General Game Playing with pUCT and Deep RL

Cosmo Bobak

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

We produce a general game playing system based on deep neural networks and an enhanced form of Monte-Carlo Tree Search that learns and improves by training against itself, and apply it to a novel pair of board games - Gomoku and Ataxx. Our system is designed to be general - it requires no domain-specific knowledge to apply to new games.

We evaluate this system against human players and a set of algorithmic baselines, and show that our system produces agents that learn to play these games to a high standard, surpassing the performance of the human players and our benchmarks and demonstrating the system's general ability to learn to play new games.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee. Relevant information can be found in the Appendix.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Cosmo Bobak*)

Acknowledgements

I would like to thank my supervisor, Dr. Kobi Gal, for his wise guidance and support throughout this project, my friends and family for their encouragement and incredible support, the School of Informatics for providing the resources necessary to complete this project, and particularly the many intelligent and shockingly patient friends I have made in the game programming community, without whom this project would not have been possible.

Table of Contents

Chapter 1

Introduction

This chapter will introduce the reader to the problem this paper addresses, and the solution that we propose.

1.1 Motivation and Problem Statement

While strong programs for popular games like Chess (Disservin, 2024) and Go (Wu, 2019) exist, less popular games may not have enjoyed corresponding levels of effort developing dedicated AI programs for evaluation, game review, and practice. Prior work (Silver et al., 2018) demonstrates that a general system for learning to play board games is possible, but focuses strictly on popular games, and many subsequent replications and improvements have not applied such a system to new games (Tian et al., 2019; Wu, 2019).

1.2 Contributions

In this project we present a general game playing framework for creating AI programs for playing board games, designed to achieve high performance in novel games with minimal adaption, and we train strong agents for the games of Gomoku and Ataxx[1], which are interesting for their complexity, strategic depth, and relative lack of previous work exploring them.

Our main contributions are the following:

- An open-source framework for training strong agents in arbitrary two-player perfect information[2] games, the main component of which is our game engine, **Veritas**, a sub-system that contains an algorithm for finding strong moves and a self-play tournament handler for training the system against itself.

[1] We describe these games in the Appendix.

[2] Perfect information games are games where all strategically relevant information is available to all players at all times - examples include Chess, Go, and Checkers.

- Two high-performance open-source board game libraries - we develop **Gomokugen**, a library for the game of Gomoku, and **Ataxxgen**, a library for the game of Ataxx. These libraries handle move generation, game state representation, serialisation, and encodings for tensor representations of actions and states for use in machine learning.
- Trained agents for both games, alongside evaluation and analysis of their performance against both humans and algorithmic benchmarks.

Chapter 2

Background

This thesis relates to work in the domains of game-tree search, deep neural networks, and reinforcement learning, specifically the problem of creating artificial agents that can play abstract strategy games. In this section we discuss the relevant history of the application of artificial intelligence techniques to board games, and then we move on to cover three critical areas: first, the foundational work in Gerald Tesauro's two Backgammon papers (Tesauro, 1990, 1995); second, the work we replicate in DeepMind's three AlphaGo/AlphaZero papers (Silver et al., 2016, 2017, 2018); and lastly, work that builds upon the AlphaZero approach from other researchers (Wu, 2019, 2024a,b; Zhao et al., 2022; Grill et al., 2020).

2.1 History of AI for games

Since the beginning of the field of artificial intelligence, there has been interest in programs that are able to play board games, and the creation of such programs has been viewed as an important milestone towards more powerful and general AI systems (Shannon, 1950).

Early programs would typically rely on hand-crafted heuristics and features, but modern game playing programs near-universally rely upon neural networks and reinforcement learning. One of the earliest successes with neural networks was Neurogammon (Tesauro, 1990), a program that used supervised learning to train an outcome-estimating neural network for the game of Backgammon which was then combined with expectiminimax[1] search (Michie, 1966), becoming the strongest Backgammon program of its time. Later came TD-Gammon, which combined this approach with reinforcement learning, where TD-Gammon could play games of Backgammon against itself to improve, becoming the first program to compete at the level of top Backgammon professionals. TD-Gammon also exhibited novel and superior strategies that it had not been programmed with, demonstrating the ability of reinforcement learning to go beyond existing human knowledge (Tesauro, 1995).

[1] Expectiminimax is an extension of standard minimax search that can handle random chance by performing a weighted average across all outcomes.

Modern programs have developed this approach further - the first system to become superhuman at the game of Go was AlphaGo (Silver et al., 2016), a program that combined deep neural networks, Monte-Carlo Tree Search, and reinforcement learning to beat one of the strongest Go players four games to one (Borowiec, 2016).

Even in Chess, where hand-designed heuristic approaches have been historically dominant, neural networks combined with reinforcement learning have been successful. AlphaZero, a generalisation of the AlphaGo system that uses iterative self-play reinforcement learning beginning from random data was applied to Chess and significantly outperformed Stockfish, the strongest program at the time (Silver et al., 2018). In the time since, Stockfish has once again become unambiguously stronger than any other program, including AlphaZero, partially as a result of the replacement of its evaluation function with a shallow neural network (Disservin, 2024).

Programs like AlphaZero are considered to be generally applicable to any two-player perfect-information game, as evidenced by the fact that the same architecture can be applied to Chess, Go, and Shogi, achieving superhuman performance in each case. There is also ongoing research to improve this approach further, resulting in systems like KataGo, which is much stronger than AlphaGo/AlphaZero at the game of Go, while requiring significantly less computational input for training (Wu, 2019), and Leela Chess Zero, an open-source implementation of AlphaZero focused on Chess, which has been developed into a much stronger program than AlphaZero, often competing with Stockfish for the title of ”strongest chess program”.

2.2 Foundations

Shannon (1950) identified the necessity for strong game playing agents to be able to estimate the value of game-states quickly, and to search ahead in the game to improve such estimates and calculate important sequences of moves. The following two papers demonstrate a system that does just that.

Tesauro (1990) introduces Neurogammon, a Backgammon program that uses neural networks trained on human games to approximate the state-value function, setting a new standard in the quality of computer Backgammon play. The success of Neurongammon demonstrated the efficacy of learned value functions for use in combination with game-tree search.

Later, Tesauro (1995) enhances Neurogammon to develop TD-Gammon, a system that improves by self-play reinforcement learning using $TD(\lambda)$ (Sutton and Barto, 2020), becoming a master-strength Backgammon player and advancing Backgammon theory as a result of its novel approach to some strategic decisions.

These papers outline the general principle of combining search with learned value approximation, and show how it can be very effective even with simple approaches. Neurogammon is relevant to this work, as it represents one of the first example of successfully applying neural networks to a complex board game. TD-Gammon is relevant, arguably even more so, for demonstrating the power of self-play to match and even surpass supervised methods.

2.3 AlphaGo, AlphaGo Zero, AlphaZero

This project aims to create strong programs for the games of Ataxx and Gomoku without game-specific knowledge, using machine learning and self-play. The following papers describe a system that can achieve this.

Silver et al. (2016) introduces AlphaGo, a system that uses deep neural networks trained on human master play to evaluate positions and to guide Monte-Carlo Tree Search, overcoming Go's problematically high branching factor.

Silver et al. (2017) introduces AlphaGo Zero, which can be trained entirely without human data, using only self-play reinforcement learning to generate successively stronger iterations of the system.

Silver et al. (2018) generalises AlphaGo Zero, removing Go-specific heuristics to create AlphaZero, a system that plays Chess, Go, and Shogi to a superhuman standard.

