Expert Iteration

Thomas William Anthony

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Computer Science
University College London

March 5, 2021

I, Thomas William Anthony, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Our Unfailing Onesidedness

Jeg sidder og hviler	I sit and rest
min livstraette Sjael	My world-weary soul
ved at spille et Slag	By playing a game
Polygon med sig selv.	Of Polygon all alone.
Der er bare den Fejl	There is only one flaw
ved min ensomme Leg:	In my solitary play:
jeg kan ikke la vaer	I cannot stop siding
med at holde med mig.	With myself all the way.

Piet Hein, inventor of the game *Hex*

(from Hayward and Toft [2019])

Abstract

In this thesis, we study how reinforcement learning algorithms can tackle classical board games without recourse to human knowledge. Specifically, we develop a framework and algorithms which learn to play the board game Hex starting from random play.

We first describe Expert Iteration (ExIt), a novel reinforcement learning framework which extends Modified Policy Iteration. ExIt explicitly decomposes the reinforcement learning problem into two parts: planning and generalisation. A planning algorithm explores possible move sequences starting from a particular position to find good strategies from that position, while a parametric function approximator is trained to predict those plans, generalising to states not yet seen. Subsequently, planning is improved by using the approximated policy to guide search, increasing the strength of new plans. This decomposition allows ExIt to combine the benefits of both planning methods and function approximation methods.

We demonstrate the effectiveness of the ExIt paradigm by implementing ExIt with two different planning algorithms. First, we develop a version based on Monte Carlo Tree Search (MCTS), a search algorithm which has been successful both in specific games, such as Go, Hex and Havannah, and in general game playing competitions. We then develop a new planning algorithm, Policy Gradient Search (PGS), which uses a model-free reinforcement learning algorithm for online planning. Unlike MCTS, PGS does not require an explicit search tree. Instead PGS uses function approximation within a single search, allowing it to be applied to problems with larger branching factors.

Both MCTS-ExIt and PGS-ExIt defeated MoHex 2.0 - the most recent Hex Olympiad winner to be open sourced - in 9×9 Hex. More importantly, whereas MoHex makes use of many Hex-specific improvements and knowledge, all our programs were trained *tabula rasa* using general reinforcement learning methods. This bodes well for ExIt's applicability to both other games and real world decision making problems.

Impact Statement

Hex, like many classical board games, is easy to learn, but difficult to master. To play well requires precision; the effects of subtle changes in strategy can be dramatic; and the number of possible positions is vast. Games with these characteristics have often proven too difficult for reinforcement learning algorithms, and in Hex no prior competitive players have been built without human knowledge.

In this work we develop and study new general reinforcement learning algorithms that outperform Olympiad-winning Hex AIs, while training *tabula rasa*.

Real-world problems, such as controlling industrial processes, managing a distribution network or driving a vehicle, are sequential decision making tasks. By and large, they are either carried out by expert humans or are automated by handcrafted rule-based systems designed by domain specialists. A sufficiently strong general reinforcement learning algorithm could reduce the need for human labour and expertise to perform these tasks, while performing them better than is possible today.

Reinforcement learning algorithms that can learn to act effectively in a subtle large-scale board game like Hex move us closer to reinforcement learning methods that can cope with the complexity and scale of problems in the real world.

Acknowledgements

There are many people whose support has been invaluable. My heartfelt thanks:

To Zheng Tian, for your tireless hard work and belief as we developed the first Expert Iteration algorithm.
To Ronnie Stafford, for our many discussions, and everything I learnt from them.
To Andrew Clarke for help with efficiently parallelising the data generation, and Alex Botev for assistance implementing the CNN, for the first Expert Iteration algorithm.
To Robert Nishihara and Philipp Moritz for your help on Policy Gradient Search, and the rest of the OpenAI team, for your warm welcome, stimulating discussions and advice.
To UCL and EPSRC for funding this PhD.
To AWS Cloud Credits for Research and OpenAI for computational resource.
To Theophane Weber, for valuable feedback on a draft of this thesis.
To David Barber, for your supervision and guidance.
To all the students in David's group, for the fun and laughter.
And most of all, to my parents Helen and Gerald, my sister Louise and my wife Qing, for your unfailing love, support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A grand ambition of Machine Learning is to create systems that autonomously interact with the world and — without reliance on limited human labour or expertise — learn to perform useful work, potentially better than people could.

Reinforcement Learning (RL) [Sutton and Barto, 2011, Szepesvári, 2010] studies how to solve this problem in general, by considering interactive tasks in terms of a sequence of decisions. In principle, a general-purpose reinforcement learning algorithm could be presented with a new task to perform, and would be able to solve it without any further input. We refer to a reinforcement learning algorithm as an agent, since it interacts with the environment at large.

Owing to the generality of the reinforcement learning problem, instances of RL problems can be difficult in many different ways. Some have huge numbers of possible observations that are difficult to summarise. For example if, as in the Arcade Learning Environment [Bellemare et al., 2013], the agent observes the world around it via a video feed, then solving the reinforcement learning problem entails solving a computer vision problem. Recent work in model-free deep reinforcement learning has invigorated the field, in particular due to progress in tackling problems with this characteristic [Mnih et al., 2015, 2016].

Nonetheless, many challenges remain. Two issues that can make sequential decision problems too hard for recent model-free algorithms are:

- If a mistake in a long sequence of decisions are enough to dramatically worsen the outcome for the agent.
- If a subtle change to the state of the world can dramatically change how good the state is.

These difficulties can be thought of as *sensitivity*, and they often appear together. Small changes in either the decisions of the agent or the state of the environment have a disproportionate impact on performance. Sensitivity can be particularly challenging in combination with large state spaces. A minor change with a large impact might be thought of as a 'corner-case', but if the state space is large, with many such crucial subtleties, it is impossible for each corner-case to be considered by the agent during training. Instead, the agent must generalise its understanding from a few examples. Generalisation, or interpolation and extrapolation, implicitly assumes some kind of smoothness[1] in the function being approximated. Unfortunately, sensitivity implies that important properties of the game - such as the values of states and actions - do not exhibit such smoothness.

Classical board games often display this kind of sensitivity. For example, high-level chess games are usually balanced on a knife edge. The locations of pieces interact in complex ways, such that if a single piece were moved or a single mistake made, it would completely change the course of the game. They also tend to have an exponential number of possible states, making heuristics or function approximation essential to play them well.

In many classical board games, learning-based methods have required supplementation with human knowledge or other techniques to achieve state-of-the-art performance, or are not used by state-of-the-art programs at all.

For example, in the chess engine Deep Blue [Campbell et al., 2002], heuristics for chess concepts such as development, trapped pieces, pins and passed pawns were used. These important features are complex functions of the state that are challenging for function approximators to learn accurately otherwise.

[1]Meant here in a colloquial, not mathematical sense.

Search based methods have also been crucial for AI in board games. These techniques explicitly plan ahead, which can help address sensitivity because the resulting look-ahead can identify and correct mistakes in heuristic evaluation. With look-ahead, the effects of initially subtle differences in moves or state can become more apparent, and therefore easier for a heuristic to capture. Continuing our example, Deep Blue used very large searches, assessing over 100 million games per second.

Moreover, Deep Blue used Quiescence search, where the sensitivity of evaluation is estimated based on a heuristic for the volatility of the position. An example of a volatile position is one where captures are likely. Where a position is volatile, heuristic evaluation is likely to be unstable and inaccurate. To alleviate this, in quiescence search search is continued until a quieter (and less sensitive) position is reached.

1.1 Problem Statement and Scope

The aim of this thesis is to answer:

Can a general reinforcement learning algorithm learn to play a large classical board game at a level competitive with human players and specialised AIs, without using human expert knowledge of game strategy?

By a classical board game, we mean a zero-sum, perfect-information alternating Markov game (see section 2.5). Famous examples include tic-tac-toe, connect 4, chess and Go. For this thesis, a game is large-scale if the number of legal positions is orders of magnitude larger than the memory of any existing computer. Such a size usually means that methods based on look-up tables or simple (i.e. linear) heuristics will fail. We care about methods that scale to such sizes because real world problems are typically large-scale. By this understanding, tic-tac-toe, with 765 states, and connect 4, with 4,531,985,219,092 states [Edelkamp, 2017], are small. On the other hand chess, with approximately 10^{43} states [Shannon, 1950], and Go, with approximately 10^{170} states [Tromp and Farnebäck, 2006], are large.

By a general algorithm, we mean one that can tackle many games of some class, in our case, alternating Markov games. Techniques that only apply to a specific game in the class would therefore not satisfy our goal. In essence, we want an algorithm that we expect to perform well out-of-the-box if applied to most classical board games.

Game specificity often involves the use of human expert knowledge of game strategy. This kind of knowledge includes datasets of human play, heuristics hand-crafted by the program designer using strategic insights, specialist algorithms that calculate strategies in game-specific ways, and auxiliary shaping rewards that wouldn't apply to other games. All these approaches at some level rely on human insight into how to play this specific game well. If the algorithms we build are to solve problems beyond the reach of human intelligence, reliance on these insights is limiting.

We refer to a method that does not use human expert strategy knowledge as learning *tabula rasa*, as it does not use concrete preconceived ideas on how to play the game. Instead the engineer of the system does not need to provide any knowledge of what would be a good decision in any situation, only decisions about machine learning techniques. Our reinforcement learning algorithms will make use only of properties generally applicable to the class of alternating Markov games, such as the theory of minimax optimal strategies (see section 2.5).

Given this reinforcement learning algorithm, the design decisions to be made by the engineer to apply it to a specific game are the representation of the game and the function approximator used. Importantly, the engineer would be able to draw on standard machine learning techniques, and require only basic knowledge — such as game rules — about the target domain.

The representation of the game determines how the AI can interact with it, and is inherently somewhat game specific. Choices here can encode different amounts of information about the game, for example a chess position could be represented simply by the moves so far in standard notation, however this would likely reduce the performance of any AI system. A natural representation would be to represent the positions of pieces in an $8 \times 8 \times 12$ binary array, where each plane represents combination of piece type and colour; this encodes the 2-d structure of the game, which is inherent in the rules. Other features based directly on game rules could also be included in the representation, such as whether a player is allowed to castle, or whether a player is in check. These do not introduce new strategy concepts that don't already exist in the rules, so we will consider them permissible. What would not be permissible is to add to the representation features such as piece value heuristics, or a measure of king safety.

To use our reinforcement learning algorithm, a function approximator must be chosen. This design decision is not a focus of the work, however the choice made will affect both generality of the overall system, and the strategies the system learns. Similarly to the choice of representation, we want to avoid using methods that result from insights into game strategy, but will choose from standard techniques a function approximator is well suited to a standard representation of our game. In this work we will use different kinds of convolutional neural networks. These are well proven in a diverse range of tasks, and they are appropriate for games with a regular 2-dimensional grid structured board. While our reinforcement learning methods could transfer to a game without this structure, such as a card game, the choice of function approximator would likely have to change for best results.

1.1.1 Hex

In this thesis we test the algorithms we propose on the board game *Hex* [Gardiner, 1957, Browne, 2000, Hayward and Toft, 2019]. Hex was invented as *Polygon* by polymath Piet Hein [Hein, 1942], and later independently invented by mathematician John Nash [Nash, 1952].

The rules of Hex are very simple: it is a two-player connection-based game played on an $n \times n$ hexagonal (i.e. honeycomb) grid. The players, denoted by colours black and white, alternate placing stones of their colour in empty cells. The black player wins if there is a sequence of adjacent black stones connecting the North edge of the board to the South edge. White wins if they achieve a sequence of adjacent white stones running from the West edge to the East edge (see figure 1.1). By convention, black moves first.

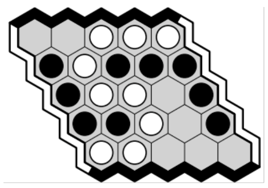

Figure 1.1: A 5×5 Hex game, won by white. Figure from Huang et al. [2013].

We chose Hex because it is an abstract strategy game with the kind of complex strategy interactions that cause sensitivity. Its simple rules make it fast to simulate games of Hex, which is convenient for research. There have been considerable efforts in creating specialist Hex AIs, which provide benchmarks for our algorithms.

We mostly study the game on a 9×9 board in this work. At this size Hex there are approximately 10^{37} possible states if states where one player has won are included. The number of non-terminal states can be loosely lower bounded by only considering positions without enough stones to complete a game, which leads to a bound of 10^{22} [Gao, 2020]. These estimates confirm that 9×9 Hex comfortably exceeds our definition of 'large'.

1.2 Hex Strategy

Hex strategy is rich,[2] good play requires both judgement and understanding of different concepts, we describe just some of those here.

1.2.1 Virtual Connection

A key concept in Hex strategy is the idea of a virtual connection. In a state s, cells c_1 and c_2 are said to be virtually connected for player i if there exists a strategy by which player i can guarantee to connect c_1 to c_2, even if player

[2]See Browne [2000] or `http://www.trmph.com/hexwiki/Basic_strategy_guide.html` for Hex strategy guides

i does not play first. Such a strategy will rely on some cells being empty, we call this set of cells the support of the virtual connection, and denote it S. A semi-connection is like a virtual connection, except that it requires that i plays first.

Some virtual connections can be calculated via combination rules [Anshelevich, 2000, Pawlewicz et al., 2015]. For example, if multiple semi-connections exist between c_1 and c_2, and those semi-connections have supports S_i, with $\cap S_i = \emptyset$, then there is a virtual connection between c_1 and c_2, with support $\cup C_i$, because the opposing player cannot play first in every semi-connection simultaneously.

If: c_1 is virtually connected to c_2; c_2 is virtually connected to c_3; these connections have disjoint support; and c_2 is empty, then c_1 is semi-connected to c_3, with a first move of playing in c_2.

Chaining rules such as these to virtual connections allows many complex connections to be automatically found. Efficiently doing this has been a key component in all recent competitive Hex programs [Pawlewicz et al., 2015], and is an important concept for beginner players to grasp.

1.2.2 Ladders

Ladders are situations where a sequence of adjacent moves are made in a chain, extending in a straight line across the board, often parallel to the board edge. First an attacking player makes some move that threatens to make a connection, and then their opponent plays a move that blocks that connection. The attacker then plays next to their original move, threatening to play the same connection, one place further along. The defender once again blocks. This can then continue until either the other edge is reached, or some other stones are reached.

When other stones are reached, their configuration can determine which player wins the fight. At it's simplest, an additional piece for the attacker can allow them to 'jump forward' one move along the ladder, resulting in two threats, which cannot both be defended. This is known as a 'ladder escape' (see `http://www.trmph.com/hexwiki/Ladder_escape.html`) for an example.

1.2.3 Mustplay

In Hex, the winning conditions for the two players are mutually exclusive. To prevent their opponent from making a connection, a player must make a transverse connection to block it. This means that offence and defence are two sides of the same coin in Hex. A player may identify the weakest point in their offence (the connection they are attempting to make) and defend that connection against threats, in doing so, they will also make new threats to connect groups of stones.

Alternatively, a player might identify what threats their opponent is posing, and look for ways to defend against them (or, to threaten a transverse connection of their own). At it's most extreme, if the opponent has a semi-connection between the two edges of the board, then a player must prevent that connection from forming to avoid defeat. Therefore, their next move must lie in the support of that semi-connection. If the opponent has multiple semi-connections, the player must play in all of them at once. In Computer Hex, this subset of moves is known as the *mustplay*, and algorithms for identifying the mustplay have been an important part of Hex programs, as it can both reduce the search space considerably, and prevent many possible blunders [Arneson et al., 2010, Huang et al., 2013].

1.2.4 First player advantage

Nash proved that player 1 has a winning strategy in Hex, via a strategy stealing argument [Hayward and Toft, 2019]. Informally, the proof says: suppose that player 2 had a winning strategy. Then player 1 could start the game with a random move, and thereafter follow the winning strategy for player 2 as though the randomly placed stone wasn't there. If the winning strategy ever requires player 1 to play on the random initial stone, they can play a new random move instead. This must be a winning strategy for player 1, which contradicts our assumption.

In practice, playing first gives a large advantage. In games between two players of similar standard, this advantage is so large as to be effectively unassailable. In human games, this is tackled by use of the swap rule:[3] player 1 plays a first move, then player 2 can choose whether to continue playing as white as normal, or to swap colours. If they choose to swap, the next move will be played by player 1, with the white stones. With the swap rule, perfect play results in a win for player 2, who can select whichever colour has the advantage after the first move. However, since player 1 can choose a move that gives no clear advantage to either side, player 2 will not gain an unassailable lead. The Hex tournament at the Computer Olympiad also uses the swap rule.

In all experiments for this thesis, we tackle the first player advantage with an alternative method common in the literature [Huang et al., 2013]. To measure performance of two agents A_1 and A_2, for each of the n^2 legal opening moves, we play two games starting with that first move: one with agent A_1 playing as black, one with agent A_2 playing as black. This allows us to compare the agents in a variety of states. We refer to this set of $2n^2$ games of $n \times n$ Hex as a *match*.

1.3 Contributions and Method

In this thesis we develop reinforcement learning algorithms that learn to play Hex to a level exceeding prior heuristic search methods. To do so, the algorithms make use of knowledge of game rules in the form of a game simulator, and in some experiments aspects of game rules are also encoded in the representation used as input to the neural network. Machine learning expertise is applied to choose the function approximation methods employed by the algorithm, but no knowledge of Hex strategy was required to apply the algorithm successfully. Thus these algorithms have achieved strong play tabula rasa, in the sense described in section 1.1.

To achieve this, we first introduce a novel algorithm family, Expert Iter-

[3]also known as the pie rule. See `https://en.wikipedia.org/wiki/Pie_rule`

ation (ExIt), which brings search and imitation learning techniques — that have been previously been successful in board games — together in a policy iteration algorithm, in order to replace the use of human expertise with a sequence of automatically generated experts. Algorithm 1 gives an informal description of the ExIt algorithm.

Algorithm 1 Expert Iteration (Informal)

```
Require: Apprentice policy function $\pi_\theta$ (and optionally value function $V_\theta$)
    with parameter space $\Theta$
Require: Initial parameter values $\theta_0 \in \Theta$
Require: Expert search policy $\pi^*(a|s, \pi_\theta, V_\theta)$, which returns a distribution
    over actions from the state $s$, and depends on apprentice value and policy
1: for $i = 0$; $i \leq$ max_iterations; $i$++ do                ▷ One expert iteration
2:     Generate dataset $D$ of state, expert policy pairs $(s, \pi^*(a|s, \pi_{\theta_i}, V_{\theta_i}))$
3:     Train parameters $\theta_{i+1} \in \Theta$ to minimise imitation learning loss on $D$
4: end for
```

The algorithm requires a function approximator π_θ to represent policies for the game (and optionally other heuristics required by search, such as state values, too), known as the 'apprentice' policy. It serves the role of learning to play well in any possible state, and is trained from examples of strong play. The other requirement is an 'expert' policy, which is a planning algorithm that can be applied to any state to produce strong actions. The expert planner will use the apprentice policy (and value function) as heuristics to guide search, but will investigate the actions suggested by the apprentice through lookahead, identifying which actions are stronger and which are mistakes. In particular, the expert can consider deviations from the apprentice policy over multiple game turns, that lead to play that is far stronger than the apprentice is currently producing. This stronger play can therefore be used as examples to retrain the apprentice policy. Because the expert calls the apprentice policy in many states, it is much slower to evaluate the expert policy than the apprentice policy. Therefore, to ensure our algorithm is performant, the apprentice should be fast to evaluate.

At a high level, the algorithm iterates through two steps. First, a dataset of states and expert actions is generated (line 2). Then, the apprentice policy is trained to minimise an imitation learning loss on this dataset (line 3). Because the expert actions generated in line 2 are much stronger than the actions of the policy, solving this imitation task leads to stronger apprentice play. As a result, on the next iteration, the expert has access to better heuristics, and produces even stronger play, leading to a virtuous cycle of improvement, as each iteration of the algorithm creates a stronger expert for the apprentice to learn from.

Both search and imitation have been extensively used in previous AI for games; typically both have relied on human strategy expertise. Multi-step search algorithms have often used heuristics handcrafted with guidance from human strategy experts. Imitation learning methods have often relied on human expert gameplay data for training. In contrast, our reinforcement learning approach uses search to generate new 'expert' knowledge automatically.

Other approaches to combining planning and learning have been tried in the literature: search based value learning methods such as TD(leaf) and TD(root) (see section 2.6.7) extend Temporal Difference learning by using search, while Guided Policy Search-style methods (e.g. Levine and Koltun [2013]) combine planning with policy gradient learning. In contrast, ExIt builds on classification-based policy iteration [Lagoudakis and Parr, 2003], and is the first to use search to find multiple-step policy improvements for a policy iteration algorithm.

We demonstrate the effectiveness of the Expert Iteration framework with a practical algorithm using Monte Carlo Tree Search (MCTS) [Browne et al., 2012] as the expert planner, and neural networks for the apprentice policy and value functions. This algorithm, first published in Anthony et al. [2017], was able to defeat MoHex 2.0, the strongest open-source Hex AI, in 9×9 Hex. We investigate different options for the imitation learning loss; how using policy heuristics affects MCTS at different points in training; and demonstrate how the final performance can far exceed the performance of early experts, unlike when using imitation learning techniques.

Concurrently to our work, [Silver et al., 2017] developed AlphaGo Zero, which also uses the ExIt approach to train policies from MCTS, achieving state-of-the-art performance. [4] They applied their algorithm to Go, and subsequently demonstrated state-of-the-art performance with a simplified version in chess and Shogi also [Silver et al., 2018]. We replicated AlphaZero and applied it to Hex, demonstrating that it too can defeat MoHex 2.0. Overall, these works convincingly demonstrate that the ExIt approach is able to learn a range of different board games.

We do additional studies using AlphaZero as a base. First we demonstrate that multiple-step improvements are important to performance in Hex, showing that ExIt is more capable than previous single-step policy iteration methods. Second, we show that using a temporal-difference learning style training targets for value functions in ExIt leads to much weaker performance than the Monte-Carlo targets used by both our work and AlphaZero. This could explain why previous attempts to combine search with temporal difference learning were less successful than ExIt.

So far, practical implementations of Expert Iteration had all used MCTS as their search algorithm. If the success was tightly coupled to this particular choice of expert, then the approach would only be applicable in those domains that MCTS is practical for. In Anthony et al. [2018] we developed Policy Gradient Search, a novel search algorithm that applies policy gradient learning in a local search. Compared to MCTS, PGS is applicable to a wider range of tasks, as it does not require the ability to build an explicit search tree. We demonstrate that this is competitive with MCTS in Hex as a test-time search method, and perform an ablation study over several design choices. Finally, we show that Expert Iteration using Policy Gradient Search is an effective learning algorithm in Hex, able to defeat MoHex 2.0. This shows that Expert Iteration can indeed be successfully applied with different search methods, and

[4]We discuss the differences between the two algorithms in section 4.5.1

motivates further study of search methods that are applicable in an expanding range of domains.

1.3.1 Thesis Outline

We proceed as follows:

- Chapter 2 reviews reinforcement learning in general, prior works that we build on in this thesis, and prior AIs for classical board games.
- Chapter 3 introduces our overall approach, a framework for training agents using search which we call Expert Iteration (ExIt). We discuss how different machine learning components are used to decompose the learning problem. We motivate this decomposition by analogy to the dual process theory of human reasoning, and hypothesise how the Expert Iteration framework can provide benefits over model-free reinforcement learning approaches.
- Chapter 4 descibes a practical implementation of Expert Iteration using Monte Carlo Tree Search (MCTS) as the 'expert' planning algorithm. We discuss how the work relates to the contemporaneous AlphaGo Zero [Silver et al., 2017], which we replicate.
- Chapter 5 introduces Policy Gradient Search (PGS), and shows that PGS can perform similarly to MCTS in Hex as a test-time search algorithm, and that it performs well as an expert in ExIt for Hex.
- Chapter 6 concludes the thesis by connecting this work to other recent works, and highlights some open questions raised by the results in this thesis.

Chapter 2

Background and Literature Review

2.1 Reinforcement Learning

In reinforcement learning (RL) [Sutton and Barto, 2011, Szepesvári, 2010], we consider sequential decision making problems where an agent interacts with an environment, making decisions at a series of discrete timesteps. In each timestep, the agent observes the state of the environment s_t, and selects an action a_t to take from an action space A. Following this action a reward r_{t+1} is received, and the environment procedes to the next state, s_{t+1}.

We call the sum of the rewards across time the return. The goal of the problem is to find actions that maximise the expectation of the return (with respect to any stochasticity from the environment or the agent's policy). Two cases may be considered here: *episodic* or *infinite-horizon* tasks. In an episodic task, at some timestep T, the environment reaches a terminal state, gives a final reward r_T, and no more reward can be obtained after. Therefore the return is $R = \sum_{t=1}^{T} r_t$. The termination condition often has some meaning, for instance when a timer runs out, or when some goal has been achieved.

In the infinite-horizon case, the episode never terminates. This means that the sum of rewards could be infinite, even if the reward at each timestep is bounded. To prevent this, we consider a discount factor $\gamma < 1$, which exponentially devalues more distant rewards. Our return in this case is $R = \sum_{t>0} \gamma^{t-1} r_t$.[1] In most respects, algorithms for episodic and infinite-horizon discounted problems are interchangeable. Because Hex games have a finite length, we will consider the episodic case in this work.

We define the return after time t_i as $R_{t_i} = \sum_{t>t_i} \gamma^{t-t_i-1} r_t$.

2.1.1 Markov Decision Processes

In order for a sequential decision making problem to be tractable, there must be some consistent structure to how rewards and observations are produced. The Markov decision process (MDP) is a common model for this. An MDP is defined by a tuple of a state space, action space, initial state distribution, transition function and reward function, denoted by (S, A, $p(s_0)$, $p(s_{t+1}|s_t, a_t)$, $r(s_t, a_t, s_{t+1})$)

The progression of the MDP occurs in timesteps. At each timestep t, the environment has a state s_t. s_t comes from a state-space S, which is a collection of all possible states. In MDPs, the agent observes this state directly. We say that MDPs are 'fully observable' (to the agent).

The agent chooses an action for the timestep a_t, which must come from the action space A. The set of legal actions may depend on the current state, particularly in board games (for example, in Hex it is illegal to place a stone in an occupied cell).

The next state for the MDP is then determined by the transition function, which is a probability distribution over possible next states s_{t+1}. It depends only on the current state and current action, so is denoted $p(s_{t+1}|s_t, a_t)$. A reward for the agent resulting from this transition is then produced by the MDP, determined by the reward function $r(s_t, a_t, s_{t+1})$. r may also be a stochastic.

When both the transition function and reward function are deterministic, we say that the MDP is deterministic.

The initial state distribution $p(s_0)$ specifies the probability distribution of

[1]Such a discount factor can be introduced in the episodic case also, this can model problems where it is desired to achieve some goal in as few steps as possible.

the first state s_0 of the MDP in every episode.

We will use $\tau = (s_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, ..., s_T, r_T)$ to denote an entire episode of an MDP.

2.1.2 Policies

The task of a reinforcement learning agent is to take all experience the agent has gathered from interaction with an environment, and use it to select actions. Any experience from states that share similarities with the current state may provide evidence about which action is best to choose. When we design a reinforcement learning algorithm, we are deciding how these experiences should impact the actions the agent takes.

The action a_t chosen by an agent should depend on all experience insofar as the experience can provide knowledge about the problem. However, the Markov property of MDPs means that when the problem has been solved, the actions selected may depend on only the current state. Therefore the dependence of the policy on the current state is particularly significant: action a_t depends on s_t directly because the agent must choose an action that is effective in s_t.

It is useful to express this dependence explicitly by selecting actions from a parametric policy distribution $\pi_\theta(a_t|s_t)$. θ are trainable parameters of this distribution that can depend on all of the experience; they store what the agent has learned about behaving effectively in this MDP. The conditioning of the action on the current state is made explicit in the notation.

One example of such a policy function π_θ is a lookup table of probabilities of each action given the state. This will be an $|S| \times |A|$ size table, each row representing a distribution over the possible actions. Entries in the table are the parameters θ, while the conditioning on the state selects which row of the table to look at. This encodes the Markov structure of the MDP problem in a useful way for our agent.

A policy $\pi(a|s)$ induces a distribution over states, which we denote $\mu(s|\pi)$

2.1.3 Value Functions

Given some policy π for selecting actions, the expected return is:

$$J(\pi) = \mathbb{E}_\pi[R] = \mathbb{E}_\pi\left[\sum_{t=0}^{T} \gamma^t r_t\right]$$

The notation $\mathbb{E}_\pi[f(\tau)]$ means an expectation of a function of an episode of the MDP, $f(\tau)$ (in this case the return), where τ is an episode sampled using the policy π.

The expected return is the average amount of reward we can expect to receive in an episode if we follow the policy π (in a particular MDP). The goal of the agent is to find a policy that maximises this. We denote an optimal policy that maximises J as π^*.[2]

The state value function is defined as:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t_0}|s_{t_0} = s] = \mathbb{E}_\pi\left[\sum_{t=t_0+1}^{T} \gamma^{t-t_0} r_t | s_{t_0} = s\right]$$

The state value function answers the question 'How good is it to be in state s, when using policy π?' If we are able to understand this quantity it allows us to reason about how effective a sequence of actions has been before we reach the end of an episode: instead of looking at returns directly, we can look at an estimate of the value of a state we transition to.

The state-action value function, known as the 'Q-function' is:

$$Q^\pi(s,a) = \mathbb{E}_\pi[R_{t_0}|s_{t_0} = s, a_{t_0} = a] = \mathbb{E}_\pi\left[\sum_{t=t_0+1}^{T} \gamma^{t-t_0} r_t | s_{t_0} = s, a_{t_0} = a\right]$$

The Q-function answers the question 'How good is action a from state s, if we follow the policy π thereafter?' Understanding a Q-function allows us to compare the values of different actions, so we can choose the action that results in the highest average return.

[2] π^* is not in general unique, in some states there might be multiple actions that are equally strong.

2.2 Value Prediction

Many reinforcement learning algorithms predict either a state value function or a Q-function for the MDP, as introduced in section 2.1.3. As we shall see in following sections, a good estimate of the value of a policy is often an important first step to understanding an MDP, and hence learning to act effectively.

2.2.1 Monte Carlo Value Estimation

A value function at a given state s is the expected value of a conditional random variable, i.e. $V^{\pi}(s) = \mathbb{E}_{\pi}[R_t|s_t = s]$. Every episode in our experience where we visit the state s provides us with a sample from this random variable. We can use the sample average as an unbiased estimate of the expectation of our random variable. Then the estimate after K steps (possibly spanning multiple episodes) is:

$$\hat{V}(s) = \frac{\sum_{t=0}^{K} R_t \mathbb{1}_{s_t=s}}{\sum_{t=0}^{K} \mathbb{1}_{s_t=s}}$$

The numerator is the sum of all the returns resulting from visits to the state s, while the denominator counts those visits. Alternatively, the following update rule can be used to keep a running average:

$$\hat{V}(s_t) := \hat{V}(s_t) + \alpha(R_t - \hat{V}(s_t))$$

Q-functions can similarly be estimated by Monte Carlo methods.

The returns of a Monte Carlo estimate depend on the full trajectory following the visit to the state s_t. This potentially consists of many timesteps. At each timestep, an action and transition are sampled, meaning the return commonly has high variance, and many samples are needed for the Monte Carlo estimate to achieve high precision. [3] Additionally, the Monte Carlo estimate is entirely on policy - it estimates the value function of the policy used to generate the data. This makes it difficult to reuse data from an old, weak

[3]For iid data, the standard deviation of errors falls as $\frac{1}{\sqrt{n}}$, where n is the number of visits to the state. If the same state is visited multiple times in an episode, only considering the first visit makes the data iid. Considering all visits is also convergent, but the analysis of convergence rates is less simple.

policy to aid estimating the value of a newer, stronger policy, further hurting the efficiency of algorithms based on Monte Carlo estimation.[4]

2.2.2 Temporal Difference Value Estimation

Temporal difference (TD) learning [Samuel, 1959, Sutton, 1988] is a method that can be used to overcome some of the problems of Monte Carlo value learning.

The Bellman equation [Bellman, 1957b] states that:

$$V(s_t) = \mathbb{E}_{s_{t+1}|s_t,\pi}[r_{t+1} + \gamma V(s_{t+1})]$$

There is an equivalent rule for the Q-function, which is:

$$Q(s_t, a) = \mathbb{E}_{s_{t+1}|s_t,a}[r_{t+1} + \gamma \sum_{a'} \pi(a'|s_{t+1}) Q(s_{t+1}, a')]$$

The Bellman equations are a temporal relationship that can be viewed as one-step rules for estimating value functions. We can replace the return from the Monte Carlo update with the target suggested by the Bellman equation, giving the update rule:

$$\hat{V}(s) := \hat{V}(s) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - \hat{V}(s))$$

Unfortunately, this update relies on evaluating the state value function V at s_{t+1}, but we were trying to learn V in the first place! We can overcome this impasse by using the current estimate of $V(s_{t+1})$, $\hat{V}(s_{t+1})$. Using the current estimate of the value function in place of the true value is known as bootstrapping [Sutton, 1988], and allows us to avoid training value estimates based on returns from long trajectories.

