python.org/3.8/library/multiprocessing.html#pipes-and-queues`. [Online; accessed 13-January-2020].

Python contributors (2019b). Thread state and the global interpreter lock. `https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock`. [Online; accessed 13-January-2020].

PyTorch contributors (2019). Multiprocessing best practices. `https://pytorch.org/docs/stable/notes/multiprocessing.html`. [Online; accessed 22-January-2020].

Rosin, C. D. (2011). Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230.

Ross, S., Gordon, G., and Bagnell, D. (2011). A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635.

Russell, S. J. and Norvig, P. (2010). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Wang, Y. and Gelly, S. (2007). Modifications of uct and sequence-like simulations for monte-carlo go. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 175–182. IEEE.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Appendix A

Extended Results

	first		second		overall	
	win	draw	win	draw	win	draw
Model						
0	0.00 ± 0.00	0.01 ± 0.02	0.89 ± 0.06	0.08 ± 0.05	0.45 ± 0.06	0.05 ± 0.03
1	0.24 ± 0.08	0.18 ± 0.07	0.22 ± 0.07	0.28 ± 0.08	0.23 ± 0.05	0.23 ± 0.05
2	0.01 ± 0.02	0.00 ± 0.00	0.91 ± 0.05	0.04 ± 0.04	0.46 ± 0.06	0.02 ± 0.02
3	0.00 ± 0.00	0.02 ± 0.02	0.06 ± 0.04	0.17 ± 0.07	0.03 ± 0.02	0.09 ± 0.04
4	0.01 ± 0.02	0.18 ± 0.07	0.49 ± 0.09	0.44 ± 0.09	0.25 ± 0.05	0.31 ± 0.06
5	0.00 ± 0.00	0.00 ± 0.00	0.36 ± 0.09	0.42 ± 0.09	0.18 ± 0.05	0.21 ± 0.05
6	0.08 ± 0.05	0.06 ± 0.04	0.53 ± 0.09	0.00 ± 0.00	0.30 ± 0.06	0.03 ± 0.02
7	0.00 ± 0.00	0.06 ± 0.04	0.89 ± 0.06	0.03 ± 0.03	0.45 ± 0.06	0.05 ± 0.03
8	0.02 ± 0.02	0.94 ± 0.04	0.00 ± 0.00	0.98 ± 0.02	0.01 ± 0.01	0.96 ± 0.02
9	0.07 ± 0.05	0.03 ± 0.03	0.91 ± 0.05	0.04 ± 0.04	0.49 ± 0.06	0.03 ± 0.02
10	0.02 ± 0.02	0.01 ± 0.02	0.42 ± 0.09	0.43 ± 0.09	0.22 ± 0.05	0.22 ± 0.05
11	0.00 ± 0.00	1.00 ± 0.00	0.00 ± 0.00	0.98 ± 0.02	0.00 ± 0.00	0.99 ± 0.01
12	0.05 ± 0.04	0.00 ± 0.00	0.80 ± 0.07	0.01 ± 0.02	0.43 ± 0.06	0.00 ± 0.01
13	0.22 ± 0.07	0.65 ± 0.09	0.64 ± 0.09	0.09 ± 0.05	0.43 ± 0.06	0.37 ± 0.06
14	0.53 ± 0.09	0.33 ± 0.08	0.50 ± 0.09	0.11 ± 0.06	0.51 ± 0.06	0.22 ± 0.05
15	0.00 ± 0.00	0.00 ± 0.00	0.93 ± 0.04	0.06 ± 0.04	0.47 ± 0.06	0.03 ± 0.02
16	0.15 ± 0.06	0.38 ± 0.09	0.50 ± 0.09	0.19 ± 0.07	0.33 ± 0.06	0.29 ± 0.06
17	0.00 ± 0.00	0.98 ± 0.02	0.00 ± 0.00	0.80 ± 0.07	0.00 ± 0.00	0.89 ± 0.04
18	0.16 ± 0.07	0.07 ± 0.04	0.09 ± 0.05	0.47 ± 0.09	0.12 ± 0.04	0.27 ± 0.06
19	0.06 ± 0.04	0.10 ± 0.05	0.42 ± 0.09	0.25 ± 0.08	0.24 ± 0.05	0.17 ± 0.05
Total	0.08 ± 0.01	0.25 ± 0.02	0.48 ± 0.02	0.29 ± 0.02	0.28 ± 0.01	0.27 ± 0.01