Tian et al. (2019) replicates AlphaGo Zero, providing a high-quality open-source implementation of the methods detailed in Silver et al. (2017).

These papers show how to apply search and learning methods to complex games for which no trivial features exist for evaluation and move prediction. Silver et al. (2016) is relevant to this work for its demonstration of how deep convolutional neural networks are able to learn and predict the outcome of games that have complex structure. Silver et al. (2017) is relevant to this work as we do not have the time nor expertise to produce thousands of high-quality examples of play in the games we create agents for, and the AlphaGo Zero paper demonstrates that their system works as well, if not better, when trained from zero knowledge. Last of all, Silver et al. (2018) is of utmost relevance, as it shows that our goal of training agents for new games is likely to work with minimal domain knowledge, as demonstrated by the variety of games that AlphaZero learns to play. Tian et al. (2019) is not especially groundbreaking, but it is relevant to this project as it achieves the same goal as us - replicating and improving upon AlphaZero - while providing an open-source implementation to build upon.

2.4 Enhancements to AlphaZero

Implementation of the two games will test the generality of the AlphaZero approach, revealing whether it can cope well with games beyond Chess, Go, and Shogi. Additionally, this project combines a number of the enhancements developed for AlphaZero since 2016, resulting in a system that incorporates the much of the state of the art, particularly those improvements in the following works:

Wu (2019) adds many enhancements to AlphaZero to produce KataGo, a Go-specific program that achieves much higher strength and training efficiency.

Zhao et al. (2022) adds a "path consistency" regularisation loss to AlphaZero, and claims significant benefits for training efficiency.

Wu (2024a), which enumerates additional enhancements to the AlphaZero approach that have been developed since the publication of Wu (2019).

These papers are relevant to this work as a result of a notable issue with AlphaZero - its data-inefficiency. DeepMind's main AlphaGo Zero run reportedly took about 41 TPU-years of compute, while Wu (2019) achieves similar results in 1.4 GPU-years, a 50x improvement over AlphaGo Zero, and Zhao et al. (2022) similarly claims an increase in the rate of self-play learning, outperforming AlphaZero trained for 900K steps after only 600K steps with PCZero, by 175 games to 163. We implement and adapt many of the methods developed by these papers, and are able to train our agents to high performance using limited resources as a result.

Chapter 3

The Veritas General Game Player

In this chapter we introduce our system, and explain how it achieves strong performance in Gomoku and Ataxx. Images and explanations of these games are provided in the Appendix. A visualisation of the overall operation of the system is shown in Figure 3.1.

3.1 Overview

Our system can learn to play any board game with all of the following characteristics:

1. The game must have exactly two players.
2. The game must be **zero-sum**, meaning that any gain for one player is an equal loss for the other player, excluding games like Pandemic, where players cooperate to win.
3. The game must be **perfect information**, meaning that all strategically relevant information is available to all players at all times, excluding games like Poker, where players cannot see each other's hands.

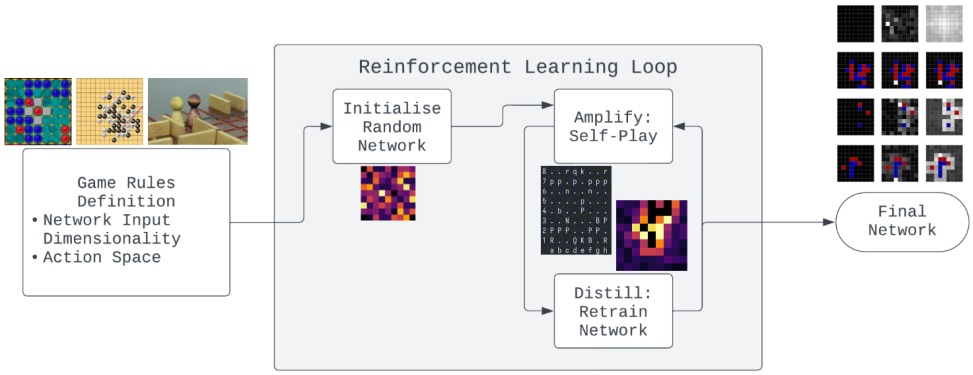

Figure 3.1: An abstract visualisation of the self-play loop, showing the inputs and outputs of the system and the process of self-improvement via reinforcement learning.

4. The game must be **non-stochastic** - all state transitions must be deterministic, excluding games like Backgammon where the outcomes of dice rolls determine future states.[1].

By the process of iterated distillation and amplification we train neural networks that progressively more accurately predict the value of states in the game and the best moves to make in those states. Our system combines these neural networks with forward planning via Monte-Carlo Tree Search to produce a strong game-playing agent that can improve its decisions if it is allowed to think for longer.

Our system has three main components - the search engine, the self-play data generator, and the training program. Components #1 and #2 are integrated in the same artefact, **veritas**, while the training is managed by PyTorch code in the **veritas-training** project. We explain the design and functioning of these components below.

3.2 Search Engine

Veritas' search engine - the system that decides which moves to play - is a modified form of Monte-Carlo Tree Search (MCTS) (Coulom, 2006), which is a search algorithm for exploring adversarial game trees, like those of Chess and Go. The variant we employ is called pUCT, which is a modification of the UCT algorithm[2] that uses a *policy predictor* to guide the search, rather than relying on uniform exploration. We use the terms "MCTS" and "Monte-Carlo Tree Search" henceforth to refer to this modified version of the algorithm, and "pure MCTS" to refer to the original UCT algorithm.

A *policy* in the reinforcement learning and game theory setting is a a function that assigns probabilities to each action in each state of the game, defining the action-probability function π as shown in Equation 3.1.

$$\begin{gathered}\pi(a|s) = \forall a \in A(s), \pi(a|s) \geq 0, \sum_{a' \in A(s)} \pi(a'|s) = 1 \\ \text{where } \pi(a|s) \text{ is the probability of taking action } a \text{ in state } s \\ A(s) \text{ is the set of all actions in state } s\end{gathered} \tag{3.1}$$

A "policy" can also refer to a *partial* policy representing a probability distribution over actions in a single game-state, and we will primarily use the term in this manner.

Monte-Carlo Tree Search constructs a best-first search tree by iterating four steps - selection, expansion, simulation, and backpropagation, until a limit is reached. Pseudocode for this procedure can be seen in Algorithm 1. During the construction of this tree, MCTS generates a *rollout distribution* at the root of the search tree that corresponds to a *policy* for the current game state. The MCTS policy is an unnormalised probability distribution over the possible actions in this game-state. Visualisations of such policy distributions are shown in Figures 3.2 and 3.3.

[1] This constraint is the least important for our system - MCTS admits the extension of "chance nodes" with little modification, which would allow our system to learn to play games like Backgammon.

[2] Essentially the "canonical" form of MCTS.

