UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Deep Reinforcement Learning for General Game Playing

by

Adrian Goldwaser

Thesis submitted as a requirement for the degree of

Bachelor of Science in Computer Science

Submitted:	October 23, 2018	Student ID:	z5076514
Supervisor:	Michael Thielscher	Co-supervisor:	Alan Blair

Abstract

Game playing is a large and growing field in artificial intelligence (AI) and has been around since the start of AI research. Games provide a useful way to compare AI algorithms as they simplify the environment for both the agents and for evaluation. The vast majority of game playing AIs, however, are extremely specialised and can only play a single game. This makes comparing algorithms difficult as they are so specialised to individual games. It also makes these comparisons less generalisable and hence less useful for investigating general intelligence.

The area of general game playing (GGP) addresses this specificity, by looking at AI systems which are programmed in such a way as to be able to play games that they have never seen before. This means that they are developed without explicit knowledge of any specific game and only given the game description at runtime.

There has been very little research in applying reinforcement learning to GGP, however, reinforcement learning has been used successfully in many new fields recently. The wide variety of new applications of reinforcement learning, especially deep reinforcement learning, shows the potential it has in a more general setting.

This work looks at applying neural network guided Monte Carlo tree search for self-play reinforcement learning in the setting of GGP. The recent AlphaZero algorithm is extended, removing the limitations that prevent its use in GGP and the new version is run on a variety of small games with different properties to evaluate the effectiveness in the more general setting. Positive results are observed for the small-scale games tested showing the potential for this method to be used more widely within GGP.

Acknowledgements

My thanks go to my supervisor, Michael Thielscher, for his excellent guidance and advice throughout the year and to my co-supervisor, Alan Blair, for his amazing help and suggestions.

I would also like to thank the following people for their help throughout the year. Jeremy Gillen for numerous discussions, Chris Lee for his use of computing resources and Joshua Lau for his help with ideas on memory usage. Additionally, I'd like to thank Tamara Diner for her help with proofreading.

Abbreviations

AI Artificial Intelligence

CSP Constraint Satisfaction Programming

DQN Deep Q-Network

GDL Game Description Language

GGP General Game Playing

GM Game Manager

IGGPC International General Game Playing Competition

MCTS Monte Carlo Tree Search

MCS Monte Carlo Search

RL Reinforcement Learning

UCB Upper Confidence Bound

UCT Upper Confidence Bounds on Trees

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Since the concepts of general computers were first explored, the question of how to make computers think has been considered, spawning the now massive field of artificial intelligence (AI). Games have been used as a testbed for AI research since the beginning as they accurately capture many of the ideas of thought, planning and reasoning in an easily evaluable way. Recent advances have seen computers beat top human players in more and more games, from DeepMind's AlphaGo beating 18-time world champion Go player Lee Sidol in 2016 [1], to both DeepStack [2] and Liberatus [3] beating professional players in No Limit Texas Hold'em, to OpenAI beating professional players in 1v1 Dota2 competitions [4]. While these AIs showed incredible improvement in different areas, they required many years of work from AI researchers and developers for each individual one.

All of these AIs are very specific and can only do the exact task that they were programmed to do. To address this gap, the field of general game playing (GGP) came into being. GGP looks at AI systems which are programmed without any explicit knowledge of a particular game; instead they must be general enough to play any game given a formal specification at runtime. This means that they cannot use game-specific knowledge or heuristics that do not apply to other games.

The field of reinforcement learning (RL) looks at agents which can learn to maximise

some notion of reward by taking actions and noting what effect these had. Recently, AlphaZero [5] has shown some limited generality in the context of games with the same algorithm being applied to Chess, Shogi and Go, reaching state-of-the-art in each. However, it was still limited to zero-sum, two-player symmetric games. This agent learnt entirely from scratch using self-play deep reinforcement learning with almost no knowledge of the game, giving it the potential to be extended and applied to the more general setting of GGP.

There has previously been very little work applying RL to the setting of GGP and this project fills that gap by extending then applying this deep reinforcement learning technique to the more general environment of GGP. This extension is then applied to a variety of games to examine how well it would work in a GGP competition environment by running it against an agent implementing Upper Confidence Bound on Trees with equivalent time limits.

An overview of the structure of the remainder of the report follows.

- **Chapter 2** explains the background of GGP and RL, as well as previous work in combining the two.
- **Chapter 3** Explains the implementation and design decisions
- **Chapter 4** Evaluates and discusses results
- **Chapter 5** Summarises results and proposes future work

Chapter 2

Background

This chapter gives an overview of the field of general game playing (GGP) and is followed by a brief description of the common approaches to GGP, looking at both current and previous state-of-the-art algorithms. It then goes on to provide a summary of some basic reinforcement learning (RL) algorithms along with some newer advances in that area that relate to GGP. Finally, it looks at previous applications of RL to GGP.

2.1 General Game Playing

The issue with using individual games as a testbed for AI is that they encourage very specified programs which perform very well in an extremely narrow domain. GGP addresses this by looking at programs which are only given the rules of the game at runtime. This means that they must be written without any explicit knowledge of any particular game, encouraging strategies which are applicable in different domains and general algorithms that result in agents which can plan and learn rather than simply using game-specific heuristics that humans have worked out. The lack of handcrafted heuristics means that performance should reflect the skill of the algorithm at that game, not the skill of the person who programmed it.

Year	Winning player	Author	Technique
2005	ClunePlayer	Jim Clune	Minimax
2006	FluxPlayer	Stephan Schiffel, Michael Thielscher	Minimax
2007	CadioPlayer	Yngvi Björnsson, Hilmar Finnsson	UCT
2008	CadioPlayer	Yngvi Björnsson, Hilmar Finnsson	UCT
2009	Ary	Jean Méhat	UCT
2010	Ary	Jean Méhat	UCT
2011	TurboTurtle	Sam Schreiber	UCT
2012	CadioPlayer	Yngvi Björnsson, Hilmar Finnsson	UCT
2013	TurboTurtle	Sam Schreiber	UCT
2014	Sancho	Steve Draper, Andrew Rose	UCT
2015	Galvanise	Richard Emslie	UCT
2016	WoodStock	Éric Piette	CSP

Table 2.1: Previous IGGPC winners

In GGP, players are given a formal specification of a game in a language such as Game Description Language (GDL) [6], players then need to play the game without any extra input from humans. The running of a game is facilitated by a game manager (GM) which sends first the specification of the game, the player's role, a *playclock* and a *startclock* to each player, all players then have *startclock* seconds to do any initial analysis of the game. All players respond when they have performed all initial computation they require. Once all players have responded or *startclock* seconds are up, gameplay commences. For each turn, the GM sends each player details of what moves were made on the previous turn (`nil` on the first move), and each player returns their move. If players do not return a move within *playclock* time then the GM selects an arbitrary valid move for them [7].

2.1.1 Approaches

The main evaluation of GGP players occurs in the International General Game Playing Competition (IGGPC) and is a good way to compare different approaches in GGP.

As can be seen in Table 2.1 [8], originally generic heuristic extraction aided minimax agents such as ClunePlayer [9] and FluxPlayer [10] won the competitions. This changed in 2007 when CadioPlayer [11] and Ary [12] entered using the upper confidence bound

on trees (UCT) algorithm. Since then all winners have used a variation of UCT, with the exception of the most recent winner, WoodStock, which used a method based on constraint satisfaction programming (CSP) [13].

2.1.2 Upper Confidence Bound on Trees

Upper confidence bound on trees (UCT) is a modification on a standard Monte Carlo Tree Search (MCTS) to enable a better choice of how to trade off exploration and exploitation.