Bootstrapping means that the random variable we use to estimate values will no longer have the same expected value as the returns, because our estima-

[4]Techniques to use such data are an active field of research, for example Precup [2000], Espeholt et al. [2018]

tor is inaccurate. In other words, it is biased. So it is not obvious that training via bootstrapping will converge to correct value estimates, or converge at all. This can be guaranteed when using a tabular function approximator for the value function, and either using a discount $\gamma < 1$ or solving bounded-length episodic tasks [Sutton, 1988, Dayan, 1992].

Both the running-average formulation of the Monte Carlo estimate and the temporal difference estimate take the form:

$$\hat{V}(s) := \hat{V}(s) + \alpha(\text{target} - \hat{V}(s)) \tag{2.1}$$

for some random variable 'target' with the same expectation as the return (notwithstanding any bias caused by the application of bootstrapping). This gives us a general recipe for value estimate update rules. For example, we can use n-step temporal difference estimates, which bootstrap after an n-step simulation:

$$V(s) = \mathbb{E}\left[\sum_{t=t_0+1}^{t_0+n} \gamma^{t-t_0-1} r_t + \gamma^n V(s_{t_0+n}) | s_{t_0} = s\right]$$

By the linearity of expectations, any convex combination of targets is also a valid target. This is used by the TD(λ) algorithm [Sutton, 1988], which uses an exponentially weighted average of n-step estimates for $n \geq 1$.

2.2.3 Value estimation with Function Approximation

So far, we have discussed how Monte Carlo and Temporal Difference methods allow us to make estimates for $V(s)$ and $Q(s, a)$. The update rules were all represented as assignments. In other words, we recall a value estimate $\hat{V}(s)$ by looking up the last value we assigned for that state. Therefore, as presented, our algorithms are estimating the value functions with look-up tables.

Tabular function approximation is fine when the number of states is small, but as the number of states grows, look-up tables become larger too, and will not fit into memory for any computer. Moreover, for values in the look-up table to become accurate, each value must be updated multiple times. With updates only occurring when states are visited, this means the amount of experience needed for the look-up table to become accurate must scale at least linearly with the number of states in the look-up table.

Function approximation can help tackle these issues. We estimate $V(s) \approx \hat{V}_\theta(s)$, where $\hat{V}_\theta$ is a parametric function with parameters θ. The task of value prediction is now to find parameters θ for which $\hat{V}_\theta(s)$ is close to the true value function $V(s)$. For well chosen parametric function classes, values of θ that achieve good accuracy on some small set of states $\tilde{S} \subset S$, $|\tilde{S}| \ll |S|$ can *generalise* to perform well on much larger subsets of the state space. When $\tilde{S}$ is representative of the states usually seen when interacting with the MDP, θ can provide a more compact representation of the value function, and allows us to achieve good value prediction with far less experience.

One class of parametric function that has proven successful in problems requiring generalisation is deep neural networks (NNs) [LeCun et al., 2015], and reinforcement learning with NNs (i.e. deep reinforcement learning) has achieved success on domains with state-spaces too large for other methods [Mnih et al., 2015].

In order to train such parametric value estimators, we must adapt update rules of the form of rule (2.1) to apply to the parameters of the function, rather than directly to its output. Observe that rule (2.1) says 'move the value estimate part way towards the target value'. When using a function approximator, we do something similar: 'move θ, so that the value estimate moves part way towards the target value'. The gradient $\nabla_\theta \hat{V}_\theta(s)$ tells us how to do this:

$$\theta := \theta + \alpha \nabla_\theta \hat{V}_\theta(s)(\text{target} - \hat{V}_\theta(s)) \tag{2.2}$$

This is the gradient of the $\mathcal{L}^2$ loss (i.e. squared error) for predicting the target value: [5]

[5]But, crucially, we do *not* differentiate through the target, even when it depends on parameters θ

$$
\begin{aligned}
\mathcal{L}^2(\theta) &= (\text{target} - \hat{V}_\theta(s))^2 \\
\theta - \frac{\alpha}{2}\nabla_\theta \mathcal{L}^2(\theta) &= \theta - \frac{\alpha}{2}\nabla_\theta \hat{V}_\theta(s) \times -2(\text{target} - \hat{V}_\theta(s)) \\
&= \theta + \alpha \nabla_\theta \hat{V}_\theta(s)(\text{target} - \hat{V}_\theta(s))
\end{aligned}
$$

Note that the theory which says that bootstrapping approaches, such as TD, converge with tabular approximators does not usually hold for the parametric function approximation case [Tsitsiklis and Van Roy, 1997]. This is because updating θ affects the estimates at all states, possibly incorrectly and by large amounts. If the errors introduced are larger than the improvements to the estimate, this can cause TD learning to diverge.

Nonetheless, with careful engineering TD learning can be applied with function approximators successfully. For example, when using n-step TD learning with larger values of n, the γ^n multiplier of the bootstrapped estimate acts to dampen unstable behaviour, making it more likely that the algorithm as a whole is stable. For a similar reason, smaller discount factors can also make using TD learning with function approximators easier [Amit et al., 2020]. Because discount factors have this stabilising effect, they are often used even in settings where evaluation is based on undiscounted returns (for example in Mnih et al. [2015]).

2.3 Control

In the previous section, we discussed techniques for learning value functions and Q-functions given an acting policy. We now discuss methods learning effective policies, which will solve the original reinforcement learning problem. Often, these will make use of learnt value functions, to detect when our actions have led to better or worse states; or Q-functions, to compare actions directly.

2.3.1 Value Based Learning

The value function and Q-function of the optimal policy π^*, V^{π^*} and Q^{π^*}, are commonly referred to by the shorthand V^* and Q^*.

A sufficient condition for a policy π to be optimal is that it always selects an action that maximises the Q-function of optimal play Q^*[6], i.e.

$$\pi(a \in M(s)|s) = 1,\ M(s) = \{a \in A \text{ s.t. } Q^*(s,a) = \max_a Q^*(s,a)\},\ \forall s \in S \tag{2.3}$$

This is not a necessary condition since there could be states which are never visited under optimal play. It is necessary and sufficient to meet this condition in all states that are visited by π with non-zero probability.

A theorem of Bellman [1957b] states that the following condition also guarantees optimality of a policy π, without reference to Q^*:

$$\pi(a \in M(s)|s) = 1,\ M(s) = \{a \in A \text{ s.t. } Q^\pi(s,a) = \max_a Q^\pi(s,a)\},\ \forall s \in S \tag{2.4}$$

This condition says that a policy is optimal if there is no state from which performance can be improved by changing the policy *in that state only.* Note, however, that it may be that a policy π is suboptimal, but can only be improved in states which are never actually reached by π. This occurs because $Q^\pi(s,a)$ can be low for the optimal action in s if π makes mistakes in the subsequent state.

2.3.2 Policy Improvement Theorem

The policy improvement theorem tells us how, if the Q-values of a policy are known, we can use them to improve on that policy.

Policy Improvement Theorem: [Bellman, 1957b] Suppose π and π' are policies. If:

$$\mathbb{E}_{a\sim\pi'}\left[Q^\pi(s,a)\right] \geq V^\pi(s)\ \forall s \in S$$

[6]See Puterman [1990] for a formal treatment

then:

$$V^{\pi'}(s) \geq V^{\pi}(s) \; \forall s \in S$$

Moreover, if the first inequality is strict in some state, then in at least one state, the second inequality is also strict. The importance of this theorem is that it means that if we estimate the Q-values of a policy using the methods from section 2.2, we can define a new, better policy. This process can then be repeated, estimating the Q-values of the new policy.

If the inequality is never strict, then it follows from equation (2.4), our policy must be optimal.

The SARSA algorithm [Sutton, 1996] directly uses the policy improvement theorem. It estimates a Q-function via TD learning. Experience is sampled by interacting with the environment with a policy defined directly from the Q-function, for example an ϵ-greedy policy.

$$\hat{Q}(s_t, a_t) := (1 - \alpha)\hat{Q}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \hat{Q}(s_{t+1}, a_{t+1}) \right)$$

This is an on-policy algorithm, as it maintains an estimate $\hat{Q}$ of the Q-function of the acting policy. This acting policy improves as the estimate improves.

2.3.3 Q-learning

SARSA estimates the value of the current policy, which is defined in terms of the Q-function $\pi(s) = f(Q(s))$, and then improves the policy with that estimate. Q-learning [Watkins and Dayan, 1992] is subtly different, in that it attempts to estimate the value of the optimal policy more directly. This holds the promise of faster convergence. A Q-function is learnt, updated by following the rule:

$$\hat{Q}(s_t, a_t) := (1 - \alpha)\hat{Q}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_a \hat{Q}(s_{t+1}, a) \right)$$

The maximisation over actions means we are continually estimating the value of the greedy policy with respect to the Q-function,[7] even if the data is generated from a different policy. In other words, Q-learning is an off-policy algorithm. This means we can select any policy we like to explore in the MDP, and provided every action is taken infinitely often in the limit, Q-learning will converge [Watkins and Dayan, 1992]. For example, an ϵ-greedy policy, which follows a uniform policy $\epsilon > 0$ of the time, and otherwise follows the greedy policy, is commonly used.

However, the usual challenges of combining bootstrapping and function approximators - bias and instability - remain. Deep Q-learning (DQN) [Mnih et al., 2015] uses two techniques to improve the stability of applying Q-learning with deep neural networks, *experience replay* and *fixed target networks.*

In experience replay, rather than always training on the most recent datapoint, data is added to a replay buffer, which contains the N most recent steps of experience. Data is then sampled from the replay buffer randomly to train the Q function. This decorrelates data that otherwise comes from a highly correlated stream of states from a single episode of experience.

This is useful, as the learning updates resulting from a mini-batch of n correlated samples of data will have higher variance than a similarly-sized mini-batch of uncorrelated samples. Furthermore, because the new parameters are used to generate the next experience, there is feedback between recent updates and the next experience generated. Without experience replay, this can lead to sudden and dramatic shifts in the training distribution. With experience replay, this feedback loop is smoothed out because the training distribution is effectively a running average of previous policies. Crucially, because Q-learning is an off-policy algorithm, it can cope with the outdated data taken from a long replay buffer, which was generated by a previous policy.

As in all temporal difference methods, the neural network is used to define its own target (i.e. bootstrapping). This results in a shifting target as the value network is updated, and is the root cause of instability. In DQN, the target

[7] i.e. a policy $\pi(a|s)$ which chooses action $\arg\max_a Q(s,a)$ with probability 1

network is fixed for 1000s of network updates, and only updated to the current network periodically.

Suppose the first Q-function estimated a value of 0 for all state action pairs. When this is used as a target network, the term $r_{t+1} + \gamma\hat{Q}(s_{t+1}, a_{t+1}) = r_{t+1}$, the immediate reward. This means that initially our Q-function would learn to predict 1-step rewards. When we update the target network, the term $r_{t+1} + \gamma\hat{Q}(s_{t+1}, a_{t+1})$ gives the immediate return plus an estimate of the 1-step greedy return from the next state, i.e. the value for a 2-step horizon. In other words, the use of fixed target networks essentially breaks the learning up into optimising for 1-step, then 2-step, then 3-step rewards, and so on, extending the reward horizon by one each time the target network is updated.

Both these improvements can be seen as an attempt to make the neural network training in Q-learning more similar to supervised learning, reducing the learning problem to a series of least-squares regressions. Deep learning in a supervised setting does not have problems with instability like those seen in model-free RL, making such reductions a promising approach to applying deep learning to RL problems.

2.3.4 Policy Gradient Algorithms

So far, all the RL algorithms we have reviewed estimate a Q-function, and then use it to define a policy, as motivated by the policy improvement theorem. An alternative approach, variants of which have proven increasingly popular in deep reinforcement learning, is that of policy gradients [Williams, 1992].

Policy gradients can be seen as 'trial and error' reinforcement learning. It follows the principle of 'if I receive a high return in an episode, change my policy so that the actions I took are more likely, if I receive a low return, make them less likely'. Formally, we estimate the gradient of the expected return with respect to the parameters of our policy. This is done using the score function gradient estimator, a black-box gradient estimation tool.

We derive this gradient estimate here. Denote an episode of experience by τ, the joint probability of all the environment transitions in τ by $p(\tau)$, and the joint probability over all action selections by $\pi(\tau)$. The return from the episode is $R(\tau)$. Then:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\pi_\theta}\left[R(\tau)\right] &= \nabla_\theta \int R(\tau)\pi_\theta(\tau)p(\tau)d\tau \\
&= \int \nabla_\theta(R(\tau)\pi_\theta(\tau)p(\tau))d\tau \\
&= \int R(\tau)p(\tau)\nabla_\theta(\pi_\theta(\tau))d\tau \\
&= \int R(\tau)p(\tau)\nabla_\theta(\pi_\theta(\tau))\frac{\pi_\theta(\tau)}{\pi_\theta(\tau)}d\tau \\
&= \int R(\tau)p(\tau)\pi_\theta(\tau)\frac{\nabla_\theta(\pi_\theta(\tau))}{\pi_\theta(\tau)}d\tau \\
&= \int R(\tau)p(\tau)\pi_\theta(\tau)\nabla_\theta \log(\pi_\theta(\tau))d\tau \\
&= \mathbb{E}_{\pi_\theta}\left[R(\tau)\nabla_\theta \log(\pi_\theta(\tau))\right] \\
&= \mathbb{E}_{\pi_\theta}\left[R(\tau)\nabla_\theta \log\left(\prod_{a,s\in\tau}\pi_\theta(a|s)\right)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\sum_{a,s\in\tau} R(\tau)\nabla_\theta \log(\pi_\theta(a|s))\right]
\end{aligned}$$

This estimator is generally high-variance, meaning policy gradient methods can be slow to converge [Marbach and Tsitsiklis, 2003, Peters and Schaal, 2006]. There are many important techniques that can be used to improve the performance of policy gradient methods. We discuss a few here.

Consider an MDP with strictly positive rewards. In this case, the vanilla Reinforce gradient derived above would increase the likelihood of whichever action was chosen, even if they were the worst possible actions. In expectation it takes the correct gradient, but only because better actions have their likelihood increased by more than worse actions do. We can correct this by subtracting a baseline b from the return, giving us an update (for a single state-action pair s_t, a_t) of the form:

$$(R - b)\nabla_\theta \log(\pi_\theta(a_t|s_t))$$

Mathematically, the purpose of the baseline is to reduce the variance of the gradient estimate. The baseline can depend on the current state s_t; so long as it is conditionally independent from $a_t|s_t$, it does not change the expectation of the gradient estimator [Williams, 1992].

Setting $b = \sum_{t'=0}^{t} r_{t'}$ means that $R - b = R_t$. This choice of baseline can be interpreted as encoding that an action cannot affect any rewards that have already been received before the action was taken. This is almost always an improvement on the vanilla policy gradient.

When a state value estimate is available, we can use it as a baseline. $R_t - V^{\pi_\theta}(s)$ measures 'how much better than expected was the return' (known as the advantage), and means we follow the intuitive rule of increasing the probability of an action only when it results in better-than-average outcomes.

So far, the policy gradient has been based on the episode's return. As was the case with Monte Carlo value prediction, this return depends on many actions, and therefore can require many samples to achieve precise estimates - in this case to determine which actions actually lead to high or low return.[8] As before, if a value function is known, we can replace the Monte Carlo sample with a TD-style target, using the value function to truncate the simulation after a finite number of steps.[9]

Policy gradient methods can use on-policy samples, or they can importance weight off-policy data to remove bias in the updates, at the cost of increased variance. This means that, in practice, they exhibit worse sample efficiency than off-policy value-based learning methods, such as Q-learning [Gu et al., 2017, van Hasselt et al., 2019], which do not require importance sampling to re-use old data. They may also struggle more with local optima, as the local nature of the gradient based rule cannot offer any global optimality guarantee.

However, because policy gradient methods estimate the gradient of the

[8]Known as the credit assignment problem.

[9]This is known as an Actor-Critic algorithm, since the value function serves as a 'critic' for the policy (or 'actor') [Konda and Tsitsiklis, 2000].

expected return directly, provided small enough learning rates and enough samples are used, they can have better stability than value-based methods, particularly when using neural network function approximators.

More recent advances, such as the TRPO and PPO policy gradient methods [Schulman et al., 2015a, 2017], further improve the stability of policy gradients by restricting them from making very large changes to the policy in a single step. Instability is a major problem for model-free RL, and these algorithms perform comparatively well in this regard.

2.4 Reinforcement Learning using an Environment Dynamics Model

The RL algorithms discussed so far are all model-free: they interact with the MDP in a sequential manner, and use the experience from those interactions to update policy or value functions directly. In particular, model-free algorithms do not try to model or explicitly reason about the dynamics of the MDP they are attempting to solve.

This means that there are many things that a strictly model-free agent does not know and cannot do: It cannot reset the environment to some state it has seen before. It does not know the dynamics of the environment. It never knows what the outcome of an action is until it tries it, or what would have happened had it tried a different action.

However, the restrictions of the model-free setting mean such algorithms have two important advantages: they tend to be simple and they can often (in principle) be applied to a very general class of problems.

In practice, model-free deep RL algorithms typically use millions or billions of interactions with an environment, far more than could ever be made without the use of simulation. Therefore, standard practice in deep reinforcement learning is to apply a model-free algorithm to experience generated by interacting with a simulator. So a natural question is, what alternative algorithms might be possible if we allowed less restricted use of the environment simulator? Can we use the ability to reset to arbitrary states — and simulate alternative actions in those states — to improve the effectiveness of our RL algorithms? In this section we review some algorithms that do make greater use of simulators.

Model-based algorithms can make different assumptions about where the environment simulator comes from. In this thesis, we assume an exact simulator has been provided: the dynamics of the environment are known precisely, and the only task is to learn policies for that environment. This is a natural assumption in a board game with explicit rules.

In other cases a simulator may be provided, but may not be perfectly exact. For example, when learning to control a physical robot, physics models can be used, but will fail to capture every detail of the real world exactly. This adds an additional challenge of ensuring that policies learnt in the simulator can transfer to the true target domain. Some methods assume that no simulator is available at all, so a model of the environment is learnt, and a model-based RL algorithm then applied using this model.

2.4.1 Dynamic Programming

Value Iteration

In value iteration (VI) [Bellman, 1957a], a value estimate is iteratively updated in all states s according to the following rule:

$$V_{k+1}(s) = \max_a \mathbb{E}\left[r + \gamma V_k(s') \mid (r, s') \sim p(r, s'|s, a)\right]$$

The subscripts refer to the iteration of the algorithm. Each iteration updates value estimates backwards in time from possible next states s' resulting from the state s.

The expectations can be evaluated by evaluating all possible r and s'. This can be calculated since we assume access to an exact model of the MDP. Given a value function V_k, we can calculate a policy in a similar way, replacing the maximisation with an $\arg\max$, this will be a 1-step greedy policy.

If we set $V_0(s) = 0$ as an initial condition, then at iteration k, value iteration will have calculated the value of the policy which is optimal for a limited horizon k. In finite episodic tasks, once k has extended beyond the maximum episode length, the value function found with value iteration will be V^*, and the resulting 1-step greedy policy will also be optimal. In an infinite-horizon task with discounted returns, the error of V can be bounded in terms of the discount factor γ and the maximum possible reward magnitude M. This bound also implies a similar bound on the sub-optimality of 1-step greedy-policy of V_k [Puterman, 1990].

$$|V^*(s) - V_k(s)| \leq \sum_{t \geq k} \gamma^t M = \frac{\gamma^k}{1-\gamma} M$$

Policy Iteration

In policy iteration (PI) [Howard, 1960, Bertsekas, 2011], we maintain a policy which is iteratively updated at all states according to the rule:

$$\pi_{k+1}(s) = \arg\max_a \mathbb{E}\left[\sum_{t>0} \gamma^t r_t | s_0 = s, a_0 = a, a_i = \pi_k(s_i) \forall i > 0\right]$$

By the policy improvement theorem, π_k is a monotonically improving sequence of policies. Indeed, by construction:

$$V^{\pi_{k+1}}(s) \geq \max_a \mathbb{E}\left[r + \gamma V^{\pi_k}(s') \mid (r, s') \sim p(r, s'|s, a)\right]$$

Hence policy iteration has at least as strong a convergence rate as value iteration, in terms of number of iterations needed [Puterman, 1990]. In practice, policy iteration typically converges in significantly fewer iterations than are needed for value iteration. However, because the horizon of the expectation is the full episode, calculating an exact step of policy iteration is usually more expensive than a single step of value iteration.

Modified Policy Iteration

Modified Policy Iteration (MPI) [Puterman and Shin, 1978, Scherrer et al., 2015] generalises VI and PI into a single algorithm, controlled by a simulation length parameter whose extremal values correspond to VI and PI themselves. Scherrer et al. [2015] found that intermediate values of this parameter yielded strongest performance in practice.

Given a value function V, we define the greedy operator G to return a policy that optimises the expected 1-step values according to V:

$$G(V)(s) = \arg\max_a \mathbb{E}\left[r + \gamma V(s') \mid (r, s') \sim p(r, s'|s, a)\right]$$

Given a policy π we define the Bellman operator T_π to return an updated value function based on a 1-step update when following the policy π:

$$T_\pi(V)(s) = \mathbb{E}\left[r + \gamma V(s') \mid (r, a') \sim p(r, s'|s, a),\, a \sim \pi(s)\right]$$

We can combine these to arrive at the modified policy iteration algorithm. Each iteration of MPI consists of two steps:

$$\pi_{k+1} = G(V_k) \qquad \text{(Greedy Step)}$$
$$V_{k+1} = (T_{\pi_{k+1}})^m V_k \qquad \text{(Evaluation Step)}$$

m is a parameter similar to the parameter n in n-step temporal difference learning methods. For $m = 1$ we recover value iteration, where in value iteration the policies π_k were implicit in the max operation. For $m = \infty$, we recover policy iteration, in policy iteration as presented above the values V_k are calculated on-the-fly in the update rule's expectation.

Finite values of $m > 1$ interpolate between the VI and PI algorithms. They enjoy faster convergence than VI in terms of number of iterations, similarly to PI, while requiring less computation per iteration than PI algorithms.

2.4.2 Approximate Dynamic Programming

Exact application of dynamic programming algorithms requires that the greedy step and evaluation step be calculated exactly for every state at each iteration. Just as in tabular model-free learning, this can be problematic:

1. Exact calculation of the greedy or evaluation steps can of themselves be prohibitively expensive.

2. Usually the state space is too large for calculating the greedy and evaluation steps in every state to be feasible.

In order to overcome the first of these limitations, we need to approximate our calculation of the greedy and evaluation steps. Rather than exactly calculating expectations by enumerating all possibilities, we can form Monte Carlo estimates for those expectations through sampling.

In order to overcome the second limitation, we use function approximation. For instance, neural networks can be used to approximate either or both of the policies π_k and value functions V_k.

To approximate a policy, a subset of states $\tilde{S} \subset S$ is chosen. For each state $s \in \tilde{S}$, the greedy step is calculated. The pairs of $(s, G(V)(s))$ then form a dataset. A policy can be trained — in a supervised manner — to approximate the mapping $s \to (G(V))(s)$ on this dataset, giving an approximate (fitted) greedy policy. Some loss must be defined to measure how strong each approximation is. In Scherrer et al. [2015], a cost-sensitive loss is suggested:

$$\mathcal{L}_{\pi_{k+1}} = \sum_{s \in \tilde{S}} \Big(\max_a Q_k(s, a) - Q_k(s, \pi_{k+1}(s)) \Big)$$

The loss is called cost-sensitive, because when the policy π_{k+1} fails to predict the optimal action according to Q_k, it is penalised by how much lower the Q-value of the action chosen is compared to the optimal. This means that predicting a slightly sub-optimal action is penalised less severely than predicting a very bad action.

Classification-based policy iteration [Lagoudakis and Parr, 2003] treats the problem of fitting a policy to the dataset as a classification problem, using a support vector machine. This disregards the costs of misclassifications, which means the learnt policy may be too willing to 'take risks' by playing actions that may be correct, but if they are not correct are likely to result in a poor outcome. On the other hand, as argued by Lagoudakis and Parr [2003], classification problems are better understood, so in practice a more effective system may be achieved through reducing the RL problem to supervised learning.

To approximate the evaluation step, a dataset consisting of pairs $(s, (T_{\pi_{k+1}})^m V_k(s))$ can be formed, and a value function fitted, usually minimising a square-loss:

$$\mathcal{L}_{V_{k+1}} = \sum_{s \in \tilde{S}} \left((T_{\pi_{k+1}})^m V_k(s) - V_{k+1}(s) \right)^2$$

These dynamic programming algorithms often make use of simulator resetting to calculate the greedy policy update or 1-step value update. It is possible to forgo this with some losses, for example the cost-sensitive loss in Scherrer et al. [2015] can be approximated with a policy gradient method.

2.5 Alternating two-player Markov Games

MDPs are single-agent problems, where one agent interacts with an environment with consistent behaviour. The alternating two-player Markov game (AMG) [Littman, 1996] extends MDPs to model contests between two agents. Agent 1 chooses actions on even timesteps $(a_0, a_2, a_4...)$, while agent 2 chooses the actions on odd timesteps $(a_1, a_3, a_5...)$. In other words, the two agents alternate moves. The state remains Markov and fully observable.

Each agent is attempting to optimise their own return $R^{(i)}$ for player i. In the zero-sum case treated in this thesis, $R^{(1)} = -R^{(2)}$, so to maximise their reward, player 2 attempts to minimise player 1's reward. Many classical board games, such as tic-tac-toe, Chess, and Hex, the test domain for this thesis, are zero-sum alternating two-player Markov games.

Optimal play in an AMG is more subtle than in an MDP. In an MDP, the outcome (or distribution over outcomes) of actions is determined entirely by the environment, so for any strategy that an agent chooses, there is an expected return that the strategy receives. A strategy is better than another strategy if the expected return is higher. In contrast, in an AMG, the outcome of a certain strategy depends on the strategy that the opponent chooses to follow, so the expected return can only be measured against a specific opponent. Nonetheless, as we will see in the next section, algorithms for AMGs apply to MDPs, and, for the most part, algorithms for MDPs can be used to solve AMGs.

2.5.1 Optimality in Alternating Markov Games

The concepts of value functions, expected return and optimality are more complex in the AMG case than in the single-agent MDP case. The expected return, value of a state or value of a state-action pair under a given policy is no longer determined only by the transition dynamics, but also by the policy chosen by an opponent, whose strategy we do not necessarily know. This is also a problem during training, when we have to decide which opponent strategy to train against.

We refer to rewards and returns with reference to a particular player's perspective, for example the reward for player i at time t is denoted $r_t^{(i)}$ and the return for player i in an episode is $R^{(i)}$.

Value functions are also defined with respect to a perspective, and also depend on policies π_1 and π_2 for player 1 and player 2 respectively. So a state value function is written as:

$$V^{(i),\pi_1,\pi_2}(s) = \mathbb{E}_{\pi_1,\pi_2}[R_{t_0}^{(i)}|s_{t_0} = s] = \mathbb{E}_{\pi}\left[\sum_{t=t_0+1}^{T} r_t^{(i)}|s_{t_0} = s\right]$$

Notation is similarly extended to define $Q^{(i),\pi_1,\pi_2}$ and $J^{(i)}(\pi_1, \pi_2)$.

In the zero-sum setting of AMGs, every reward, return and value for player 2 is the negative of that of player 1. This allows us to simplify our notation to be closer to the MDP case:

$$r_t^{\pi_1,\pi_2} = r_t^{(1),\pi_1,\pi_2} = -r_t^{(2),\pi_1,\pi_2}$$
$$V^{\pi_1,\pi_2}(s) = V^{(1),\pi_1,\pi_2}(s) = -V^{(2),\pi_1,\pi_2}(s)$$
$$\text{And so on for } R,\ Q \text{ and } J$$

This return is not independent of opponent strategy, and so to maximise/minimise return is not a well defined objective unless the opponent strategy is also specified. We instead adopt the goal of finding *minimax* policies π_1, π_2. We say that π_1 and π_2 are minimax if they satisfy the following generalisation of condition 2.4 to AMGs:

$$\begin{aligned}&\pi_1(a \in M(s)|s) = 1,\ M(s) = \{a \in A \text{ s.t. } Q^{\pi_1,\pi_2}(s,a) = \max_a Q^{\pi_1,\pi_2}(s,a)\}\\&\text{and}\\&\pi_2(a \in M(s)|s) = 1,\ M(s) = \{a \in A \text{ s.t. } Q^{\pi_1,\pi_2}(s,a) = \min_a Q^{\pi_1,\pi_2}(s,a)\}\end{aligned} \tag{2.5}$$

If π_1, π_2 are minimax policies in an AMG, and player 1 follows π_1, for any opponent strategy π_2' and state s, $V^{\pi_1,\pi_2'}(s) \geq V^{\pi_1,\pi_2}(s)$. That is to say, player 1 will receive an average return from any state at least as high as the average return from the minimax strategies, regardless of the opponent's policy.

Furthermore, if some policy π_1' is not minimax, then there exists some policy π_2' and state s such that $V^{\pi_1',\pi_2'}(s) < V^{\pi_1,\pi_2}(s)$. This means that a minimax strategy provides the best possible guarantee on the average return, that is independent of the opponent strategy and start state. In other words, minimax policies match the colloquial term of 'perfect play' used in classical games such as chess.[10]

Suppose we want to train an agent to play well as player 1. We would like to reduce the AMG to an MDP, and then apply an RL algorithm to this

[10]These results are presented here from the perspective of player 1, but trivially also apply to player 2

problem. We could do this by choosing some opponent, and then treating the opponent's actions as part of the environment. However, even with an RL algorithm that converges to an optimal policy in the MDP, this approach will not find minimax strategies if our opponent is not minimax. Instead we find the optimal strategy for countering the specific opponent against whom we are playing [Littman, 1996].

To allow us to use RL algorithms to train our policies, we train both the policies π_1 and π_2 with RL. For the training of π_1, we treat the actions of player 2 as part of the environment's transition function, and vice-versa for training π_2. This means that the MDPs being solved by the RL algorithms change their dynamics during training. Nonetheless, Littman [1996] showed that this approach will converge to minimax solutions if the RL algorithms simultaneously converge to optimal policies for the MDPs. This means (subject to some caveats, see Littman [1996] for details) we can solve an AMG with algorithms for MDPs.

It is possible to train a single policy to play as both player 1 and player 2 using the same approach. A single RL agent sees each game from the perspectives of both player 1 and player 2, and updates its policy on both states with player 1 to play and with player 2 to play. This method, known as *self-play* [Littman, 1996] also converges to minimax if the policy converges to an optimal policy for the MDP. For nearly symmetrical games such as Hex, there is common knowledge between playing as player 1 and playing as player 2. By sharing learning between the agents, self-play can learn to play faster than independently learning policies for the two players [Littman, 1996].

This means that, in RL, agents are usually trained to play AMGs by using a single agent RL algorithm with self-play.

There is a trivial construction to convert any MDP into a degenerate AMG by adding a 'phantom' player 2. Every time it is player 2's turn to move, they have a single action available, 'pass', which has no effect on the state (except to return control back to player 1). There is only one policy for player 2, and the minimax policy for player 1 is the optimal strategy for the original MDP. So any algorithm for solving an AMG is also an algorithm for solving an MDP.

This correspondence between optimality of policies in MDPs and AMGs does admit one subtlety: in an MDP, if a policy is poor in some states, it can often be approximately (or exactly) optimal in terms of the expected return because the states it is poor in occur only rarely (or never). This is quite a common occurrence, as such states tend to appear less often during training, so learning is likely to be less advanced at any given stage in those states rarely visited

In an AMG, the policy in some rare states may be poor for a similar reason — the state is rarely visited during self-play — however, against a different, adversarial opponent strategy, the state can become much more common. In other words, to be approximately optimal in an AMG, a policy needs to perform well against all possible opponents, meaning diversity of experience is even more important in this setting than it was in the setting of MDPs. These exploits severely limit the effectiveness and stability of RL in learning to play games.

Because of the similarity in algorithms for AMGs and MDPs, from here on we will abuse terminology slightly by referring to MDPs and AMGs collectively as MDPs, and reserve the term AMG for discussions applying specifically to that case.

2.6 Online Planning

In most traditional board games, players are given a significant amount of time to choose each action they take. This deliberation time can be used to consider possible forward lines of play, and plan actions accordingly. This *online* task differs from the offline reinforcement learning task we have discussed so far in two important ways:

1. The amount of time available is usually significantly shorter than for offline reinforcement learning.

2. We only care about performing well in selecting the right action a_t in the current state s_t. Learning in any other state is interesting in the online problem only insofar as it helps in selecting an action for the current state.

Such online planning can be more effective than offline pre-training in terms of compute efficiency, because it is not required to find good policies across a wide range of states in the environment [Kearns et al., 2000]. Instead, computation can be focused narrowly on the particular states encountered during the game at hand.

Online planning always requires a simulator, separate from the genuine interaction with the environment. This simulator is used to 'try out' different sequences of actions. Similarly to the model based methods of section 2.4, this model could be given or learnt. For board games rules are assumed to be known, and so the model can be given.