Algorithm 1 Monte-Carlo Tree Search

```
1: s ← Root State
2: while time limit not reached do
3:     s′ ← Select(s)              ▷ Select a child node according to pUCT (3.2; 3.3)
4:     if s′ is not terminal then
5:         s′ ← Expand(s′)                          ▷ Add a child node to the tree
6:         v ← Simulate(s′)                         ▷ Estimate the value of the new node
7:     else
8:         v ← Value(s′)                            ▷ Get the value of the terminal node
9:     end if
10:    Backpropagate(s′, v)                         ▷ Update the statistics of the path s → s′
11: end while
12: return Rollout Distribution(s)
```

$$\text{Select}(s) = \begin{cases} \text{Select}\left(\underset{s' \in \text{Children}(s)}{\text{argmax}}\ \text{pUCT}(s')\right) & \text{if } \exists s' \in \text{Children}(s) \text{ s.t. } N(s') > 0 \\ s & \text{otherwise} \end{cases} \tag{3.2}$$

$$\text{pUCT}(s) = \begin{cases} \text{fpu} + c_\text{p}\pi(a_{\to s})\sqrt{N(s)+1} & \text{if } N(s) = 0 \\ Q(s') + c_\text{p}\pi(a_{\to s})\frac{\sqrt{N(s)+1}}{1+N(s')} & \text{otherwise} \end{cases}$$

where $\pi(a_{\to s})$ is the prior policy for the move to s
$Q(s)$ is the value of s
$N(s)$ is the number of visits to s
s' is the parent state of s
fpu is the first-play urgency heuristic, $\frac{1}{2}$ in Veritas
c_p is the exploration constant, $\frac{5}{2}$ in Veritas

(3.3)

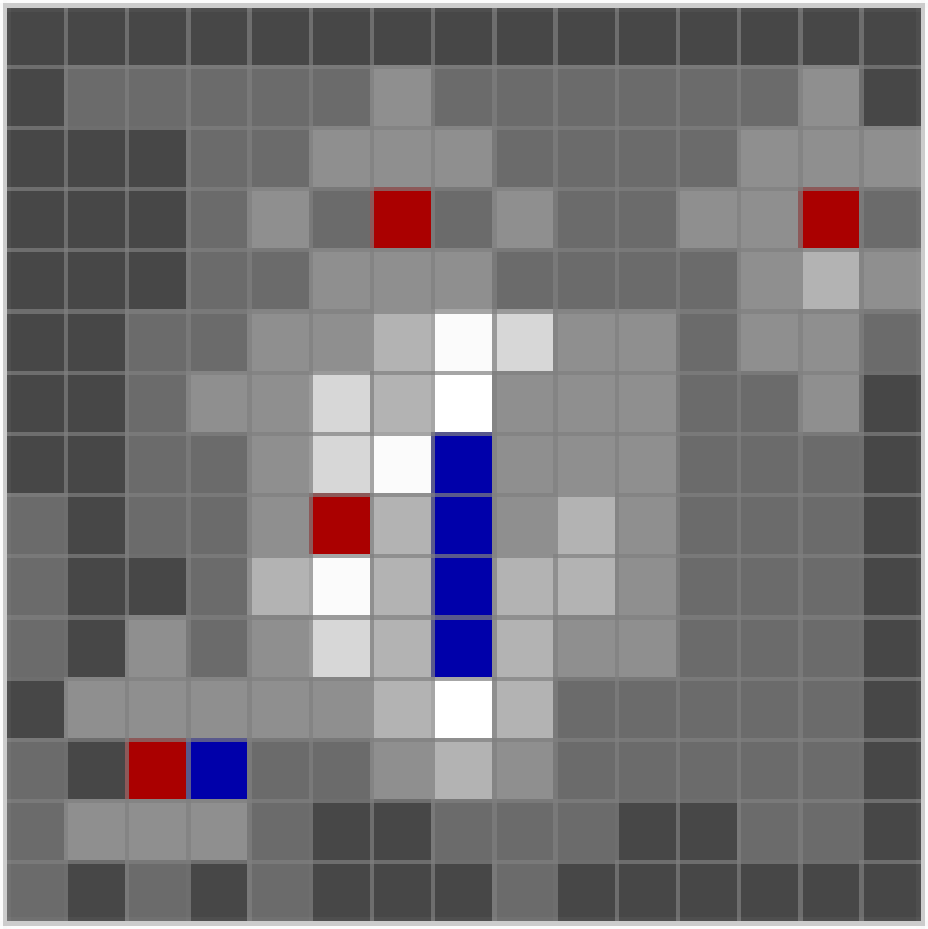

Figure 3.2: A two-dimensional visualisation of the rollout distribution in a 15×15 Gomoku position. Red stones belong to the side to move, blue stones the opponent. The brightness of the grayscale indicates the number of rollouts that MCTS allocated to the move, indicating the probability that placing a stone on the corresponding square is the best move.

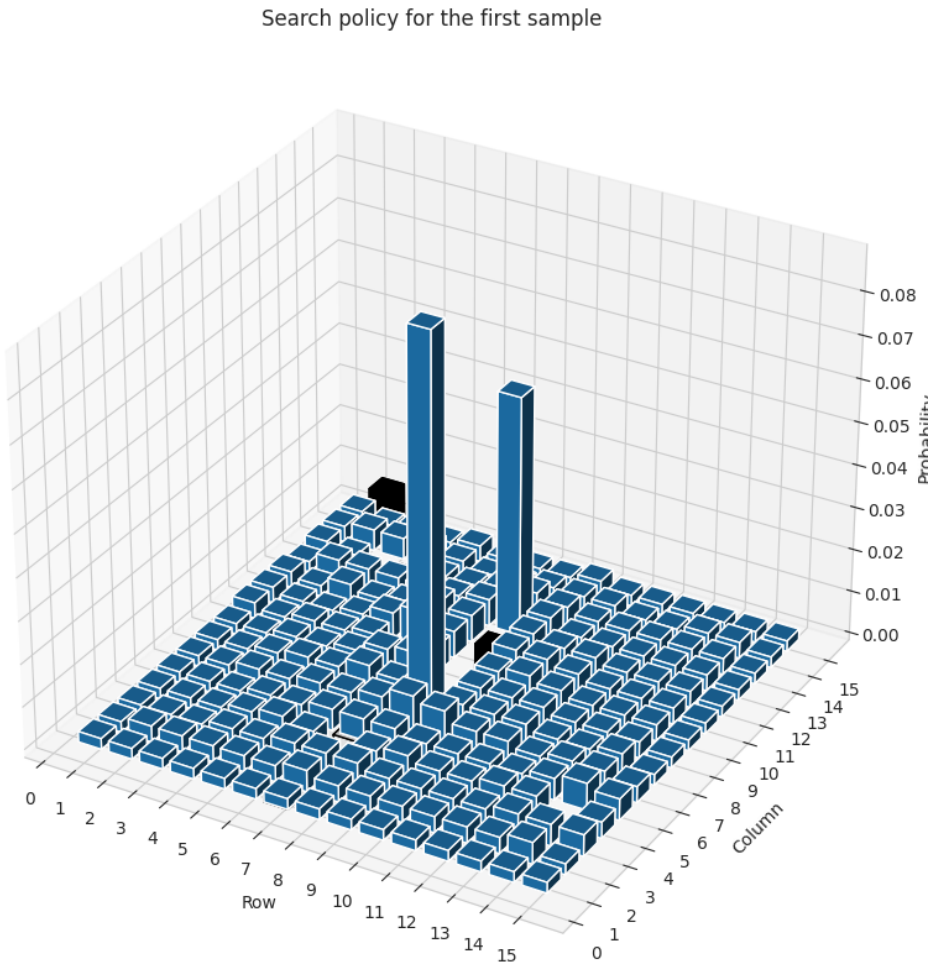

Figure 3.3: A three-dimensional visualisation of the same rollout distribution shown in Figure 3.2. The height of the bars corresponds to the number of rollouts that MCTS allocated to the move, indicating the probability of the move being played. It can be seen how Monte-Carlo Tree Search finds ”sharp” policies that allocate a large fraction of the probability mass to only a few moves.