Consider a standard Monte Carlo Search (MCS), where a state is evaluated by running a number of simulations of random games and counting which player wins each one. This has the disadvantage that a move that looks promising could have a simple counter that was missed by the random playouts. One extension of this is MCTS; in this version a game tree is kept track of with scores at each node as shown in Figure 2.1a. The tree is initially empty and again a number of game playouts are simulated, each expanding a single state at the bottom of the current tree. On each playout, a path is followed down to the bottom of the tree. Once a leaf node is reached, it is expanded and a single MCS simulation is performed from the new node giving an initial score for that node, as shown in Figure 2.1b. This initial score is propagated backwards to update all nodes above it in the tree with the result that the most recent MCS playout returned, as shown in Figure 2.1c. Running many of these playouts gives progressively better approximations at each node on the tree of how good that state is and hence who is likely to win from there.

Finally, this method can be extended to UCT. The missing section in the above description of MCTS is that there is no good way to choose a path through the currently expanded game tree; doing it randomly would provide no major advantage over standard MCS. Focusing on promising nodes is useful to fully explore these nodes, however too much focus there could cause the player to miss a good move that may have randomly resulted in a loss in the earlier MCS playouts. Instead a variant on upper confidence bounds (UCB) [14], a common solution to the multi-armed bandit problem

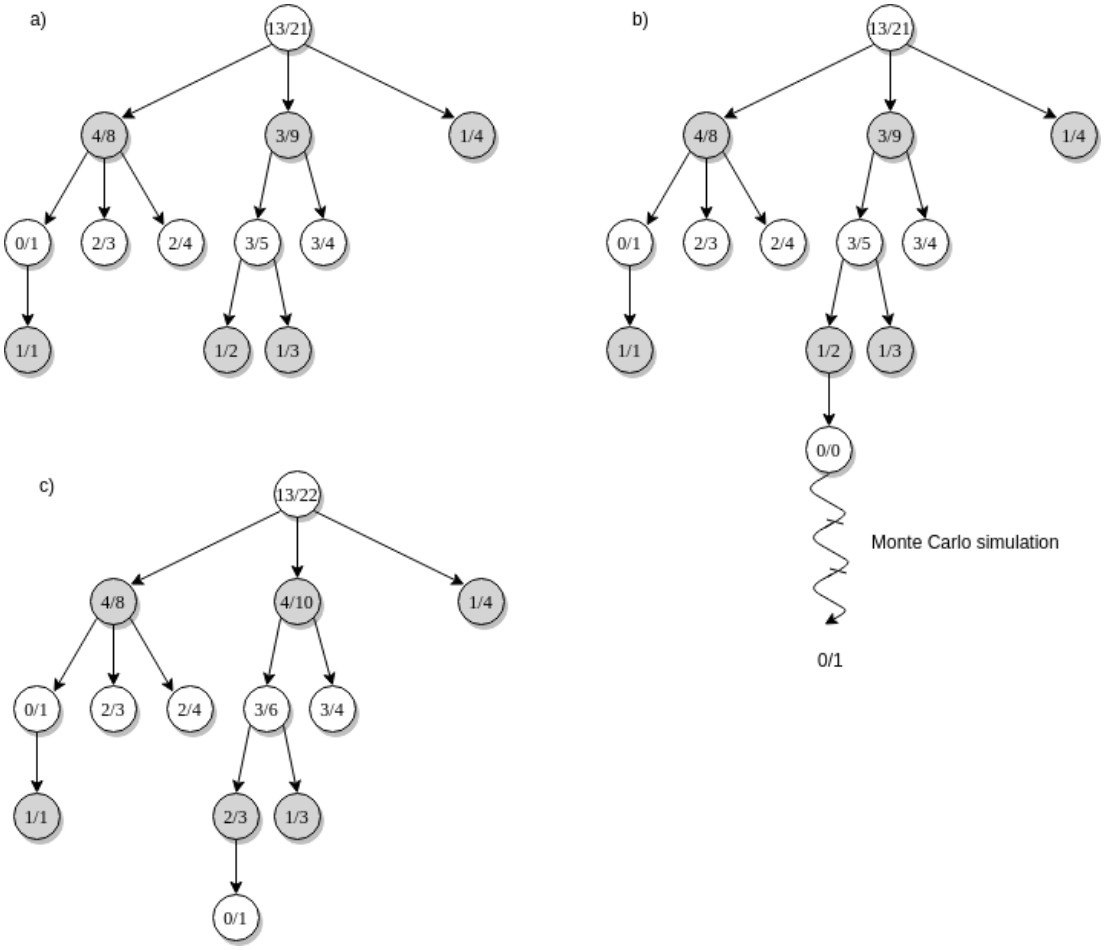

Figure 2.1: One simulation of an MCTS (a) is the initial game tree (b) is the Monte Carlo playout (c) is the updated game tree

of trading off exploration and exploitation, is modified to work in this setting.

Here the idea is to go down to the child node with the greatest upper confidence bound (expected value + confidence interval). The reasoning for this is to optimistically prioritise nodes which have high potential and either they will turn out well or the confidence in a bad value will increase, bringing the confidence bound lower and resulting in a different child being chosen in subsequent simulations.

Each node stores a node count, $N(s)$, a count of the number of times action a has been taken from node s, $N(s,a)$, and a value for how good the current state, $V(s)$. The choice of action at a state is then given by $\mathbf{argmax}_{a \in A(s)} \left(V(\delta(s,a)) + C_{\text{uct}} \sqrt{\frac{\ln(N(s))}{N(s,a)}} \right)$, where $A(s)$ is the set of legal actions in state s, $\delta(s,a)$ is the transition function — the next state after taking action a from state s and C_{uct} is a constant.

2.2 Reinforcement Learning

The idea of RL is to have an agent learn to maximise some notion of reward by taking actions and seeing the resultant reward.

2.2.1 Q-learning

One of the most common versions of RL is called Q-learning. In this technique, the aim is to learn a function $Q(s,a)$ which represents the expected infinite horizon discounted reward resulting from taking action a in state s. The infinite horizon discounted reward is defined as $\sum_{i=0}^{\infty} \gamma^i r_i$ with r_i being the reward on the ith step and $\gamma < 1$ being the *discount factor*. Once $Q(s,a)$ has been learnt, actions can then be chosen in state s using $\mathbf{argmax}_{a \in A(s)} Q(s,a)$ where $A(s)$ is the set of legal actions from state s.

In order to learn this function, the following fact is observed. For an optimal $Q(s,a)$, the equation $Q(s,a) = R(s,a) + \gamma \mathbf{max}_{a' \in A(s)} Q(\delta(s,a), a')$ always holds in deterministic environments, where $R(s,a)$ is the immediate reward received from taking action a in

state s and $\delta(s, a)$ is the resultant state after taking action a from state s. From this, the following iterative update rule is derived: $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a) + \gamma \mathbf{max}_{a' \in A(s)} Q(\delta(s, a), a'))$, where α is the learning rate. This update rule also is guaranteed to converge in both deterministic and non-deterministic environments for suitable learning rates [15].

2.2.2 Deep Q-learning

One issue with Q-learning is that the input space for $Q(s, a)$ can be incredibly large for more complex systems. One approach in dealing with this is using function approximation through a Deep Q-Network (DQN). In this version, the function $Q(s, a)$ is replaced by a neural network. After doing this, the method of learning switches to off-policy learning and instead gathers triples of state, action and discounted reward. These are then sampled and used to continue to train the neural network. DQNs have been used to achieve superhuman performance in many of the Atari games [16].

2.2.3 AlphaZero

In 2016, DeepMind used a neural network-guided MCTS agent, AlphaGo, to beat 18-time world champion Go player Lee Sidol [1]. This was then generalised to AlphaGo Zero, which beat the original and learnt entirely from self-play reinforcement learning, as opposed to initially learning from expert games [17]. Finally, this was generalised to AlphaZero which was separately trained on Go, Chess and Shogi and achieved state-of-the-art in each [5].