Table A.1: The win and draw ratio for policy network rollouts against random rollouts with 95% confidence intervals in **Othello with a state evaluator**. Note that the loss ratio is not shown, but it's fixed given the two others. Values are shown separately both by model and by whether the agent using policy network rollouts played first or second, thus those values are based on 20 matches. In addition there are values given across the starting player (240 matches) and across the models (2400 matches). The values across both variables are based on all 4800 matches (these are also showed in table 5.5). In each case there are two outcomes ("win" or "draw/loss" and "draw" or "win/loss"), meaning that they form Bernoulli distributions. The variance of the estimator is therefore calculated as $\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}$, where $\hat{p}$ is the maximum likelihood estimator $\frac{n_{\text{wins}}}{n_{\text{games}}}$ and $\frac{n_{\text{draws}}}{n_{\text{games}}}$ for the win and draw ratio respectively. The confidence intervals are calculated as $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}}$. Note that this relies on the assumption that the sum of independent identically distributed Bernoulli random variables approach a normal distribution. This is most reliable when n is large and p isn't close to 0 or 1.

Model	first win	first draw	second win	second draw	overall win	overall draw
0	0.84 ± 0.07	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.42 ± 0.06	0.00 ± 0.00
1	0.98 ± 0.02	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.49 ± 0.06	0.00 ± 0.00
2	0.00 ± 0.00	0.00 ± 0.00	0.98 ± 0.02	0.00 ± 0.00	0.49 ± 0.06	0.00 ± 0.00
3	0.22 ± 0.07	0.00 ± 0.00	0.88 ± 0.06	0.00 ± 0.00	0.55 ± 0.06	0.00 ± 0.00
4	0.62 ± 0.09	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.31 ± 0.06	0.00 ± 0.00
5	1.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.50 ± 0.06	0.00 ± 0.00
6	0.17 ± 0.07	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.08 ± 0.03	0.00 ± 0.00
7	0.01 ± 0.02	0.00 ± 0.00	0.50 ± 0.09	0.00 ± 0.00	0.25 ± 0.06	0.00 ± 0.00
8	0.97 ± 0.03	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.48 ± 0.06	0.00 ± 0.00
9	0.67 ± 0.08	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.34 ± 0.06	0.00 ± 0.00
10	0.93 ± 0.05	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.46 ± 0.06	0.00 ± 0.00
11	0.67 ± 0.08	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.33 ± 0.06	0.00 ± 0.00
12	0.23 ± 0.08	0.00 ± 0.00	0.17 ± 0.07	0.00 ± 0.00	0.20 ± 0.05	0.00 ± 0.00
13	0.00 ± 0.00	0.00 ± 0.00	0.31 ± 0.08	0.00 ± 0.00	0.15 ± 0.05	0.00 ± 0.00
14	0.50 ± 0.09	0.00 ± 0.00	0.03 ± 0.03	0.00 ± 0.00	0.27 ± 0.06	0.00 ± 0.00
15	0.93 ± 0.04	0.00 ± 0.00	0.97 ± 0.03	0.00 ± 0.00	0.95 ± 0.03	0.00 ± 0.00
16	0.63 ± 0.09	0.00 ± 0.00	0.32 ± 0.08	0.00 ± 0.00	0.47 ± 0.06	0.00 ± 0.00
17	1.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.50 ± 0.06	0.00 ± 0.00
18	0.97 ± 0.03	0.00 ± 0.00	0.16 ± 0.07	0.00 ± 0.00	0.57 ± 0.06	0.00 ± 0.00
19	0.45 ± 0.09	0.00 ± 0.00	0.02 ± 0.02	0.00 ± 0.00	0.23 ± 0.05	0.00 ± 0.00
Total	0.59 ± 0.02	0.00 ± 0.00	0.22 ± 0.02	0.00 ± 0.00	0.40 ± 0.01	0.00 ± 0.00

Table A.2: The win and draw ratio for policy network rollouts against random rollouts with 95% confidence intervals in **Hex with a state evaluator**. Note that the loss ratio is not shown, but it's fixed given the two others. Values are shown separately both by model and by whether the agent using policy network rollouts played first or second, thus those values are based on 20 matches. In addition there are values given across the starting player (240 matches) and across the models (2400 matches). The values across both variables are based on all 4800 matches (these are also showed in table 5.5). In each case there are two outcomes ("win" or "draw/loss" and "draw" or "win/loss"), meaning that they form Bernoulli distributions. The variance of the estimator is therefore calculated as $\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}$, where $\hat{p}$ is the maximum likelihood estimator $\frac{n_{\text{wins}}}{n_{\text{games}}}$ and $\frac{n_{\text{draws}}}{n_{\text{games}}}$ for the win and draw ratio respectively. The confidence intervals are calculated as $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}}$. Note that this relies on the assumption that the sum of independent identically distributed Bernoulli random variables approach a normal distribution. This is most reliable when n is large and p isn't close to 0 or 1.