Monte-Carlo Tree Search has a few notable characteristics:

First, MCTS is *anytime* - it conducts iterative simulations, each taking a short time, and performs backup after each simulation, progressively growing and improving the search tree. This means that it can be stopped at any time, and longer time will result in progressively higher-quality results.

Second, MCTS is an **improvement operator**. The models we use generate a policy and a value estimate for a game state, possessing a type signature of

$$\text{NN} : S \to (\pi, V) \tag{3.4}$$

Our extended MCTS algorithm uses such a model and with it searches to improve the policy and value estimates, and then uses the improved estimates to search further, and so on, and as such has the signature

$$\text{MCTS} : (S \to (\pi, V)) \to S \to (\pi, V) \tag{3.5}$$

taking an existing policy-value predictor and *improving* it, to produce a new, stronger policy-value predictor. This is what it means to be an improvement operator, and it is the core property that makes MCTS useful for reinforcement learning in this setting. An explicit view of exactly how MCTS acts to generate improved policies can be found in Grill et al. (2020), which reframes MCTS as an online learning algorithm. They show that MCTS converges on a rollout distribution that is the solution to the optimisation problem of finding the policy π that maximises

$$\begin{aligned} &\sum_a \pi(a)Q(a) - \lambda_N D_{KL}(P||\pi) \\ \text{where } &\pi(a) \text{ is the probability of taking action } a \\ &Q(a) \text{ is the value of taking action } a \\ &P(a) \text{ is the prior policy for taking action } a \\ &D_{KL}(P||\pi) \text{ is the KL divergence between } P \text{ and } \pi \\ &\lambda_N \text{ is a regularisation term that decays with N} \end{aligned} \tag{3.6}$$

This is just a maximisation of the expected value of acting under π, minus the weighted KL divergence between the prior policy and the new policy, which is a regularisation term that encourages the new policy to be close to the prior policy, and which decays as the number of visits to a node increases. The KL divergence term is reversed from the usual $D_{KL}(\text{Posterior} \mid\mid \text{Prior})$, and this has the benefit of allowing MCTS to allocate high likelihood to moves that the policy network considers very unlikely without incurring a large penalty (Wu, 2024b). As explicated by Wu (2024b):

> "*This also gives context to our earlier observations on why the visit distribution can be treated so much like a high-quality policy, and for example why in AlphaZero the visit distribution is a good policy training target for future neural net training. Except for the discretization of the visits, it basically is the policy of a continuous learning algorithm that broadly resembles a lot of classical regularized reinforcement learning algorithms.*"
>
> - David Wu, Monte-Carlo Graph Search from First Principles

3.3 Self-Play Loop

To train our agent to play a given game we use a technique called self-play, where the agent plays games against itself to generate training data. This data is then used to train a new model, which is then used to play more games, and so on. This process is visualised in Figure 3.1.

3.3.1 Game Generation

To generate games, we use the procedure outlined in Algorithm 2.

Algorithm 2 Game Generation Procedure

1: Start with the initial game position
2: $n \leftarrow 8 + \text{Bernoulli}(0.5)$ ▷ For Gomoku and Ataxx
3: Play n random moves
4: **while** game is not over **do**
5: Perform and record either 800 or 200 rollouts via *playout cap randomisation*
6: Make the move with the largest number of allocated rollouts
7: **end while**
8: Record the sequence of rollout distributions in conjunction with the game outcome

This procedure forces exploration by starting each game from an essentially unique initial position, but after the initial period of randomisation the agent's choices are fully deterministic and always correspond with the move that is allocated the largest number of rollouts. This marks a divergence from Silver et al. (2018), which uses Dirichlet noise to induce exploration throughout the game.

This alternative method is motivated for two reasons. The first is simplicity of implementation, while the second is subtler - initial runs of Leela Chess Zero, an open-source replication of AlphaZero for the game of Chess suffered from weak endgame play, as a result of noise inducing blunders in the data that caused misevalation of endgames requiring precise play. When the full evaluation of AlphaZero was released, it was discovered that applying temperature (random noise) to the moves ought only be done in the early stage of the game, which our approach achieves cleanly and avoids such issues as were had by the Leela Chess Zero team.

3.3.2 Playout Cap Randomisation

In order to improve the quality of the data generated by self-play, we apply a technique called *playout cap randomisation*, which involves randomly selecting a number of rollouts to perform for each move, rather than using a fixed number of rollouts. This technique was introduced in Wu (2019), and sacrifices some of the quality of the data for a significant increase in the number of games generated, which is very important for training the value-prediction head of the neural network, as each full game only provides a single Win/Loss/Draw result - an extremely noisy and low-information signal.

3.3.3 Parallelism

In order to take full advantage of a GPU, it is important to fully saturate the GPU with work. In order to achieve this, we must ensure that the GPU is never waiting for data to be loaded from main memory, and that the CPU is never waiting for the GPU to finish its work. To this end, we develop an asynchronous data generation pipeline, where hundreds of agents play in parallel while all sharing the same GPU executor, which batches states to the GPU for evaluation and policy generation, and dynamically routes the results back to the correct agent.

All finished games are sent through a channel to a writer thread, which outputs into three files - a file of game outcomes, a file of game positions, and a file of rollout distributions. Each file is formatted in csv, for ease of processing.

3.4 Training Procedure

Networks are trained with the PyTorch deep learning package (Paszke et al., 2019), using the AdamW optimiser (Loshchilov and Hutter, 2017) with a learning rate of 10^{-4}, weight decay of 0.03, and a batch size of 64.

We encountered significant issues with training stability, which motivated the use of a number of techniques to improve stability, including gradient clipping, batch normalisation, and post-activation residual connections in the network architecture, which we describe below.

3.4.1 Model Architecture

Given a game-state in tensor representation, the model predicts both the value of the position and the policy distribution. As such, it has a shared backbone, which is then split into two heads - a value head and a policy head. The value head outputs a scalar value from 0 to 1, representing the the expected value of the current state, where 1 is a guaranteed win, 0 is a guaranteed loss, and 0.5 is a draw or a position where both sides have equal chances. The policy head determines the likelihood of the moves in the position, producing a probability distribution over the actions available in the position. We describe the architecture of these components in greater detail below.

Backbone

The model has a 5-block 128-filter residual backbone, using 3x3 convolutions throughout. The blocks are similar to those of Leela Chess Zero, making use of Squeeze-and-Excitation (Hu et al., 2017) modules and two convolutions per block. Notable differences from the Lc0 architecture include the use of ReLU nonlinearities after every convolution, the use of explicit batch normalisation, and the use of post-activation residual connections, which were found to significantly reduce the rate at which the network diverges during training while improving the accuracy of the model's predictions.

Value Head

The value head applies a 1x1 convolution to the latent representation to reduce it to a single channel, which is then passed through a fully connected layer to produce a 128-unit hidden vector, which is then activated with a ReLU and passed through another fully connected layer and a sigmoid to produce the final value estimate.

Policy Head

The policy head is implemented differently for each game. For Gomoku, the policy head applies two steps of 1x1 convolution to generate a $N \times N$ policy distribution, where N is the board size.

For Ataxx, we make use of *attention policy*, which involves the generation of $N \times N$ sets of *source* and *target* vectors, which are then used to compute logits for the full Cartesian product of the source and target sets - the logit for moving from some source to some target is equal to the dot product of the source and target vectors.