AlphaZero uses a single neural network which outputs both a probability distribution over actions and an expectation as to who will win. It plays through games with itself, at each state running a MCTS/UCT to produce a better probability distribution over actions and sampling from that distribution. After a self-play game, the winner is recorded and passed back to all states that were played in that game. These triples of state, new action distribution and winner are then sampled in order to train the neural

network [5]. This process is shown in Figure 2.2.

The main part of this algorithm is the MCTS/UCT search improvement operator. This works in a similar way to the standard UCT algorithm. It has an initially empty tree and each simulation expands a single node at the bottom of the tree. Each node stores a number of values. $N(s, a)$ is the visit count from taking action a in state s, $P(s, a)$ is the prior of taking action a from state s using the neural network policy output, and $Q(s, a)$ is the average of all final evaluations in the nodes below $\delta(s, a)$ [17]. Noise drawn from a Dirichlet distribution is combined with the prior at the root node of the MCTS in order to encourage exploration

The tree is followed down again based on an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) = C_{\text{puct}} \frac{P(s,a)}{1+N(s,a)}$. Once a leaf node is reached, an option is expanded and that node is evaluated using the neural network. The evaluation result is then propagated back up to all parent nodes in the tree. This process is shown in Figure 2.3.

2.3 Reinforcement Learning in General Game Playing

Very little previous work exists in applying RL to GGP. A framework, RL-GGP [18], created by José Luis Benacloch-Ayuso allowed testing of different RL algorithms, but little was published evaluating the performance of each in GGP.

Initial work in 2018 looked at applying Q-learning to GGP and found that it does converge, but quite slowly [19]. It also looked at extending the Q-learning with MCS which saw an improvement but still did not outperform UCT. However, this shows that there is potential for RL based GGP players.

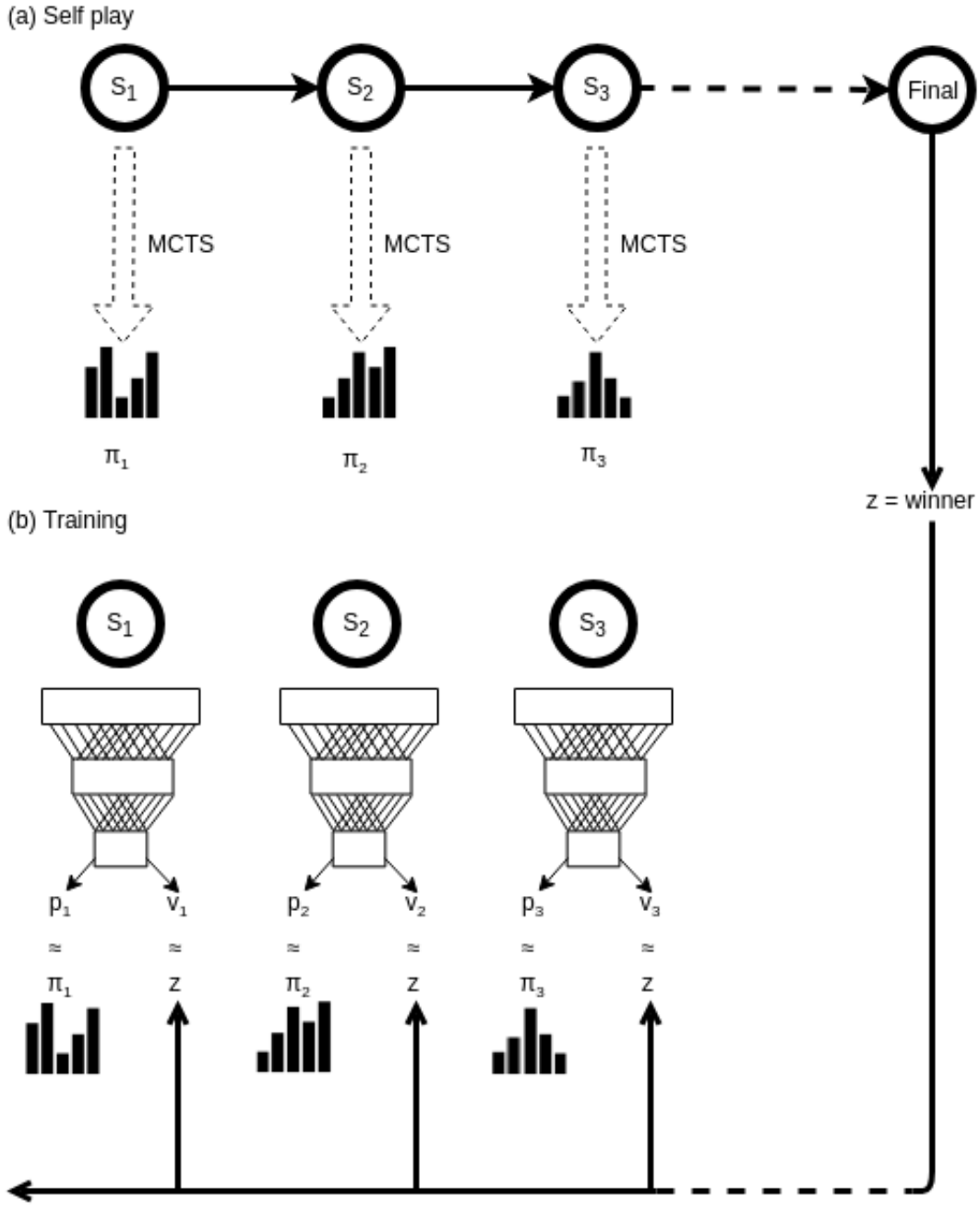

Figure 2.2: The self-play algorithm and training algorithm (a) shows the self-play process for a game (b) shows the new training data generated for each state visited in the game, image adapted from [17]