Model	first win	first draw	second win	second draw	overall win	overall draw
0	0.43 ± 0.09	0.03 ± 0.03	0.50 ± 0.09	0.09 ± 0.05	0.47 ± 0.06	0.06 ± 0.03
1	0.47 ± 0.09	0.06 ± 0.04	0.45 ± 0.09	0.04 ± 0.04	0.46 ± 0.06	0.05 ± 0.03
2	0.43 ± 0.09	0.03 ± 0.03	0.38 ± 0.09	0.03 ± 0.03	0.41 ± 0.06	0.03 ± 0.02
3	0.42 ± 0.09	0.07 ± 0.04	0.41 ± 0.09	0.02 ± 0.02	0.41 ± 0.06	0.04 ± 0.03
4	0.38 ± 0.09	0.05 ± 0.04	0.38 ± 0.09	0.03 ± 0.03	0.38 ± 0.06	0.04 ± 0.02
5	0.35 ± 0.09	0.09 ± 0.05	0.41 ± 0.09	0.03 ± 0.03	0.38 ± 0.06	0.06 ± 0.03
6	0.33 ± 0.08	0.08 ± 0.05	0.49 ± 0.09	0.04 ± 0.04	0.41 ± 0.06	0.06 ± 0.03
7	0.40 ± 0.09	0.04 ± 0.04	0.43 ± 0.09	0.03 ± 0.03	0.42 ± 0.06	0.04 ± 0.02
8	0.37 ± 0.09	0.04 ± 0.04	0.43 ± 0.09	0.07 ± 0.05	0.40 ± 0.06	0.06 ± 0.03
9	0.48 ± 0.09	0.05 ± 0.04	0.41 ± 0.09	0.07 ± 0.04	0.44 ± 0.06	0.06 ± 0.03
10	0.48 ± 0.09	0.04 ± 0.04	0.40 ± 0.09	0.06 ± 0.04	0.44 ± 0.06	0.05 ± 0.03
11	0.43 ± 0.09	0.06 ± 0.04	0.40 ± 0.09	0.07 ± 0.04	0.42 ± 0.06	0.06 ± 0.03
12	0.40 ± 0.09	0.04 ± 0.04	0.42 ± 0.09	0.03 ± 0.03	0.41 ± 0.06	0.03 ± 0.02
13	0.44 ± 0.09	0.05 ± 0.04	0.46 ± 0.09	0.03 ± 0.03	0.45 ± 0.06	0.04 ± 0.03
14	0.45 ± 0.09	0.05 ± 0.04	0.43 ± 0.09	0.05 ± 0.04	0.44 ± 0.06	0.05 ± 0.03
15	0.45 ± 0.09	0.04 ± 0.04	0.42 ± 0.09	0.06 ± 0.04	0.44 ± 0.06	0.05 ± 0.03
16	0.48 ± 0.09	0.04 ± 0.04	0.46 ± 0.09	0.07 ± 0.05	0.47 ± 0.06	0.06 ± 0.03
17	0.42 ± 0.09	0.04 ± 0.04	0.40 ± 0.09	0.08 ± 0.05	0.41 ± 0.06	0.06 ± 0.03
18	0.45 ± 0.09	0.03 ± 0.03	0.40 ± 0.09	0.07 ± 0.04	0.43 ± 0.06	0.05 ± 0.03
19	0.32 ± 0.08	0.05 ± 0.04	0.45 ± 0.09	0.07 ± 0.04	0.38 ± 0.06	0.06 ± 0.03
Total	0.42 ± 0.02	0.05 ± 0.01	0.43 ± 0.02	0.05 ± 0.01	0.42 ± 0.01	0.05 ± 0.01

Table A.3: The win and draw ratio for policy network rollouts against random rollouts with 95% confidence intervals in **Othello without a state evaluator**. Note that the loss ratio is not shown, but it's fixed given the two others. Values are shown separately both by model and by whether the agent using policy network rollouts played first or second, thus those values are based on 20 matches. In addition there are values given across the starting player (240 matches) and across the models (2400 matches). The values across both variables are based on all 4800 matches (these are also showed in table 5.5). In each case there are two outcomes ("win" or "draw/loss" and "draw" or "win/loss"), meaning that they form Bernoulli distributions. The variance of the estimator is therefore calculated as $\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}$, where $\hat{p}$ is the maximum likelihood estimator $\frac{n_{\text{wins}}}{n_{\text{games}}}$ and $\frac{n_{\text{draws}}}{n_{\text{games}}}$ for the win and draw ratio respectively. The confidence intervals are calculated as $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}}$. Note that this relies on the assumption that the sum of independent identically distributed Bernoulli random variables approach a normal distribution. This is most reliable when n is large and p isn't close to 0 or 1.