Here, S and T are $N \times N \times D$ tensors, where D is the length of the attention policy vectors, and the policy is computed as in Equation 3.7.

$$\begin{aligned} S, T &= \text{NN-Policy}(\text{State}) \\ \text{Logit}_{s,t} &= S_s \cdot T_t \\ \pi(a) &= \sigma(\{\text{Logit}_{s,t} : s \in S, t \in T\})_{\text{source}(a),\text{target}(a)} \end{aligned} \tag{3.7}$$

This technique was proposed informally by Connor McMonigle, and is also employed by the open-source Leela Chess Zero project (borg, 2022). The main advantage of this approach is that it reduces the dimensionality of the policy head by a quadratic factor, predicting $N \times N \times 2 \times D$ values, where D is the length of the attention policy vectors, rather than $(N \times N)^2$ values, and we additionally speculate that it provides an inductive bias that allows the network to learn that moves that use the same stone or which land on the same square are likely correlated in quality.

3.4.2 Softmax Policy Temperature

When the prior policy from the neural network and the posterior MCTS policy are compared, the MCTS policy is often found to be a significantly ”sharper” distribution than the prior policy. This leads to an issue where successive iterations of reinforcement learning will tend to over-sharpen the neural policy, and this can lead to a lack of exploration or ”policy blindness”, where the neural policy so strongly believes that some small set of moves are best that the search is unable to recover if it is incorrect. To counteract this, we apply temperature to the output of the network during training, to force it to learn a smoother policy. This is done by multiplying the logits by a scalar before softmax is applied. We use a policy softmax temperature of 1.3 for both games.

In order to encourage exploration during search, we also apply temperature to the neural network’s policy output in the root node of the MCTS tree when the agent is playing, which has the added benefit of empirically improving playing strength. This latter technique is used in KataGo (Wu, 2024a).

3.4.3 Auxiliary Soft Policy Head

In order to incentivise the network to learn the difference between moves that are poor quality (e.g. the difference between a move that is merely ”not great” and a move that would lose on the spot), we add an auxiliary policy head that is trained to predict the output of the MCTS policy with a very high temperature (4.0, in our system). This massively flattens the target distribution, and so the loss of this head depends far more strongly on the relative quality of weaker moves in comparison to the main policy head, which does not need to discriminate in this manner to achieve low loss.

This head is not used during play, and in fact it is pruned from the network before it is converted to ONNX format for deployment. This technique is used in KataGo, but was added after the publication of Wu (2019), and their implementation is described in detail in Wu (2024a).

3.4.4 Legal Move Masking

During training, we manually set the values of all logits corresponding to illegal moves to $-\infty$, after the neural network generates them. This is done as an optimisation, as we always know which moves are legal in any given position, and as such there is no need for the model to waste effort learning to predict move legality. This technique masks off the gradients that would ordinarily originate from mispredictions of move legality and, this should free up more of the model's representative capacity for learning to predict the value of the position and the quality of the moves.

3.4.5 Policy / Value Weighting

During early generations, we weight the policy loss more strongly than the value loss, as data generated early in the reinforcement learning process has a particularly poor signal for the value of the position.

To understand why this is reasonable, consider that if an agent makes significant mistakes in $\frac{1}{10}$ of the states in a self-play game, then the value of all states prior to a mistake becomes decorrelated with the actual outcome of the game, corrupting the value target. In constrast, such a mistake-prone agent will still produce good data for training the policy head in $\frac{9}{10}$ of positions, and so the policy data generated by weak agents is comparatively much more trustworthy.

Naturally, as the quality of the data improves, we remove this weighting, allowing the networks to dedicate greater effort to value prediction once the data is sufficiently high-quality to support this.

Chapter 4

Evaluation

We evaluate our agents against human opponents and against algorithmic baselines. We find that our agents significantly outperform the baselines, and our final agents are able to beat human players in both games.

4.1 Results against Human Players

We tested our agents against human players in both Gomoku and Ataxx. Each human player played three games against each configuration of our system, and were given a practice game prior to the test games. The human players were given extensive explanation of the rules of the games, and two were novices, while one had prior experience with both games. Our system played in three configurations for each game - a pure MCTS agent with no neural networks, an agent that uses the very first trained network from the game's training run, and the final agent from the training run. In all matches our agent was limited to one second of thinking time per move, and ran on a GTX 1060 6GB GPU[1].

4.1.1 Ataxx

In Ataxx, our final agent was able to beat all three human players in all three games. Table 4.1 shows the results of the human matches. We find that the final agent is strong enough that it wins every game against human players, even when they have prior experience with the game. We additionally note a significant improvement in the performance of the agent over the course of training, as the generation 1 agent was only able to beat the novice players.

One of the human testers noted an interesting behaviour of the fully-trained agent in one of the games - the agent got into a position where it was clearly completely winning, and then did not make much effort to end the game quickly, instead playing "slack" moves that retained its advantage but that did not increase the agent's number of stones or attempt to capture all of the human player's stones. We hypothesise that this is an

[1]UUID: GPU-c3d3665e-0c94-3ad5-2269-9542afe7e6d7

artefact of the system's learning objective, as the system is only trained to maximise the probability of winning, and not to maximise the margin of victory, and as such it sees no difference between winning by a single stone or by forty stones.

Agent	Player 1 - Novice	Player 2 - Novice	Player 3 - Expert
MCTS Agent	0-3	1-2	0-3
Generation 1 Agent	3-0	1-2	0-3
Final Agent (Gen 17)	3-0	3-0	3-0

Table 4.1: Results of human matches in Ataxx. Results given in the format of "wins for the agent - wins for the human player".

4.1.2 Gomoku

In Gomoku, our agents were comparatively weak, though still outperformed the human testers by a large margin. We attribute this almost entirely to the significantly shorter training run for Gomoku, which was only 9 generations long, compared to 17 for Ataxx. The final agent was able to beat one novice player and the expert every time, but one novice showed surprising skill and managed to win a game against the final generation-9 agent. The results of the human matches in Gomoku are shown in Table 4.2.

Figure 4.1 shows the position in which the final agent blundered against the novice player. The agent takes longer than the allocated second of time to see that it must play in either of the F9 or B13 squares, and instead plays in the D14 square, allowing the human player to create a double-threat that the agent cannot defend against. In post-game analysis, we find that if the agent is allowed to search for longer, even as little as 1.5 seconds, it manages to find the F9 move and avoid a loss. This indicates that this is not a fundamental flaw in the agent's policy (as the agent is able to find the correct move with more time), and we strongly suspect that the agent would overcome this issue with more training.

Agent	Player 1 - Novice	Player 2 - Novice	Player 3 - Expert
MCTS Agent	0-3	2-1	1-2
Generation 1 Agent	0-3	0-3	0-3
Final Agent (Gen 9)	3-0	2-1	3-0

Table 4.2: Results of human matches in Gomoku. Results given in the format of "wins for the agent - wins for the human player".