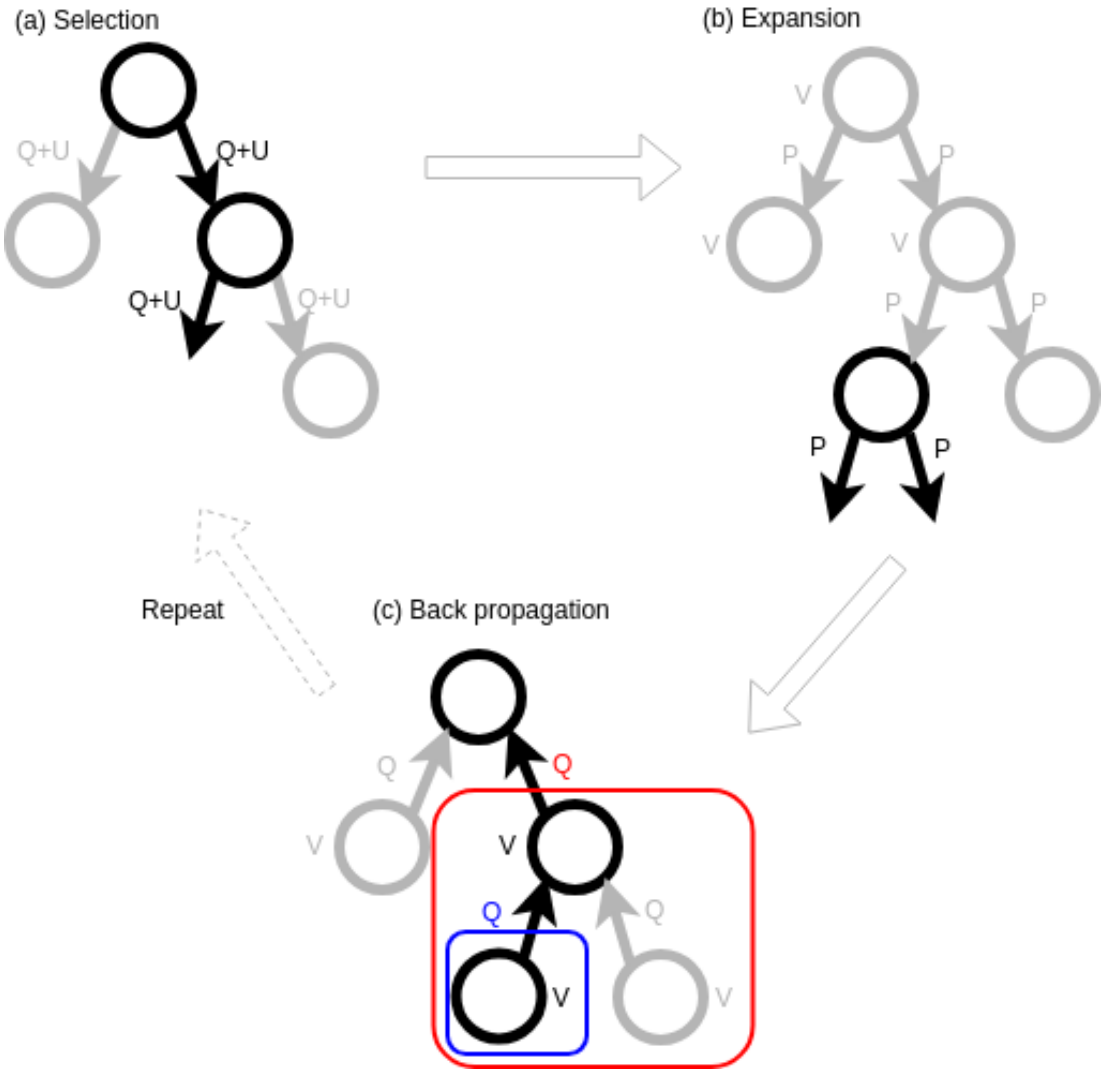

Figure 2.3: The MCTS improvement operator steps (a) shows the selection of a state based on the upper confidence bound (b) shows the expansion and evaluation of the new state (c) shows the propagation of the new evaluation back up the tree, image adapted from [17]

Chapter 3

AlphaZero Implementation

3.1 Overview

The implementation of the system was in Python 3.6 and Cython 3.6 using Tensorflow [20] for neural networks and was split into three parts. These are listed below along with their interfaces with each other.

- Game manager communication: This section deals with the network protocols used by the GM to communicate with the player. It calls the GDL and agent initialisation on receipt of the `start` message and the agent play method on receipt of the `play` message.
- Propositional network (propnet): This section initialises internal structures and provides methods to extract the current state of the game and to enumerate legal moves in the current state. It also provides a memory efficient state representation.
- Learning agent: This section performs the actual learning and playing, it provides an initialisation and a play method, which set up networks and choose actions respectively. Most actual learning is done in the initialisation phase.

For simplicity when both training and testing, the game manager was generally by-

passed. Its main use ended up being for easily trialling against other players during the debug stage.

The full code is available at `http://github.com/AdGold/ggp-beta`.

3.2 Propositional Network

The propositional network (propnet) was written in Cython for speed once it became clear that the Python version was too slow. This gave a 6× speedup over the original Python code.

One issue that appeared when using the propnet on large games was memory usage. Storing the entire state at each node in a MCTS proved to be too memory intensive for larger games, which inspired the following approach. The main observation is that typically a constant amount of the state changes at each step, this amount is very small compared to the size of the state so a way is needed of storing only the changes. This was accomplished using a persistent array to store the state of the propnet which allowed using $O(\log n)$ extra memory per change.

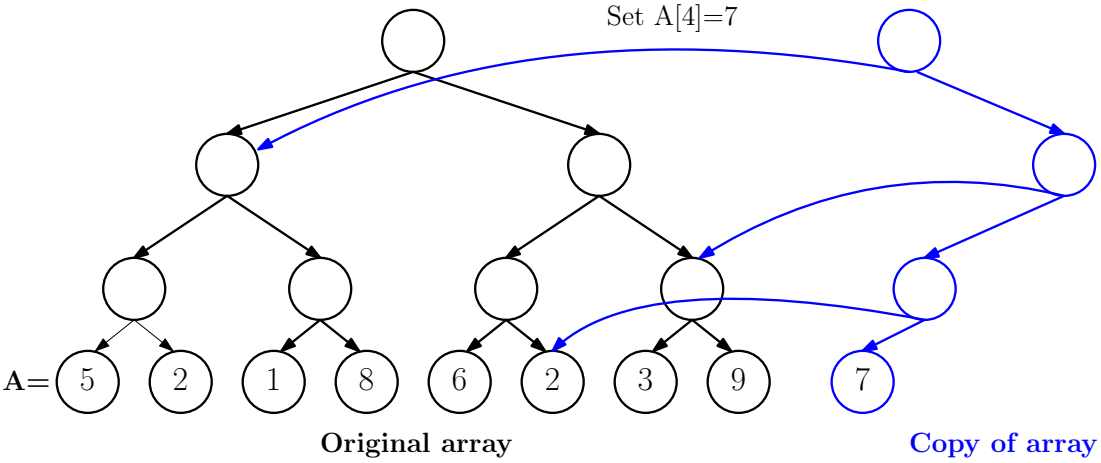

Figure 3.1: A persistent array with a single modification

The persistent array was implemented in the standard way by building a binary tree on top of the state array after which updates can be accomplished through simply

creating a new root node and new nodes down the path to the element that changed, but pointing to original nodes for subtrees which have not changed. This modification procedure is shown in Figure 3.1.

3.3 Learning Agent

There are a number of assumptions that AlphaZero makes about games that it plays, namely it assumes the following:

- the game is zero-sum
- the game is symmetric between players
- the game is for two players
- the game is turn based
- a neural network has been created specifically for each game
- the rules of the game are known

The last assumption is also assumed in GGP, so only the first five need to be addressed. The first is addressed by replacing the *expected winner* output of the neural network with an *expected reward*, or more specifically only allowing rewards between 0 and 1 — the rewards from GDL between 0 and 100 are rescaled by dividing by 100. After this change, each agent will simply try to maximise their own reward with no care for the others, so cooperative policies can be learned.

To deal with asymmetry between players, one method would be to keep a separate neural network for each player which is trained separately, however, it can be noted that these will all be trained to extract very similar features from the game state, giving rise to the option of combining the early layers of all the players' neural networks, as shown in Figure 3.2, this is a common technique in neural network architecture design as the different heads regulate the earlier layers as seen in Neural Machine Translation with a combined section then a separate head for each language [21]. A similar unification of earlier layers was implemented in AlphaGo Zero when the value and policy networks were combined [17]. This method has the added benefit of easily

generalising to multiple players and allowing simultaneous play without having to make a pessimistic assumption of the other players knowing your move — simply have all players predict a move at once and use the combined move as the move choice.