We can generally assume that some preparation has been made prior to tackling the online problem. Just as human experts practice their games, an AI player will typically have access to a globally accurate, but still imperfect, value function and policy. Generally these are produced by some combination of handcrafting and/or learning methods. In this section we review three important algorithms for tackling this online RL problem.

2.6.1 Game Trees and Search Trees

An AMG can be described by a game tree. Nodes of the tree are identified with states of the game. The (directed) edge from state s_i to s_{i+1} represents the action a taken in s_i to reach s_{i+1}, and is identified by the pair (s_i, a). If there is no action a leading from s_i to s_{i+1} then there is no such edge. In games with transposition (i.e. when there can be more than one sequence of actions that leads the same state), the game tree will have multiple nodes corresponding to the state, one for each possible route to the state.

So in the game tree, the root node corresponds to the original state s_0, its children correspond to the states resulting from a single move from it, etc.

When considering an online planning problem, we use a search tree whose root node is the current state s, this is the game tree of the subgame starting at s. [11]

2.6.2 Minimax Search

The value of a state s if both players play minimax strategies can be found by recursively applying by the rules:

- If the state is terminal, the minimax value is the return for player 1 in this terminal state
- If the state is not terminal, and player 1 is to move, it is the maximum minimax-value of all states reachable from this state in one move by player 1
- If the state is not terminal, and player 2 is to move, it is the minimum minimax-value of all states reachable from this state in one move by player 2
- If the state is a chance node, it is the expectation of the minimax-values of the states possible following the chance action.

Provided that the game has finite length, recursively calculating the values of states according to these rules will terminate, giving the value of the game under perfect play. This is known as minimax search [Borel, 1921, Neumann, 1928]. When complete, we can look at the values of the states that are reachable from the current state to choose the best action in that state.

This algorithm is only feasible in toy problems, such as tic-tac-toe, as it enumerates states. In the problems with large state spaces, we can use a value function to perform a depth-limited minimax search. Now our rules are:

[11]In non-deterministic games, as well as the states where the players make decisions (known as choice nodes / states), we introduce chance nodes / states between the choice nodes, where a chance player 'plays' a fixed stochastic policy corresponding to the distribution over possible chance outcomes in the game.

- If the state is terminal, the value is the return for player 1 in this terminal state
- If the state is d moves after the starting state s, the value is the estimate of the state's value from the value function
- If the state is not terminal, and player 1 is to move, it is the maximum minimax-value of all states reachable from this state in one move by player 1
- If the state is not terminal, and player 2 is to move, it is the minimum minimax-value of all states reachable from this state in one move by player 2
- If the state is a chance node, it is the expectation of the minimax-values of the states possible following the chance action.

This version reduces the number of nodes that must be expanded, and can be viewed as a d-step generalisation of the greedy operator from section 2.4.1, in that it finds an optimal strategy based on the value function to depth d, rather than just to depth 1.

Minimax search algorithms find paths through the game tree of optimal actions, starting from the root state. In deterministic games, we call such a path a *principal variation* (PV), it is unique unless there are two optimal actions in some state. The value of the game is the value if both players follow a PV.

Sometimes, before a node's value has been fully calculated, we can determine bounds on its possible values. These bounds may allow us to realise that a node cannot possibily be on the minimax path. In this case, it is possible to *prune* the node from search, and not calculate its value exactly, saving time. α-β-pruning is an algorithm that keeps track of these bounds in an efficient way, maintaining a bound for player 1 (α) and for player 2 (β) [Knuth and Moore, 1975].

2.6.3 Monte Carlo Search

Monte Carlo search (MCS) was introduced by Tesauro and Galperin [1997] to improve backgammon policies during a game. From the current state of a game s_t, Monte Carlo search simulates many games following a pre-trained policy π. These simulations allow an accurate Monte Carlo estimate of action values in the current state $Q^\pi(s_t, a)$ to be made. Once the simulations are complete, the action $\text{argmax}_a Q^\pi(s_t, a)$ is played. This procedure is justified by the policy improvement theorem, as it implements a single step of policy improvement during the match.

Unlike improvements found via pre-training with offline reinforcement learning, the policy improvements found by MCS do not have to be represented by the parameters of a policy network or generalised across the state-space. As a result, such online improvement can be far more effective than increasing the training time of a policy iteration algorithm. However, it can only perform a single policy improvement, so may not find the optimal policy no matter how many search iterations are used.

2.6.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [Kocsis and Szepesvári, 2006, Coulom, 2006] is an any-time best-first tree search algorithm for online RL that extends MCS. It too uses repeated game simulations to estimate the value of states, but it improves the simulation policy in all states that are visited multiple times, storing improvements in a partial game tree. This means it can find policies that are stronger than can be found with a single policy improvement step.

MCTS is used by the leading algorithms in the AAAI general game-playing competition [Genesereth et al., 2005]. As such, it is the best known algorithm for general game-playing without a long RL training procedure. Unlike MCS, the policy of MCTS can guarantee convergence to the optimal policy with enough search iterations.

Each simulation consists of two parts. First, a *tree phase*, where the tree is traversed by taking actions according to a *tree policy*. Second, a *rollout phase*, where some *default policy* is followed until the simulation reaches a terminal game state. The tree policy is based on statistics stored in the game tree (such as average values of states), and is designed to explore possible actions and gradually improve. The default policy is fixed; it is the policy taken in states outside the partial tree. The result returned by this simulation is then used to update estimates of the value of each node traversed in the tree during the first phase.

At each node of the game tree we store $n(s)$, the number of iterations in which the node has been visited so far. Each edge stores both $n(s, a)$, the number of times it has been traversed, and $R(s, a)$ the sum of the returns obtained in simulations that passed through the edge. The tree policy depends on these statistics. The most commonly used tree policy is to act greedily with respect to the upper confidence bounds for trees formula (UCT) below [Kocsis and Szepesvári, 2006], which is based on the UCB algorithm for bandits [Auer, 2002].

$$\mathrm{UCT}(s, a) = \frac{R(s, a)}{n(s, a)} + c_b\sqrt{\frac{\log n(s)}{n(s, a)}} \tag{2.6}$$

The first term of the UCT formula is the average return achieved so far when taking the action a from state s (i.e. an estimate of the Q value). The second term is an upper confidence bound on that value, meaning that UCT gives an optimistic estimate of the value of an action. c_b is a hyperparameter, which will usually be set by playing games to test the strength of the agent with different values.

When an action a in a state s_L is chosen that takes us to a state s' not yet in the search tree, the rollout phase begins. In the absence of domain-specific information, the default policy used is simply to choose actions uniformly from those available. Often hand-crafted default policies using domain knowledge give sizeable performance improvements.

Rather than storing the entire game tree, the search tree is built incrementally: when the simulation moves from the tree phase to the rollout phase, we perform an expansion, adding s' to the tree as a child of s_L.[12] Once a rollout is complete, the reward signal is propagated through the tree (known as a *backup*), with each node and edge updating statistics for visit counts $n(s)$, $n(s, a)$, and total returns $R(s, a)$.

When all simulations are complete, and a move must be chosen, the most explored move at the root node is taken. Because UCT will explore higher value actions more often, the most explored action is usually also the highest value action. However, if an action's value changes near the end of the search procedure, then the most explored action and the highest value action differ. Value estimates for the actions at the root are available, so it we can directly select the highest value action, but this has been found to be in practice slightly weaker than taking the most explored action [Enzenberger et al., 2010].

2.6.5 Rapid Action Value Estimation

Rapid action value estimation (RAVE) [Gelly and Silver, 2007] is a technique for providing estimates of the values of moves in the search tree more rapidly than is achieved with MC estimates alone, particularly in the early stages of searching from a state. This is important because the Monte Carlo value estimates require many samples to achieve a low variance. The high variance is particularly problematic when there are many actions available, between which the search budget must be divided.

A common property of many games is that a move that is strong at time t_2 is likely to have also been strong at time $t_1 < t_2$. For instance, in stone placing games such as Go and Hex, if claiming a cell is useful, it may also have been advantageous to claim it earlier. RAVE exploits this heuristic to harness estimates for many actions from a single rollout.

RAVE statistics $n_{\text{RAVE}}(s)$, $n_{\text{RAVE}}(s, a)$ and $R_{\text{RAVE}}(s, a)$ are stored that correspond to the statistics used in normal UCT. After a simulation

[12]Sometimes multiple nodes are added to the tree per iteration, adding children to s' also. Conversely, sometimes an *expansion threshold* is used, so s_L is only expanded after multiple visits.

$s_1, a_1, s_2, a_2, ..., s_T$, with result R, RAVE statistics are updated as follows:

$$\begin{aligned}
n_{\text{RAVE}}(s_{t_i}, a_{t_j}) &:= n_{\text{RAVE}}(s_{t_i}, a_{t_j}) + 1 \quad \forall\ t_i < t_j \\
R_{\text{RAVE}}(s_{t_i}, a_{t_j}) &:= R_{\text{RAVE}}(s_{t_i}, a_{t_j}) + R \quad \forall\ t_i < t_j \\
n_{\text{RAVE}}(s_{t_i}) &:= \sum_a n_{\text{RAVE}}(s_{t_i}, a) \quad \forall\ t_i
\end{aligned}$$

In other words, the statistics for state s_{t_i} are updated for each action that came subsequently as if the action were taken first. This is also known as the *all-moves-as-first* heuristic, and is applicable in any domain where actions can often be transposed.

To use the statistics, an upper confidence bound UCT_{RAVE} is calculated, and averaged with the standard UCT into the tree policy to give $\text{UCT}_{\text{U,RAVE}}$, which then chooses the action. Specifically:

$$\begin{aligned}
\text{UCT}_{\text{RAVE}}(s,a) =& \frac{R_{\text{RAVE}}(s,a)}{n_{\text{RAVE}}(s,a)} + c_b\sqrt{\frac{\log n_{\text{RAVE}}(s)}{n_{\text{RAVE}}(s,a)}} \\
\beta(s,a) =& \sqrt{\frac{c_{\text{RAVE}}}{3n(s) + c_{\text{RAVE}}}} \\
\text{UCT}_{\text{U,RAVE}} =& \beta(s,a)\text{UCT}_{\text{RAVE}}(s,a) \\
& + (1-\beta(s,a))\text{UCT}(s,a)
\end{aligned}$$

The weight factor $\beta(s, a)$ trades the low variance values given by RAVE with the bias of that estimate. As the number of normal samples $n(s)$ increases, the weight given to the RAVE samples tends to 0. c_{RAVE} governs how quickly the RAVE values are down-weighted as the number of normal samples increases. In our experience in Hex, performance was not particularly sensitive to the value of c_{RAVE}, with a performance appearing to be unimodal with respect to the parameter, and values within a factor of 2 of the best found performing almost as well.

2.6.6 Combining Deep Neural Networks with MCTS

Maddison et al. [2015], Clark and Storkey [2015], and Silver et al. [2016] trained policy networks to play Go [13] by imitating datasets of expert human play (see section 2.7). These networks can be used to bias MCTS towards more plausible moves, resulting in strong amateur level play. The biasing policy is often known as a 'prior' policy.[14]

For AlphaGo, Silver et al. [2016] improved their imitation/supervised learning (SL) network with policy gradient reinforcement learning. The RL network beat the SL network it was warm-started with in over 80% of games, but was not more effective as a prior policy for MCTS. It seems that the RL network showed less diversity in strategies than the SL network. Diversity may not be necessary for performance in head to head games against a fixed opponent, but is necessary for biasing an MCTS algorithm effectively, for learning to play against a variety of opponents, or for exploration in RL.

The RL network of Silver et al. [2016] was used to train a value network, using Monte Carlo value estimation. These value estimates were then used to aid in evaluating the leaf nodes during MCTS. Both rollout and value network evaluations were taken at each leaf node, and the average of their values was backed up through the tree. In 2016 AlphaGo defeated professional Go player Fan Hui, the first time an AI has played Go at the level of professional humans.

2.6.7 Using Search for Learning

Prior works have used minimax searches to train value functions. Examples include the TD-root, TD-leaf and TreeStrap algorithms [Samuel, 1959, Schaeffer et al., 2001, Lai, 2015, Veness et al., 2009].

The TD-root algorithm of Samuel [1959] trained a value heuristic for the game checkers by updating the value at state s_t with a target value calculated by a search at the next state s_{t+1}. TD-leaf uses the same target as TD-root, but instead uses it to update the value of a *search* from s_t. Since the value of a minimax search is the value of the leaf along the principal variation, the TD-leaf algorithm adjusts the value of this leaf towards the target value.

[13]Prior to the first publication resulting from this thesis, Anthony et al. [2017], work combining deep learning and MCTS was focused on Go

[14]Though it is not a prior in a Bayesian sense

Treestrap also uses a minimax search to train value predictions. In this algorithm, a minimax search is performed, and then at all nodes in the search tree the heuristic value function is updated towards the recursive value estimates calculated during the search. By updating on all nodes in the search tree, TreeStrap can be more efficient than the TD-root and TD-leaf algorithms. For an individual node, the update is similar to a value iteration update, but with search acting as a multiple step generalisation of the greedy operator.

In Giraffe, Lai [2015] used deep learning with the TD-Leaf algorithm, extending the state-of-the-art for reinforcement learning methods in chess, the heuristic was as strong as traditional chess heuristics, but was much slower to evaluate.

Guided Policy Search [Levine and Koltun, 2013] also used online planning within a learning algorithm. The planning approach used is trajectory optimisation [Kalakrishnan et al., 2011]. Trajectory optimisation is only applicable on tasks where the mapping from an action sequence to resulting value is differentiable with respect to the actions themselves. This is not the case when the action space is discrete, as in classical board games.

Guo et al. [2014] observed that MCTS policies in the Arcade Learning Environment outperformed the best model-free RL policies at the time. By training a neural network policy to imitate the MCTS policies, they created policies that outperformed the best model-free RL algorithms. Imitating a search-based policy is a key component in the algorithms developed in this thesis.

2.7 Imitation Learning

Sometimes data from strong policies is available. For example, in the game of Go, large datasets of expert human play exist [Maddison et al., 2015]. Instead of attempting to learn policies for MDPs via reinforcement learning, we can instead try to act similarly to the players in the dataset.

The simplest approach, behavioural cloning [Pomerleau, 1991], is to treat the prediction of expert moves as a classification task. The game's state is the input, and the chosen action is the label. This is an easier problem than learning to play a game with reinforcement learning: the algorithm no longer needs to discover what good play looks like, we 'only' need to learn to represent and generalise it.

However, good performance in behavioural cloning - i.e. when few mistakes are made in move prediction on the test set - doesn't necessarily translate to good performance in the MDP or game by following the *apprentice* policy learnt.

One reason for this is that the mistakes made can be particularly costly in some way. Perhaps there are some actions that immediately cause defeat, which the experts never make, but the behavioural cloning agent makes occasionally; these mistakes can dominate the reward. Conversely, it could be that the agent systematically fails to follow one key part to the policy, and so although every other part can be carried out effectively, reward is not obtained. Cost-sensitive losses [Daumé et al., 2009], as seen before in some policy iteration algorithms, offer some defence against this kind of issue.

A second reason a policy from behavioural cloning may perform poorly is that following it results in a shift in state distribution. If expert policies are gathered on a state distribution resulting from expert play, when we follow the apprentice policy we can reach states that never appeared in the dataset, because the apprentice and expert strategies differ. This state may even be perfectly fine, in that strong play from it will solve the task, but the apprentice policy has never seen strong play from this kind of state. Therefore it must extrapolate to predict what an expert would do here, which it may fail to do.

DAgger [Ross et al., 2011] is an intuitive and theoretically motivated approach to solving this second problem. If we can sample expert actions on the apprentice policy's state distribution, we can train our policy on these states. This gives a new apprentice policy, we can alternate between gathering more data on the current apprentice's distribution, and training the apprentice on the data generated so far.

2.8 Artificial Intelligence in Classical Games

The application of artificial intelligence to playing classical board games has a history as long as that of computer science itself, dating back to work by Charles Babbage. This history has included both general approaches, such as learning-based methods, and techniques specific to particular board games.

2.8.1 Traditional Search Based Methods

Shannon [1950] discusses the use of heuristic evaluations for chess based on piece values, and combining them with minimax search. Extensions to this have remained the most common and successful approach to creating a chess AI for many years. DeepBlue [Campbell et al., 2002], which famously defeated world champion Gary Kasparov, used this approach. In DeepBlue, the heuristic was a linear function on a large set of hand-crafted features, which were designed by human experts. Their weights were largely tuned by hand. A very deep search on specialised hardware found strategies that were minimax on that value heuristic. Modern chess engines use even more hard-won chess programming knowledge, including endgame tablebases that store solutions to all chess states with few enough pieces, extensive heuristics for ordering moves, and heuristic evaluation parameters tuned at great expense in both experts' time and computation for testing.

In Go, similar techniques were relatively less successful [Müller, 2002]. Go has a larger branching factor than chess, but evidence suggests this is not the only thing that prevented the methods that worked well in chess from performing well in Go. In particular, 9×9 Go AIs were also poor, despite a much more similar branching factor to chess. Rather, it seems, a key problem for computer Go was the difficulty of crafting knowledge for heuristic functions.

MCTS was a major breakthrough in computer Go, lifting the strength of programs from the weak *kyu* ranks to advanced amateur *dan* ranks, largely due to the way that Monte Carlo value estimates tackled the value heuristic problem. However, uniform policy rollouts provide a flawed heuristic, so better rollout policies were designed, either by hand or using data from human play. The design of rollout policies proved taxing, since stronger policies can actually lead to weaker play by the MCTS program [Silver and Tesauro, 2009].

AlphaGo [Silver et al., 2016] made a major breakthrough for MCTS algorithms with two main advances:

- Biasing the search towards prior knowledge from imitating human play with a deep neural network.
- Supplementing rollout evaluation with a deep neural network value function trained by a combination of imitation learning, RL and Monte Carlo value estimation.

It was able to defeat professional Go players for the first time, conclusively proving that deep learning could be the missing piece of the puzzle of mastering Go. Nonetheless, it took advantage of substantial human data and expertise in Go, so for many traditional games, how to train an AI to play at expert or superhuman level *without* recourse to data from matches between expert human games players or human expert knowledge of strategy for the game remained (for example to design a position evaluation heuristic or to generate dataset of human play) an open problem. The aim of both this thesis and the similar concurrent work AlphaGo Zero [Silver et al., 2017] is to obviate the need for such knowledge in AI for games.

2.8.2 Temporal Difference Learning in Games

Tesauro [1994] applied Temporal Difference learning to the game of backgammon. The approach performed strongly with a 1-ply search. When combined with 2-ply search, and using some hand-crafted features, the program was competitive with master level human players. Superhuman play was achieved not long after. It was particularly strong in aspects of judgement. In the words of backgammon analyst Kit Woosley [Tesauro, 1995],

"TD-Gammon has definitely come into its own. There is no question in my mind that its positional judgement is far better than mine. Only on small technical areas can I claim a definite advantage over it . . . . I find a comparison of TD-Gammon and the high-level chess computers fascinating. The chess computers are tremendous in tactical positions where variations can be calculated out. Their weakness is in vague positional games, where it is not obvious what is going on . . . . TD-Gammon is just the opposite. Its strength is in the vague positional battles where judgement, not calculation, is the key. There, it has a definite edge over humans. Its technique is less than perfect in such things as building up a board with no opposing contact, and bearing in against an anchor. In these sorts of positions the human can often come up with the better play by calculating it out . . . . In the more complex positions, TD has a definite edge. In particular, its judgement on bold vs. safe play decisions, which is what backgammon really is all about, is nothing short of phenomenal."

This result was particularly exciting since it illustrated an important possibility: applying learning approaches to playing games of intellect. Off the back of the success of TD-gammon, temporal difference approaches were applied to many other games, including Chess [Thrun, 1995, Beal and Smith, 1997, Baxter et al., 2000], Shogi [Beal and Smith, 2001], and Go [Schraudolph et al., 1994]. The results of these works were, unfortunately, far less spectacular, with the algorithms not reaching human-level play.

In light of these results, a likely hypothesis is that a particular feature of backgammon made TD-Gammon possible: randomness [Pollack and Blair, 1997]. Before a player takes their turn in backgammon, they must roll dice to determine which actions are legal. As such, very precise move sequences are often not as important in backgammon: good plans must instead be robust to the chance events that could derail them. Furthermore, a small difference in the state can often be reversed in the next dice roll; as such, small state changes don't have the same capacity to change the value of the game.

In other words, the use of dice in backgammon makes the game far less *sensitive*, in the sense described in chapter 1. In contrast, deterministic board games are exceptionally sensitive, and the poorer technical play noted by Kit Woosley proved to be a critical flaw, from which RL agents never escaped.

TD(leaf) based learning techniques have achieved stronger results than direct application of TD learning. It seems that the search component of TD(leaf) might be alleviating the tactical naivety that arose from vanilla TD learning. A very early success was achieved in checkers [Samuel, 1959], training a polynomial value function over some checkers-specific features, referring to important strategic concepts. Besides these features, the procedure did not use checker's knowledge; this work is an early attempt to tackle a game by replacing domain expertise with machine learning expertise. The resulting program was rated by players as 'better than an average human', and in particular was a stronger player than the system designer. TD(leaf) methods continued to find success in checkers, for example Schaeffer et al. [2001] learnt parameters for the evaluation function for the Chinook program that were as strong as hand-crafted weights (this evaluation function used a linear mapping, again from hand-chosen features).

While these efforts in checkers made some use of human knowledge in their feature sets, a strong positive is the generality of the TD(leaf) learning procedure used. This has allowed it to be applied successfully to other domains. A prominent example is Lai [2015], which applied TD(leaf) to chess. They were able to train a value function of similar strength to the hand-crafted heuristics used by traditional chess engines, although the neural network was much slower to evaluate than traditional heuristics.

Although more successful in sensitive domains than TD-learning, general learning methods using TD(leaf) have not matched or surpassed classical expertise-intensive methods for many games.

2.9 Computer Hex

Computer Hex has undergone considerable study, in this section we review those programs. In sections 2.9.1 to 2.9.4 we review some individual components used by successful Hex AIs. Section 2.9.5 discusses how these methods can be used to shrink the search space and weakly solve Hex up to 9×9. In section 2.9.6 we discuss the MoHex programs, which have been the leading classical Hex programs for over a decade, and, as they are the strongest open source Hex programs, will serve as the benchmark for the methods introduced in this thesis. Finally, in section 2.9.7 we discuss other machine learning approaches to Hex, their performance compared to the MoHex benchmark we use in this work, and contrast the objectives of those works to our goal to learn without expert strategy knowledge.

2.9.1 Electrical Resistance

Shannon [1953] developed an evaluation heuristic for Hex based on viewing the board as an electrical circuit. Cells with the player's own stones have a resistance of 0, cells with the opponents stones have infinite resistance, while empty cells have a resistance of 1. The total resistance between two board locations then represents the difficulty of connecting them. The resistance for black between the North and South edges, and the resistance for white between East and West edges can therefore be compared to give an evaluation of the current state.

The Shannon heuristic is able to capture some strategic understanding of the board, but greedily playing with respect to it results in play that is generally tactically naive. Searching for a strategy that is minimax to a fixed depth using the Shannon heuristic (e.g. via α-β search) can lead to strong play, and this heuristic was used by Olympiad winning Hex programs until 2008 (e.g. Melis and Hayward [2003]), although other techniques were also used in those programs.

2.9.2 H-Search

Virtual connections (see section 1.2.1) can be automatically calculated via an algorithm known as H-Search [Anshelevich, 2000, Pawlewicz et al., 2015]. A virtual connection from one edge of the board to the opposite edge will prove that a particular player has won, along with a strategy to force that victory. Versions of H-search take advantage of insights on how different virtual connections and semi-connections can be combined to create new, more complicated connections. Designing these rules required human insight into Hex strategy.

In complicated middle-game positions, it is common that a player who has just moved has a semi-connection from one edge of the board to another. In other words, H-search can find a winning strategy for them if it was their turn, but cannot find one given that it is their opponent to move. This semi-connection relies on a specific support (set of empty cells), and to prevent their opponent from winning, the current player is forced to play in one of those empty cells (which is therefore known as the *mustplay*). Any other move returns control to their opponent without blocking the semi-connection, and so will lose the game. Hex AIs can prune moves outside of the mustplay from their search tree, reducing the state-space considerably.

2.9.3 Captured Cells and Inferior Cells

Sometime playing a move in a particular cell does not add any connections for a player, because too many of the adjacent cells already have opponent stones in them. When this happens, it is known as a captured cell, and it is possible to add a stone of a particular colour to the board without changing the position strategically. Adding the extra stone reduces the branching factor for the whole game tree, which helps search methods.

Sometimes an action a_1 can be proven to be no stronger than another action a_2, for instance because all connections that a_1 makes would also be made by a_2. This is another reason that it might be possible to prune an action from search.

Some captured and inferior cells can be calculated recursively: once one cell is captured, the stone that is added it means that another cell becomes inferior or captured. Virtual connections can also sometimes play a role in determining if a cell is captured. Again, this technique relies on human insight into the strategy of Hex.

2.9.4 Learning Patterns Weights for Hex

Huang et al. [2013] learn pattern weights for Hex. Patterns are local configurations of the board around a possible next move, for example including the positions of stones in the 12 closest cells (see figure 2.1. The weights of the patterns are learnt values representing the correlation between playing the move represented by the pattern and winning the game.

These patterns can then be used to inform the default policy in an MCTS algorithm, and to prioritise the exploration of new leaves in a search tree. Pattern weights have been successfully applied to games other than Hex, such as Go [Enzenberger et al., 2010], however they require a dataset of strong play for training. In Huang et al. [2013] this used human expert games and games between previous AI systems. Therefore their method relies on human understanding of Hex strategy both directly from the human players who contributed to the dataset, and indirectly through its application to the design of the AI agents.

Because pattern weights only use local features of the board, they can be cheap to calculate, which is important for their efficient use in MCTS, but this also limits the total possible improvement in playing strength.

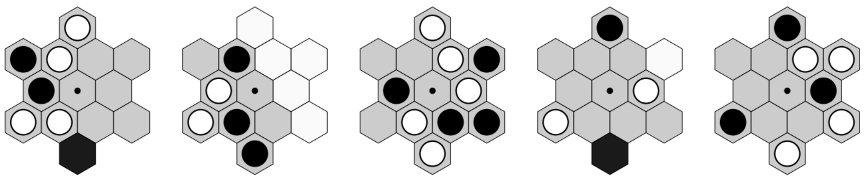

Figure 2.1: Examples of Hex patterns with black to move on the centre cell. Stones filled in white or black are the board edges. In each instance, the pattern was given a high weight. The moves also all appear urgent, as they prevent white from connecting (or virtually connecting) separate groups of stones. From Huang et al. [2013]

.

2.9.5 Solving Hex

Several works have also considered solving Hex [Henderson, 2010, Pawlewicz and Hayward, 2013]. Combining the ideas discussed in this chapter with proof number search [Kishimoto et al., 2012] can solve end-game states in 13×13 and 11×11 Hex. For boards sized 9×9 and below, Hex is weakly solved using these techniques. Hex is PSPACE-complete, so these solvers do not scale well to larger boards [Reisch, 1981], where heuristic play is needed. These solvers would also not have been possible without the extensive Hex-specific insights.

2.9.6 MoHex

Versions of MoHex [Arneson et al., 2010, Huang et al., 2013, Pawlewicz et al., 2015], a Hex program developed by the Hex research group at the University of Alberta, have won every computer games olympiad since 2009. MoHex uses MCTS, but enhanced by many Hex specific ideas. All versions have virtual connection engines, inferior cell engines, and action pruning based on these. MoHex 1.0 uses a hand-crafted rollout policy, while MoHex 2.0 uses pattern weights in its rollout policy (see section 2.9.4), trained on expert human games. Both have endgame solvers to ensure perfect play towards the end of their games, and are built on a highly optimised MCTS implementation, Fuego [Enzenberger et al., 2010].

Short exhibition matches between a highly ranked human player and MoHex and DeepHex [Pawlewicz and Hayward, 2015] were played at the 2015

Computer Games Olympiad. The games were played on an 11×11 board, with 15 minutes a side. The AIs won all four matches [Hayward et al., 2015].

MoHex achieves its strength using the ingenious but Hex-specific techniques described in this section. However, many of these do not apply to even similar games, such as Go. Requiring this sort of intensive case-by-case effort limits the potential of AI in real world applications, both due to the cost of conducting the necessary research and because domain specific ideas may be less effective in other domains (such as in Go, where classical MCTS programs are far from superhuman).

2.9.7 Machine Learning in Hex

In recent years, there has been a growing interest in applying modern machine learning techniques to Hex, particularly using deep learning methods.

NeuroHex [Young et al., 2016] used DQN to train a Q-network to play Hex. Before applying DQN, the network was trained to predict the Shannon resistance heuristic for action values, a supervised regression task. The dataset consisted of positions resulting from self-play of the Wolve search-based program [15]. Additionally, the same dataset of positions was used as a set of starting positions for self-play games for DQN, to ensure a diversity of positions was maintained throughout learning. The resulting network could win some games against MoHex, for example winning 8% against MoHex with 1 second per move, but was much weaker than the state-of-the-art.

Gao et al. [2018a] used versions of actor-critic algorithms to learn to play Hex. Like NeuroHex, learning is warm-started using supervised learning of a classical Hex program, in this case, by predicting the moves made by the MoHex program. Reinforcement learning was able to improve the winrate of this network against Wolve (with search depth 1) from $\approx 20\%$ to $\approx 40\%$. When combined with MoHex 2.0, the resulting program could beat MoHex 2.0 in around 60% of games with equal time controls (the exact win rate varied slightly depending on board size).

[15]Wolve is an $\alpha - \beta$ search player of similar strength to MoHex 1.0 [Arneson et al., 2010]

Takada et al. [2017] train a value network to predict the value estimates of a 1-ply search program. Concurrently, the search program uses the value network as its evaluation heuristic, meaning that this approach effectively implements Value Iteration, but augmented using the Hex specific features of the searcher, such as use of pruning to the mustplay and automatically playing out any forced wins discovered by H-search. The final agent was able to win 60% of games against MoHex 2.0. Unlike other works for learning in Hex, the network was not warm-started using a strong previous Hex program, however it still uses knowledge of Hex strategy from the design of the search method.

Gao et al. [2017] trained a neural network to predict the moves made by MoHex 2.0, and incorporated the resulting network back into the search program. This can be seen as incorporating a single iteration of Expert Iteration in the development of a Hex-specific program. The neural-network enhanced MoHex-CNN achieved a winrate of 69.9% against MoHex 2.0 in 13×13 Hex. [Gao et al., 2018b] extended this method by training an additional Q-function, winning 82.4% of games against MoHex 2.0. This is the strongest reported result against MoHex 2.0 in the literature on any board size. [16]

All these works using learning for Hex made significant use of expert knowledge of Hex strategy in some form during training. When the goal is to achieve the highest possible performance on the specific game of Hex, this is a sensible approach, and compared to training an agent without that knowledge can be much more cost effective. These works also mostly focus on 11×11 or 13×13 Hex, as these are the variants played in the Computer Hex Olympiad.

However, when algorithms are highly reliant on game-specific knowledge, then to apply them to other domains, similar game-specific knowledge must be developed, which may require much effort and ingenuity, or it may not actually be possible to transfer the ideas, because they rely on strategy knowledge that cannot be generated for the new domain.

[16]This agent's neural network predicts the moves of MoHex 2.0, this means it is likely that it achieves particularly high winrates against MoHex, because it will tend to search the actions MoHex will take.

The goal of this thesis is to play Hex well *tabula rasa*, i.e. in a way that could be reasonably expected to generalise to many more games. As such, we will eschew many of the techniques of this section, even though they could help performance, in favour of combining deep learning models with new, general reinforcement learning algorithms.

Chapter 3

Expert Iteration

When solving large-scale sequential decision making problems, an agent must both discover strong policies and generalise those policies across the state-space. In this chapter we develop the Expert Iteration framework, a class of algorithms that separates the discovery and generalisation tasks. They are solved via planning and supervised learning methods respectively.

In contrast, model-free algorithms attempt to learn policies and value functions directly from experience. Combining model-free approaches with deep learning has lead to substantial improvements in the ability of RL agents to generalise their policies across large and high-dimensional state spaces (such as video input). This improved ability for agents to perceive the world has precipitated state-of-the-art performance in benchmark domains such as Atari games [Mnih et al., 2015] or robotic control tasks [Schulman et al., 2015a].

Nonetheless, these successes of deep reinforcement learning have come in domains where the underlying policy being learnt is conceptually fairly simple. Most Atari games or motor control tasks would appear to humans to be games of *mechanical* skill, rather than being deep *intellectual* challenges. Actions normally have a fairly immediate impact, such as successfully hitting a target or missing, and the correct approach does not usually depend on the state in a complex way.

In classical board games actions can have subtle implications that are only apparent after many more moves have been taken. Even the smallest possible change to the state can disproportionately affect both how a player should play and which player is currently winning. It isn't possible to perform well when making mistakes, and the game is very different when played with many mistakes, compared to playing well.

Successful AIs in classical board games have relied on the use of prior knowledge in the form of handcrafted heuristics, features, or data from expert human play, as we saw in section 2.8. Additionally, at test-time online search is used to improve the quality of play. Such search focuses computational effort to the current state, and is able to correct weaknesses in the heuristics used, whether hard-coded or learnt. In other words, the search is able to plan, handling subtleties of the game that heuristics fail to understand.