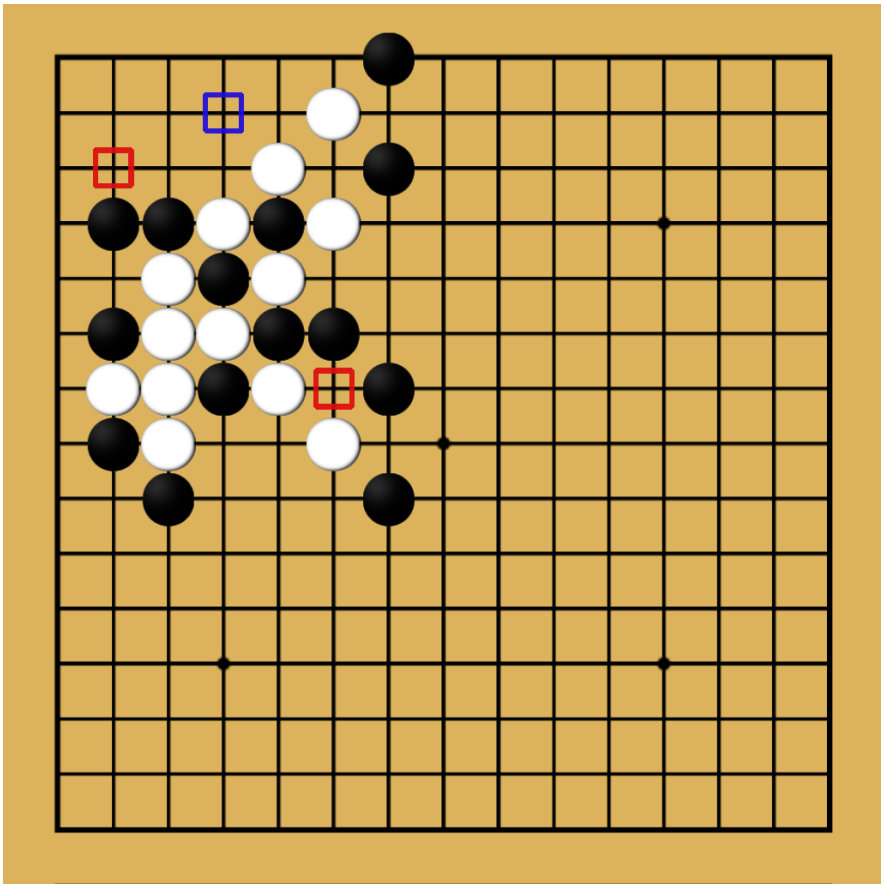

Figure 4.1: The position in which the agent blundered against a human tester (White to move). The agent needed to play in either of the marked red intersections to avoid a loss, but instead played on the marked blue intersection, allowing the human player to create an unstoppable double-threat. The human player went on to win this game.

4.2 Results against Algorithmic Baselines

We compare our agent to the following baselines:

- **Random Agent** - An agent that follows the uniform policy - for each available action the probability of taking that action is the reciprocal of the number of available actions, $\pi(a) = |A|^{-1}$.
- **Pure MCTS** - A agent that uses Monte-Carlo Tree Search with no neural network (uniform policy, determines state-value estimates via stochastic playouts). This is a classical algorithm for general game playing that achieves strong performance in many games without the need for domain-specific knowledge.
- **Least-Captures** *(Ataxx-specific)* - An agent that selects the move that flips the fewest pieces to the player's colour, with ties broken by selecting moves that do not create new pieces.
- **Most-Captures** *(Ataxx-specific)* - An agent that selects the move that flips the most pieces to the player's colour, with ties broken by selecting moves that create new pieces.

4.2.1 Ataxx

4.2.1.1 Random Agent, MCTS Agent, and Least-Captures

Ataxx is a complex, tactical game, and does not admit strong performance from agents that rely on random simulation or that use weak heuristics.

As such, the three weaker benchmarks - random agent, MCTS agent, and Least-Captures agent - lose every single game to the very first trained network playing at 1 node (no lookahead).

This is a very pleasing result - our system soundly outperforms pure MCTS, which is already a sophisticated general game playing algorithm - but it is not the most rigorous test of the system's strength. As such, we also test the system against the Most-Captures agent.

4.2.1.2 Most-Captures

Most-Captures is a surprisingly strong heuristic agent (human players regularly report finding it extremely difficult to beat), that relies on domain knowledge about the game of Ataxx to select moves. It always picks the move that flips the most stones to its own colour, breaking ties by selecting single-moves that create new stones.

The Ataxx networks were benchmarked primarily against this agent, as it is by far the strongest of the four baselines. Despite the fact that Most-Captures is weak in an absolute sense, performing no forward search, we can compare it against low-compute configurations of our agent and gain very useful insight into our agent's strength. We measure performance using the Elo rating system, where a difference of 400 Elo corresponds to a 10:1 win ratio.

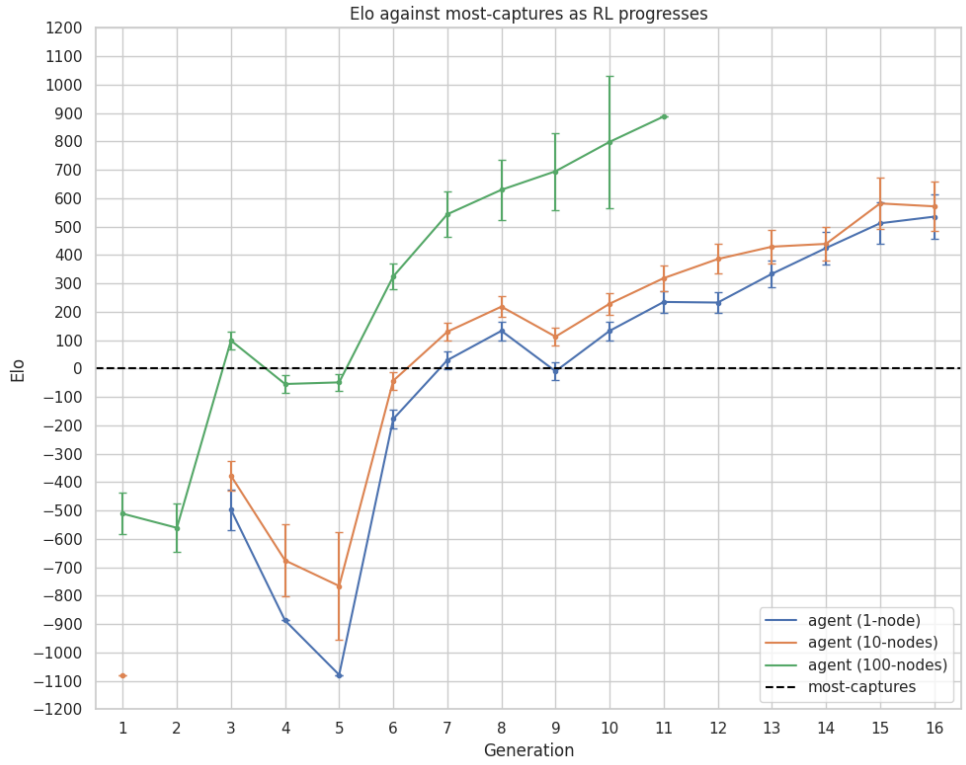

Figure 4.2: Results from the main Ataxx reinforcement learning run. Missing data points and error bars result from the fact that very large strength differences cannot be accurately converted into an Elo rating. Note that after generation 11 the network becomes so strong that it wins every single game at 100 nodes, and as such we chart no 100-node data beyond that point.

As training progressed, we tested our agent with varying levels of computational power, both to determine its strength against the baseline and also to measure how well the agent's performance scales with increasing compute. To achieve this, we set the number of nodes (future game-states) that the agent is permitted to evaluate per move to a fixed value and then run matches from varied opening positions against Most-Captures. When the agent is limited to one node, it can do no forward search at all, and must rely only on the learned policy of the neural network. We test our agent with node budgets of 1, 10, and 100, and Elo ratings of the these configurations are shown in Figure 4.2.