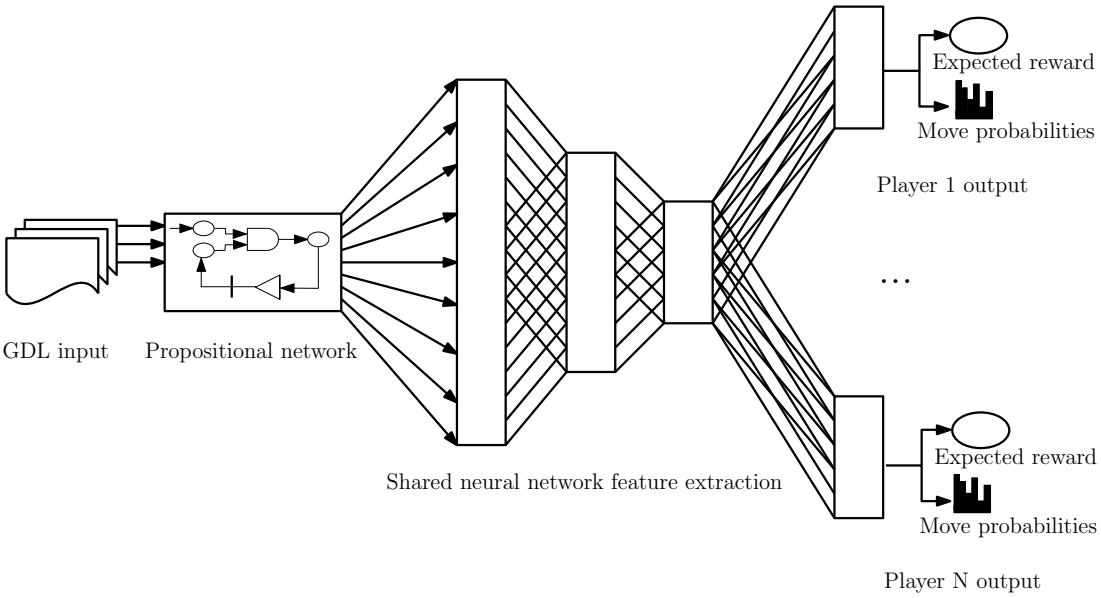

Figure 3.2: The network architecture of the neural network and its interface with the game specification.

Finally, the neural network can be constructed by initially taking every fluent from the game state in GDL as an input, then having a number of automatically generated fully connected layers.

These fully connected layers were generated as follows. Assume there are f fluents, the initial input of all f fluents is passed through a to fully connected hidden layer of size $\frac{f}{2}$ which is then passed through a fully connected hidden layer of size $\frac{f}{4}$, this is continued until $\frac{f}{2^k} \leq 50$ at which point it is passed into a layer of size 50. Then each head comes off that layer of size 50 as a common start, doubling the size of the layer with more fully connected layers until it reaches the number of legal actions. At this point there is also a fully connected layer which goes to a single output for expected return prediction. All hidden layers use `ReLU` activation, the expected reward uses `tanh` and policy output uses `softmax`. This construction is shown in Algorithm 1.

This diverges significantly from the convolutional residual blocks used in AlphaGo

Algorithm: setup_network(*num_fluents*, *num_outputs*)

size = *num_fluents*

while *size* $\geq$ *50* **do**

  Add new fully connected layer with *size* nodes and `ReLU` activation

  size = *size* / 2

end

Add new fully connected layer with 50 nodes and `ReLU` activation

Store current layer as *state*

for *each role* **do**

  Initialise current layer to build on to *state*

  head_size = 50

  while *head_size* $\leq$ *num_outputs[role]* **do**

    Add new fully connected layer with *head_size* nodes and `ReLU` activation

    head_size = *head_size* $\times$ 2

  end

  Store current layer as *head*

  Add new fully connected layer with *num_outputs[role]* nodes for policy head using `softmax` activation

  Add new fully connected layer to *head* with 1 node for value head using `tanh` activation

end

Algorithm 1: Network initialisation

Algorithm: train(*agent*, *model*)

while *Time left to train* **do**
  Re-initialise game state
  while *Game not finished* **do**
    Perform MCTS with *agent*
    Record new action probabilities, π, and new expected value, q, for each player
    Make moves proportionally to new action probabilities
  end
  Record final result z for each player
  Add triples of $(\text{state}, \pi, r \times z + (1 - r) \times q)$ to replay buffer
  Sample and train *model* on 10 mini-batches from replay buffer
end

Algorithm 2: High-level training loop

Zero [17] and AlphaZero [5]. It has far smaller value and policy heads and contains different heads for each player. These changes were made for two main reasons, most importantly was that convolutional layers require knowledge about ordering which is not given in GDL. Though there is some work with extracting this information [10, 22], it was outside the scope of this project. Second is that the large networks used for Go, Chess and Shogi all require large amounts of computational resources and a long time to train, making them difficult to use for GGP when training time is so limited.

The training loop was also very different. AlphaZero used a separate process dedicated to evaluating positions, one for training on new data and many for generating games [5]. This technique only makes sense if it is being run in extreme parallelisation, when running with access to only a single graphics card, it doesn't make sense to run training and simulation in parallel and without large clusters it does not make sense to run so many games simultaneously. Thus for both efficiency and simplicity, it was implemented by running a sample game, adding recorded data to the replay buffer, then training on 10 mini-batches from the replay buffer and repeating. This process is shown in Algorithm 2.

The choice of training on 10 mini-batches (of size 128) was made to make it so that it will train on around 5% of the data in the replay buffer (of max size 20,000) at each stage, this means that each state will be used for training around 20 times before it leaves the replay buffer. This strikes a good balance between not training on each sample enough, which would end up requiring significantly more training data as it would forget information, and training on each sample too much causing it to overfit.

Another difference between AlphaZero and the given implementation was in what value the expected reward was trained to approximate. AlphaZero simply used the final result of the self-play game (referred to as z) as after many games it should average to a good estimate. Following Prasad [23], information from the new expected reward at the root node after finishing the MCTS simulations was also considered — this is referred to as q below. These were combined using parameter r with the reward being trained to approximate $r \times z + (1 - r) \times q$. This parameter was varied in the range $r \in \{0, 0.5, 1\}$ for Connect-4 and the best value (0.5) was used for all further tests.

Finally, where AlphaZero used a single value α as a parameter for the Dirichlet noise added to the root of the MCTS search and scaled inversely with the average number of legal moves [5], this implementation explicitly scaled the noise inversely with the number of legal moves at that state. This is both more tailored to the state causing improved exploration and also simplifies the implementation as there is no need for initial randomised playouts to calculate the average number of legal moves.

Chapter 4

Evaluation

4.1 Evaluation Methodology

Instances were evaluated as follows. A set of 50 games were simulated of AlphaZero against a UCT agent with a time limit for both agents of 2 seconds and the first 2 moves randomised. The randomised moves were typically a `noop` and an actual move for each player.

UCT was chosen as a comparison as it forms a good benchmark due to variants of it being state-of-the-art for many years. However, both AlphaZero and UCT are mostly deterministic, with non-determinism only when two actions have the exact same count. This makes it difficult to run multiple tests to compare the two as all tests will result in the same game playouts. By randomising the first 2 moves, this allowed a number of independent tests to be run to give better results.

One important artefact of this randomisation method is due to it testing a wide variety of states and initial moves. This shows the breadth of the state space which the neural network has learnt but also means that it could win most games against a UCT agent but still lose when removing all randomisation and running a full game from the initial state.

All parameters were tuned on Connect-4 with a 6×8 board, then evaluated on the following games:

- Connect-4 (6×7 board) — 2 player, zero-sum, player-symmetric, turn based
- Breakthrough (6×6 board) — 2 player, zero-sum, player-symmetric, turn based
- Babel — 3 player, cooperative, player-symmetric, simultaneous
- Pacman 3p (6×6 board) — 3 player, cooperative/zero-sum, player-asymmetric, mixed turn-based/simultaneous

4.2 Results and Discussion

As can be seen from Figure 4.1, a choice of $r = 0.5$ is the best option of the three considered. This makes sense intuitively as when using $r = 1$, it relies only on the final result meaning even if it is in a good state, a single bad action might cause a low final reward. To get a reasonable estimate here more runs are required. On the other extreme with $r = 0$ it doesn't care enough about the long term so even though it becomes a good player quicker, the lack of grounding in final results causes more forgetting during training as seen in Figure 4.1. In earlier testing runs, using $r = 0$ also resulted in very unstable learning. Using $r = 0.5$ results in a simple but effective balance between the long term and short term expected rewards as can be seen by its far superior performance in tests. This reflects the results of Prasad [23] who additionally found even better performance with a linear dropoff between the two, however, this is less applicable to GGP as it requires knowledge of how long it will take to train and hence how quickly it should vary from 0 to 1. Moving average smoothing has been applied to Figure 4.1 in order to more clearly see the trends. Due to this averaging, error bars have been omitted, both for clarity and as they are less meaningful after smoothing.