In Expert Iteration (ExIt), we aim to harness the ability of online search to improve on heuristics as a training aid for those heuristics. This essentially decomposes the RL problem into separate planning and generalisation tasks: online search plans in order to discover new policies, then a function approximator, such as a neural network, generalises those plans across the state space. Subsequently, the function approximator — now improved by imitating the searches better plans — guides future searches, increasing the strength of the search's plans in the next iteration.

When people play board games such as Hex, they use planning combined with intuitive judgements. An analogy between the techniques used in the Expert Iteration framework and human approaches to such games is explored in section 3.2.5.

At a low level, ExIt can be viewed as an extension of Imitation Learning (IL) methods to domains where the best available experts (human or machine) are unable to achieve satisfactory performance. In IL, an *apprentice* is trained to imitate the behaviour of an *expert* policy. In ExIt we iteratively re-solve the IL problem. Between each iteration, we perform an expert improvement, where we bootstrap the apprentice policy as a heuristic to improve the performance of the expert. Using search to generate our own expert play obviates the need to rely on human expert players.

From another perspective, by ExIt we mean the class of algorithms which extend versions of policy iteration (section 2.4.1, Lagoudakis and Parr [2003], Puterman and Shin [1978], Scherrer et al. [2015]) by using multiple step planning in place of the single-step greedy improvements.

3.1 Expert Iteration

3.1.1 Preference for Imitation Learning

A central idea of Expert Iteration is that, if we have access to some *expert* policy π^*, which is far stronger than the learner's policy π, then progress can be made most quickly by constructing a supervised learning task: either predicting the expert policy π^*, as in behavioural cloning, or estimating the value of that policy V^{π^*}, classification and regression tasks respectively. (This is instead of attempting to improve the learner policy π directly via reinforcement learning).

Supervised learning is generally an easier problem than reinforcement learning, as we will discuss in section 3.2.1, and so this proxy problem can be solved more successfully than the original. Furthermore, by solving it we can potentially improve our policy to be as strong as π^* right away, rather than through a succession of incremental improvements.

In other words, when a suitable expert is available, we should prefer to use imitation learning over reinforcement learning. Indeed, in tasks too complex for model-free reinforcement learning algorithms, state-of-the-art and often superhuman systems have been created in part by imitating the behaviour of humans. By imitating humans, the AI can side-step the challenge of discovering strong policies for itself, to focus merely on generalising human policies instead.

However, once the policy is imitated, if the resulting performance is unsatisfactory, any further improvement must rely on reinforcement learning. And if a reinforcement learning algorithm struggles to learn human-level play from scratch, we might expect that, starting from human-level play, it will only learn to improve that play to a limited degree, meaning the final performance of the agent is still dependant on the quality of human play it began with. Therefore our approach is to automatically generate experts that outperform our current policy with a search based planner, removing the need for a human expert.

Another approach to using automatically generated expert demonstrations is Guided policy search [Levine and Koltun, 2013]. This algorithm utilises trajectory optimisation methods to generate 'guiding samples' to aid policy learning. Similarly to expert iteration, a planning method is used to generate training data from higher-quality play than the current policy is capable of. Guided policy search uses this data differently to expert iteration, however. Whereas Expert Iteration attempts to imitate the expert play, in guided policy search, the objective is to estimate the gradient of expected return with respect to policy parameters (at the current parameters), but using guiding (or 'expert') policy samples. The gradient estimate is made by using importance weights to correct for the mis-match between the guiding policy and parametric policy. As a result, if the guiding policy deviates far from current policy, for example to correct a bad mistake, the importance weights will be low. This has the disadvantage that the important data contributes less to the policy update.

The PEGASUS algorithm [Ng and Jordan, 2000] also uses a model to improve the performance of a policy gradient learning approach, and has shown success in application to robotics (e.g. Ng et al. [2003]). Unlike a model-free algorithm, PEGASUS is able to contrast multiple simulations from a single state, and importantly, by assuming control over the model's random number generation, it can use the same random numbers for the different simulations. The common random numbers remove a source of noise in the gradient estimation, and thereby reduce the variance of the policy gradient updates. In this thesis, we tackle different a different challenge: in Hex the transition model is already deterministic, but we need multi-step planning to overcome the complexity of the required strategy, which policy gradient based approaches do not provide.

3.1.2 Expert Improvement

In order to take advantage of the benefits of imitation learning throughout training, ExIt enriches imitation learning with an *expert improvement* step. Improving the expert player and then re-solving the Imitation Learning problem allows us to exploit the fast convergence properties of Imitation Learning even in contexts where no strong player was originally known, including when learning *tabula rasa*.

To do this, rather than using a human expert, we use a search algorithm to provide an expert policy. Such algorithms require significant deliberation time to make their decisions, and make use of fast, approximate heuristics to aid those decisions.

Deep neural networks can act as heuristics, and stronger heuristics result in stronger play. As such a mutual cycle of improvement can be achieved where apprentice neural networks imitate an expert, and then an improved expert can be defined by using the stronger neural network as heuristics.

Including both policy and value approximation for the apprentice, the general form of ExIt is:

Algorithm 2 Expert Iteration

```
Require: Apprentice value function $V$ and policy function $\pi$ with parameter
    space $\Theta$
Require: Initial parameter values $\theta_0 \in \Theta$
 1: for $i = 0$; $i \leq$ max_iterations; $i$++ do                ▷ One expert iteration
 2:     $D = \emptyset$
 3:     for $j = 1$; $j \leq$ data_set_size; $j$++ do     ▷ Create a datapoint each loop
 4:         $s \sim \mu(s|\pi^{\text{sampling}}(\cdot|\cdot;\theta_i))$ ▷ State sampled from exploratory self-play
 5:         $q \leftarrow \pi^*(a|s;\theta_i)$                     ▷ Expert policy target at state $s$
 6:         $z \sim p\left(\sum_{t=1}^{n} \gamma^{t-1} r_t + V(s_n;\theta_i)|s_0 = s, \pi^{\text{exploit}}(\cdot|\cdot;\theta_i)\right)$
 7:                                                              ▷ Sample Value target
 8:         $D \leftarrow D \cup \{(s, q, z)\}$
 9:     end for
10:     $\theta_{i+1} \leftarrow \arg\min_{\theta\in\Theta} \mathbb{E}_{(s,\pi,z)\sim\text{Unif}(D)} \left[\mathcal{L}_\pi(q, \pi(a|s;\theta)) + \mathcal{L}_V(z, V(s;\theta))\right]$
11: end for
```

The outer for loop of this algorithm represents one expert iteration. In an iteration of the algorithm, we collect a dataset of play from our current expert (lines 2-9), and train our next apprentice to approximate the expert policy using this data (line 10).

To create a datapoint, we first need sample a state from the state distribution induced by some state-sampling policy π^{sampling} (line 4). To generate a single sampled state, first simulate a self-play game using π^{sampling}, and then sample a single state from that game. To cheaply generate multiple correlated samples, multiple or all states from the simulation, but to generate uncorrelated data, a new game must be simulated each time. The role of the sampling distribution is to ensure good coverage of the state-space of the game.

Once a state s has been sampled, we need to generate our expert policy target (line 5). This target q is a distribution over the actions available from state s. One straightforward option would be to sample an action a from the expert policy, and use this as target. In this case q would place all it's mass on the sampled action a. The expert $\pi^*(a|s;\theta_i)$ is a policy that results from a search procedure. The policy of search depends on the current apprentice, which is used for policy and/or value heuristics, which is made explicit here as a dependence on the previous iteration apprentice's policy parameters θ_i.

We leave the choice of expert search algorithm to future chapters, the crucial requirement is that $\pi^*(a|s;\theta_i)$ be stronger than $\pi(a|s;\theta_i)$.

If a value approximation is required by the expert's search algorithm, we also sample data to train an apprentice value target (line 8), which will be a sample to estimate the expected return from the state s. To do this, we sample the rest of the game from the state s by following some exploitation policy $\pi^{\text{exploit}}(a|s;\theta_i)$. From this simulation we calculate a value target for the current player, possibly using bootstrapping, which will become the training target z for the apprentice.

The tuple (s, q, z) is then added to the dataset (line 8). When fitting the parameters for the next apprentice policy (line 10), the loss consists of two losses. First, a policy loss $\mathcal{L}_\pi$ measuring the error in prediction of the expert policy target q by the apprentice policy, such as the KL divergence between the target and apprentice prediction. Second, a value loss $\mathcal{L}_V$, measuring the error in prediction of the value target z by the apprentice value function $V(s;\theta)$, for example the square error. With the dataset now sampled, this step is to solve an imitation learning/supervised learning problem on that dataset, predicting the targets given the input of the state s

For a practical implementation of expert iteration we must choose: our state sampling policy π^{sampling}; our expert (i.e. multi-step improvement operator) π^*, the policy for generating value targets, π^{exploit}; our policy and value losses $\mathcal{L}_V$ and $\mathcal{L}_\pi$; and the function classes of our apprentice.

Each component of the algorithm is crucial for the success of an ExIt algorithm; they are the topics of chapters 4 and 5. A strong initial expert can also be important for accelerating learning, which we employ in chapter 4. In this chapter we discuss the high-level motivations and hypothesised advantages of the Expert Iteration framework.

3.2 Why Expert Iteration?

In this section, we discuss the differences between the ExIt approach and alternative RL algorithms. In section 3.2.1 we discuss an advantage which it has in common with classification-based policy iteration. In section 3.2.2 we explain how simulation based search methods produce lower variance targets than model-free methods; this benefit applies to both ExIt and classification-based policy iteration algorithms, but may be greater in ExIt methods. In sections 3.2.3 and 3.2.4, we discuss how multiple step improvements of ExIt can be beneficial, particularly in classical board games; these advantages may be common between ExIt and search-based value iteration methods such as TD(root). In section 3.2.5, we discuss human reasoning, studies on how people play chess, and how the ExIt approach may be analogous.

3.2.1 Reinforcement Learning as Supervised Learning

ExIt recasts the reinforcement learning problem as a succession of supervised learning problems, similarly to other classification based policy iteration algorithms. This is because a dataset of experience is sampled, and then the function approximators are trained to classify actions and regress against value targets given the state.

One advantage of this approach is that, as noted by Lagoudakis and Parr [2003], supervised learning is in general better understood than reinforcement learning.

This is still the case: recent major successes in deep learning for supervised tasks, such as AlexNet [Krizhevsky et al., 2012] have predated (and indeed precipitated) similar success in deep reinforcement learning, with RL practitioners adapting techniques to RL once they have been proven in supervised problems. In both classification based policy iteration and ExIt, such advances are used in the context they are best understood as out-of-the-box tools.

Several recent successful approaches to reinforcement learning problems have used supervised learning, or borrowed ideas from it. AlphaGo [Silver et al., 2016] achieved strong play in Go, with imitating human play in a supervised manner a key component. This serves as a proof of concept that, if the right supervision data can be generated, a neural network can be trained to play such a game at a high level; in contrast, methods such as TD-learning or policy gradients have shown comparatively less success (see section 2.8). As discussed in section 2.3.3, DQN can be seen as a partial casting of reinforcement learning as a succession of regression problems.

As well as these overall trends in the literature, there are some concrete results pointing to advantages to supervised learning. For learning policies, the classification-based approach allows for greater freedom in the choice of optimisation objective than model-free RL methods. This is because a target policy is constructed, and then any classification loss measure can be used to compare the apprentice policy to the target policy.

Most crucially, classification-based learning allows us to employ cross-entropy losses. The model-free RL approaches to policy optimisation instead use policy gradient losses.[1] A 'policy-gradient' could be used in classification too. Given a classifier $f(y|x)$, we can use an $\mathcal{L}^1$ loss:

$$\mathcal{L}^1 = 1 - f(y^*|x) = \mathbb{E}_{y\sim f(y|x)}\left[1 - \mathbb{1}_{y=y^*}\right]$$

where y^* is the true label. If we sampled predicted labels from the classifier $f(y|x)$ and estimated the gradient of the $\mathcal{L}^1$ loss with a score function estimate, we would end up essentially treating the supervised task as a contextual bandit, with a reward of 1 whenever the correct action (label) is chosen, and 0 otherwise, and doing policy gradient learning on it.

However, the empirical performance of using the $\mathcal{L}^1$ (with either exact or score function gradient) is considerably worse than when using the cross-entropy (or equivalently, KL) loss. This was noted by, for example, Chen et al. [2019]. In other words, contrary to the popular wisdom that variance is the largest problem for reinforcement learning [Ahmed et al., 2019], even

[1]Or the similar approach of evolution strategies, which is typically still less efficient

in a zero-variance scenario, policy gradient methods fall behind cross-entropy methods.

3.2.2 Low Variance Samples

After taking the discussion of the previous section into account, it remains the case that high variance updates are also a serious issue for reinforcement learning algorithms [Konda and Tsitsiklis, 2000, Schulman et al., 2015b]. The training targets used by ExIt algorithms are much lower variance than the advantage estimates or value estimates used in model-free RL. This is because the searching algorithm uses multiple simulations for selecting an action, and averages between those simulations.

Another view is that using N simulations in each search means that the effective batch size is N times larger. A model-free algorithm could compensate for high variance samples with larger batches or lower learning rates for a similar effect. This is more difficult to parallelise, requires more episodes of experience, and is expensive in terms of the number of network updates necessary.

In a policy gradient method, an effective baseline is crucial for low-variance learning, even when the batch size is large. The ideal baseline is the true state value, so a common approach is to use a value estimate of the state $V_\theta(s)$ [2]. The advantage is therefore calculated by comparing the value estimate of some future state $V(s')$ to the value estimate of the current state.

In a simulation based search we instead compare many possible future states to each other. Effectively, when considering action a, it is being compared to a 'baseline' of the average value of all simulations. Because this is based on many simulations, it is likely to be a more accurate estimate of the value of the current state than the point estimate provided by the learnt $V_\theta(s)$

However, when using search to drive learning, there is a trade-off between the amount of search per data-point, and the number of datapoints in the dataset, or the number of policy iterations completed (given a fixed computa-

[2]See section 2.3.4 for more discussion of baselines

tion budget). With 1-step greedy policy iteration algorithms, the most efficient approach can therefore be to use a minimal number of samples per data point [Scherrer et al., 2015]. Sampling many samples from a single point in ExIt is not deleterious in the same way, as the samples do double duty: they reduce the variance of the data, and they improve the simulation policy, so fewer policy iterations are needed. Therefore, a large dataset of low variance samples can be maintained.

Whether the optimal trade-off is for many cheap samples or fewer expensive samples is difficult to say: it depends on the problem, the whole learning system and even the stage of learning. Nonetheless, ExIt makes low-variance sampling substantially more practical than for 1-step policy iteration, because longer searches lead to much stronger policies, as well as reducing variance.

3.2.3 Multiple Step Improvement and Exploration

ExIt generalises policy iteration in that it does not employ a 1-step greedy policy improvement. Instead, through the online planning algorithm, a multiple step improvement is calculated. A multiple step improvement has the potential to give a larger improvement than a single step improvement, by finding new plans that require deviation from the current policy in multiple moves. This can accelerate the learning process, and may be of particular importance in domains where good strategies consist of multiple step plans.

During search, the planning algorithm repeatedly simulates play around the current state. We can compare the distribution of states reached during search to that of model-free RL algorithms. In a model-free method, typically states arise from playing out some single policy, with subsequent trajectories being independent (apart from the gradual effect of learning). Compared to this, ExIt generates a significantly different distribution. Each state reached is the result of two different policies. First, the sampling policy that was played to reach the state at the root of the search, then a different, exploratory policy internal to the search algorithm, such as the policy determined by the UCT formula in MCTS.

The search's exploration therefore will usually differ from typical methods in model-free learning. It can take several exploratory, off-policy actions in a row, which might be helpful for discovering plans consisting of sequences of actions. With on-policy sampling or epsilon-greedy exploration, there is no such correlation between the sampling of consecutive actions.

Secondly, because it explores multiple times from a single state, it can use information gained to correlate the exploration, investigating a promising line in more depth. This contrasts to the gradual approach of model-free algorithms, where deviating across sequences of actions, or investigating a new line require modifying the parametric policy.

We can also choose specific policies for determining the set of states considered by search and the exploitation policy evaluated. For example, we found that training value functions with greedy policies resulted in stronger play than evaluating stochastic versions of policies, but because we only used those samples for value learning, it did not affect the policy exploration. In Silver et al. [2017], an expert policy is used to find states to be considered for search, this might lead to exploration effort being expended in more relevant states than when following the parametric policy.

Finally, an advantage we did not anticipate, but found particularly important when conducting this research: usually when studying an exploration approach, success can only be measured by running the full learning algorithm. This is typically expensive, and has many moving parts that might affect the success for an exploration method. Changing the exploration strategy in an ExIt algorithm normally means changing the search algorithm. Success can be quickly tested by self-play games, since we typically expect better self-play results to translate to better learning in ExIt algorithms. This approach provides a practical research process. Although it is not necessarily the case that better winrates in self-play of the search algorithm would translate to better learning when that search is used in ExIt, in our experimentation we found no example where it was false.

3.2.4 Multiple Step Improvement in Sensitive Domains

Multiple step improvements can increase the ability of an algorithm to withstand policy approximation error. In deep reinforcement learning, all policies and value estimates learned include some degree of error. When a policy is then optimised to perform well based on the single-step improvement on the current policy, it may be that the direction of improvement is determined mostly by the approximation errors in future steps. In other words, the agent is mostly solving the task of avoiding states where it plays poorly, or reaching states where its opponent performs poorly, rather than learning to play well overall.

In a classical board game, the sensitivity of the optimal policy and value functions to their inputs means that we should expect these errors to be both more important, because small errors have a large impact on the game, and more common, because less smooth functions are more difficult to approximate accurately. An improved search-based expert policy can detect subtle differences that are missed by the apprentice policy. This then allows training to proceed in the correct overall direction, in spite of approximation errors that might otherwise dominate learning. Crucially, the expert policy only corrects these errors when they appear within the planning horizon. The more steps ahead the expert is able to plan, the more likely it is for the correction to occur.

Observing overall trends in the literature, it has always been the case state-of-the-art methods for many classical board games have made extensive use of search. This by itself suggests that using search within a reinforcement learning algorithm might be particularly beneficial in tackling these domains.

3.2.5 Thinking Fast and Slow

According to dual process theory [Kahneman, 2011], human reasoning consists of two different kinds of thinking. *System 1* is a fast, unconscious and automatic mode of thought, also known as *intuition* or *heuristic process*. *System 2*, an evolutionarily recent process unique to humans, is a slow, conscious, explicit and rule-based mode of *reasoning*.

Psychologists can tease apart the differences between these two systems by examining people's answers when posed problems with an intuitive *but wrong* solution [Frederick, 2005]. For example:

> A bat and a ball cost $1.10. The bat costs $1.00 more than the ball. How much does the ball cost? __ cents

Many participants answered 10 cents (the correct answer is 5 cents). People who answer more quickly are far more likely to give the wrong answer, and slowing participants down, for instance with a hard to read typeface, reduces the number of mistakes. The explanation of dual-process theory is that this is a problem that is difficult to solve via the heuristic reasoning of system 1, but easy via a rule based method, such as simple algebra; or a guess-check-refine approach (guess 10 cents, notice 10 cents plus $1.10 is too large, so try a smaller value).

Both system 1 and system 2 play crucial and complementary roles in human intelligence. System 1 provides good ideas far quicker than system 2 could, while system 2 can refine those ideas or correct them when they err.

This certainly corresponds to the subjective experience of expert games players. Skilled players report having an intuitive sense of what are good moves, and who is winning, but also that they spend time during games planning ahead through sequences of moves, explicitly and consciously applying the rules of the game.

Research can confirm this. Studies by De Groot [1946] showed that expert chess players are far superior at memorising a chess board than novices. Work by Chase and Simon [1973] found this effect was much stronger for positions that arose from actual games of chess. This is indicative of different pattern recognition processes in expert players - these players have refined heuristic understanding of chess positions that is aiding their memory, but with a much stronger effect on familiar positions they trained on.

De Groot [1946] also studied the search performed by chess players.

World-class players and strong amateur level players showed remarkably similar patterns in their search behaviour, searching to a similar depth and considering a similar number of positions. The differences were apparent in where they searched: grandmasters examined more relevant moves than the weaker players. It seems that their intuitive grasp of the game is what really differentiates the most skilful players of chess.

This is not to say that search is not important in human chess play, it just isn't what changes as players increase in skill. Chabris and Hearst [2003] analysed chess games from tournaments with different amounts of time allowed to play. They found that decreasing grandmaster players' thinking time from ≈ 3 minutes per move to ≈ 30 seconds per move increased their error rate by 36%.

As well as performing search to reduce errors in games, chess players train through intensive study, making heavy use of system 2 as they attempt to improve their intuitions to become more skilful players.

The techniques used by ExIt are in some ways analogous to the techniques people use to play these games:

The strengths and weaknesses of neural networks have much in common with system 1. They can be evaluated rapidly, and can provide good guesses of correct moves in positions never before seen. But they can also be fooled, particularly by the subtle changes in position that occur frequently in classical board games.

Planning methods such as minimax search or MCTS are analogous to system 2: deliberative, slow and rules-based, using both the rules of the game being played and rules governing the search direction. Their strength is underpinned by the strength of heuristics, whether hand-crafted or provided by neural networks.

In ExIt, heuristics are trained with the aid of a 'system 2'-like search. Better heuristics lead to stronger searches, completing a virtuous cycle of improved policies. Much as with expert chess players, the difference between a skilful ExIt player at the end of training and unskilled players at the start is the strength of their 'system 1'.

Chapter 4

Monte Carlo Tree Search Expert Iteration

In this chapter we present a practical implementation of Expert Iteration using Monte Carlo Tree Search experts and neural networks as apprentices, as first presented in Anthony et al. [2017]. Following the methods presented in this chapter, our program was able to outperform both MoHex and MoHex 2.0, having learnt to play *tabula rasa*.

We describe our methodology in sections 4.1, 4.2, 4.3. Section 4.1 covers the details of our neural network enhanced search algorithm, including different versions for different stages of training. Section 4.2 describes our imitation learning methods, including our novel method to produce policy targets from an MCTS expert. Section 4.3 discusses how datapoints are sampled, and how to efficiently generate the large datasets required by ExIt.

Section 4.4 presents the main results of this chapter, which study some of the methodological choices in the previous sections, and demonstrate that our MCTS-ExIt algorithm can learn to outperform previously state-of-the-art heuristic search algorithms MoHex and MoHex 2.0.

Section 4.5 compares the algorithm to the contemporaneous Alpha Zero algorithm, which also uses MCTS policies to train a neural network. Alpha Zero uses a much simplified Neural-MCTS algorithm, allowing us to directly compare an Expert Iteration algorithm to a Policy Iteration algorithm with

1-step greedy policy improvements, and examine the hypotheses of chapter 3.

4.1 Neural Monte Carlo Tree Search

In this chapter we use MCTS as our choice of expert. Both MCTS and $\alpha - \beta$ search algorithms have performed well in prior works on Hex, but MCTS can play games with no heuristic value function by employing random rollouts. This is useful for learning from scratch, since it gets us 'off the ground' much more quickly, imitating a higher level of play without resorting to specialised knowledge.[1]

We use several different versions of MCTS in our final algorithm, depending on the neural-network heuristics available. Initially, neither policy nor value functions are trained, meaning a classical MCTS algorithm - using random rollouts - is superior to the arbitrary heuristc functions. Less data was needed to train a policy network than a value network, so we also developed a version using only policy heuristics. Our final MCTS version used both a policy and a value neural network to perform its search.

4.1.1 Vanilla MCTS

Our vanilla MCTS algorithm follows standard methods as described in section 2.6.4. The tree policy of our agent uses the UCT formula:

$$\text{UCT}(s, a) = \hat{Q}(s, a) + c_b \sqrt{\frac{\log n(s)}{n(s, a)}}$$

When $n(s, a) = 0$, we define a 'first-play urgency' (FPU) [Gelly and Wang, 2006] value for $\text{UCT}(s, a) = \infty$. [2] The default policy is uniform across all actions, and RAVE is also used to reduce the variance of the value estimates $\hat{Q}$.

[1]How to learn in games where random rollouts do not provide any useful signal (for example, because they always result in a draw) is beyond the scope of this work.

[2]In Hex the branching factor is small enough that all actions can eventually be searched, which makes this choice appropriate in this case, in games with a much larger branching factor, a finite FPU or an alternative method such as Progressive Widening [Chaslot et al., 2008], could be needed.

4.1.2 Use of a Policy Network in MCTS

A heuristic policy function, such as the apprentice policy in ExIt, can be used to improve the performance of MCTS by providing fast evaluations of action plausibility at the start of search. We use this to focus our MCTS tree policy in favour of the plausible actions, and avoid searching actions that the policy function believes are weak.

Insofar as our neural network's policies have learnt to choose good actions, those actions they assign higher probability to will in general be stronger. We can therefore use our neural network policy to bias the MCTS tree policy towards those moves that we expect to be stronger, allowing it to begin searching from a better starting policy.

When a node is expanded, we evaluate the apprentice policy $\hat{\pi}$ at that state, and store it. We then modify the UCT formula by adding a bonus proportional to the probability mass of the action under the policy $\hat{\pi}(a|s)$:

$$\mathrm{UCT}_{\mathrm{P-NN}}(s,a) = \mathrm{UCT}(s,a) + w_a \frac{\hat{\pi}(a|s)}{n(s,a)+1}$$

where w_a is a hyperparameter that weights the neural network against the simulations, $\hat{\pi}(a|s)$ is the probability assigned to the action a by the apprentice policy, and $n(s,a)$ is the number of times action a has been searched so far. This formula is adapted from one found in Gelly and Silver [2007]. It initially follows the recommendations of the neural network, but as the iteration count increases, it approaches the normal UCT formula.

When using vanilla MCTS, with no prior knowledge, an infinite FPU makes sense, as no action can be preferred until some information has been gained about its strength. This is no longer the case, because we want to initially take only actions recommended by the policy network, so a finite FPU value of 12 was chosen. w_a was set to 100, which allows the searcher to search a move favoured by the prior network multiple times before searching other moves, allowing deeper searches. These values were chosen by assessing play strength (see section 4.1.4 for details).

Our use of the policy to bias our search has two beneficial effects. Firstly, it means that more simulations can be spent on determining which is the strongest of the plausible actions, while fewer simulations are spent on the weaker actions. This will reduce the effective search width, as initially plausible actions are searched several times before implausible actions are tried; only when many simulations have been made does the tree attain full-width. This allows for deeper searches focused on the most promising lines, but does not entirely discount any action to allow exploration and exploitation — both of knowledge provided by the neural network and knowledge gained by search — to be traded-off, this tradeoff is controlled by the MCTS hyperparameters.

Secondly, it means that the simulations made by the tree policy are of higher quality play, particularly in nodes with low visit counts. Note that the simulation policy of the search in one node is also the policy for which the parent node attains a Q-value estimate. Therefore having a good policy in the first simulations from a node also means that the node above is optimising actions assuming subsequent play is stronger, and therefore we require fewer simulations of a pair of actions to accurately determine which of two actions are superior under strong play (as opposed to determining which is superior under random play, as is initially estimated by vanilla MCTS).

The neural network's final layer uses a softmax output. Because there is no a priori reason to suppose that the optimal bonus in the UCT formula should be linear in the policy probability, we view the temperature of the network's output layer as a hyperparameter for the N-MCTS and tune it to maximise the performance of the N-MCTS in self-play.

Our MCTS program was able to simulate games faster than the neural network can be evaluated. In order to balance the loads of performing rollouts (on CPU) and evaluating the neural network (on GPU), we use an expansion threshold [Silver et al., 2016] of 1. This means that when an action $a|s$ is taken for the first time by the tree policy, we do not add a node to the tree, but instead immediately perform a rollout.

The second time we take action $a|s$, we add a new node for the resulting state s' and evaluate the neural network at s', blocking search while doing so. Once the network policy is evaluated, we choose the next move of this simulation with the tree policy at s' (this will be the preferred move of the prior). The rest of the simulation will follow the uniform default policy.

4.1.3 Use of a Value Network in MCTS

To use a value network V in the expert, we estimate $V(s_L)$ whenever a new node s_L is created. This value estimate is backed up through the tree to the root in the same way as rollout results are: each edge stores the average of all the network evaluations made in nodes in its subtree. Because an expansion threshold was used, not every simulation results in a network evaluation.

In the tree policy, the value is estimated as a weighted average of the network estimate and the rollout estimate, weighting the value network more strongly ($w_{net} = 0.75$, $w_{rollouts} = 0.25$).

4.1.4 MCTS Hyperparameters

MCTS hyperparameters were hand tuned, by a mostly informal inspection of results of self-play matches using prototype models. [3] Prior to training, parameters were chosen for vanilla MCTS, Neural-MCTS with only a policy network, and Neural-MCTS with a dual policy and value network; these parameters are common to all ExIt training runs.

Three parameters did not undergo hyper-parameter tuning, and chosen in a different way. Firstly, the number of search iterations. Longer searches are consistently stronger than shorter, but are also more expensive. The number of search iterations was chosen to allow large datasets to be generated within the available computation budget for Expert Iteration experiments.

Secondly, the expansion threshold. For P-MCTS and PV-MCTS, it was chosen to be the maximum value that did not result in the GPU idling. For Vanilla MCTS, adding a node to the search tree does not require a neural

[3] i.e. models from individual training runs during development

network evaluation, so is not costly, and expansion thresholds are not necessary to maintain fast searching. For retuned MCTS (see below), rollouts are not used, and all value estimates are derived from the neural network evaluation, so again there is no need to balance CPU and GPU load, and an expansion threshold is not required.

Thirdly, for vanilla search, the FPU value was set to ∞ and not tuned. For P-MCTS and PV-MCTS, the hyperparameter was hard-coded and not tuned. For the retuned MCTS this parameter was made configurable, and tuned in the way described below.

To tune the other hyperparameters, a first guess was made for reasonable values for all hyperparameters. A particular hyperparameter was then chosen to be tuned, keeping the other hyperparameter values fixed. Once this hyperparameter was tuned, another hyperparameter was chosen and tuned. Once all parameters were tuned, the process was repeated. Usually the second tuning of each parameter involved only very small changes to the values, and the third tuning resulted in no change.

The order hyperparameters were selected for tuning was not random: we tuned hyperparameters for which we were more uncertain of the correct value first. For example, when tuning P-MCTS, we were less confident of our inital guesses for the new hyperparameters (e.g. policy network weight w_a) than for hyperparameters that were used in vanilla search (e.g. c_{RAVE})

To tune a hyperparameter, a grid of possible values around the current guess is chosen for consideration. The admissible values for most of our parameters are non-negative real numbers (c_b, c_{RAVE}, FPU, policy net weight and policy net temperature). For these parameters, the grid would be chosen to span approximately 4 orders magnitude around the current value, spaced by multiplicative factors of around $\times 2$ to $\times 4$. The value of 0 was also considered. For example, when c_{RAVE} was tuned based on an initial guess of 100, then the grid used was $(0, 1, 3, 10, 30, 100, 300, 1000, 3000, 10000)$. For value network weight, the admissible values are the range $[0, 1]$, so we searched values

$(0, 0.1, 0.25, 0.5, 0.75, 0.9, 1.0)$.

Once the grid was set, it was searched based on assumption that performance was strictly unimodal in the parameter. That is, there was a single maximal value, and performance strictly decreased away from that value.

To perform the search, two parameters were chosen, at approximately 30th and 70th percentiles of the values under consideration, and a match (consisting of 162 games, see section 1.2.4) was played between them. In our example, this would be a match between $c_{RAVE} = 10$ against $c_{RAVE} = 1000$. If a particular parameter setting wins by a sufficient margin to pass a hypothesis test with 95% confidence level, then the other value, and all more extreme parameters would be discounted from the search — based on the assumption of unimodal performance — and we repeat the process on the smaller set of possible values. In our example, $c_{RAVE} = 10$ loses, so we continue searching with the values above 10: $(30, 100, 300, 1000, 3000, 10000)$.

When matches were inconclusive according to the hypothesis test, new matches were played first with twice as many games (324), and then with four times as many (648). If either of these resulted in a significant result, then that result was used in the same way as before.

If the extra matches were also inconclusive, then (given our assumption of strictly unimodal performance) there were two possibilities: either (1) this hyperparameter has only a small effect on overall performance, and subsequent tuning will also return inconclusive results and is not useful, or (2) the optimal parameter value lies in between these two values tested. To determine which, we ran a pair of matches between the values just tested and the intermediate value. If these are also inconclusive, then we are in case (1), and select this intermediate value. If either of these matches have statistically significant results, we use those results to reduce the space of considered parameters.

For example, when tuning c_b, suppose we are considering the values $(0, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5)$, and a match between $c_b = 0.025$ and $c_b = 0.5$ was inconclusive. 0.1 is halfway between 0.025 and 0.5 on our grid, so we play extra matches, between $c_b = 0.025$ and $c_b = 0.1$ and between $c_b = 0.1$ and $c_b = 0.5$. The first of these two extra matches is won by $c_b = 0.1$, while the second is inconclusive. We use this to shrink our search to the parameter values greater than 0.025, i.e. $(0.05, 0.1, 0.25, 0.5, 1.0, 2.5)$, and continue.