Between generations 7 and 9, the agent's raw network learns to implement a strategy that is as strong as Most-Captures[2], and subsequent networks continue to learn far stronger strategies, and by generation 16 the raw network outperforms Most-Captures by an enormous 534.80 ± 78.52 Elo margin[3], without performing any forward search. An example of the raw network's policy prediction is shown in Figure 4.3.

When the agent is allowed to search ahead, it performs far better, and by the later generations it simply wins every game. This result is a clear demonstration of our

[2] in that the raw network wins as many games as it loses against Most-Captures.

[3] This corresponds to a win ratio of approximately 22 to 1.

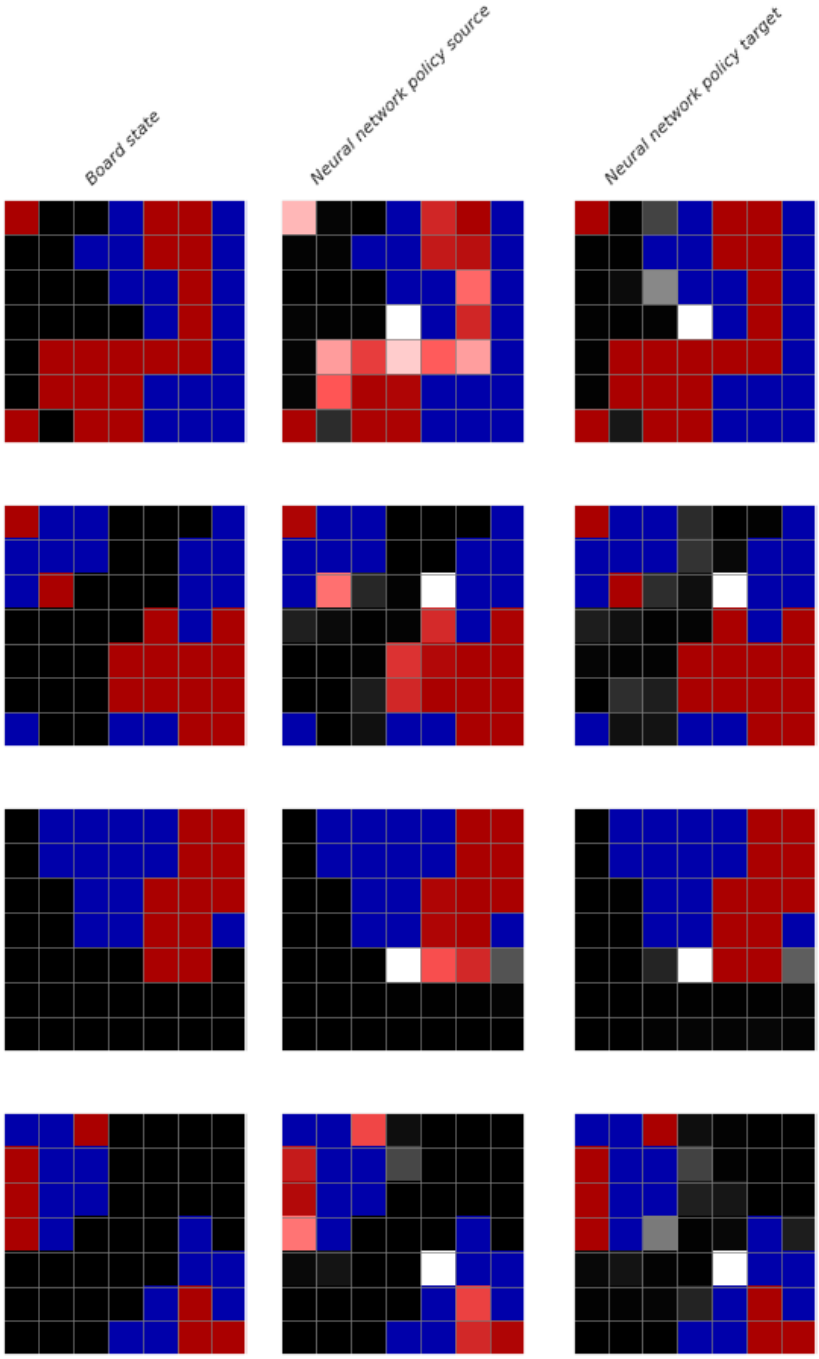

Figure 4.3: A visualisation of the policy output of the Ataxx policy head at generation 16. The policy is split into source- and target-boards, due to the nature of Ataxx's move structure. The clean game board is shown on the left, the policy source board is shown in the centre, and the policy target board on the right. Red stones belong to the side to move, blue stones the opponent. The brightness of the grayscale indicates the likelihood that a stone will be moved from the square or to the square, and self-square moves indicate the placement of a new stone.

system's ability to learn strong strategies in novel games, outperforming systems that rely on hand-crafted domain knowledge.

To properly contextualise the results in Figure 4.2, we note that our system searches and evaluates hundreds of positions per second on even relatively weak hardware[4], and as a result our agents would search two to four orders of magnitude more game-states were they playing under standard time-controls used for official tournament play, and as such would exhibit play of a quality commensurate with such an increase in computational budget.

4.2.2 Gomoku

During the development of the system, Gomoku was used as the main testing ground for correctness and improvements. As such, many experimental networks were produced before the first proper training run began, although even these networks showed strong performance against the baselines.

4.2.2.1 Random Agent

The random agent loses every game to the first trained network playing at 1 node (no lookahead). This is expected, as the random policy is *extremely* weak in Gomoku. Nevertheless, it is pleasing that even the very first trained network is able to outperform the random agent.

4.2.2.2 MCTS Agent

The first trained network for Gomoku performs very well (wins every single game) when compared node-for-node against pure MCTS search, demonstrating that the network has learned a policy that outperforms the uniform policy, and is able to evaluate positions to an accuracy better than that of pure MCTS's random playouts.

This is remarkable, as pure MCTS is rather well-suited to board-filling games like Gomoku, so the fact that the network is able to outperform it is a strong indicator of the network's strength.

[4] 539 nodes/sec on an Nvidia GTX 1060 6GB GPU

Chapter 5

Conclusions and Future Work

In this chapter, we summarise the results of our project, and review the strengths and weaknesses of our system. We also discuss potential future work that could extend or enhance our system.

5.1 Conclusions

This project has introduced a system for training game-playing agents that is generally applicable across different games, and has demonstrated the system's effectiveness in the games of Gomoku and Ataxx. By testing our agents against human players and multiple algorithmic baselines, we prove that our agents learn strong strategies in these games. We observe that the final agents learn robust strategies that perform well across a range of testing conditions and opponents, beating our baselines by enormous margins even in our system's lowest-resource configuration, and demonstrate that our agents reliably benefit from increased computation, be that in the form of longer training runs or greater computational budget at test-time.

We find that our system outperforms classical general game playing algorithms like MCTS by using deep reinforcement learning to learn more accurate policy and value estimates, and that the system is able to learn strong strategies in games with complex rules without the need for hand-crafted heuristics. When tested against human players, we show that our agents outperform human players across both games, although we note that there remain weaknesses that human players can successfully exploit, particularly in Gomoku.

5.2 Future Work

The possible future work for this project is extensive, as the generality of the system admits extension to a wide range of domains and applications. We discuss possible improvements to the algorithms used in the system, the architecture of the deep neural networks used, and possible extensions of the system to new games.