Figure 4.1 also shows that this method performs comfortably better than a straight UCT player on Connect-4 after 1500 games of self-play (around 15 hours real time) and is significantly better after 3000 games of self-play (around 30 hours of real time).

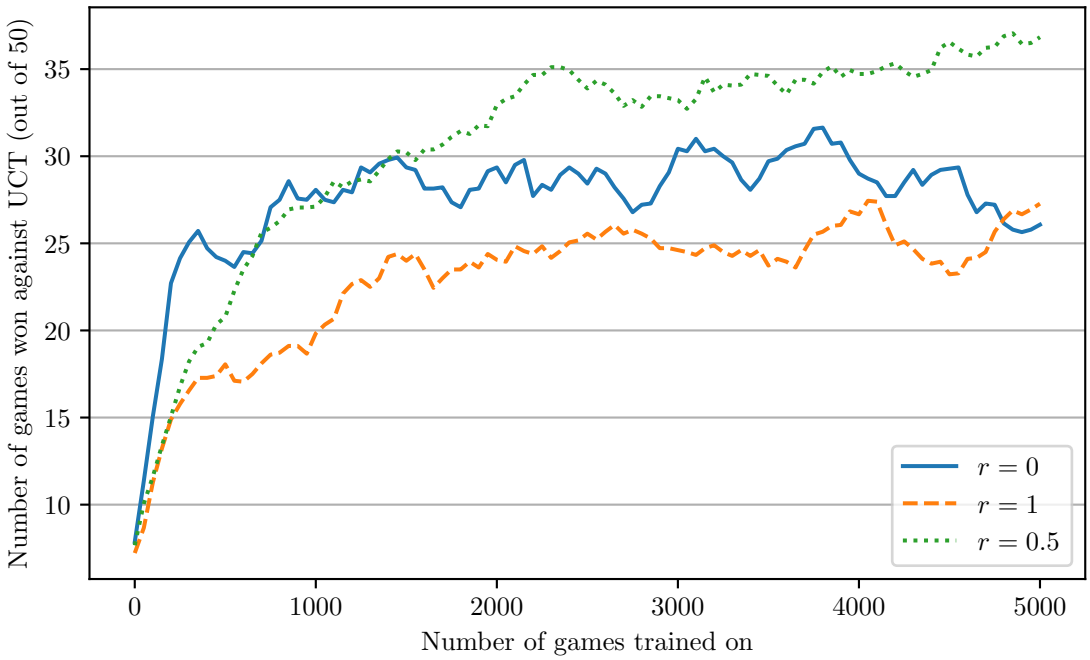

Figure 4.1: Three runs of training on Connect-4 (6×7 board), comparing $r \in \{0, 0.5, 1\}$, a moving average of kernel size 9 is applied

This is, of course, much too slow to use within a typical 10-minute *start clock* but shows the potential if learning was sped up through better algorithms or more computational power. Figure 4.2 shows the total training time compared to the number of self-play games.

Figure 4.3 shows AlphaZero's performance against UCT for Breakthrough. Interestingly it quickly reaches winning 49–50 games out of 50 after just 400 games of self-play (just over 10 hours real time). This is significantly better performance than was experienced for Connect-4, which is the type of game that the parameters were tuned on. While this is good evidence of the generalisability of the methods given, it is also affected by the game in this instance. Breakthrough has a much higher branching factor of typically around 16 as opposed to just under 7 for Connect-4. UCT struggles with higher branching factors, but AlphaZero was originally designed for the massive branching factor of Go so is able to cope with it more easily. Another advantage in Breakthrough is that the games can take a while if played out randomly, so the Monte Carlo rollouts in UCT take a fair amount of time, reducing the number of simulations

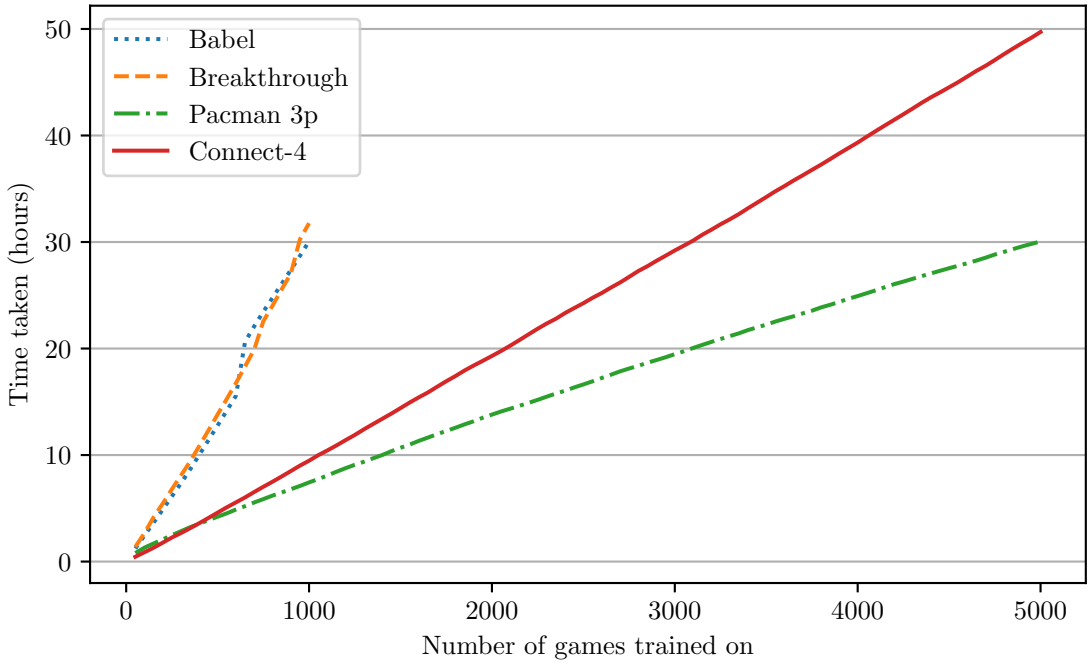

Figure 4.2: Time taken to train on different numbers of self-play games, for all tested games

that can be achieved in the 2 second time limit, comparatively, AlphaZero simply does a single forward pass through the neural network on the expansion of a leaf so can fit in many more simulations.

From the perspective of training time, Breakthrough is more achievable than Connect-4 with it beating UCT comfortably after only a single hour of training — meaning only a 6× speedup would be necessary to fit into the 10-minute limit. This would be very achievable even without parallelisation or major changes, simply by rewriting in C++ and optimising the game generation. While it took 10 hours to achieve results of 50 to 0, it got to high 40s after only 5 hours as can be seen in Figure 4.2. This shows that with a bit of work this method is definitely applicable to GGP on smaller games or games with high branching factors.