We tune our MCTS for two reasons, either to improve the strength of a search algorithm used by a final agent, or to choose the parameters for the expert in Expert Iteration. Particularly in the second case, the self-play win-rate measure is only a proxy of what we actually want to achieve. A particular possible problem is that for maximising win-rate of a single search, it might be best to never search certain actions. However, this is not something we want to have happen in our experts, as this would mean that action would never be explored again. We note that with 10,000 search iterations from the root node, the hyperparameters we found for P-MCTS and PV-MCTS have a large enough FPU and c_b, and small enough policy network weight w_a, that every action will be searched several times even if the neural network policy gives them 0 probability.

After the final, large-scale training run, new hyper-parameters were chosen based on the model output by that training run. This parameter setting results in significantly stronger play than those used during training, it is denoted 'retuned MCTS' in the table below. The biggest difference was that, with a fully trained value function, random rollouts should be dropped entirely, and value estimation comes entirely from the neural network value function (unlike for the much weaker prototype model, where rollouts still helped). Therefore the expansion threshold is set to 0, and the number of search iterations reduced to 2000 to maintain a similar time per move. We also found that a policy network temperature of 1.0 was now optimal, compared to the value of 0.1 that was optimal for the prototype networks. It might be because the fully trained network has learnt to give very low probabilities to weak actions, whereas the prototype networks, which were trained to imitate vanilla MCTS, still gave them moderately high probability. Finally, the previously hard-coded FPU

Setting	Vanilla	P-MCTS	PV-MCTS	Retuned MCTS
Iterations	10,000	10,000	10,000	2,000
Exploration c_b	0.25	0.05	0.05	0.05
c_{RAVE}	3000	3000	3000	N/A
FPU	∞	12	12	0
Expansion Threshold	0	1	1	0
Policy Net Weight w_a	N/A	100	100	25
Policy Net temperature	N/A	0.1	0.1	1.0
Value Network Weight	N/A	N/A	0.75	1.0

Table 4.1: Monte Carlo Tree Search parameters

value was tuned.

4.2 Imitating Monte Carlo Tree Search

4.2.1 Policy Targets

In prior work on imitating MCTS policies [Guo et al., 2014], the policy target used was simply the move chosen by MCTS; we will refer to this as *chosen-action targets*, (CAT), and optimise the Kullback Leibler (KL) divergence between the output distribution of the network and this target. So the loss at state s is given by the formula:

$$\mathcal{L}_{\text{CAT}} = -\log[\pi(a^*|s)]$$

where $a^* = \arg\max_a(n(s,a))$ is the move selected by MCTS.

In most imitation learning tasks, the only information known about the expert is the chosen action, for example because the expert moves come from a dataset of human play. In our case, we simulate our experts, so have more detailed information about how the MCTS chose its action. Can we take advantage of this to construct a more useful, informative target?

We propose an alternative target, which we call *tree-policy targets* (TPT). The tree policy target is the average tree policy of the MCTS at the root. In other words, we try to match the network output to the distribution over actions given by $n(s,a)/n(s)$ where s is the state we are scoring.[4] This gives the loss:

$$\mathcal{L}_{\text{TPT}} = -\sum_a \frac{n(s,a)}{n(s)} \log[\pi(a|s)]$$

Unlike CAT, TPT induces a cost-sensitive loss: when MCTS is less certain between two moves (because they are of similar strength), $\mathcal{L}_{\text{TPT}}$ penalises misclassifications less severely. On the other hand, it will relatively more severely punish misclassifications choosing very poor moves. When an agent trained using a cost-sensitive target does misclassify, it will still tend to select stronger actions. CAT offers no training signal to select stronger actions in the event of misclassifications.

Cost-sensitivity is a desirable property for an imitation learning target, as it induces the IL agent to trade off accuracy on less important decisions for greater accuracy on critical decisions. Prior work has used expected cost as a target to achieve cost-sensitivity (e.g. Scherrer et al. [2015]), but as discussed in section 3.2.1, a cross entropy target will usually perform better empirically. We found this to be true in our setting; tests where we trained our networks to maximise the expected value, or to minimise a mean square error of policy prediction, resulted in much weaker policies than cross-entropy/log-loss.

In ExIt, there is additional motivation for such cost-sensitive targets, as our networks will be used to bias future searches. Accurate evaluations of the relative strength of actions not taken by the current expert are still important, since future experts will use the evaluations of all available moves to guide their search.

4.2.2 Value Targets

The guiding principle of Expert Iteration is to use search methods to create the best quality data possible for training neural networks. Ideally, therefore, we would construct value targets to estimate $V^{\pi^*}(s)$, training a value function directly from the expert's play. However, we found that to train a

[4] $n(s)$ and $n(s,a)$ are defined in section 2.6.4

value function without severe overfitting requires approximately 10^5 independent samples. Playing this many expert games was not feasible, so instead we approximate $V^{\pi^*}(s)$ with the value function of the apprentice, $V^{\hat{\pi}}(s)$. Monte Carlo estimates of $V^{\hat{\pi}}(s)$ are much cheaper to produce, as they only require playing out a game following the cheaper apprentice policy, starting from state s.

For our exploitation policy π^{exploit}, we use greedy action selection from our policy network, resulting in the strongest play the network is capable of, and substantially improving performance compared to using actions sampled from the network with a temperature of 1, as done by π^{sampling}. This means that, in the future, search will aim to reach states where best play is strong, rather than being cautious due to mistakes made by a deliberately exploratory policy.

Hex has two game outcomes: winning and losing, resulting in returns of 1 and 0 respectively. The value of a state $V(s)$ is therefore the probability of winning the game from state s, and value learning can be treated as a binary classification problem. So, to train the value network, we use a KL loss between a Bernoulli distribution with probability $V(s)$ and the sampled (binary) result z. [5]

$$\mathcal{L}_{\text{V}} = -z \log[V(s)] - (1-z)\log[1-V(s)]$$

4.2.3 Optimisation and Early Stopping

At each training step, a mini batch of 250 samples is randomly sampled from the training dataset. The Adam optimiser [Kingma and Ba, 2015] is used to update network parameters. We employ early stopping to avoid overfitting. The early stopping point is the first epoch after which the validation errors increase 3 times consecutively. By an epoch we mean one iteration over each data point in the data set. We then use the checkpoint which had lowest loss on our validation set.

[5] This loss function implicitly relies on the win-loss property of Hex. In later experiments we successfully used the more generally applicable mean square error loss.

In order to continue using early stopping, training must be started from scratch each Expert Iteration. To reduce the cost of training from scratch each epoch, a maximum of 7 epochs was used. Because the ADAM optimiser is able to train the neural network rapidly, restricting the duration of training in this way was not found to hurt performance: in testing, when the early stopping point was later than the 7th epoch, matches between the network at the early stopping point and the 7th epoch network did not consistently favour either stopping point.

4.2.4 Network Architecture

Our network architecture and input are designed to give the network basic information about the game rules, such as the board topology, win condition and legal moves, following previous work in deep learning for Hex [Young et al., 2016]. The network uses standard methods from the deep learning toolkit, and is not itself a novel contribution of this work.

Input Features. We use the same state representation as Young et al. [2016]: the two-dimensional state of 9×9 Hex board is extended to a 6 channel input. The 6 channels represent: black stone locations, white stone locations, black stones connected to the North edge, black stones connected to the South edge, white stones connected to the West edge and white stones connected to the East edge. [6]

Encoding this information about the winning condition was found to give a small but consistent benefit over a less informative input using only stone locations, particularly in terms of requiring less training data. Encoding game rules did not require *strategy knowledge*, but could still inhibit generality if other domains have difficult to encode rules. We do not believe these features have contributed significantly to our final performance, as in chapter 5 we successfully train agents using input that only describes stone locations.

As in Young et al. [2016], we pad the network input to represent the edge of the board. To do this we expand the board, adding two extra rows or

[6]These features only relate to literal connections, and do not consider virtual connections

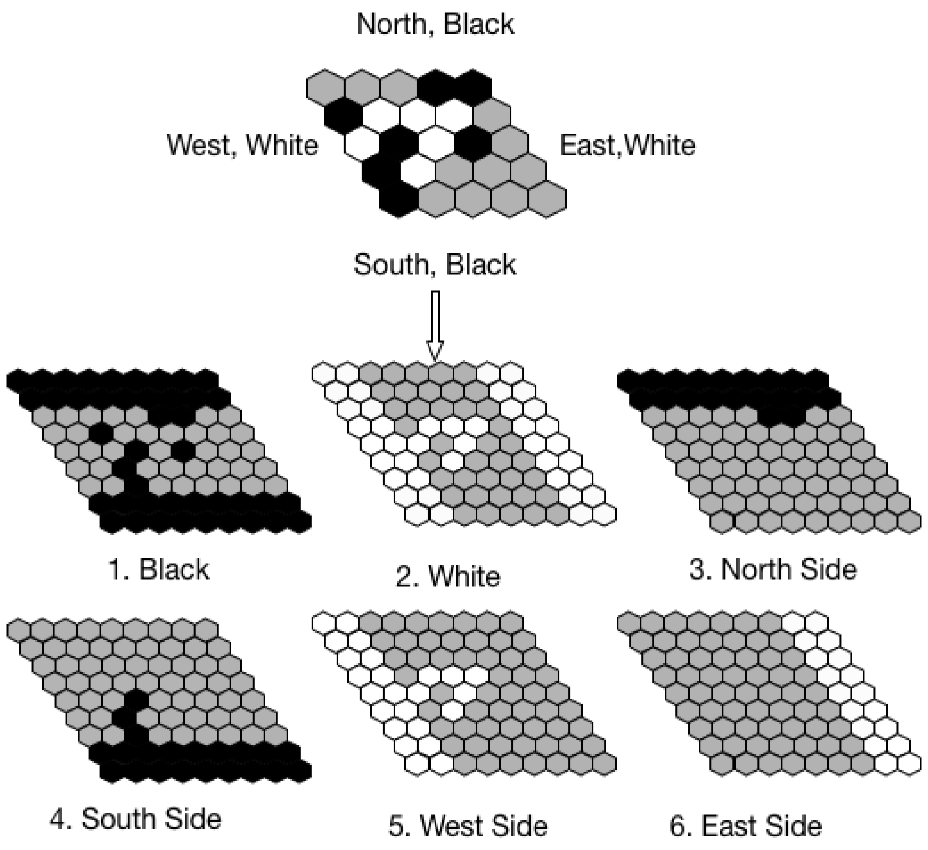

Figure 4.1: The 6-channel encoding used as neural network input

columns to each side. On the extra cells thus created, we add dummy stones: black stones along the North and South edges, and white stones along the East and West edges. In each corner of the padding, we 'place both a black and a white stone'. The resultant encoding of the board is shown in figure 4.1.

Playing Hex on this expanded board, with the dummy stones providing connections in the same way as stones played by players, does not change the game, but it means that the orientation of the board is made clear from the network input, and that convolutions centred at the edge of the board have more meaningful input than would be provided by zero-padding these cells.

Neural network architecture. We illustrate our neural network architecture in figures 4.2 and 4.3.

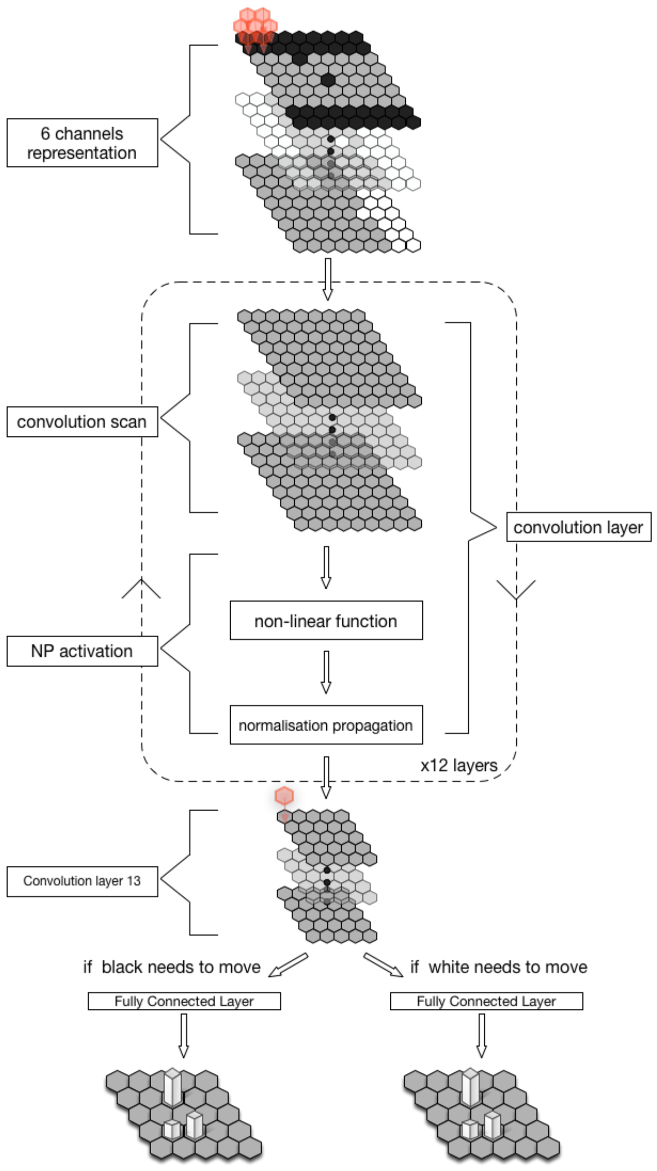

Figure 4.2: The policy network architecture

Our network has 13 convolution layers followed by 2 parallel fully connected softmax output layers.

The parallel softmax outputs represent the move probabilities if it is black to move, and the move probabilities if it is white to move. Before applying the softmax function, a mask is used to remove those moves which are invalid in the current state (i.e. those cells that already have a stone in them).

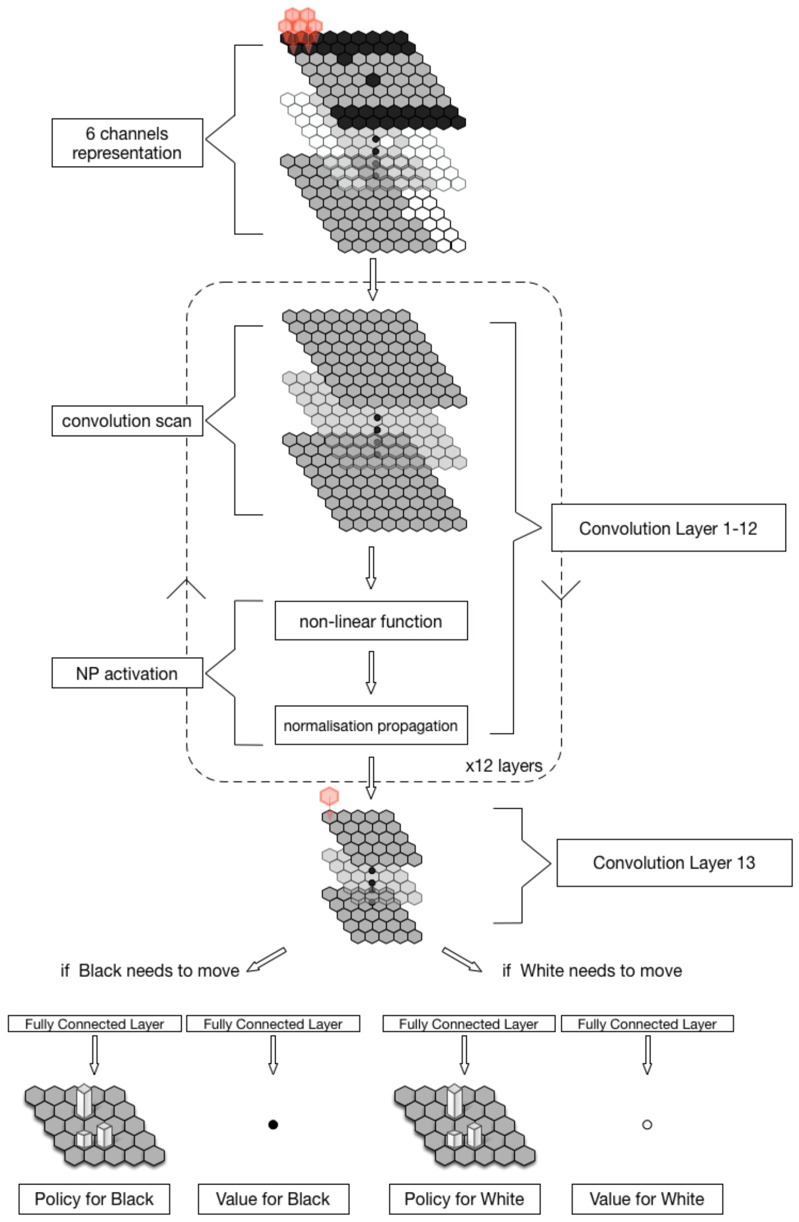

Figure 4.3: The policy and value network architecture

When also predicting state value, we add two more outputs, each outputting a scalar with a sigmoid non-linearity. They estimate the winning probability, depending on which player is next to move. This multitask network removes the need to evaluate two separate networks during the time-sensitive search and also regularises value prediction. Training losses for policy and value prediction are summed to give an overall loss. We considered weighting the two losses in a ratio other than 1:1. This was only useful for training from very small datasets, and, even then, only conferred a small benefit.

Because the board topology is a hexagonal grid, we use hexagonal filters for our convolutional layers. A 3×3 hexagonal filter centred on a cell covers that cell, and the 6 adjacent cells. This is implemented using a square filter, with top left and bottom right weights set to 0. This may actually have worsened performance compared to square filters, as it reduces the networks ability to share information along the long diagonal, so we dropped this for later experiments (in chapter 5).

In convolution layers 1-8 and layer 12, the layer input is first zero padded and then convolved with 64 3×3 hexagonal filters with stride 1. Thus the shape of the layer's output is the same as its input. Layers 9 and 10 do not pad their input, and layers 11 and 13 do not pad, and use 1×1 filters.

Our convolution layers use exponential linear unit (ELU) [Clevert et al., 2016] nonlinearities. Separate biases are used in each location for all convolution layers and normalisation propagation [Arpit et al., 2016] is applied in layers 1-12.

4.3 Dataset Generation Methods

4.3.1 Dataset Generation in Expert Iteration

Access to an environment simulator gives freedom over which states to consider for training and cost-intensive search. Each datapoint is costly, requiring a full search and 1000s of neural network evaluations, and correlations between states searched might reduce the usefulness of the data from search. We therefore chose to ensure that each datapoint came from an independently sampled game.

First, a game was played using the apprentice network, with a policy temperature of 1. From this, a single state is randomly sampled. A 10,000 iteration search was performed from this state to create policy targets, and a new apprentice network game is played with a temperature of 0 to form the value target.

10,000 iteration searches typically use ≈ 2000 network evaluations, while the network self-play used < 100 network evaluations. So the overhead from generating independent states compared to performing searches is small. A cost is incurred because we cannot reuse partial search trees when all states are decorrelated. If we searched multiple states from the same game, we could reuse part of one search for the search in the next state.

One could use internal nodes from the search tree as additional datapoints, as in the treestrap algorithm, we do not explore the possibility in this thesis.

Because the testing procedure is to play one game per legal opening move, we generate equal amounts of data for each legal first move. To reduce the cost of data generation, data was augmented by rotating each position by 180 degrees; this results in an equivalent state in Hex. This technique has been used in previous Hex agents, for example NeuroHex [Young et al., 2016].

Algorithm 3 Sample Data Point (with Greedy Apprentice Value Targets)

```
Require: $\pi$, $\pi^*$                                   ▷ Apprentice and Expert Policies
 1: $s \leftarrow$ initial_game_state()          ▷ For Hex, start after a random first move
 2: $\tau \leftarrow []$
 3: while $s$ is not terminal do
 4:     $\tau$.push_back($s$)
 5:     $a \leftarrow$ sample($\pi(a|s)$, temp $= 1$)        ▷ Sample action from apprentice
 6:     $s \leftarrow$ next_state($s, a$)
 7: end while
 8: $s \leftarrow$ uniform_sample($\tau$)                   ▷ Sample single state from game
 9: policy_target $\leftarrow \pi^*(a|s)$
10: $s' \leftarrow s$
11: while $s'$ is not terminal do
12:     $a \leftarrow$ sample($\pi(a|s')$, temp $= 0$)       ▷ Greedy action from apprentice
13:     $s' \leftarrow$ next_state($s', a$)
14: end while
15: value_target $\leftarrow s'$.return_player_one
16: return ($s$, policy_target, value_target)
```

In algorithm 3, lines 3-7 play a game using the apprentice policy at a high temperature. Line 8 selects a single state from that game, and line 9 produces the expert policy target. Lines 10-14 play out the game according the the greedy version of the apprentice policy, the outcome of which gives us the value target (line 15).

4.3.2 Dataset Re-use

In order to successfully imitate a complicated expert policy, it is necessary to generate a dataset of many samples from it. Furthermore, because each sample requires calculating a move via a computationally intensive expert, compared to sampling experience for model-free algorithms, generating data for ExIt accounts for a greater proportion of the total computation cost than in model-free algorithms.

To make most use of the expensive data, rather than creating a dataset anew each iteration, it is more efficient to retain data between iterations. Although data from old experts represents weaker play than more recent data, it remains stronger than the apprentice for some time. Aggregating datasets across multiple iterations allows improved heuristics to be used for expert improvement sooner, we call this online ExIt. It can be seen as applying the DAgger procedure to the case of Expert Iteration.

Because we terminate our supervised learning after between 4 and 7 epochs (depending on the behaviour of early stopping), and each epoch is a single pass through all our training data, the length of time taken by the supervised learning step is proportional to the dataset size. In online ExIt, the dataset grows on each iteration, potentially meaning the supervised learning takes longer and longer. If a fixed amount of data was added in each iteration and data never discarded, the supervised learning step would eventually dominate run time, because each iteration generates the same amount of data, but network retraining is increasing each iteration without limit.

We test two designs of online ExIt that avoid this phenomenon. In the first, we create fixed number of moves each iteration, and train on the most recent k iterations. In the second, we use all our data in each training step, and expand the size of the dataset by a constant factor each iteration. We refer to these as ‘buffer’ and ‘exponential’ methods, respectively.

4.3.3 Parallelism

The sampling of data is trivial to parallelise. Each search is relatively slow, taking ≈ 0.3 seconds on our hardware, and the imitation learning targets to be communicated after a search are small, measuring approximately 1KB. This means it is easy to efficiently deploy large amounts of distributed resource as multiple data-generation workers. In our largest experiment we do this, with supervised learning being performed parallel to data generation on a (single) separate machine. Communication of network parameters can be infrequent, as the large expert improvements mean that data from old experts remains relevant for a considerable amount of time.

When using neural networks in MCTS programs there is a trade-off when having multiple search threads run simultaneously. Using more search threads improves wall clock time per search iteration and allows for larger, more efficient batch calculations of the neural network on GPU accelerators. On the other hand, large batches mean that search iterations are run with less information than would be available if each search was sequential, meaning less important lines of play might be searched. This issue can be partially remedied by using virtual losses [Segal, 2010], but it still can mean that iteration-for-iteration, highly parallelised searches are weaker [Sephton et al., 2014].

In ExIt, unlike the case of playing a single match, many searches are run from different states. These can be done simultaneously allowing for batching between different searches, thereby gaining the efficiency benefit of large calculation batches, while having each individual search's simulations run entirely in series.

4.4 Empirical Results

4.4.1 Tree Policy Targets

We tested our imitation learning procedure on data generated with vanilla MCTS. With no policy network to sample the state set, states were instead sampled by playing games using vanilla MCTS with the number of search iterations per move reduced to 1,000. The reduction in search iterations reduces the cost of sampling the dataset, and can promote a greater diversity of states, as shorter searches are more stochastic.

We independently sampled 66 datasets of 100,440 states in this way (1240 per legal first move). Policies were trained using CAT and TPT on each of these datasets, with 10% of the data held back as a validation set for early stopping, using the CAT or TPT loss as the measure for the early stopping point. For each of the 66 datasets, an additional 10,044 datapoints were created as a test set (meaning each independent experiment here also had an independent test set).

The accuracies of TPT and CAT networks were similar. TPT had a top-1 accuracy of $47.7\% \pm 0.1$, compared to $46.9\% \pm 0.1$ for CAT. Top-3 accuracies were $65.7\% \pm 0.1$ and $65.3\% \pm 0.1$ for TPT and CAT respectively. (Margins given are 95% confidence intervals across the 66 experiments, see appendix A for details)

To compare the strengths of TPT and CAT networks, we played a round robin tournament between the 132 networks, i.e. each pair of networks plays a 162 game match against each other network. [7] CAT and TPT may induce different temperatures in the trained networks' policies. During play, each network greedily selected the most likely move to equalise the different temperatures from training. This selection procedure was also stronger for either type of network than using any positive temperature for the softmax. We calculated Elo scores for each network using BayesElo [Coulom, 2005].

In matches between the networks, the TPT network is 50 ± 13 **Elo stronger than the CAT network trained on the same data**, despite their apparently similar accuracies in move prediction. (Margin is the 95% confidence intervals, see appendix A.) This suggests that the cost-awareness of TPT indeed gives a performance improvement.

When using a training target such as CAT, which is not cost-sensitive, we

[7]See section 1.2.4 for a full description of the matches.

expect prediction errors to be more damaging to performance than with TPT. This indeed seems to be the case: we found that, **for networks trained with CAT, top-1 prediction accuracy correlated with network strength, as measured by Elo ratings** ($n = 66$, $\beta = 0.30$, $p = 0.012$)**. For networks trained with TPT, no correlation was found** ($n = 66$, $\beta = -0.065$, $p = 0.61$)**. The difference between the correlations is significant** ($p = 0.0384$)

We continued training one of the TPT networks (randomly chosen) with the DAgger algorithm, iteratively creating 3 more batches of 100,440 moves. This additional data resulted in an improvement of 120 Elo over the first TPT network. Our final DAgger TPT network achieved similar performance to the MCTS it was trained to emulate, winning just over half of games played between them (87/162).

4.4.2 Policy-Enhanced MCTS

Our policy enhanced MCTS was much stronger than a vanilla MCTS: **N-MCTS using the final network of section 4.4.1 beats our vanilla MCTS, in 97% of games.** The neural network evaluations caused a factor of 2 slowdown in search. For comparison, a doubling of the number of iterations of the vanilla MCTS to 20,000 results in a win rate of just 56% against 10,000 iteration MCTS.

We compared TPT and CAT networks as policy networks for MCTS, using the same 132 networks compared in section 4.4.1. To do this, we played paired matches between TPT and CAT networks trained on the same dataset, and estimated the difference in Elo from this match.[8] Under this setting, **TPT networks were again superior to the CAT networks, by 42 ± 11 Elo, a similar margin to that found with the networks playing directly.**

We measured the effect of a prior policy on behaviour of the search by looking at the average number of nodes created at each depth. A search was

[8]This structure requires many fewer matches than the round robin used in section 4.4.1, necessary since matches between MCTS players are far more expensive than policy network matches

performed from every Hex state resulting from a single opening move. To keep the number of nodes constant and allow like-for-like comparison between the different searches, we used an expansion threshold of 0 for all search algorithms, otherwise parameters are identical to those used elsewhere.

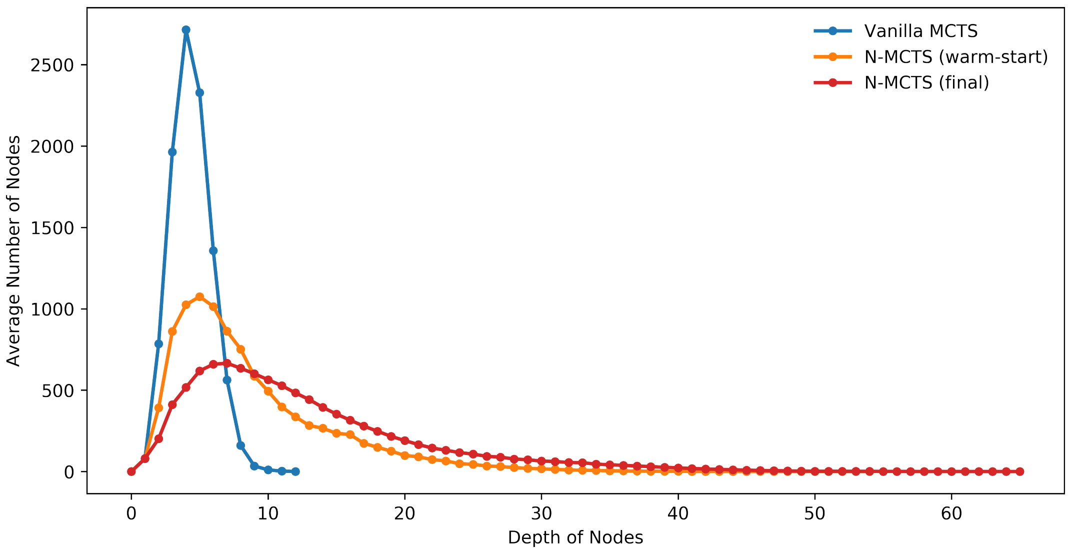

Figure 4.4: Average number of nodes at different search depths without a policy network, and with two different policy networks. Each line represents the behaviour of search with a particular neural network, averaged over searches from 162 positions. The plot shows that the policy networks result in substantially deeper, but narrow searches, suggesting that they are prioritising search along lines of play recommended by the neural network. Compared to the initial policy network (orange), the final policy network makes more confident predictions, and results in an even deeper search. The deeper searches are also the strongest Hex players. Each line represents data from a single network; while this behaviour is as expected, retraining might lead to different results.

We can see in figure 4.4 that the initial vanilla MCTS has by far the shallowest (and therefore widest) search tree. The **neural network trained to imitate this search algorithm is able to direct a much deeper search**, while using the strongest network from section 4.4.3 results in even deeper searches. In each case, the deeper searches result in play far stronger than the shallower searches.

4.4.3 Policy-Only Expert Iteration

Our goal is not just to improve the quality of our MCTS once, but to repeatedly retrain the neural network using a succession of improved experts. We do this here using the policy network enhanced MCTS. We also compare batch and online versions of ExIt.

For batch ExIt, we perform 3 training iterations, each time creating an entirely new dataset of 243,000 moves. In the buffer version, we create 24,300 moves each iteration and train on the last 10 iterations. In the exponential version, we start by creating a dataset of 24,300 moves, and grow this by 10% each iteration.

We compare ExIt to the policy gradient algorithm used by Silver et al. [2016], which achieved state-of-the-art performance for a neural network player in the related board game Go.[9] In Silver et al. [2016], the algorithm was initialised by a network trained to predict human expert moves from a corpus of 30 million states, and then Reinforce [Williams, 1992] was used. We initialise with the best network from section 4.4.1. Such a scheme, Imitation Learning initialisation followed by Reinforcement Learning improvement, is a common approach when known experts are not sufficiently strong.

For this experiment we did not use any value networks, so that network architectures between the policy gradient and ExIt are identical. This limits ExIt, as without a value function and with uniform random rollouts, it cannot converge to an optimal policy. All policy networks are warm-started to the best network from section 4.4.1.

As before, we measure performance by the strength of the neural network playing greedily. Elo ratings for the greedy networks are estimated with the BayesElo program, using data from a round robin tournament of all checkpoints from all training runs. See Appendix A for details.

ExIt learns far stronger policies than Reinforce. As can be seen in figure 4.5, ExIt also shows no sign of instability: the policy improves monoton-

[9]It has since been outperformed by an ExIt-style algorithm (see section 4.5), but remains the strongest model-free RL agent in Go.

ically on each iteration and there is little variation in the performance between each training run. In contrast, the Reinforce algorithm displays much more variation in performance, with highly non-monotonic progression in rating. This agrees with findings in other domains that policy gradient methods can give inconsistent results between training runs [Henderson et al., 2018], possibly as a result of the high variance of gradient estimate [Mohamed et al., 2020], or due instabilities particular to self-play in games [Silver et al., 2016, Balduzzi et al., 2018]. The exceptional consistency of our ExIt algorithm suggests that the benefits of the approach hypothesised in chapter 3 are being realised by this algorithm.

The ExIt algorithms explored in this thesis are very expensive, and our subsequent experiments will generally be at a larger scale than this one, so this experiment is the only instance where we repeat an experiment multiple times with different random seeds. It is important therefore that performance of ExIt algorithms appears to be very consistent between training runs, as our subsequent findings are supported only insofar as this is true.

Online Expert Iteration substantially outperforms the batch mode, as expected. Compared to the 'buffer' version, the 'exponential' dataset version appears to have learned marginally faster. The difference appears to have emerged during the period when training was cheap, and the exponential version retrained the network more frequently. It appears that the long run behaviour of both will be similar, however for training runs where relatively little data is generated in total, it may be preferable to use the version that keeps all data.

Training curves for online ExIt show a small dip at the start. This is because the first network retraining is based on a small but high quality dataset, whereas the initial network was trained with a much larger dataset of vanilla MCTS. It may have been better to initialise the dataset with some vanilla MCTS data, or to collect more data before retraining the networks.