5.2.1 Improving the Search Engine

Many improvements can be made to the core search algorithm, including the use of *virtual loss* (Chaslot et al., 2008) for enhanced parallelism during match play, *progressive widening* and *dynamic* C_p for improved exploration (Browne et al., 2012), and Monte-Carlo Graph Search (Wu, 2024b) for handling transpositions in the game tree. We are confident that the use of these techniques would improve the strength of the agent significantly, but they are less important for reinforcement learning performance and so were not implemented in this project.

5.2.2 Improving the Training Procedure

Many standard training enhancements could be applied to the system, including hyperparameter tuning, application of additional regularisation techniques, and the use of more advanced optimisers. We would be particularly optimistic that the addition of additional predictive heads, like a moves-left head or a final-score head for Ataxx, would give the agent a richer training signal and potentially improve characteristics like the Ataxx agent's "slack" play, which we conjecture originated from the score-oblivious training objective. Such auxiliary predictive heads have proven effective in other systems (Wu, 2019), and we believe that they would be similarly effective in our system.

5.2.3 Improving the Neural Network

The open-source Leela Chess Zero project has demonstrated that the use of transformer architectures over convolutional neural networks can significantly improve the strength of a game-playing agent (Monroe, 2024), and it is likely that the same would be true for our system. Additionally, the use of augmented input representations like *history planes*, where the network is shown the previous states of the game, would almost certainly improve the strength of the agent, as they would allow the agent to perform conditional reasoning about its strategic choices based on the previous moves in the game.

Our results, particularly those shown in Figure 4.2, show that the agent's strength reliably increases with training time, and as such an obvious extension of this work would be to train the agents for longer. We strongly suspect that our agents would continue to improve in strength with more data and more training time.

5.2.4 Application to Other Games

Our system has been tested on two games with strong results, but there are many more games that could be tested. In particular, we would be interested to apply the system to mainstream games, like Chess or Go, or to popular Chess variants like Crazyhouse and Antichess, to see if it is able to learn strong strategies in these games as well. In addition, results like those in Yen et al. (2013) show that MCTS can be extended further to stochastic games like Backgammon via the addition of *chance nodes*, and we believe that such an extension is extremely promising for our system.

Bibliography

borg. Lc0 release v0.29.0, Dec 2022. URL https://lczero.org/blog/2022/12/lc0-release-v0.29.0/.

Steven Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. *The Guardian*, 2016. URL https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

Guillaume Chaslot, Mark Winands, and H. Herik. Parallel monte-carlo tree search. pages 60–71, 09 2008. ISBN 978-3-540-87607-6. doi: 10.1007/978-3-540-87608-3_6.

Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In Paolo Ciancarini and H. Jaap van den Herik, editors, *5th International Conference on Computer and Games*, Turin, Italy, May 2006. URL https://inria.hal.science/inria-00116992.

Disservin. NNUE | Stockfish, 2024. URL https://disservin.github.io/stockfish-docs/pages-nnue/docs/nnue.html. [Online; accessed 22-March-2024].

Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Rémi Munos. Monte-carlo tree search as regularized policy optimization, 2020.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. *CoRR*, abs/1709.01507, 2017. URL http://arxiv.org/abs/1709.01507.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017. URL http://arxiv.org/abs/1711.05101.

D. Michie. Chapter 8 - game-playing and game-learning automata. In L. FOX, editor, *Advances in Programming and Non-Numerical Computation*, pages 183–200. Pergamon, 1966. ISBN 978-0-08-011356-2. doi: https://doi.org/10.

1016/B978-0-08-011356-2.50011-2. URL https://www.sciencedirect.com/science/article/pii/B9780080113562500112.

Daniel Monroe. Transformer progress, Feb 2024. URL https://lczero.org/blog/2024/02/transformer-progress/.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Claude E. Shannon. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, March 1950. ISSN 1941-5982, 1941-5990. doi: 10.1080/14786445008521796. URL http://www.tandfonline.com/doi/abs/10.1080/14786445008521796.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, October 2017. ISSN 1476-4687. doi: 10.1038/nature24270. URL https://doi.org/10.1038/nature24270.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362 (6419):1140–1144, December 2018. doi: 10.1126/science.aar6404. URL https://doi.org/10.1126/science.aar6404. Publisher: American Association for the Advancement of Science.

Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts London, England, second edition edition, 2020. ISBN 978-0-262-03924-6.

G. Tesauro. Neurogammon: a neural-network backgammon program. In *1990*

IJCNN International Joint Conference on Neural Networks, pages 33–39 vol.3, San Diego, CA, USA, 1990. IEEE. doi: 10.1109/IJCNN.1990.137821. URL http://ieeexplore.ieee.org/document/5726779/.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995. ISSN 0001-0782, 1557-7317. doi: 10.1145/203330.203343. URL https://dl.acm.org/doi/10.1145/203330.203343.

Yuandong Tian, Jerry Ma, Qucheng Gong, Shubho Sengupta, Zhuoyuan Chen, James Pinkerton, and C. Lawrence Zitnick. ELF OpenGo: An Analysis and Open Reimplementation of AlphaZero. 2019. doi: 10.48550/ARXIV.1902.04522. URL https://arxiv.org/abs/1902.04522. Publisher: arXiv Version Number: 5.

David J. Wu. Accelerating Self-Play Learning in Go. 2019. doi: 10.48550/ARXIV.1902.10565. URL https://arxiv.org/abs/1902.10565. Publisher: arXiv Version Number: 5.

David J. Wu. Other methods implemented in katago, 2024a. URL https://github.com/lightvector/KataGo/blob/master/docs/KataGoMethods.md. [Online; accessed 22-March-2024].

David J. Wu. Monte-carlo graph search from first principles, 2024b. URL https://github.com/lightvector/KataGo/blob/master/docs/GraphSearch.md. [Online; accessed 22-March-2024].

Shi-Jim Yen, Cheng-Wei Chou, Jr-Chang Chen, I-Chen Wu, and Kuo-Yuan Kao. The art of the chinese dark chess program diable. In Ruay-Shiung Chang, Lakhmi C. Jain, and Sheng-Lung Peng, editors, *Advances in Intelligent Systems and Applications - Volume 1*, pages 231–242, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-35452-6.

Dengwei Zhao, Shikui Tu, and Lei Xu. Efficient learning for AlphaZero via path consistency. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 26971–26981. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/zhao22h.html.

Appendix A

Participants' Information and Ethics

Participants were taught the rules of each game in detail by the author, and guided through a practice game. No other information was given to the participants.

Our participants were several students, who were offered the option to play informal games against our agents. No forms were signed, and all participants were free to leave or give up at any time during testing.

Appendix B

Game Descriptions

B.1 Gomoku

Gomoku is a two-player perfect information game played on a square board, typically 15×15. Players take turns placing stones of their colour on the board, and the first player to get five stones in a row, either horizontally, vertically, or diagonally, wins the game. The game is a draw if the board fills up before either player wins. A visualisation of a Gomoku board is shown in Figure B.1.

B.2 Ataxx

Ataxx is a two-player perfect information game played on a square board, typically 7×7. Players begin with two stones each, placed in diagonally opposing corners. Players take turns either adding a new stone adjacent to one of their existing stones, or moving one of their stones two squares, and each time a stone arrives on a square, all the opposing stones adjacent to it are flipped to the colour of the arriving stone. The game ends when neither player can make a move (usually as a result of the board filling up), and the player with the most stones on the board wins. A visualisation of an Ataxx board is shown in Figure B.2.

Figure B.1: A 15×15 Gomoku board.

Figure B.2: A 7×7 Ataxx board.