Both Connect-4 and Breakthrough are zero-sum, two-player player-symmetric games so in terms of changes from AlphaZero, these first two results mostly show three things:

- The improvement due to use of $r = 0.5$

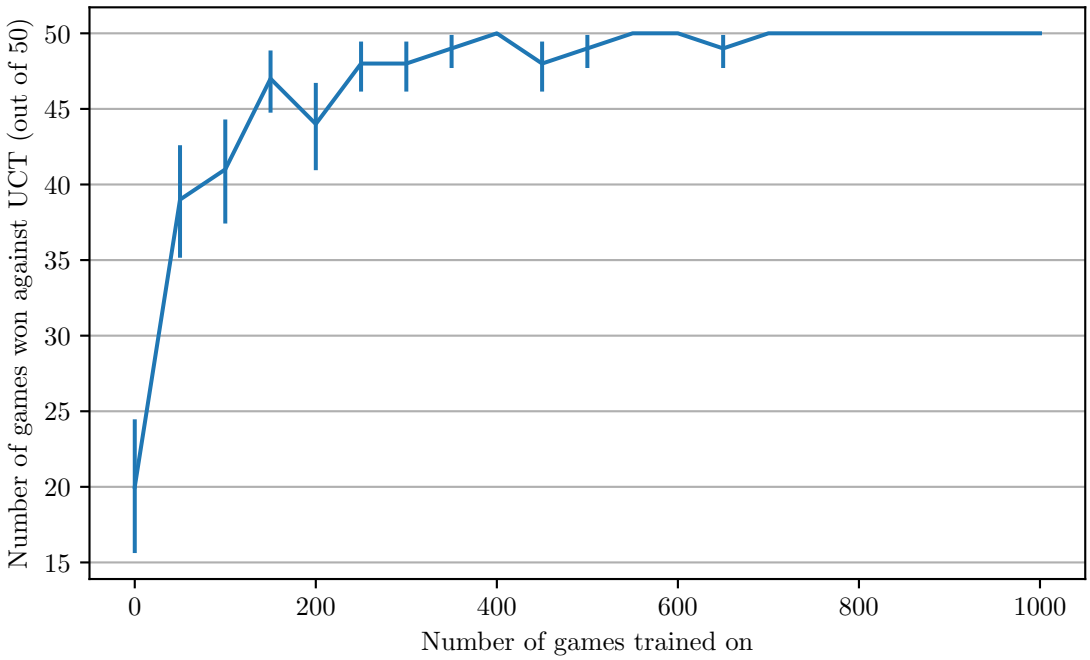

Figure 4.3: A training run of Breakthrough vs a UCT agent, error bars show the 80% confidence intervals.

- That for small games they can be quite efficient in the number of games needed
- That the major changes in network architecture and generalisation compared to the original AlphaZero implementation do not negatively impact the performance on standard games

Pacman 3 player is a game where one player controls Pacman and two players work together against the first, each controlling a ghost. For the evaluation, it was played in both directions — with Pacman being a UCT agent and both ghosts being AlphaZero agents as well as the reverse with Pacman being an AlphaZero agent and both ghosts being UCT agents. On all runs, the ghosts caught Pacman so reward as ghosts has been omitted from the graph shown in Figure 4.4 for clarity. The 'points against' series shows how well Pacman as a UCT agent scored against AlphaZero — the learning of the network as the ghosts, while 'points for' shows how well AlphaZero performed as Pacman against UCT — the learning of the network as Pacman. It is quite clear from the graph that it learnt very well as the ghosts but far less convincingly as Pacman. There are a few explanations for this, the first is that there is a far simpler strategy

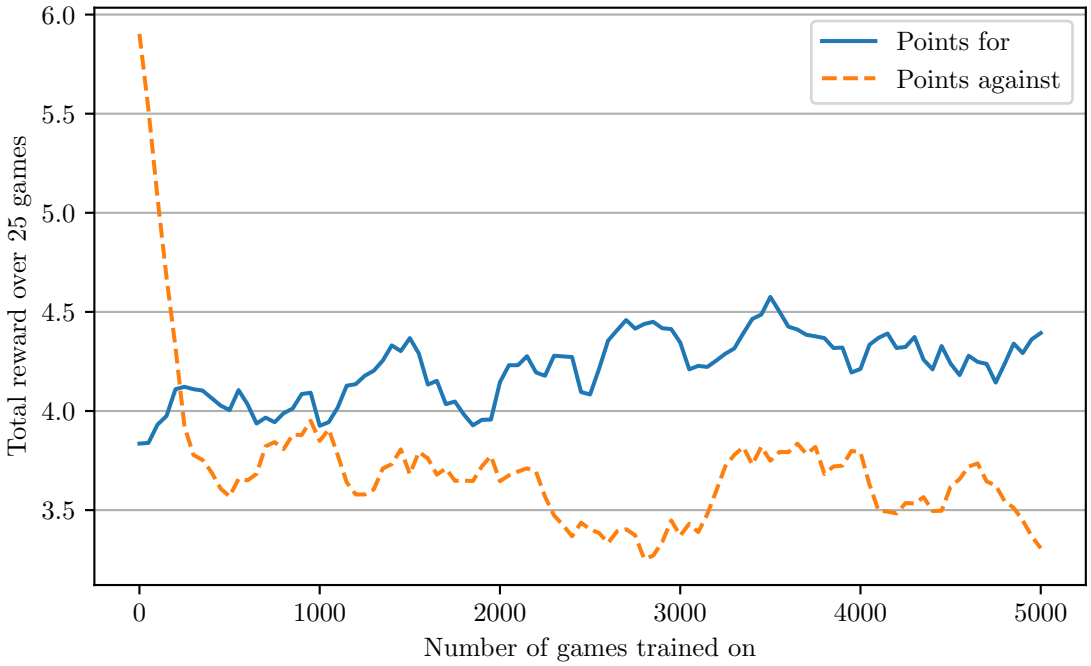

Figure 4.4: Points won by AlphaZero while playing as Pacman vs points won against AlphaZero while playing as the ghosts, a moving average of kernel size 9 has been applied

for ghosts (namely to simply move towards Pacman) than there is for Pacman who has to strategise to evade the ghosts. This simpler strategy can be more easily learnt by the neural network but also good strategies for Pacman may simply be too complex for the very small network architecture used to be able to encode or need a far larger set of games to train on in order to be able to learn. The final point that may have had a large impact is that it typically has a branching factor of around 2–3 (maximum 4) as Pacman which is the type of game where AlphaZero doesn't have as much of an advantage over UCT.

Despite all this, AlphaZero performed comfortably better than UCT, even quite early in training. This shows that even for games outside of its ideal area, using combined cooperative, zero-sum, simultaneous, turn-based and player asymmetric games, AlphaZero is still able to beat UCT after training. Note that error bars have again been omitted for clarity as they are less meaningful after smoothing.

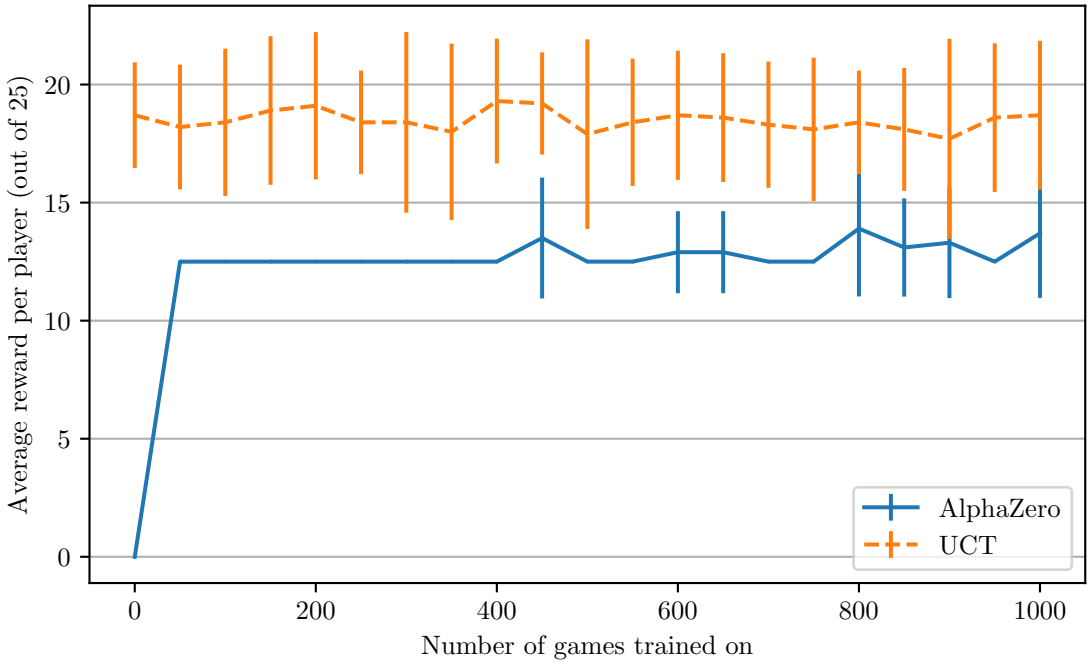

Figure 4.5: Three instances of AlphaZero playing Babel vs three instances of UCT playing Babel, error bars show 80% confidence interview