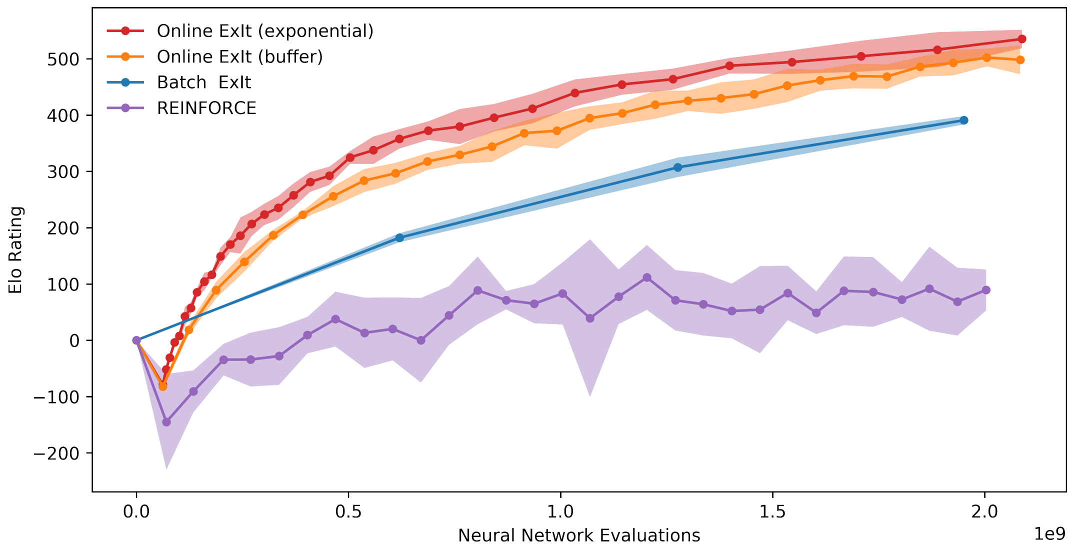

Figure 4.5: Elos throughout training of policy only MCTS-ExIt algorithms and Reinforce. Lines represent the means across five independent random seeds, shaded regions represent 95% confidence intervals. The x-axis counts the number of times a neural network is evaluated; because all methods use the same network architecture, this gives a comparable measure of computation cost. Expert Iteration methods all outperform REINFORCE, with online ExIt methods performing best. Whereas REINFORCE shows inconsistent performance improvements and a high variance of outcomes between runs, expert iteration methods learn consistently both through a single training run, and between different random seeds.

4.4.4 Policy and Value Expert Iteration

To train value networks we needed a larger dataset than for the policy network. When training value networks with smaller datasets, clear signs of overfitting were seen. These overfitted value networks proved detrimental to performance of MCTS in self-play games, compared to using random rollouts and RAVE.

To avoid these issues, we started by running online ExIt using only a policy network until our dataset contained $\approx 550,000$ states, which is comfortably large enough for value learning. We then used the most recent apprentice to add value estimates, using a greedy self-play game from each of the states in our dataset. We trained a combined policy and value apprentice on this dataset, giving a substantial improvement in the quality of expert play.

From a network trained on this data, we then ran ExIt with a combined value and policy network, creating another $\approx 7,400,000$ move choices. This was still small enough that all data could be held in memory, so we followed the 'exponential' version of MCTS that doesn't discard any data.

For comparison, we continued the training run without using value estimation for equal time. We then ran a tournament (see appendix A.1.3) to estimate Elo scores for all agents.

Our results are shown in figure 4.6, which shows that value-and-policy-ExIt significantly outperforms policy-only-ExIt. In particular, the improved plans from the better expert soon manifested in a stronger apprentice.

We also see an increased rate of improvement of the expert when the expert provides a larger improvement over the apprentice policy. This provides evidence for the hypothesis of section 3.2.3 that improvements can be tested via search self-play, rather than by running the full RL pipeline.

We can also clearly see the necessity of expert improvement, with later apprentices comfortably outperforming experts from earlier in training.

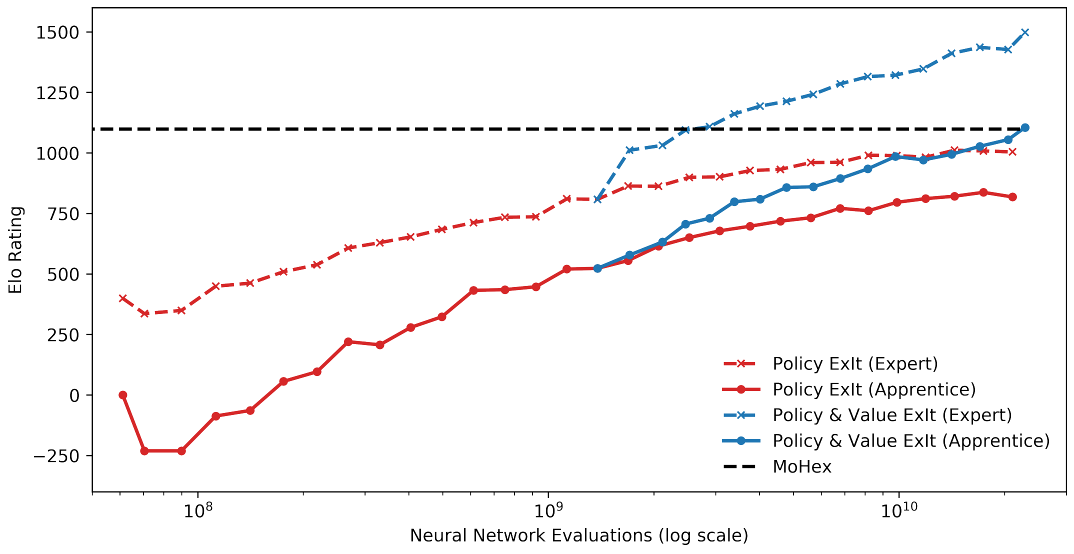

Figure 4.6: Elo ratings of apprentices and experts in distributed online ExIt, with and without neural network value estimation. MoHex 1.0's rating (10,000 iterations per move) is shown by the black dashed line. The figure demonstrates a clear advantage to learning value functions, with the stronger experts giving rise to stronger apprentices later on in training. For both experiments, the final apprentices outperform the initial experts. Data is from a single training run with each method.

ExIt Setting	s/move	ExIt win rate	MoHex Setting	Solver	s/move
10^4 iterations	≈ 0.3	$75.3\% \pm 6.6$	10^4 iterations	No	≈ 0.2
10^4 iterations	≈ 0.3	$59.3\% \pm 7.5$	10^5 iterations	No	≈ 2
10^4 iterations	≈ 0.3	$55.6\% \pm 7.6$	$4s$/move	Yes	4

Table 4.2: Results of matches between ExIt and MoHex 1.0, with 95% confidence intervals

4.4.5 Performance against MoHex

We compared our final MCTS-ExIt agent to the most recent available version of MoHex (as of 2017), MoHex 1.0 [Arneson et al., 2010].

To fairly compare MoHex to our experts with equal wall-clock times is difficult, as the relative speeds of the algorithms are hardware dependent: MoHex's theorem prover and search make heavy use of the CPU, while not utilising the GPU, whereas for our experts, the GPU is the bottleneck. On our machine MoHex is approximately 50% faster.[10] We tested ExIt against 10,000 iteration-MoHex on default settings, 100,000 iteration-MoHex, and against 4 second per move-MoHex (with parallel solver switched on).

Each match consisted to 162 games, following the procedure described in section 1.2.4. **ExIt won all matches using just 10,000 iterations per move, results in table 4.2**.

In 2018, MoHex 2.0 was made publicly available for benchmarking.[11] MoHex 2.0 contains multiple improvements over MoHex 1.0, and defeats MoHex 1.0 in 74.7% of 9×9 games head to head [Gao et al., 2018a]. This is a similar win rate to our program. But in head-to-head matches, **ExIt defeats MoHex 2.0 on its default settings (see table 4.3), even with 10 times fewer search iterations**. On our machine, MoHex 2.0 was slower per simulation than both MoHex 1.0 and our program, likely due to the pattern-based rollout policy that is more expensive to calculate.

[10]This machine has an Intel Xeon E5-1620 and nVidia Titan X (Maxwell), our tree search takes 0.3 seconds for 10,000 iterations, while MoHex takes 0.2 seconds for 10,000 iterations, with multithreading.

[11]github.com/cgao3/benzene-vanilla-cmake

ExIt Setting	s/move	ExIt win rate	MoHex 2.0 Setting	s/move
10^4 iterations	≈ 0.3	$63.6\% \pm 7.4$	10^4 iterations	≈ 0.6
10^4 iterations	≈ 0.3	$51.2\% \pm 7.6$	10^5 iterations	≈ 6.7

Table 4.3: Results of matches between ExIt and MoHex 2.0, with 95% confidence intervals

ExIt Setting	s/move	ExIt win rate	MoHex 2.0 Setting	s/move
2000 retuned	≈ 0.3	$73.5\% \pm 6.8$	10^4 iterations	≈ 0.6
2000 retuned	≈ 0.3	$58.6\% \pm 7.5$	10^5 iterations	≈ 6.7

Table 4.4: Results of matches between ExIt with retuned hyperparameters and MoHex 2.0, with 95% confidence intervals

Parameters for our training experts were chosen based on a prototype network trained for a much shorter time than our final network. By retuning the MCTS parameters with the final network, performance could be improved. The main difference in parameters is that, with the stronger value function, random rollouts should be dropped entirely. Therefore the expansion threshold is set to 0, and the number of search iterations reduced to 2000 to maintain a similar time per move. The match results are in table 4.4, they show that **returned parameters increased in ExIt's win rate vs MoHex 2.0**

4.5 AlphaZero

AlphaGo Zero and AlphaZero [Silver et al., 2017, 2018] are contemporaneous algorithms which use similar techniques to this work. Among the contributions of AlphaGo Zero was an RL algorithm that trains policy networks to imitate the outcome of a MCTS algorithm, so we understand it as an instance of ExIt. AlphaZero achieved state-of-the-art performance in Chess, Go and Shogi with a general learning algorithm.

4.5.1 Some Differences from this Work

Both AlphaZero and this work use tree policy targets for training the apprentice policy. The policies π^{sampling} and π^{exploit} of AlphaZero differ from what we used in this chapter. For these, AlphaZero used self-play of the expert policy, not of the apprentice policy.

Using the expert makes generating independent samples considerably more expensive, and AlphaZero does not decorrelate its dataset as we did. The advantage is that these policies focus the search on a state distribution of stronger play and provide value targets for a much stronger policy. Early in training, using expert games for value targets likely slows learning, since obtaining enough data for value learning is the bottleneck in both ExIt algorithms, and searches are less informative if the value function is weak.

Near convergence, however, using expert self-play games becomes the better option. This is because, after a while, improvements from training become relatively gradual, and very old experts remain stronger than the current apprentice. As such, a large dataset of old expert games is better than a cheaper to generate, 'fresher' dataset of apprentice games. It is unclear where the crossover point lies between apprentice and expert games.

Because the expert, sampling and exploitation policies are based on the same search in AlphaZero, computation was shared between them. Using the same policy for state sampling and value prediction introduces a trade-off between either exploring more diverse sets of states or ensuring accurate play during value prediction. To manage this trade-off, AlphaZero played with a policy temperature of 1 (i.e. the search distribution) for the first 30 moves of the game (as in our π^{sampling}), and switched to playing greedily thereafter (as in our π^{exploit}).

When creating training data, AlphaGo Zero adds Dirichlet noise to the prior policy at the root node of the search. This guarantees that all moves have a chance of being tried by the tree search, and no actions are excluded entirely from search. We did not use this idea, but our UCT formula includes an exploration term that guarantees every move at the root will be attempted at least once every search, meaning that the expert policy will never completely disregard a move.

Compared to the CNN used in this chapter, AlphaZero used much larger residual neural networks, this architecture achieved state-of-the-art performance in imitating human play in Go.

19×19 Go is a much larger game than 9×9 Hex. Correspondingly, the scale of the experiments in Silver et al. [2017] was much greater than presented in this work. AlphaGo Zero used approximately 4 orders of magnitude more compute in training, split between the cost of the larger network and making a larger number of inference calls to that network, since much more training data was generated.

AlphaZero's MCTS exclusively uses the neural network value function to evaluate simulations. Rather than transitioning to a rollout phase, the value function is called at the leaf node of the tree, and this estimate is backed up through the tree.

In contrast, this work uses different search algorithms at different stages in training. By using traditional rollouts at the start of training, our initial expert play is much stronger, and avoids wasting computation on inference calls to untrained networks. This is particularly valuable for our rather short training runs, but comes at the cost of an increased number of hyperparameters and code complexity.

4.5.2 Reimplementation of AlphaZero for Hex

We re-implemented AlphaZero [Silver et al., 2018], and applied it to Hex. We summarise the details of this here. Hyperparameters used are the same as reported by Silver et al. [2018], except for the Dirichlet noise parameter that varied between games in that work. We chose a value between the values used in Silver et al. [2018] as 9×9 Hex has an average branching greater than Chess, but less than 19×19 Go.

Data is generated via self-play of the expert search algorithm. Whenever a game is finished, all states s_i from the game are added to a replay buffer, with the game result z and expert search policy $p(s_i, a) = n(s_i, a)/n(s_i)$ for each state. Asynchronously, the neural network is trained on the data in the replay buffer.

For the first 30 moves of each self-play game, the expert takes actions according to the search distribution, i.e. $\pi_{expert}(s, a) = n(s, a)/n(s)$. Thereafter actions are chosen greedily. At the root node, Dirichlet noise is added to the prior policy in the PUCT formula: $\pi'(s_0, a) = 0.75\pi(s_0, a) + 0.25\eta$, $\eta \sim Dir(0.12)$.

In 90% of games, resignation is used: if the average return of search simulations from the current state is below a resignation threshold, the game is resigned. In the other 10% of games no resignation is used, these games are used to calculate a threshold for each player with a false positive rate below 5%.

The replay buffer stored the 10,000,000 most recent states. The neural network was trained with a batch size of 1024, optimised with a fixed learning rate of 0.01. We used a momentum optimizer with a momentum parameter of 0.9. A cross-entropy loss was used for optimising the policy, mean square error for optimising the value function, and an $\mathcal{L}^2$ parameter regulariser with weight 10^{-4} was used.

By choosing the ratio of data-generation workers (performing search) to learners (calculating network updates on the data), we can approximately control the number of times a datapoint is sampled from the replay buffer (on average) before 10,000,000 new datapoints are added and it is evicted. We chose numbers of workers of each type to ensure that on average data was sampled ≈ 5 times by the learners.

The network architecture used was a residual neural network with 256 channels and 19 blocks, with batch normalisation used throughout. Policy and value heads with the same architecture as AlphaZero were used. For states with white to move, input was transformed to an equivalent state with black to move. The input consisted of 3 planes: white stones, black stones and a plane representing whether white or black was to move.

The PUCT formula was used in MCTS:

$$\text{PUCT}(s,a) = Q(s,a) + c_{\text{puct}}\pi(a|s)\frac{\sqrt{n(s)}}{1+n(s,a)}, \text{ with } c_{\text{puct}} = 5$$

4.5.3 AlphaZero Replication Results

We ran AlphaZero on a much larger compute cluster than our previous runs in this chapter, and achieved higher final performance. In head-to-head games with equal wall clock time, AlphaZero's search reached parity to the final expert from section 4.4.4 after approximately 43 million self-play moves, just under 6 times as many searches as performed in our longest run. The total training run used approximately 18 times the compute used in section 4.4.4, equivalent to 100 days on an nVidia 1080Ti GPU.

The slower-per-floating point operation learning of AlphaZero is not surprising: it was designed with a larger scale in mind. It can be attributed to a combination of the larger neural network, not using vanilla MCTS at the start of training and, at least initially, using more expensive search-intensive self-play games for value estimation.

4.5.4 Comparison between Multiple Step and 1-Step Policy Improvements

The AlphaZero MCTS algorithm is simpler than ours: it does not use rollouts or RAVE, and a single search algorithm is used throughout training.

This allows us to perform an ablation to study the impact of using a multiple step, rather than single step, improvement in π^*, π^{sampling} and π^{exploit} (algorithm 2, lines 4-6). The multiple step improvement occurs because the simulation policy defined by the PUCT formula becomes stronger than the neural network policy as search progresses. The Q-values estimated at the root node are then the Q values of the average search policy, not of the original network policy.

We can adapt the MCTS algorithm to perform a 1-step policy improvement by replacing the PUCT action selections in the non-root nodes with

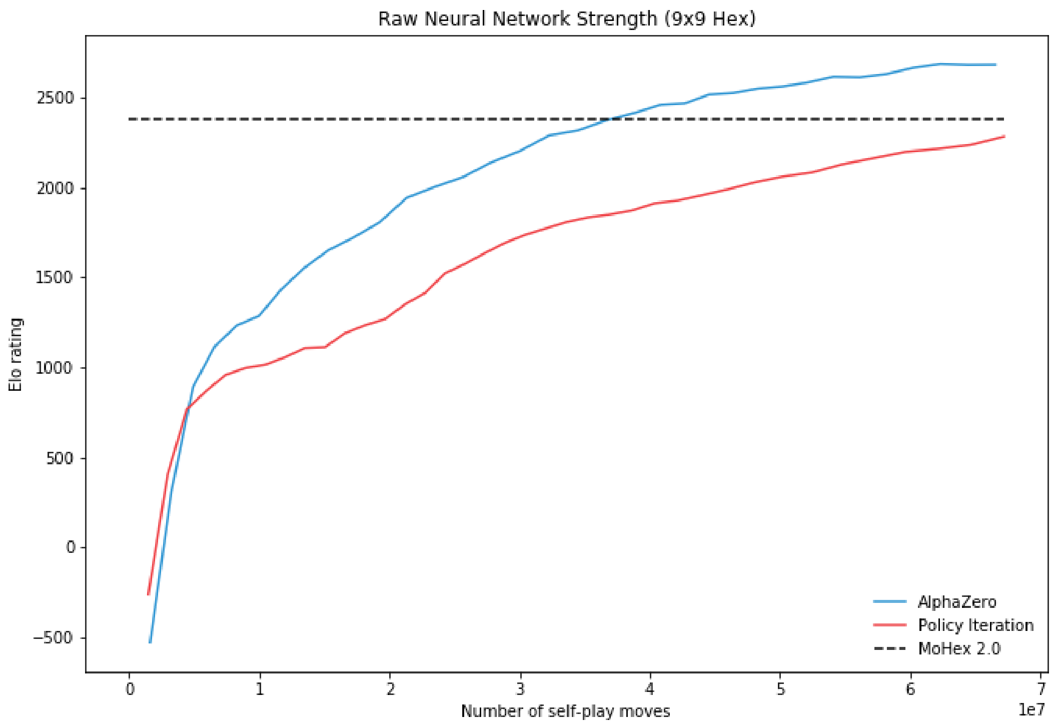

Figure 4.7: Comparison of Elo ratings for networks trained using AlphaZero and an otherwise identical Policy Iteration algorithm which only uses single-step planning. Performance of both methods is strong, but with a substantial advantage for AlphaZero, which shows the advantage of multi-step search. Data from single training runs. The x-axis is the number of self-play moves generated in self-play, because each algorithm uses the same number of search iterations per move, this is proportional to total compute cost: the total number of network evaluations is around 500 times larger

samples from the prior policy. Now, because the leaf nodes are sampled from the search tree according to the prior policy, the Q-values estimated at the root are for that policy also. This is a version of Monte Carlo Search with truncated rollouts.

Since the play of MCTS, which finds a multiple step improvement is much stronger than that single step improvement found by MCS, we expect to see that it also performs better as a policy improvement operator in RL. This indeed was the case, the neural-network trained with a multiple step improvement both learned faster and achieved a much stronger final performance (see figure 4.7).

Nonetheless, the training run using single-step improvements showed stable learning. This contrasts starkly from the experience with model-free policy gradients, which has periods where the performance regresses significantly, suggestive of cycles of learning and forgetting. This indicates that the stability of Expert Iteration algorithms is a result of using large replay buffers (rather than streamed experience from only the current policy), supervised policy targets (rather than policy gradient losses), and low variance policy targets formed as the amalgamation of multiple simulations.

4.5.5 Use of Monte Carlo Value Estimates

Prior works using tree-search for learning, such as TD(leaf), TD(root) and TreeStrap trained their value functions using the value predicted by a minimax tree search, either in the current state, or after a single turn.[12]

In contrast, both this work and AlphaZero used the final outcome of a self-play game played by search algorithm, or a self-play game of a policy that imitates the search algorithm. In other words, in line 7 of algorithm 2, we use $n = \infty$, whereas prior works use $n = 1$ and bootstrap with an expert value function.

How important is this difference to prior works for the reinforcement learning algorithm?

Given a dataset generated by the AlphaZero algorithm, we can retrain the networks to predict not the outcome of the game, but the value predicted by the search, in line with the prior works. Doing this we achieve a very slightly improved performance in the task of predicting the outcome of the game on a validation set (figure 4.10), and in policy prediction accuracy (see figure 4.9). Figure 4.8 shows the training curves; note that the value losses are not comparable due to the different targets.

An aside: The training losses and accuracy figures are taken from the data trained on each iteration, averaged over the most recent 1000 updates.

[12]A notable exception is the chess program Giraffe [Lai, 2015], which played out multiple steps before dropping to a search value.

The test losses, on the other hand, are measured by taking a snapshot of the parameters after each 1000 updates, and measuring the performance of those parameters on the test set. This means that the training loss value is smoothed over 1000 updates, whereas the test loss is not, this explains why test loss metrics are less smooth/more stochastic in the following figures.

However, this is not the whole story. Based on a short training run, we found that if we use this search value training target during training, the performance of the RL algorithm as a whole is worse, as measured by the Elo ratings of the neural networks, plotted in figure 4.11.

Figure 4.12 can show us why. These figures plot the mean squared error of value predictions as a function of the distance from the end of the game. This shows us how well the value function understands different stages of the game. Prediction typically will become easier closer to the end of the game. Each line corresponds to a different stage in training, giving an insight to how training progresses.

At the start of training, the value function is uninformative, so search only accurately predicts game values when close enough to terminal states to find them. In AlphaZero, the value function gradually learns to predict the outcome of the game, becoming accurate from increasingly far from the ends of games. Importantly, although accuracy near the end of the game is always best, the reduction in prediction accuracy as the distance from the end of the game increases is always gradual.

When using the search value, however, the reduction in accuracy remains very severe, with a cut-off point where predictions suddenly become no better than random. This cut-off point slowly moves further from the end of the game, but never becomes less severe as training progresses.

The reason for this bad behaviour when predicting the search value is that the search values themselves are only useful very near the end of the game when searching with randomly initialised parameters. Predicting this search value is only able to improve performance on the frontier of where predictions are accurate, learning how to estimate state values in the middle game can only commence once the end game value prediction problem is solved; whereas the smooth curves displayed by AlphaZero represent that some learning in the middle game can be achieved before end game states are solved.

Note that this is similar to the difference in behaviour between value iteration and policy iteration algorithms, where value iteration algorithms require more iterations before they can converge to an optimal policy, and after a finite number of iterations they only optimise for a finite-horizon. In board games, optimising for a small horizon often means there is no signal at all, because when the game does not end within that horizon, no reward is seen.

An aside: the characteristic zig-zag patterns seen in 4.12 are a consequence of a particular property of the game Hex: in Hex, the last player to move is always the winner of the game. MCTS generally makes somewhat optimistic value estimates, because it builds policies that maximise an inaccurate value function. If a state is an even number of moves from the end of the game, this implies the current player to move must have lost the game (because they weren't the last player to move). Therefore, when there is an even number of turns-to-go, the optimism results in a slightly worse prediction, while when there are an odd number of turns-to-go the optimism slightly improves the prediction accuracy.

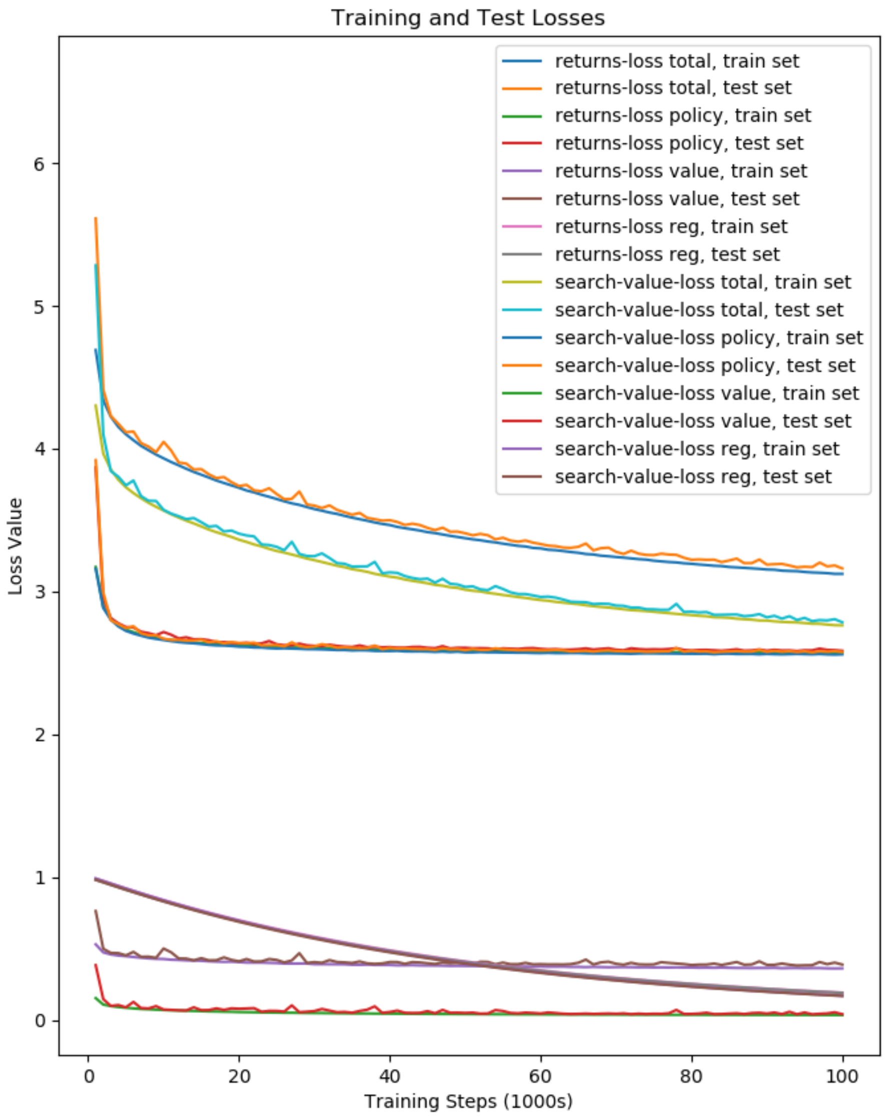

Figure 4.8: Training and test set losses for retraining networks with either episode return targets, or search value targets. For supervised learning on a fixed dataset, performance is very similar for both methods. The difference between the value losses is because episode return targets have higher variance than search value targets, thought their expectations are similar. Results from a single dataset, and single training runs on that data.

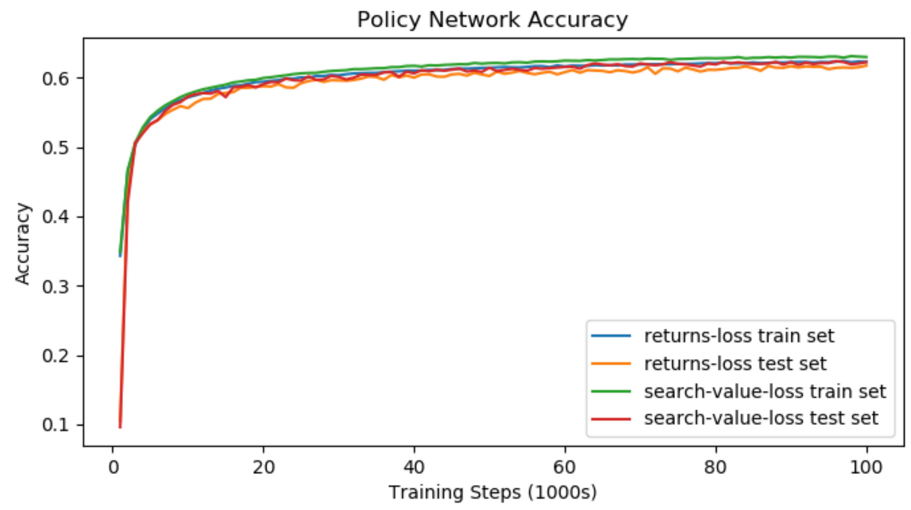

Figure 4.9: Policy prediction accuracy for retraining networks with either episode return targets, or search value targets, showing little difference in performance.

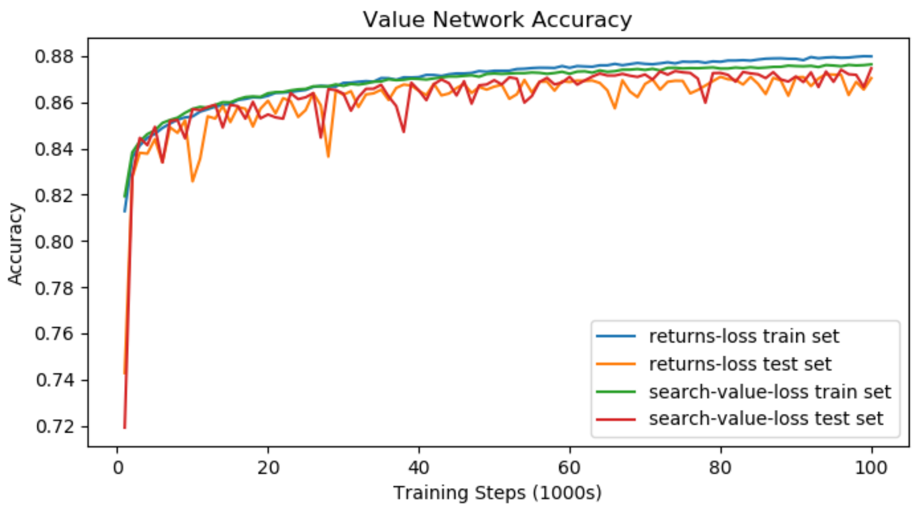

Figure 4.10: Value prediction accuracy for retraining networks with either episode return targets, or search value targets. Performance on the test set is similar between the two methods.