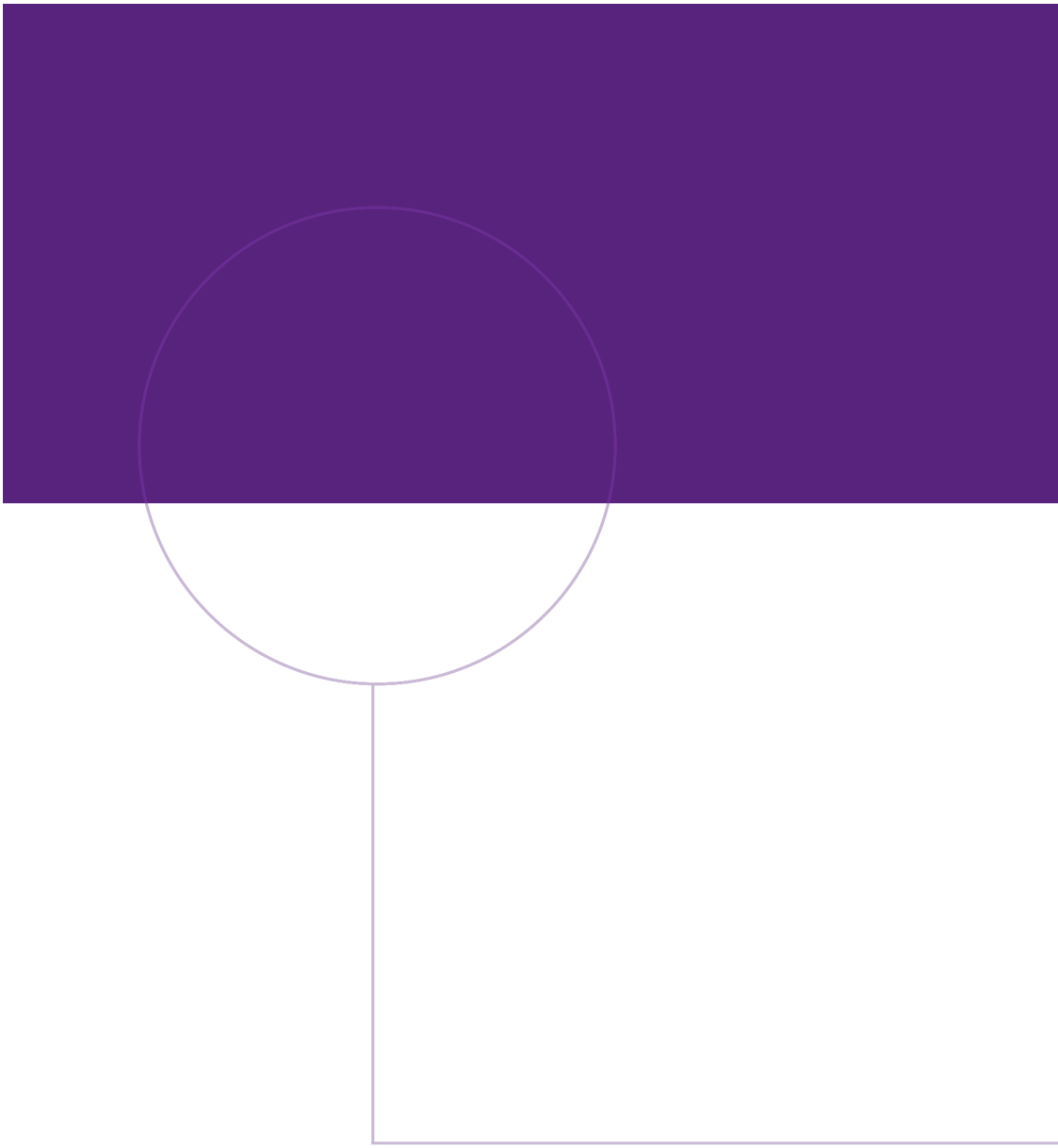
Model	first win	first draw	second win	second draw	overall win	overall draw
0	0.38 ± 0.09	0.00 ± 0.00	0.35 ± 0.09	0.00 ± 0.00	0.36 ± 0.06	0.00 ± 0.00
1	0.47 ± 0.09	0.00 ± 0.00	0.25 ± 0.08	0.00 ± 0.00	0.36 ± 0.06	0.00 ± 0.00
2	0.31 ± 0.08	0.00 ± 0.00	0.23 ± 0.08	0.00 ± 0.00	0.27 ± 0.06	0.00 ± 0.00
3	0.31 ± 0.08	0.00 ± 0.00	0.28 ± 0.08	0.00 ± 0.00	0.29 ± 0.06	0.00 ± 0.00
4	0.37 ± 0.09	0.00 ± 0.00	0.39 ± 0.09	0.00 ± 0.00	0.38 ± 0.06	0.00 ± 0.00
5	0.31 ± 0.08	0.00 ± 0.00	0.35 ± 0.09	0.00 ± 0.00	0.33 ± 0.06	0.00 ± 0.00
6	0.26 ± 0.08	0.00 ± 0.00	0.33 ± 0.08	0.00 ± 0.00	0.30 ± 0.06	0.00 ± 0.00
7	0.40 ± 0.09	0.00 ± 0.00	0.33 ± 0.08	0.00 ± 0.00	0.37 ± 0.06	0.00 ± 0.00
8	0.22 ± 0.07	0.00 ± 0.00	0.27 ± 0.08	0.00 ± 0.00	0.24 ± 0.05	0.00 ± 0.00
9	0.15 ± 0.06	0.00 ± 0.00	0.11 ± 0.06	0.00 ± 0.00	0.13 ± 0.04	0.00 ± 0.00
10	0.47 ± 0.09	0.00 ± 0.00	0.37 ± 0.09	0.00 ± 0.00	0.42 ± 0.06	0.00 ± 0.00
11	0.28 ± 0.08	0.00 ± 0.00	0.41 ± 0.09	0.00 ± 0.00	0.34 ± 0.06	0.00 ± 0.00
12	0.34 ± 0.08	0.00 ± 0.00	0.27 ± 0.08	0.00 ± 0.00	0.31 ± 0.06	0.00 ± 0.00
13	0.35 ± 0.09	0.00 ± 0.00	0.32 ± 0.08	0.00 ± 0.00	0.33 ± 0.06	0.00 ± 0.00
14	0.40 ± 0.09	0.00 ± 0.00	0.23 ± 0.08	0.00 ± 0.00	0.32 ± 0.06	0.00 ± 0.00
15	0.35 ± 0.09	0.00 ± 0.00	0.26 ± 0.08	0.00 ± 0.00	0.30 ± 0.06	0.00 ± 0.00
16	0.39 ± 0.09	0.00 ± 0.00	0.31 ± 0.08	0.00 ± 0.00	0.35 ± 0.06	0.00 ± 0.00
17	0.50 ± 0.09	0.00 ± 0.00	0.42 ± 0.09	0.00 ± 0.00	0.46 ± 0.06	0.00 ± 0.00
18	0.26 ± 0.08	0.00 ± 0.00	0.35 ± 0.09	0.00 ± 0.00	0.30 ± 0.06	0.00 ± 0.00
19	0.35 ± 0.09	0.00 ± 0.00	0.26 ± 0.08	0.00 ± 0.00	0.30 ± 0.06	0.00 ± 0.00
Total	0.34 ± 0.02	0.00 ± 0.00	0.30 ± 0.02	0.00 ± 0.00	0.32 ± 0.01	0.00 ± 0.00