Babel is a game where players work together to build a tower. The results of AlphaZero are shown in Figure 4.5 alongside the results of a UCT agent, each run consists of either three AlphaZero agents or three UCT agents — mixed teams were also tested but fell monotonically in between as expected so are omitted for clarity. In this instance, AlphaZero never outperformed a basic UCT algorithm. The consistency between the three players due to the setup of the UCT agent caused a large advantage here because players tended to perform correlated actions which worked well as a strategy. Despite this, it is clear that AlphaZero was able to learn and improve still and it is likely that with some extra complexity in the neural network architecture and more training time it would be able to learn a more sophisticated approach. Note that some data points have error bars of 0 as they got the same reward on all runs of the program.

The timings for Babel and Pacman were similar to Connect-4 and Breakthrough and are also shown in Figure 4.2. For Pacman, it took 30 hours for the entire run, but only 7 hours before learning stabilised. Babel was a similar story with again 30 hours for the entire training run and 5 hours until learning stabilised. Again this is on the upper

edge, but with optimisation and faster or more parallelised computing, this could fit in the 10-minute *startclock* of GGP competitions. As can be seen in Figure 4.2, all games except for Connect-4 were run for around 30 hours, Connect-4 was run for longer in order to make sure the results for varying r were meaningful. Tests were run on an Intel Core i5 running at 2.9GHz and used a GeForce GTX 780Ti graphics card for neural network operations.

The speed of training appears to have been mostly dependant on the branching factor and the typical number of moves in a game. The branching factor changes how big a tree it has to look through, hence how quickly it can converge to a good path, with Pacman having by far the lowest branching factor and hence fastest training time. Breakthrough has the highest branching factor, however both Connect-4 and Babel have similar branching factors so the main difference in training speed there was due instead to Connect-4 having a strict 42 move limit whereas games of Babel can go for much longer, causing it to be on par with Breakthrough with respect to training time.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This project has proposed an extension to the AlphaZero algorithm [5] which removes limitations based on assumptions that the games being played are zero-sum, turn-based, two-player, player-symmetric and have a handcrafted neural network. It extends the policy-value network to remove these restrictions and presents a way of automatically scaling it based on the size of the state of the game.

The results of this project clearly show the potential for using AlphaZero within a GGP environment. It has performed noticeably better than the UCT benchmark agent in a number of games and has the potential to improve further with larger networks, more training time and further tuning.

The training times needed for the experiments have been well outside the typical 10 minute *startclock* of GGP competitions, however with simple speedups through rewriting in C++, parallelisation and faster hardware a large amount of training could be squeezed in to at least perform better than a basic UCT agent in a competition format. Other improvements in the tuning of parameters would also help significantly here as they could likely optimise for learning speed.

Some games were not tested due to lack of high specification hardware, specifically many took too long to train or required too much memory even with the memory optimisations outlined. In order to push this to a level where it could be used in a GGP competition, these computational issues would need to be addressed through either more optimised code or better hardware. A combination of both of these would result in quite a strong GGP agent with the potential to do extremely well in a competition setting.

5.2 Future Work

The largest missing area in this project is to do with scale, rerunning this on larger games with more compute power would give far more reliable results about the usefulness of this method in a general setting. With more power, it would also be interesting to run with larger neural networks to allow the learning of more complex strategies.

Additionally, there are a number of parameters that rather than tuning once, should ideally scale based on the properties of the game, in particular based on branching factor, complexity and size of the propnet. These could be used to vary, for example, the replay buffer size, the number of simulations in the MCTS during both training and evaluation, the minimum layer size, as well as the parameters r and C_{puct} for the tradeoff between long/short term reward and amount of exploration/exploitation respectively. On top of these, the network structure could include known facts about the game such as player symmetries to improve regularisation, convolutions over extracted ordinals in the game description and increasing network depth more strategically than simply based on the state size.

One of the remaining problems is improving learning speed, a promising approach here is to use transfer learning to allow knowledge learnt from other games to be used as a basis to be tailored specifically for each new game. This could help improve the amount of useful training which is fit into the *startclock* when playing.

Bibliography

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan 2016.

[2] M. Moravčík, M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, and T. Davis, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, pp. 508–513, Jan 2017.

[3] N. Brown and T. Sandholm, "Libratus: The superhuman ai for no-limit poker," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pp. 5226–5228, 2017.

[4] OpenAI, "Openai dota 2 1v1 bot." `https://openai.com/the-international/`, August 2017. Accessed: 26/04/2018.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *ArXiv e-prints*, Dec 2017.

[6] M. Genesereth, "Game definition language." `http://games.stanford.edu/games/gdl.html`. Accessed: 16/05/2018.

[7] M. Genesereth and M. Thielscher, *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, Mar 2014.

[8] "International general game playing competition - past winners." `http://ggp.stanford.edu/iggpc/winners.php`, 2016. Accessed: 10/04/2018.

[9] J. Clune, "Heuristic evaluation functions for general game playing," in *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pp. 1134–1139, AAAI Press, 2007.

[10] S. Schiffel and M. Thielscher, "Fluxplayer: A successful general game player," in *Proceedings of the AAAI National Conference on Artificial Intelligence*, pp. 1191–1196, AAAI Press, 2007.

[11] H. Finnsson and Y. Bjrnsson, "Simulation-based approach to general game playing," in *Proceedings of the 23rd National Conference on Artificial Intelligence*, vol. 1, pp. 259–264, AAAI Press, 2008.

[12] J. Méhat and T. Cazenave, "A parallel general game player," *Künstliche Intelligenz*, vol. 25, pp. 43–47, Mar 2011.

[13] F. Koriche, S. Lagrue, E. Piette, and S. TabaryKJ, "General game playing with stochastic csp," in *Constraints*, vol. 21, pp. 95–114, 2016.

[14] P. Auer, "Using confidence and bounds for exploitation-exploration trade-offs," *The Journal of Machine Learning Research*, vol. 3, pp. 397–422, 2003.

[15] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb 2015.

[17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, Oct 2017.

[18] J. L. Benacloch-Ayuso, "Rl-ggp." `http://users.dsic.upv.es/~flip/RLGGP/`. Accessed: 14/04/2018.

[19] H. Wang, M. Emmerich, and A. Plaat, "Monte carlo q-learning for general game playing," *ArXiv e-prints*, Feb 2018.

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[21] J. Gu, H. Hassan, J. Devlin, and V. O. K. Li, "Universal neural machine translation for extremely low resource languages," *ArXiv e-prints*.

[22] G. Kuhlmann and P. Stone, "Automatic heuristic construction in a complete general game player," in *AAAI*, 2006.

[23] A. Prasad, "Azfour: Connect four powered by the alphazero algorithm." `https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191`, 2018. Accessed: 15/09/2018.