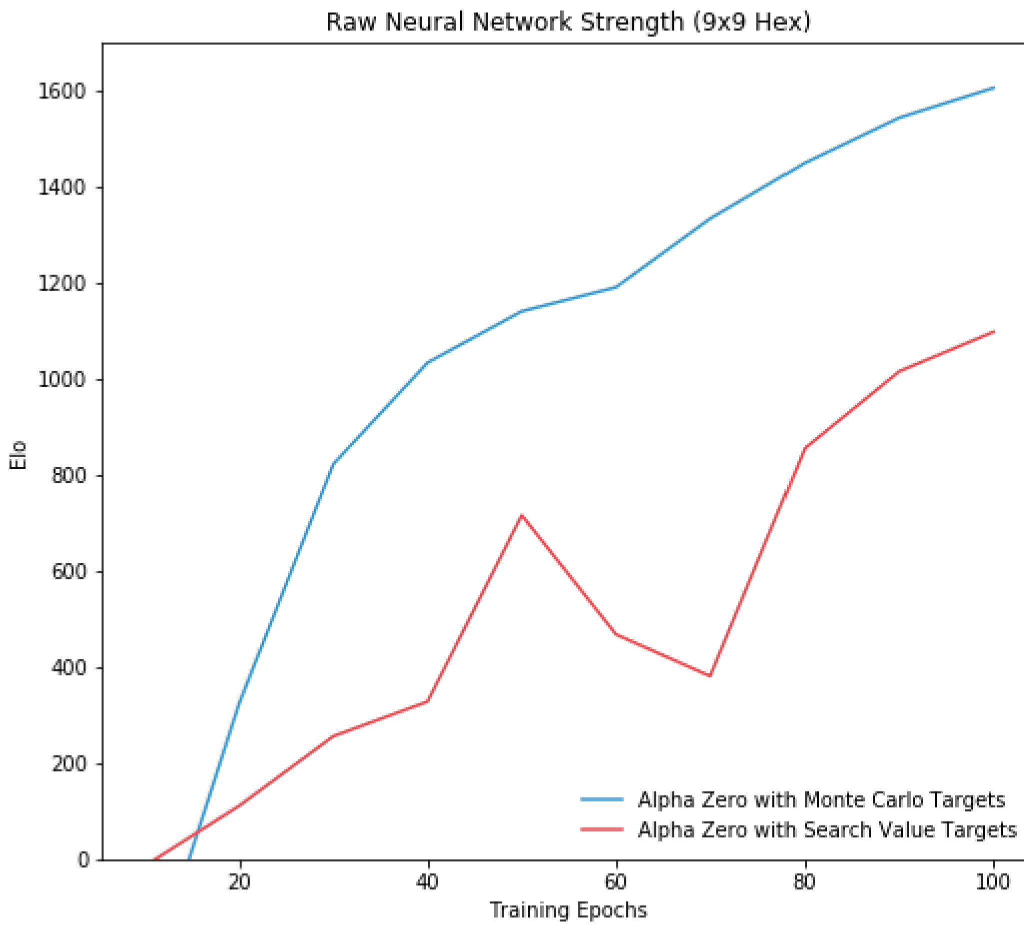

Figure 4.11: Raw network play strength for AlphaZero trained with either episode return targets, or search value targets. Training with search value targets gives substantially weaker final performance, the difference of approximately 500 Elo is equivalent to a head-to-head winrate of 95%. Single training runs, 1-epoch is 1000 updates with batch size 1024

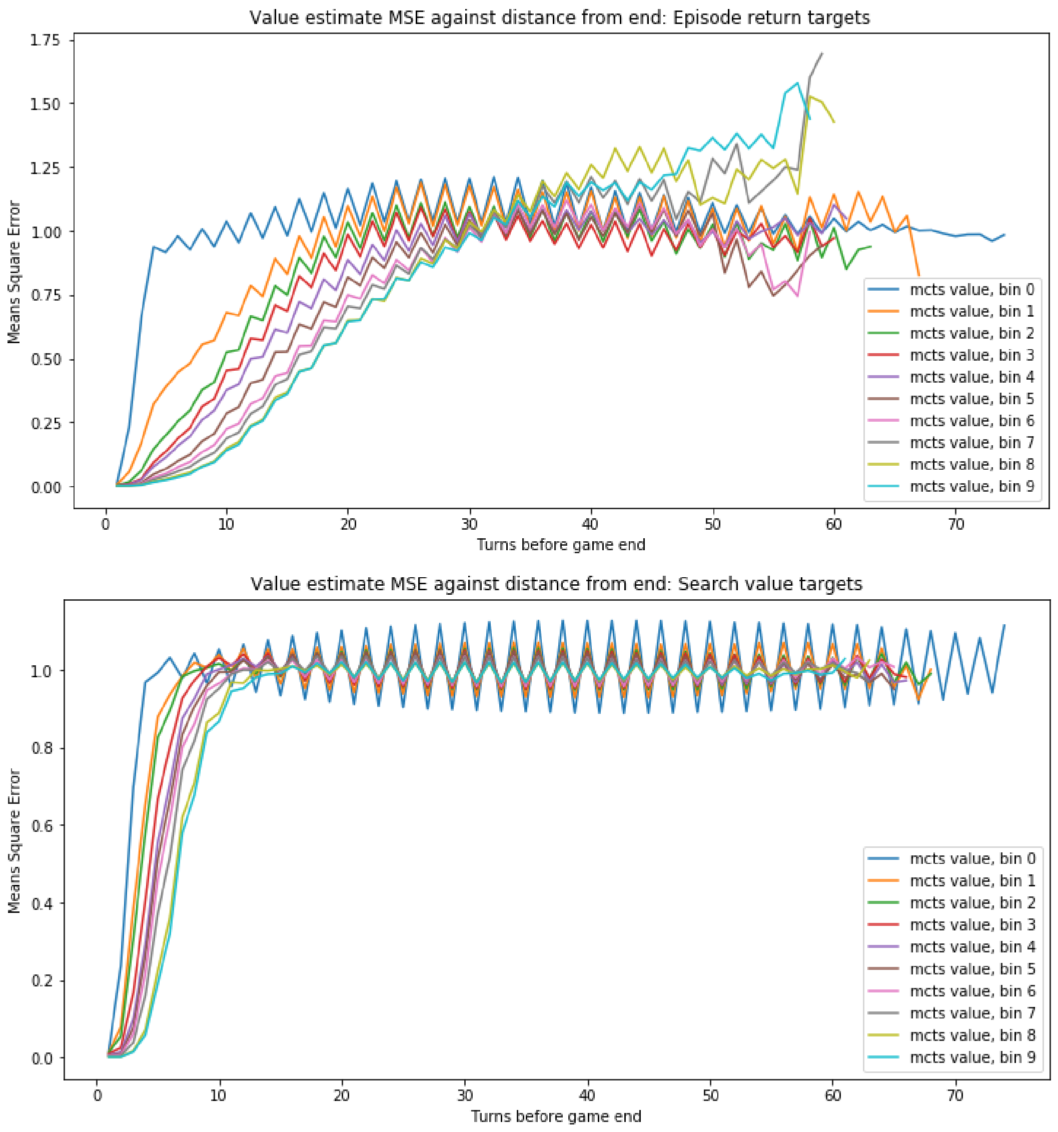

Figure 4.12: AlphaZero search value prediction error progression against distance from the end of the game, for two methods of training the value function. Rewards were scaled as +1 for a win, −1 for a defeat, so a constant prediction of 0 results in a mean square error of 1. The bins split the data chronologically through training, so bin 0 represents data generated between the 1st and 10,000th training step, bin 1 data between the 10,001st and 20,000th step, etc. In both cases prediction accuracy improves, however for search value targets still only improve in predicting the outcome near the end of the game, whereas episode return targets improve predictions from much further before the game's end.

4.6 Conclusions

In this chapter, we have demonstrated the effectiveness of Expert Iteration in the game of Hex when using Monte Carlo Tree Search experts. Our Expert Iteration algorithm substantially outperformed REINFORCE with only policy learning. A version combined with value learning was able to defeat MoHex versions 1.0 and 2.0.

We showed how tree policy targets outperform the chosen action targets used by prior work on imitating policies from Monte Carlo Tree Search, and illustrated how those learned policies can be used to drive a much deeper search.

The concurrent work AlphaZero extended ours by employing the expert for value learning, as well as policy learning. This makes data generation considerably more expensive, but stronger value targets should result in better play in the limit.

We performed two ablations on AlphaZero. First we compared it to a version that used Monte Carlo Search. This ablation demonstrated that the multiple step policy improvement that distinguishes the Expert Iteration framework from prior Policy Iteration methods leads to a sizeable improvement in the performance of the algorithm.

Second, we showed how Monte Carlo value estimates significantly improve the performance of AlphaZero, as they allow faster propagation of information backwards through the game. This may partially explain why prior methods that combined TD-learning with minimax search were less successful.

Chapter 5

Policy Gradient Search

In chapter 4 and in Silver et al. [2017, 2018], we saw that Monte Carlo Tree Search, as well as being an effective online planning algorithm, can also be used as an expert for training RL agents in the Expert Iteration framework. These methods achieved state-of-the-art performance in several classical board games without human knowledge for the first time.

However, MCTS builds an explicit search tree, storing visit counts and value estimates at each node - in other words, creating a tabular value function for parts of the game tree near the current state. To be effective, this requires that nodes in the search tree are visited multiple times. This is true in many classical board games, but real world problems often have branching factors so large that MCTS is hard to use.

Large branching factors can be caused by very large action spaces or chance nodes. In the case of large action spaces, a prior policy can be used to discount weak actions, reducing the effective branching factor. Stochastic transitions are harder to deal with, as prior policies cannot be used to reduce the branching factor at chance nodes.

In contrast, Monte Carlo Search (MCS) [Tesauro and Galperin, 1997] algorithms have no such requirement. Whereas MCTS uses value estimates in each node to adapt the simulation policy, MCS algorithms have a fixed simulation policy throughout the search. However, because MCS does not improve the quality of simulations during search, it produces significantly weaker play than MCTS.

To adapt simulation policies in problems where we don't visit states multiple times, we can search over a restricted version of the problem, where multiple visits to states do occur, as in progressive widening and double progressive widening [Couëtoux et al., 2011]. Alternatively, we can generalise knowledge between different states during search. In this chapter, we take the latter approach. To this end, we develop Policy Gradient Search (PGS), a search algorithm which trains its simulation policy during search using policy gradient RL. This gives the advantages of an adaptive simulation policy, without requiring an explicit search tree to be built.

In this chapter we ask: Could a planning algorithm that doesn't use tabular function approximation perform as well as MCTS in Hex, a game where MCTS excels?

In section 5.1, we discuss in detail some of the assumptions that need to be fulfilled in order for MCTS based approaches to be most effective. Section 5.2 describes the PGS algorithm, which can tackle some of these issues. In section 5.3 we assess the strength of PGS as a test-time decision maker, while in section 5.4 we show results from using PGS within ExIt.

5.1 MCTS and Branching Factor

Compared to traditional tree search algorithms, MCTS is particularly strong in games with large branching factors: for example the best $\alpha-\beta$ search programs are very strong in chess, which has around 30 legal moves in typical states; competitive with (but weaker than) MCTS programs in Hex (for example, Wolve is only slightly weaker than MoHex 1.0, and still wins some games against MoHex 2.0 [Huang et al., 2013]); and is not competitive in Go, with around 300 legal moves in typical states.

The crucial characteristic that allows MCTS programs to succeed in the games with larger branching factors is how it builds an asymmetric search tree. Strong moves are searched many times, resulting in deep lookahead. Weaker moves are searched much less. When accurate heuristics are available, the weakest moves may not need to be searched at all.

This ‘soft pruning’ effectively reduces the necessary breadth of the search tree from the actual breadth of the game tree, to being approximately the number of *plausible* moves in a state. In many classical board games, most moves are blunders, leaving only a few options needing to be searched in depth.

However, this approach will not result in narrow search trees in all problems. Implicitly, it relies on features of classical board games that do not hold in all problems of interest. If search trees are not narrow, we do not visit nodes multiple times, and MCTS will fail.

5.1.1 Deterministic Transitions

If an MDP has deterministic transitions, then each action has a single child node, however if a chance event occurs with k possible outcomes, then k nodes are needed. This multiplies the branching factor by k.

The pruning heuristic MCTS uses obviously does not apply to the outcomes of chance events, and any likely outcomes of chance events need to be considered. Even for small values of k, the branching will make the search tree impractical after only a couple of steps.

It may be sufficient to search just a few of the chance outcomes to reduce the branching factor. However, with few samples, the variance of the estimates will remain high, so search will likely select whichever actions sampled ‘lucky’ chance outcomes, and the benefit of searching with improving policies may be dominated by this effect. In situations with several actions that are ‘risky’, and some ‘safe’ actions, combining high variance estimation of values with greedy action selection will lead to an overall bias towards the risky actions, because it is highly likely that some risky action’s value will be overestimated.

When the chance outcomes share structure, it may be possible instead to search them all, while generalising on the fly between the different chance events to overcome the disadvantages of a high branching factor [Powley et al., 2014, Sironi and Winands, 2017].

5.1.2 'Nice' Action Space

Although ignoring sub-optimal actions is effective for games such as Hex, in other settings, there may either be too many actions for learning to get started (since at the beginning of training the agent cannot know which actions to ignore), or there may be too many actions that are near-optimal in any given state. For example, if in Hex every action was duplicated 100 times, there would be at least 100 optimal moves in every state. Classical MCTS would search through each of these actions independently, resulting in a branching factor of over 100, which would be too large to be practical.

Continuous action spaces do not have finite branching factors. They could be discretised, either with a grid or via sampling from some policy network, but such sub-sampling must be carefully managed: too many samples, and the branching factor will be too large, with 'nearby' actions all being approximately optimal. Not enough samples and optimal actions may be missed both at the root node, resulting in a search that cannot find the correct answer; and at an internal node, resulting in misleading value estimates being provided for the choice at the root. An algorithm that can generalise between different search tree branches may more gracefully cope with finer grained discretisation, by generalising between the similar branches.

5.2 Policy Gradient Search

The core idea of PGS is that, because simulation-based search algorithms are a form of reinforcement learning algorithm applied to the online RL problem (see section 2.6), we can apply any modern deep reinforcement learning algorithm to this online RL problem.

The algorithm must represent everything it learns through non-tabular function approximators, otherwise it will suffer the same drawbacks as MCTS. MCTS can be viewed as a form of self-play RL [Silver, 2009], however we cannot directly adapt it to use function approximation, because versions of the UCB formula rely on count-based exploration rules.

Instead, PGS works by applying a model free RL algorithm to adapt the simulations in Monte Carlo Search. We will assume that a prior policy π and a prior value function V are provided, which have been trained on the full MDP.

We use policy gradient RL to train the simulation policy. Our simulation policy π_{sim} is represented by a neural network with identical architecture to the global policy network. At the start of each game, the parameters of the policy network are set to those of the global policy network.

Because evaluating our simulation policy is expensive, we do not simulate to a terminal state, but instead use truncated Monte Carlo simulation. Choosing when to truncate a simulation is not necessarily simple; the best choice may depend on the MDP itself. If simulations are too short, they may fail to contain new information, or not give a long enough horizon search. Too long simulations will be wasteful.

Reasonable strategies could include simulating for a fixed horizon H; or for a gradually increasing horizon $H(n)$ on the nth simulation; or simulating until a threshold on the probability density of the action sequence is reached, to induce deeper search down the principal variation. For Hex, we run each simulation until the action sequence for that simulation is unique. [1]

Once we reach a final state s_L for the simulation after t steps, we estimate the value of this state using the global value network V, and use this estimate to update the simulation policy parameters θ using REINFORCE (Williams [1992], for a derivation see section 2.3.4):

$$\theta = \theta + \alpha V(s_L) \sum_{i=1}^{T} \nabla_\theta \log \pi_\theta(a_i|s_i)$$

where α is a learning rate. For PGS, we re-scaled the reward to be -1 for losing, and a reward of 1 for winning so that a baseline of 0 would be sensible. In testing, we found that using the value function as a baseline had no measurable effect on performance, but for other problems, such a non-zero baseline may

[1]Noting that an alternative would be needed in domains with very large action spaces.

be necessary [2]. The policy gradient updates can be seen as fine-tuning the global policy to the current sub-game.

Because the root node is visited in every simulation, as with MCS, we can use a bandit-based approach to select the first action a_0 of each simulation. We adopt the PUCT formula [Silver et al., 2017, Rosin, 2011] for this, greedily choosing the action that maximises:

$$\text{PUCT}(s_0, a) = Q(s_0, a) + c_{\text{puct}}\pi(s_0, a)\frac{\sqrt{n(s_0)}}{1 + n(s_0, a)}$$

Where c_{puct} is a hyperparameter. $Q(s_0, a)$ is the average return from all simulations so far that started with the action a. $\pi(s_0, a)$ is the original global policy at s_0. Every subsequent action of the simulation $a_1, a_2, ..., a_t$ is sampled from π_{sim}.

Algorithm 4 Policy Gradient Search

```
Require: $\pi_\theta, V_\phi, s_0, \alpha$        ▷ Policy, value function, state, learning rate
 1: $N \leftarrow 0$
 2: prior $\leftarrow \pi_\theta(s_0)$
 3: for $a$ in legal_actions($s_0$) do
 4:     $n[a] \leftarrow 0$                                  ▷ Counts for first actions
 5:     $v[a] \leftarrow 0$                                  ▷ Total value for first actions
 6: end for
 7: while $N < N_{\max}$ do                   ▷ Loop is parallelised for network batching
 8:     $a_0 \leftarrow \arg\max_a \text{PUCT}(n[a], v[a], \text{prior})$    ▷ Chose first action by PUCT
 9:     $N, n[a_0], v(a_0) \leftarrow N, n[a_0] + 1, v(a_0) - 1$                ▷ Add Virtual Loss
10:     $t \leftarrow 1$
11:     $s_t \leftarrow$ next_state($s_0, a_0$)
12:     while not termination_condition($s_0, a_0, ..., s_t$) do
13:         $a_t \leftarrow \pi_\theta(s_i)$
14:         $s_{t+1} \leftarrow$ next_state($s_t, a_t$)
15:         $t{+}{+}$
16:     end while
17:     $R \leftarrow V_\phi(s_t)$
18:     $v[a_0] \leftarrow v[a_0] + r + 1$              ▷ Update value and remove virtual loss
19:     $\theta \leftarrow \theta + \alpha R \sum_{i=1}^{t} \nabla_\theta \log \pi_\theta(a_i|s_i)$           ▷ Policy Gradient Update
20: end while
21: return $\arg\max_a n[a]$
```

[2]see section 2.3.4 for a discussion of baselines in policy gradient methods

In algorithm 3, lines 1-6 set up variables. Lines 7-20 are our main search loop. This loop is parallelised across several threads (we used 32 threads) to allow the network evaluations (lines 13 and 17) and the parameter update (line 19) to be performed in batches. For simplicity we synchronised this, so all threads would join at lines 13, 17 and 19.

Lines 8 and 9 select a first action at the root node with a bandit formula, and update virtual loss statistics; these should be done atomically. The loop from line 12-16 samples the rest of the trajectory according to the current policy, until a termination condition is reached. The final state in that trajectory is then evaluated (line 17), and the statistics for the PUCT formula at the root are updated with this value (line 18) and the policy network parameters are updated (line 19), using the actions chosen by the policy network (and not using the first action chosen deterministically by the bandit formula)

5.2.1 Parameter Freezing during Online Adaptation

During testing, online search algorithms are usually used under a time constraint, so it is important to ensure that our algorithm does not require too much computation per simulation step. When used for offline training in Expert Iteration, the efficiency of the search method is still crucial. If it is too slow, it would be more efficient to use a worse but faster planner and run for a greater number of iterations.

We use a residual neural network with the architecture used by Silver et al. [2017], with 19 residual blocks and separate policy and value heads. In order to learn a policy across the entire state space from a dataset of millions of states, the global neural network is very large and therefore expensive to evaluate. In contrast, we apply our search algorithm with only hundreds or thousands of search iterations. Far fewer parameters should be sufficient for the online adaptation.

PGS is more expensive than MCS because it must perform the policy gradient update to the neural network parameters. The backward pass through our network takes approximately twice as long as the forward pass, making

PGS 3-4 times more expensive than MCS. In order to reduce the computational cost of the algorithm, during policy gradient search we adapt the parameters of the policy head only. This reduces the computation used by the backward pass by a factor of over 100, making the difference in computational cost between MCS and PGS negligible.

We found that freezing the shared residual stack in this way did not only reduce the computational cost of PGS, but also improved performance of the algorithm on a per-iteration basis. This suggests that although useful for training the global policy and value function, during search the large number of parameters available in our neural network is detrimental to performance.

(In games such as Hex where states are visited multiple times, an additional optimisation can then be made: the forward pass through the fixed part of the network can be cached, rather than being recalculated for every visit of each simulation. This is similar to storing the prior policy at each node of MCTS, and substantially reduced the runtime of our experiments.)

5.2.2 Note on Batch Normalisation

Our neural network uses batch normalisation [Ioffe and Szegedy, 2015]. In all instances, the global neural networks have been trained on datasets of states from many independently sampled games of Hex.

During search, the input distribution is substantially changed to consist of highly correlated states, as each simulation is started from the same initial state. Using mini-batch statistics for batch normalisation therefore results in a large shift in the policy. So during PGS we freeze the parameters for batch normalisation, and calculate the normalisation using population rather than mini-batch statistics, as is usual during inference.

5.3 Policy Gradient Search as an Online Planner

To allow for comparison to a strong baseline, we continue to test on the domain of Hex. If we can match the performance of MCTS, then PGS is a demonstrably capable approach to planning. As noted in section 5.2.1, faster (and cheaper) experimentation is possible in this domain.

In this section, we test the performance of PGS for maximising agent performance at test-time, independently of learning policies.

5.3.1 Baselines

In our experiments we use two baseline search algorithms: MCTS and MCS. MCTS provides a strong baseline, but becomes harder to use in some MDPs with very large branching factors. In contrast, MCS is easier to apply to general MDPs, but is weaker than MCTS in Hex.

The MCTS algorithm used is the same as used by AlphaGo Zero [Silver et al., 2017]. That is, we use the PUCT formula for our tree policy, $c_{\text{puct}} = 5$, and use a value network for leaf evaluations. Our MCS is as described in section 4.5.4, the same as the MCTS but sampling from the prior policy instead of using PUCT at every node except the root. It is therefore also the same as our PGS algorithm with a learning rate of 0.

For all search algorithms, simulations are completed in batches of 32, with virtual losses added wherever the PUCT formula was used to encourage diversity in the simulations [Segal, 2010].

5.3.2 Description of the Neural Networks

To show general applicability, we test using multiple different global neural networks, trained with different variants of Expert Iteration. In each case, we compare PGS to MCS and MCTS using the same neural network. For PGS, we tuned the learning rate for the policy gradient updates during search independently for each neural network, as we found that the networks trained by different methods needed different learning rates. This was done in the same approach as used in section 4.1.4.

For the UCB formula, we did not find that different exploration parameters were necessary as we switched the network. All networks produce distributions as outputs, which are on the same scales. In contrast, the magnitudes of the trained parameters of the different networks, and how quickly the outputs change as the parameters are changed, could vary substantially. It makes sense that this would mean PGS's learning needs more careful fine-tuning for each network than the UCB exploration parameter does.

The networks used are:

1. A residual neural network for 9×9 Hex trained on the dataset generated in the distributed training run from chapter 4.
2. The network at the end of training for our version of AlphaZero (see section 5.4) applied to 9×9 Hex
3. A network from early in training with Policy Gradient Search Expert Iteration (PGS-ExIt, section 5.4), applied to 9×9 Hex i.e. at epoch 10 of 450
4. A network from approximately half way through training with PGS-ExIt on 9×9 Hex, i.e. at epoch 230 of 450
5. The network at the end of training with PGS-ExIt applied to 9×9 Hex, i.e. at epoch 450 of 450
6. The network at the end of training with our version of AlphaZero, applied to 13×13 Hex

5.3.3 Round Robin Tournament

For each neural network, we ran a round robin tournament between the raw neural network and three search algorithms MCS, MCTS, and PGS. That is, a match of $2n^2$ games of $n \times n$ Hex was played between each pair of algorithms

Each search algorithm used 800 search iterations per move. Elo ratings were calculated using BayesElo [Coulom, 2005], with the scale shifted so the mean estimate for the raw neural network's Elo was 0 in each case, giving a different Elo scale for each tournament. Results are presented in table 5.3.3, along with the values of α used in PGS.

NN	NN Elo	MCS Elo	MCTS Elo	PGS Elo	PGS α
1	0 ± 42	351 ± 26	$\mathbf{473 \pm 27}$	$\mathbf{430 \pm 26}$	5e-4
2	0 ± 26	109 ± 24	$\mathbf{160 \pm 24}$	$\mathbf{140 \pm 24}$	1e-4
3	0 ± 43	328 ± 27	$\mathbf{428 \pm 27}$	$\mathbf{457 \pm 27}$	1e-4
4	0 ± 28	153 ± 25	$\mathbf{238 \pm 25}$	$\mathbf{205 \pm 25}$	1e-4
5	0 ± 27	126 ± 24	$\mathbf{195 \pm 25}$	$\mathbf{185 \pm 28}$	1e-4
6	0 ± 25	161 ± 25	$\mathbf{269 \pm 23}$	$\mathbf{239 \pm 23}$	2e-5

Table 5.1: Strengths of different search algorithms in online planning, for the 6 networks enumerated in section 5.3.2. Ratings and 95% confidence intervals from BayesElo. The multi-step search algorithms PGS and MCTS consistently outperformed MCS.

In all cases, all search algorithms significantly outperformed the raw neural network. Most differences between the search algorithms are smaller, but overall trends are clear: **on average PGS is ≈ 70 Elo stronger than MCS, and MCTS is ≈ 20 Elo stronger than PGS.**

5.3.4 Effect of Parameter Freezing in Policy Gradient Search

We tested the impact of training all policy network parameters in PGS. Compared to the other algorithms, this is significantly more expensive per iteration, nonetheless we still tested with equal number of iterations. We added this version of PGS to our round robin tournament by playing a match between this algorithm and each of the neural network, MCS, MCTS and the original version of PGS.

Note that we never train the value head, and when fine-tuning the whole simulation policy network we change the parameters of the shared layers. Because the value is not simultaneously being trained, this would corrupt value estimates. Instead we maintain two copies of the neural network, one fixed copy for the value function, and one that we fine tune for the simulation policy.

PGS without freezing parameters (PGS-UF) was found to be consistently weaker than PGS, even disregarding the additional computational cost, results are given in table 5.2 (with Elo scores on the same scales as in table 5.1 for reference). The performance of PGS-UF is quite similar to that of MCS, suggesting that little effective adaptation has occurred.

During training, deep neural networks may learn hierarchical features [LeCun et al., 2015], where the early hidden layers learn low-level features such as shapes and edges, which subsequent layers can combine to produce concepts such as objects and classes. In Hex therefore, we might expect our early layers to detect simple patterns, such a small virtual connections or semi-connections. Subsequent layers would assemble those into more complex structures, and final layers would evaluate the importance of those structures relative to the whole board's position.

If this is the case it affords an interesting interpretation to the failure of PGS without frozen parameters: it implies that adapting the strategy chosen given the structures on the board and adapting the assessment of how structures interact at a high level are beneficial, but attempting to adapt the lower-level feature detection that identifies those structures based on only a few local simulations is misguided. This seems intuitive, as these higher-level nuances might be both harder to generalise and easier to detect through studying the particular position.

5.3.5 Regularising Policy Gradient Search

Policy Gradient methods have been improved for the standard reinforcement learning setting with regularisers that prevent the policy from moving too far in a single update (e.g. Schulman et al. [2015a, 2017]). We can do this in policy gradient search with a KL cost on the difference between the fine-tuned

NN	MCS	PGS Elo	PGS-UF Elo	PGS-UF α
1	351 ± 26	430 ± 26	355 ± 30	1e-6
2	109 ± 24	140 ± 24	121 ± 27	5e-6
3	328 ± 27	457 ± 27	350 ± 30	5e-5
4	153 ± 25	205 ± 25	178 ± 28	5e-6
5	126 ± 24	185 ± 28	156 ± 27	5e-6

Table 5.2: Comparison of Policy Gradient Search with and without parameter freezing; ratings and 95% confidence intervals from BayesElo. Without parameter freezing performs only slightly better than the non-adaptive Monte Carlo search algorithm. Note also that the best learning rates found are much lower than for the frozen parameters version (table 5)

policy and policy of the original network:

$$\mathcal{L}_{\text{reg}} = cKL(\pi_{\text{orig}}, \pi_{\text{fine}})$$

Here the KL is between the distributions over actions of the fine-tuning policy and the original policy on the states visited during the simulations in search.

This regularisation method improves PGS. With regularisation, PGS performs similarly to or is slightly stronger than MCTS. Results are reported in table 5.3.

One effect of the regularisation is that, for all networks tested, a larger learning rate was optimal. This implies that, for the unregularised policy gradient search algorithm, one function of the learning rate was to restrict the extent to which the policy deviates from the global policy. This happens because over a relatively small number of learning steps, the learning rate can control by how much the parameters change, and hence how much the policy can change. The KL regularliser is able to restrict policy deviation directly. This means learning rate can be set to optimise the speed of adaptations, without having to also to control the final extent of those adaptations.

5.3.6 Scaling with number of search iterations

We also tested how the performance of the different search algorithms scales with different numbers of search iterations, in a range from 200 to 1600 search

NN	MCTS Elo	PGS Elo	KL-PGS	α	c
1	473 ± 27	430 ± 26	473 ± 30	1e-3	0.02
2	160 ± 24	140 ± 24	173 ± 28	2e-4	0.02
3	428 ± 27	457 ± 27	512 ± 31	5e-4	0.05
4	238 ± 25	205 ± 25	241 ± 28	2e-4	0.02
5	195 ± 25	185 ± 28	196 ± 28	1e-4	0.02

Table 5.3: Comparison of Policy Gradient Search with and without KL regularisation; ratings and 95% confidence intervals from BayesElo. Adding regularisation improves the performance of PGS. With a regulariser controlling the extent of deviations, KL-PGS is able to use a higher learning rate

iterations per move, our results are plotted in figure 5.1. **In both cases we see that the adaptive search algorithms, PGS[3] and MCTS, scale much more effectively with number of search iterations than does MCS.** This supports the hypothesis in section 3.2.2 that large search budgets are more useful with multi-step search algorithms than for single-step planners.

These round robin tournaments have 12 search algorithms and a neural network agent, so consist of rather large numbers of matches. Adding more search iterations also increases the cost of the experiment. We therefore assess the scaling with the strongest MCTS and PGS trained networks, which were networks 2 and 5.

PGS might scale less well than MCTS if the capacity for the policy head to represent adaptations were saturated. We find no evidence that this occurs, but note that 1600 iterations per move is still a fairly short search; such an effect may still take place in longer searches.

5.4 Policy Gradient Search Expert Iteration

A key motivation for Policy Gradient Search is the utility of online planning algorithms in the ExIt framework. To this end, we used PGS as an expert in ExIt, comparing again to the baseline MCS and MCTS agents. We use the same Expert Iteration set up as for our reimplementation of AlphaZero

[3]Without the regularisation technique in section 5.3.5

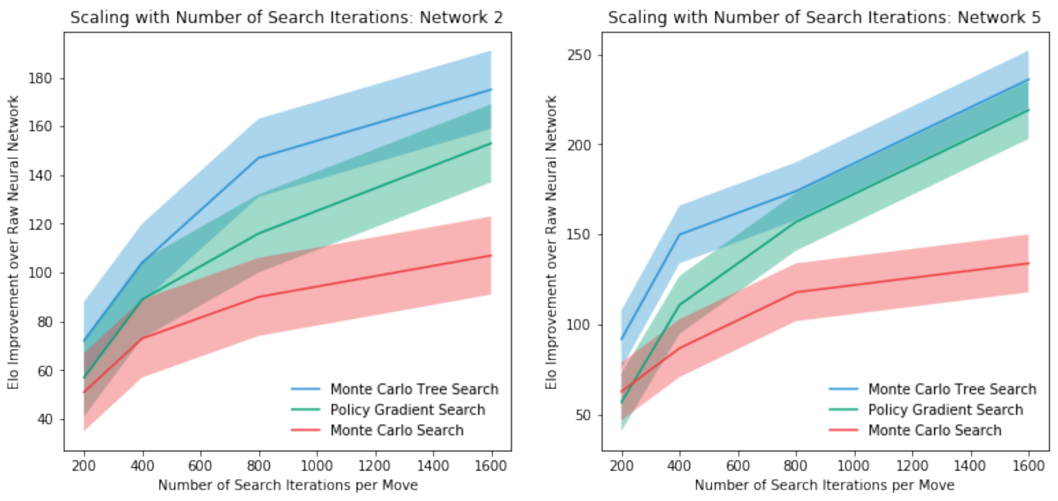

Figure 5.1: Graphs showing how the strength of search algorithms scales with the number of search iterations. While the multi-step planning algorithms MCTS and PGS are able to effectively use additional simulation budget to further improve their performance, MCS improves less with the increased search budgets. Shaded regions represent 95% confidence intervals.

in section 4.5.2. For all search algorithms, $c_{puct} = 5$ wherever the PUCT formula is used. For PGS-ExIt, we used an 'inner' learning rate during PGS of $\alpha = 0.0005$, and no regularisation was used.

5.4.1 Results

Our results are in line with those from section 5.3, with PGS performing better than MCS, but not as well as MCTS. Over the course of training the differences in strength between the agents have compounded through repeated use of better or worse experts. **AlphaZero (i.e. MCTS-ExIt) outperforms PGS-ExIt, which in turn substantially outperforms Policy Iteration (i.e. MCS-ExIt)**. We show the strength of the raw policy networks (with no search) throughout training in figure 5.2.

The 'inner' learning rate α for PGS-ExIt was chosen based on the optimum value for Network 1 from section 5.3.2. Subsequent tests showed this not to be optimal for networks trained by PGS-ExIt. Indeed, the benefit of using PGS over MCS is much reduced by this sub-optimal α. We also did not use KL-regularisation in this experiment, as we had not added this when we

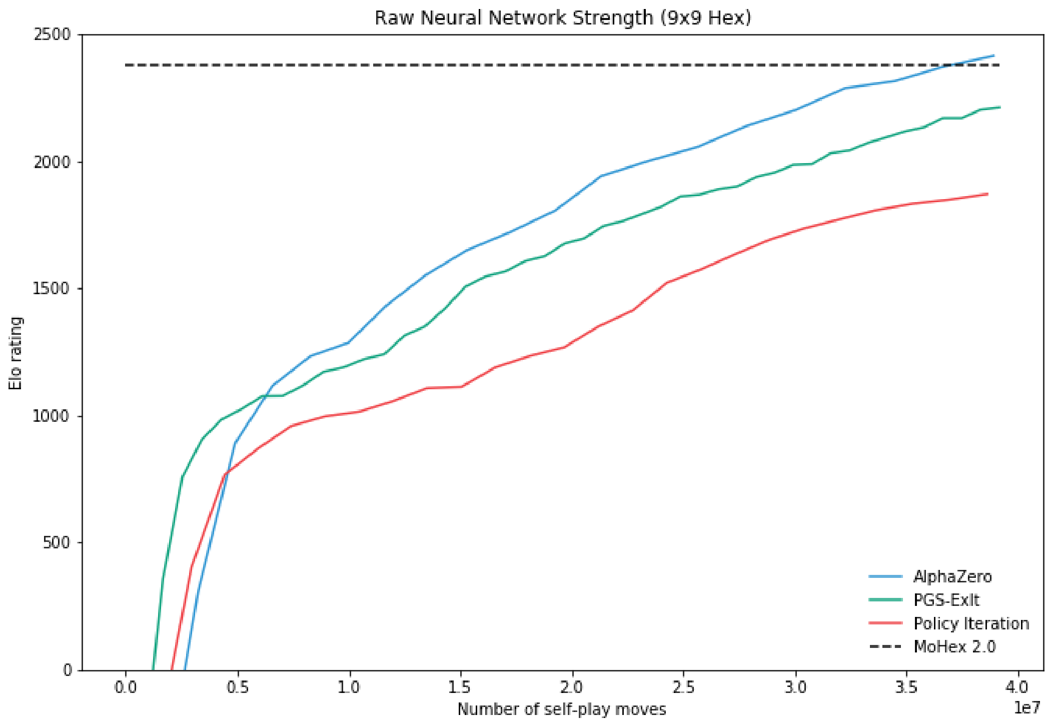

Figure 5.2: Strength of the raw neural networks throughout training with MCTS, MCS or PGS experts, calculated by a round robin tournament between the networks. Results for training in Expert Iteration are in line with results from test-time searching: using MCTS performs best, followed by PGS, while using MCS is much further behind. Data is from single training runs.

ran these experiments. [4]

Previous results suggest that better experts lead to better performance in ExIt, so we expect that with a better setting for this parameter and the KL regularisation, the performance of PGS-ExIt could be improved. Over the course of training, approximately 2 million games were played. In contrast, tuning this hyperparameter requires 2000 games; automatic tuning would not significantly increase the cost of the algorithm.