Table A.4: The win and draw ratio for policy network rollouts against random rollouts with 95% confidence intervals in **Hex without a state evaluator**. Note that the loss ratio is not shown, but it's fixed given the two others. Values are shown separately both by model and by whether the agent using policy network rollouts played first or second, thus those values are based on 20 matches. In addition there are values given across the starting player (240 matches) and across the models (2400 matches). The values across both variables are based on all 4800 matches (these are also showed in table 5.5). In each case there are two outcomes ("win" or "draw/loss" and "draw" or "win/loss"), meaning that they form Bernoulli distributions. The variance of the estimator is therefore calculated as $\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}$, where $\hat{p}$ is the maximum likelihood estimator $\frac{n_{\text{wins}}}{n_{\text{games}}}$ and $\frac{n_{\text{draws}}}{n_{\text{games}}}$ for the win and draw ratio respectively. The confidence intervals are calculated as $\hat{p} \pm 1.96\sqrt{\frac{\hat{p}(1-\hat{p})}{n_{\text{games}}}}$. Note that this relies on the assumption that the sum of independent identically distributed Bernoulli random variables approach a normal distribution. This is most reliable when n is large and p isn't close to 0 or 1.

Appendix B

Source Code, Models and Raw Data

The source code of the system, trained models and raw data of the experiments can be found in this GitHub repository: `https://github.com/henribru/deep-mcts`.



NTNU

Norwegian University of
Science and Technology