5.4.2 Performance against MoHex

We played a head-to-head match between MoHex 2.0, with 10,000 iterations, and PGS-ExIt with 800 iterations. Playing 4 games from each first move with each colour, **PGS-ExIt defeated MoHex 2.0 by** 375 **games to** 273**, 55**

[4]We did not repeat after identifying these improvements as the full expert iteration experiments were particularly expensive to run.

Elo stronger. The final agent trained with Policy Iteration lost to MoHex 2.0 by 540 games to 108, a gap of 280 Elo.

All previous competitive Hex agents have used explicit tree search algorithms at test-time, and many also use them during training (chapter 4), did not learn to play tabula rasa [Arneson et al., 2010, Huang et al., 2013, Takada et al., 2017], or both [Gao et al., 2017]; in this chapter we have presented the first competitive agents that entirely forgo both tree search and prior Hex knowledge.

5.5 Related Work

Temporal Difference Search (TDS) [Silver, 2009] is another search algorithm that employs a model-free RL algorithm for online search. Like PGS, it does not require an explicit search tree because it uses function approximators. Because TDS defines the simulation policy with value functions, it requires many more function evaluations per move than does PGS. This is okay with linear function approximators, as used by Silver [2009], but would not be feasible with the large neural networks used in this work. Graf and Platzner [2015] use policy gradient updates to improve a linear default policy of a classical MCTS Go program, this can be understood as a combination of PGS and MCTS. They show that this combination results in stronger play than applying MCTS with a fixed default policy.

PGS-ExIt decomposes the overall RL problem into many sub-problems, one per self-play game, and attempts to solve (or at least make progress on) each of the sub-problems with a model-free RL algorithm. The solutions to sub-problems are distilled back into a global network. Recent works on multi-task RL, such as Distral [Teh et al., 2017], follow a similar pattern. Divide and Conquer RL [Ghosh et al., 2018] also attempts to solve a single MDP by considering it as multiple different sub-problems, the division in Divide and Conquer RL is based on auxiliary context information, which determines which of a small set of sub-problems any given episode belongs to. In PGS-

ExIt, every episode effectively belongs to its own unique context, a more rudimentary technique which nonetheless has the advantage of requiring no additional information.

As discussed in section 2.9.7, other works have applied model-free Reinforcement Learning algorithms to train networks to play Hex, albeit with warmstarting using human knowledge. NeuroHex [Young et al., 2016] used deep Q-learning, while Gao et al. [2018a] proposed several variants of policy gradient algorithms for alternating move games. These networks are quite strong: they can win some games against MoHex 2.0 without test-time tree search, but they remain substantially weaker than it. Gao et al. [2018a] also showed that combining their neural network with MoHex 2.0 resulted in an tree search algorithm stronger than MoHex 2.0.

5.6 Conclusions

In this chapter, we have presented Policy Gradient Search, a search algorithm for online planning that does not require an explicit search tree. We have shown that PGS is an effective planning algorithm. In our tests it performed at a similar level to MCTS, and substantially outperformed the non-adaptive MCS, for test-time decision making in both 9×9 and 13×13 Hex.

It is also effective during training when used within the Expert Iteration framework, resulting in the first competitive Hex agent trained tabula rasa without use of a search tree. Of note, the policy gradient algorithm we used during individual searches is similar to the method that was unsuccessful in learning the game of Hex as a whole without Expert Iteration in section 4.4.3.

The results presented here are on the deterministic, discrete action space domain of Hex, this allowed for direct comparison to MCTS. But the most exciting potential applications of PGS are to problems where MCTS cannot be readily used, such as problems with stochastic state transitions or much larger action spaces with redundancies. We leave applying PGS and PGS-ExIt to such domains to future work.

Chapter 6

Discussion

6.1 Contributions

In this work we have presented a novel reinforcement learning family, Expert Iteration, which builds on previous policy iteration methods by using multi-step search to improve policies. We developed two examples of algorithms within the family, which used two different search algorithms, MCTS and PGS. We demonstrated that both algorithms were able to play the challenging classical board game Hex to a level exceeding that achieved by traditional heuristic search methods. We show through an ablation that the use of multi-step search improves the performance of ExIt over standard policy iteration methods; this ablation also suggested that policy iteration in general holds advantages in stability over model-free methods such as policy gradients. We explored some important design decisions for Expert Iteration algorithms, such as the choice of policy target, which data is used for training, and the choice of value target. Finally, the second search algorithm we used, Policy Gradient Search, is itself novel. It shows by example that model-free RL algorithms can themselves be applied as search methods, even when using function approximation, and can perform competitively with traditional methods such as MCTS. Since model-free methods can often be applicable to a wider class of domains, this holds promise for extending the problems to which ExIt can be applied.

This thesis tackled a single game, Hex. We believed that our generalist approach would mean that our results were likely to apply to many similar games; this has been borne out by other works — both concurrent and subsequent to ours — that had success with similar methods. Previously, achieving superhuman or state-of-the-art AI in interesting classical board games usually required large amounts of game strategy expertise in algorithm design, and large amounts of engineering to implement and test the resulting approach. It is now the case that for many classical 2-player games, given a sufficient computation budget, this approach could be outperformed by applying an Expert Iteration algorithm with well chosen but standard function approximators.

As well as a broad class of problems that can be effectively tackled by the methods in this work, this thesis has contributed a state-of-the-art deep reinforcement learning algorithm that differs in several ways to the model-free methods such as DQN and Actor-Critic methods that were most prominent before. As we will discuss in more detail in the next section, such an example affords a new opportunity to study the differences between reinforcement learning algorithms and advance our understanding of both reinforcement learning algorithms, and the reinforcement learning problems they tackle.

6.1.1 Limitations

Expert Iteration can solve highly challenging interesting problems, but several limitations remain. We discuss these here in terms of three categories. First, what the algorithm requires to be run on a new game, from the perspective of knowledge and resource provided by the AI's developer. Second, is the limitation of the method to the class of MDPs and AMGs, which means that it would not work on problems with different traits. Finally, some required properties of a game within the class of MDPs and AMGs for the method to work, or to confer an advantage over other approaches.

To apply Expert Iteration to a game, the developer must be able to provide an accurate simulator of the game to be used in search. In classical board games, the rules are known and so this is not a problem, but for application to a real world decision making problem, such as controlling a chemical plant, it may be a substantial hurdle. The algorithm makes heavy use of the simulator, so the simulator should also not be too expensive to query. The developer must also choose a representation of the state of the MDP, and a function approximator that can effectively map this to actions. Recent progress in deep learning has improved the function approximators available to be used, and Expert Iteration methods may find it easier to re-purpose these advances than other RL algorithms, because it casts RL as supervised learning. Nevertheless, it remains that good function approximation is necessary for the method to work. Finally, the developer must provide sufficient computation resource to run the ExIt algorithm: all examples in this thesis generated millions or tens of millions of moves, requiring billions of neural network evaluations; the high cost of training the algorithm could be prohibitive for some applications.

The class of reinforcement learning problems tackled, MDPs and perfect information two-player zero sum games with discrete action spaces, is quite broad. The ExIt algorithms we present are specialised to this class due to assumptions required by the search method. Characteristics which would place games outside the class we tackle include: imperfect information or partial observability; more than two players; general-sum rather than zero-sum payoffs; and continuous action spaces. When these are salient features of a reinforcement learning problem, the algorithms we developed may not be successful. New Expert Iteration algorithms, that use a different kinds of search, could extend the method to such games and are promising future work.

Finally, our algorithms have some implicit requirements on the dynamics of the game to succeed. When we start training from scratch, our first simulated games use an essentially random policy. In many settings, this policy contains some information about good play, and can be used to learn better than random play (albeit slowly, it can takes a large amount of training to reach the level of even a novice human). However, some games exist where such random play is totally uninformative, for instance, if it always leads to a draw. This would prevent our algorithms from learning anything at all.

Expert Iteration may not outperform single-step learning methods if several sequential game turns do not constitute a meaningful planning horizon. For example, if a game is presented in real time, with 40 timesteps per second, then a planner that looks 4 steps ahead still only plans for a tenth of a second. In constrast, board games are usually abstracted so that each turn is a meaningful unit of time for planning. Such problems are difficult for reinforcement learning in general, with techniques such as action repetition used to shorten the planning horizon [Mnih et al., 2015]. Such techniques might also be necessary to achieve good performance from ExIt, or any benefit over single-step learners. ExIt may also not outperform single-step methods in domains where the optimal policy is simple enough for single-step learning to solve the problem. It isn't that ExIt would fail to solve such problems, rather that it would do so at an unnecessary additional expense, as the searches are not needed.

6.2 Future Work

In chapter 3, we discussed motivations for the algorithm, particularly in terms of why the approach might outperform previous learning methods. We now revisit these ideas, in light of our results, and discuss where our results support our hypotheses, and where further research is needed to achieve a full understanding of how, why and when Expert Iteration works, and to determine the best methods for ExIt algorithms. We also highlight some problems that the Expert Iteration algorithms presented in this work would not be equipped to tackle, but to which future work might be able to extend the approach to other reinforcement learning problems.

6.2.1 Supervised Learning in Reinforcement Learning

Our ablation between multiple step Expert Iteration and policy iteration in section 4.5.4 suggested that the ideas around constructing supervised targets with low variance may indeed be important elements to achieving stability in self-play learning, since the policy iteration approach also achieved stable learning. This contrasts markedly with our experience with policy gradients in section 4.4.3, where we achieved some learning, but it was far from stable.

Other works have pointed to important differences in the loss landscapes of supervised and policy gradient learning, making similar observations to section 3.2.1. In particular, they have begun to offer explanations as to why the cross entropy losses favoured in supervised learning are more amenable to learning than the landscape of policy gradients. Chen et al. [2019] observe that surrogate losses outperform policy gradient estimates in a contextual-bandit setting, and show that the policy gradient loss landscape can contain many local optima. Ray Interference [Schaul et al., 2019] explains how policy gradient losses create large plateaus in loss landscapes, which do not appear for the cross entropy loss.

The use of entropy regularisers in policy gradient reinforcement learning has become standard practice in recent years [Mnih et al., 2016, Haarnoja et al., 2018]. This has proven an important technique, despite the fact that the solution to this regularised RL task is biased from the solution to the original task. One explanation may be that entropy regularisers can mitigate some of the drawbacks to using policy gradient losses [Ahmed et al., 2019]. These entropy regularisers may be having a similar effect to the supervised training regime of ExIt.

MPO [Abdolmaleki et al., 2018] is a model-free policy optimisation method that moves away from using policy gradient estimates altogether. Instead a non-parametric target policy is constructed from a Q-function, and then a parametric policy trained to minimise cross entropy to the target. This pattern is very similar to that used by classification-based policy iteration, including Expert Iteration.

6.2.2 Multiple Step Policy Improvement

Throughout this thesis we saw that algorithms using a multiple step policy improvement outperformed those that use or approximate a single step improvement, often by large margins. But we do not know what aspects of planning are bringing this advantage. Theory on the effects of multiple step improvements indicates there are risks with such operators [Efroni et al., 2018], so more work is needed to understand when and why they are effective.

A simple explanation might be that the multiple step improvement made it easier to efficiently deploy larger amounts of computation on the problem. In this view, single step policy iteration algorithm could reach the same level as Expert Iteration, if only trained for sufficiently long periods of time. Another explanation is that multiple step improvement operators change the way exploration takes place; if this is the case it may provide insights for improving model-free learning methods, or prescribe an increased focus on model-based techniques. A third possible explanation is that larger improvements allow expert iteration approaches to handle more lossy policy and value distillation, and so are essential to achieving strong final performance, particularly in domains where function approximation is challenging.

To establish which (if any) of these hypotheses are correct will require careful ablations, with multiple full training runs of ExIt-like algorithms. This was beyond the scope and computational budget of this work. However, by so doing we could gain a better understanding of which domains benefit from combining search and learning.

Compared to our first implementation, AlphaZero used the search policies to aid more aspects of solving the RL task: both the state distribution and value targets are from search-algorithm self-play. It is not yet known how important the use of search is for different parts of the algorithm, answering this through systematic study would be computationally intensive, but could also lead to improvements to Expert Iteration algorithms, and provide insights that allow us understand how similar advantages can be extended to other domains.

As one example, Wu [2019] accelerated learning (for a given compute budget) in AlphaZero by using shorter searches in most states of self-play games, with only a few states being searched for a longer period. Only the states that were searched for longer were used as training data. This is intermediate between AlphaZero, which used full searches in state sampling and value target self-play, and our approach in chapter 4, where the self-play used no search at all.

6.2.3 Extensions

Our methods are specific to MDPs and 2-player zero-sum AMGs, but as highlighted in section 6.1.1, many interesting problems fall outside this class. Extensions of ExIt to other kinds of reinforcement learning problem are likely to be fruitful directions for future research.

Hex has perfect information. In multiplayer games without perfect information, a planning algorithm must reason about the knowledge other agents have, and MCTS or PGS cease to be appropriate search strategies. Some work has already looked at extending ExIt to this setting [Kitchen and Benedetti, 2018, Zhang et al., 2019]. Even in single-player settings, partial observability can complicate planning.

Hex's action space is discrete and small enough to enumerate. Policy Gradient Search is a promising approach for search in games with even higher branching factor, but we have not tested it on such a domain in this work. Extending planning to such games would allow Expert Iteration to be applied in more real-world settings where the decision space can be combinatorially large or continuous.

Our Policy Gradient Search algorithm used a parametric function approximator to generalise between states during simulation based search. It may be interesting to consider whether a non-parametric function approximation scheme might outperform this. Such a scheme could be more complementary to the parametric imitation learning step in Expert Iteration. Yee et al. [2016] use kernel regression to generalise between actions at each node when searching over a continuous action space. More benefits could be realised by generalising between states, and by integrating prior knowledge from a neural network into the search. Expert Iteration itself gives new motivation to focusing on improving local search methods.

All algorithms presented in this work begin with random play, either in the form of self-play by (search on) a randomly initialised neural network, or with random rollouts inside MCTS. That such rollouts produce meaningful data was necessary for our algorithms to begin learning at all. Combining directed exploration strategies with our method would be necessary in scenarios where this is not the case.

A final assumption that we make use of throughout this thesis is that we have access to a perfect dynamics model, provided by the game rules. In real world problems, this is rarely the case. Instead, to apply Expert Iteration, a model would have to be learnt. This is an opportunity as well as a challenge: if the search could use a learnt model, it might be able perform exploration without incurring costs from pursuing an exploratory policy in the true environment, and might be more sample efficient in terms of steps in the true environment than a model-free algorithm ever could be. Some exciting recent works have looked into this model learning case [Sun et al., 2018, Schrittwieser et al., 2019]. Of particular note is that Schrittwieser et al. [2019] achieved state-of-the-art results for sample efficiency in the Atari domain.

These recent methods differ from previous model-based RL methods such as Dyna [Sutton, 1991] or PEGASUS [Ng and Jordan, 2000] that use the model to generate simulated experience, but then apply a single-step learning rules to that experience. This does not take advantage of the possibility of using search on the model for multi-step planning during learning, which, as we have shown in this thesis, can outperform single-step learning.

6.3 Conclusion

By eclipsing MoHex 2.0, the algorithms presented in this work have achieved something remarkable. MoHex was built over the course of many years, much of the effort was dedicated to approaches exploiting mathematical and logical properties of the game. In contrast, this work required little knowledge or understanding of the game; indeed neither I nor my collaborators are skilled players of Hein's game. We believe this to be a meaningful step towards artificial intelligences that solve real problems better than humans are able to.

However, as we highlighted in section 6.1.1, it is only a small step. Classical board games are in many ways the easiest case for AI: the rules are known; games can be cheaply simulated; the state is observed in full, with no imperfections; both state and actions are presented at an appropriate level of abstraction; the game is a purely adversarial contest between two players, allowing for assumptions about the intentions of the other agent in the contest. We now need to understand more fully how this approach was able to achieve success in Hex, so that we can achieve similar successes in ever more general classes of problems.

Appendix A

Statistical Methods

We detail here how the statistics presented in this thesis were calculated.

A.1 From Chapter 4

A.1.1 Chosen Action Targets and Tree Policy Targets

66 networks were trained with tree policy targets or chosen action targets, one network of each type for each of 66 independently sampled train/validation/test set tuples. Accuracies were calculated on test sets of 10,044 positions drawn from the same distribution as the training and validation data. Elo scores were calculated by BayesElo based on a round robin tournament of all 2×66 networks.

The confidence intervals reported in section 4.4.1 are 95% intervals based on a normal-distribution assumption (with unknown variance). The confidence intervals for the Elo scores are similarly calculated: this includes an assumption that the Elo scores calculated from the round robin can be treated as independent of one another

The correlations between top-1 accuracy and Elo scores are based on a linear regression.

In section 4.4.2 TPT and CAT were compared by playing head-to-head matches between MCTS algorithms using networks trained on the same dataset. An estimate of the Elo difference was calculated based on that match, and the mean and confidence interval on this margin were calculated based on a normal-distribution assumption (with unknown variance).

A.1.2 Tournament of Algorithms from Section 4.4.3

Elo scores in section 4.4.3 were calculated based on a round robin tournament between all neural networks checkpoints, with each neural network playing greedily.

In figure 4.5, confidence intervals are calculated using the same assumptions as previously. They are based on 5 independent training runs. Due to the high computational cost of each training run, elsewhere we present the results of single training runs. Nonetheless, the tight error margins on this experiment give us confidence that the comparisons we make between methods are representative.

A.1.3 Tournament of Distributed ExIt algorithms

Elo scores in section 4.4.4 are calculated from a tournament consisting the following matches:

1. Matches between greedy neural network players at every checkpoint and MoHex 1.0 with 10,000 iterations per turn and default settings

2. Matches between the expert players at every checkpoint and MoHex 1.0 with 10,000 iterations per turn and default settings

3. Matches between greedy neural network players at every checkpoint and the greedy neural network at checkpoints 17, 33, 49 and 63 for both training runs (i.e. every quarter of the way through training)

4. Matches between the expert players at every checkpoint and the greedy neural network at checkpoints 17, 33, 49 and 63 for both training runs (i.e. every quarter of the way through training)

This structure was chosen because MCTS players are expensive, and a full round robin would therefore have been prohibitively costly. The tournament design uses performance against MoHex 1.0 and the network checkpoints at each quarter through training as reference agents.

A.1.4 AlphaZero

The Elo ratings in 4.7 are based on a greedy network round robin tournament including the policy gradient search Expert Iteration presented in chapter 5, see section A.2.2.

The training losses reported in section 4.5.5, are the average losses on the last 1000 mini-batches sampled for training. Validation losses are the average loss on a validation set, calculated on the entire validation set from checkpoints taken every 1000 steps.

The plots of value prediction against length of game are taken from search self-play games that did not use resignation.

A.2 From Chapter 5

A.2.1 Search results

The Elo ratings of raw networks, MCS, MCTS and versions of PGS were calculated using BayesElo based on an independent tournament for each network. The tournaments were round robins, excluding matches between KL regularised PGS and full-network adaptation PGS. Error margins are as reported by BayesElo.

The results in figure 5.1 are based on a round robin tournament between the search algorithms and the raw neural network. The shaded region represents the margins according to BayesElo.

A.2.2 Expert Iteration

The Elo scores for the raw neural networks are calculated by BayesElo based on a round robin of checkpoints from the three training algorithms. Checkpoints were taken every 10,000 learner steps (batch size 1024). No confidence intervals are plotted since results are from single training runs. (BayesElo reports very tight margins as the tournament is large, however this represents the error in the measurement of skill of the particular training run, rather than the variation of performance across repetitions of the RL algorithm)

Appendix B

Tools

The original MCTS-ExIt was written in Python, Cython and C++. Neural Networks were implemented in Theano [Bergstra et al., 2010]. The re-implementation of AlphaZero and Policy Gradient Search were written in Python using Ray [Moritz et al., 2018] and Tensorflow [Abadi et al., 2016].

Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yu Yuan, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016.

Abbas Abdolmaleki, Jost Tobias Springenberg, Yuval Tassa, Remi Munos, Nicolas Heess, and Martin Riedmiller. Maximum a posteriori policy optimisation. *International Conference on Learning Representations*, 2018.

Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. Understanding the impact of entropy on policy optimization. In *International Conference on Machine Learning*, 2019.

Ron Amit, Ron Meir, and Kamil Ciosek. Discount factor as a regularizer in reinforcement learning. In *International Conference on Machine Learning*, 2020.

Vadim V Anshelevich. The game of hex: An automatic theorem proving approach to game programming. In *AAAI Conference on Artificial Intelligence*, 2000.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, 2017.

Thomas Anthony, Robert Nishihara, Philipp Moritz, Tim Salimans, and John Schulman. Policy gradient search: Online planning and expert iteration without search trees. *Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems*, 2018.

Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte Carlo tree search in hex. In *IEEE Transactions on Computational Intelligence and AI in Games*, 2010.

Devansh Arpit, Yingbo Zhou, Bhargava U Kota, and Venu Govindaraju. Normalization propagation: A parametric technique for removing internal covariate shift in deep networks. In *International Conference on Machine Learning*, 2016.

Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3, 2002.

David Balduzzi, Sebastien Racaniere, James Martens, Jakob Foerster, Karl Tuyls, and Thore Graepel. The mechanics of n-player differentiable games. In *International Conference on Machine Learning*, 2018.

Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3), 2000.

Donald F Beal and Martin C Smith. Learning piece values using temporal differences. *International Computer Games Association Journal*, 20(3), 1997.

Donald F Beal and Martin C Smith. Temporal difference learning applied to game playing and the results of application to shogi. *Theoretical Computer Science*, 252(1-2), 2001.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, 2013.

Richard Bellman. A Markovian decision process. *Journal of Mathematics and Mechanics*, 1957a.

Richard Bellman. Dynamic programming. *Princeton University Press*, 1957b.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Python for Scientific Computing Conference*, 2010.

Dimitri P Bertsekas. Approximate policy iteration: A survey and some new methods. *Journal of Control Theory and Applications*, 9(3), 2011.

Emile Borel. La théorie du jeu et les équations intégralesa noyau symétrique. *Comptes rendus de l'Académie des Sciences*, 173, 1921.

Cameron Browne. *Hex strategy.* AK Peters, 2000.

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

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2), 2002.

Christopher F Chabris and Eliot S Hearst. Visualization, pattern recognition, and forward search: Effects of playing speed and sight of the position on grandmaster chess errors. *Cognitive Science*, 27(4), 2003.

William G Chase and Herbert A Simon. Perception in chess. *Cognitive Psychology*, 4(1), 1973.

Guillaume M J-B Chaslot, Mark HM Winands, H Jaap Van Den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03), 2008.

Minmin Chen, Ramki Gummadi, Chris Harris, and Dale Schuurmans. Surrogate objectives for batch policy optimization in one-step decision making. In *Advances in Neural Information Processing Systems*, 2019.

Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. *International Conference on Learning Representations*, 2015.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units(ELUs). In *International Conference on Learning Representations*, 2016.

Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *International Conference on Learning and Intelligent Optimization*, 2011.

R Coulom. BayesElo. 2005. URL `http://remi.coulom.free.fr/Bayesian-Elo/`.

Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, 2006.

Hal Daumé, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine Learning*, 75(3), 2009.

Peter Dayan. The convergence of TD(λ) for general λ. *Machine learning*, 8 (3), 1992.

Adriaan D De Groot. *Thought and Choice in Chess.* De Gruyter, 1946.

Stefan Edelkamp. BDDs for minimal perfect hashing: Merging two state-space compression techniques. 2017.

Yonathan Efroni, Gal Dalal, Bruno Scherrer, and Shie Mannor. Multiple-step greedy policies in online and approximate reinforcement learning. In *Advances in Neural Information Processing Systems*, 2018.

Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. Fuego—an open-source framework for board games and go engine based on Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4), 2010.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. Impala: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, 2018.

Shane Frederick. Cognitive reflection and decision making. *Journal of Economic perspectives*, 19(4), 2005.

Chao Gao. *Search and Learning Algorithms for Two-Player Games with Application to the Game of Hex.* PhD thesis, University of Alberta, 2020.

Chao Gao, Ryan B Hayward, and Martin Müller. Move prediction using deep convolutional neural networks in hex. *IEEE Transactions on Games*, 2017.

Chao Gao, Martin Mueller, and Ryan Hayward. Adversarial policy gradient for alternating markov games. *Workshop track at International Conference on Learning Representations*, 2018a.

Chao Gao, Martin Müller, and Ryan Hayward. Three-head neural network architecture for Monte Carlo tree search. In *Internation Joint Conference on Artificial Intelligence*, 2018b.

Martin Gardiner. Mathematical games: Concerning the game of hex, which may be played on the tiles of the bathroom floor. *Scientific American*, 1957.

Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *International Conference on Machine Learning*, 2007.

Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for Monte-Carlo go. In *Neural Information Processing Systems Conference Online trading of Exploration and Exploitation Workshop*, 2006.

M Genesereth, N Love, and B Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2), 2005.

Dibya Ghosh, Avi Singh, Aravind Rajeswaran, Vikash Kumar, and Sergey Levine. Divide-and-conquer reinforcement learning. *International Conference on Learning Representations*, 2018.

Tobias Graf and Marco Platzner. Adaptive playouts in Monte-Carlo tree search with policy-gradient reinforcement learning. In *Advances in Computer Games*, 2015.

Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic. In *International Conference on Learning Representations*, 2017.

Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline Monte-Carlo tree search planning. In *Advances in Neural Information Processing Systems*, 2014.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

Ryan Hayward, Jakub Pawlewicz, Kei Takada, and Tony van der Valk. MoHex wins 2015 hex 11× 11 and hex 13× 13 tournaments. *Journal of International Computer Games Association*, 39(1), 2015.

Ryan B Hayward and Bjarne Toft. *Hex: The Full Story*. CRC Press, 2019.

Piet Hein. Vil de laere polygon. *Article in Politiken newspaper*, 1942.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Conference on Artificial Intelligence*, 2018.

Philip Thomas Henderson. *Playing and Solving the Game of Hex*. PhD thesis, University of Alberta, 2010.

Ronald A Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. MoHex 2.0: a pattern-based MCTS Hex player. In *International Conference on Computers and Games*, 2013.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, 2015.

Daniel Kahneman. *Thinking, Fast and Slow*. Macmillan, 2011.

Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *International Conference on Robotics and Automation*, 2011.

Michael J Kearns, Yishay Mansour, and Andrew Y Ng. Approximate planning in large POMDPs via reusable trajectories. In *Advances in Neural Information Processing Systems*, 2000.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Akihiro Kishimoto, Mark HM Winands, Martin Müller, and Jahn-Takeshi Saito. Game-tree search using proof numbers: The first twenty years. *Journal of the International Computer Games Association*, 35(3), 2012.

Andy Kitchen and Michela Benedetti. ExIt-OOS: Towards learning from planning in imperfect information games. *arXiv preprint arXiv:1808.10120*, 2018.

Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4), 1975.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, 2000.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Michail G Lagoudakis and Ronald Parr. Reinforcement learning as classification: Leveraging modern classifiers. In *International Conference on Machine Learning (ICML-03)*, 2003.

Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. Master's thesis, Imperial College London, 2015.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553), 2015.

Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, 2013.

Michael L Littman. *Algorithms for Sequential Decision Making.* PhD thesis, Brown University, 1996.

Christopher J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. *International Conference on Machine Learning*, 2015.

Peter Marbach and John N Tsitsiklis. Approximate gradient methods in policy-space optimization of markov reward processes. *Discrete Event Dynamic Systems*, 13(1), 2003.

Gábor Melis and Ryan Hayward. Six wins hex tournament. *Journal of the International Computer Games Association*, 26(4), 2003.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wiestra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540), 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 2016.

Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte carlo gradient estimation in machine learning. *Journal of Machine Learning Research*, 21(132), 2020.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *Symposium on Operating Systems Design and Implementation*, 2018.

Martin Müller. Computer go. *Artificial Intelligence*, 134(1-2), 2002.

John Nash. Some games and machines for playing them. *Rand Corporation Technical Report D-1164*, 1952.

John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1), 1928.

Andrew Y Ng and Michael I Jordan. PEGASUS: A policy search method for large MDPs and POMDPs. *Conference on Uncertainty in Artificial Intelligence*, 2000.

Andrew Y Ng, H Jin Kim, Michael I Jordan, Shankar Sastry, and Shiv Ballianda. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 16, 2003.

Jakub Pawlewicz and Ryan B Hayward. Scalable parallel DFPN search. In *International Conference on Computers and Games*, 2013.

Jakub Pawlewicz and Ryan B Hayward. Sibling conspiracy number search. In *Annual Symposium on Combinatorial Search*, 2015.

Jakub Pawlewicz, Ryan Hayward, Philip Henderson, and Broderick Arneson. Stronger virtual connections in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2), 2015.

Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225, 2006.

Jordan B Pollack and Alan D Blair. Why did TD-Gammon work? In *Advances in Neural Information Processing Systems*, 1997.

Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1), 1991.

Edward J Powley, Peter I Cowling, and Daniel Whitehouse. Information capture and reuse strategies in Monte Carlo tree search, with applications to games of hidden information. *Artificial Intelligence*, 217, 2014.

Doina Precup. Eligibility traces for off-policy policy evaluation. *Computer Science Department Faculty Publication Series*, 2000.

Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2, 1990.

Martin L Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24 (11), 1978.

Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Informatica*, 15(2), 1981.

Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3), 2011.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *International Conference on Artificial Intelligence and Statistics*, 2011.

Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3), 1959.

Jonathan Schaeffer, Markian Hlynka, and Vili Jussila. Temporal difference learning applied to a high-performance game-playing program. In *International Joint Conference on Artificial intelligence*, 2001.

Tom Schaul, Diana Borsa, Joseph Modayil, and Razvan Pascanu. Ray interference: a source of plateaus in deep reinforcement learning. *arXiv preprint arXiv:1904.11455*, 2019.

Bruno Scherrer, Mohammad Ghavamzadeh, Victor Gabillon, Boris Lesner, and Matthieu Geist. Approximate modified policy iteration and its application to the game of tetris. *Journal of Machine Learning Research*, 16:1629–1676, 2015.

Nicol N Schraudolph, Peter Dayan, and Terrence J Sejnowski. Temporal difference learning of position evaluation in the game of go. In *Advances in Neural Information Processing Systems*, 1994.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, 2015a.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015b.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Richard B Segal. On the scalability of parallel UCT. In *International Conference on Computers and Games*, 2010.

Nick Sephton, Peter I Cowling, Edward Powley, Daniel Whitehouse, and Nicholas H Slaven. Parallelization of information set monte carlo tree search. In *Congress on Evolutionary Computation*, 2014.

Claude E Shannon. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41 (314):256–275, 1950.

Claude E Shannon. Computers and automata. *Proceedings of the Institute of Radio Engineers*, 41(10), 1953.

David Silver. *Reinforcement learning and simulation-based search.* PhD thesis, University of Alberta, 2009.

David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In *International Conference on Machine Learning*, 2009.

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), 2016.

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 Thore Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676), 2017.

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), 2018.

Chiara F Sironi and Mark HM Winands. On-line parameter tuning for Monte-Carlo tree search in general game playing. In *Workshop on Computer Games*, 2017.

Wen Sun, Geoffrey J Gordon, Byron Boots, and J Andrew Bagnell. Dual policy iteration. In *Advances in Neural Information Processing Systems*, 2018.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1), 1988.

Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *Association for Computer Machinery Sigart Bulletin*, 2(4), 1991.

Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems*, 1996.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2011.

Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1), 2010.

Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. Reinforcement learning for creating evaluation function using convolutional neural network in hex. In *Conference on Technologies and Applications of Artificial Intelligence*, 2017.

Yee Teh, Victor Bapst, Wojciech M Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, 2017.

Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2), 1994.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the Association for Computer Machinery*, 38(3):58–68, 1995.

Gerald Tesauro and Gregory R Galperin. On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing Systems*, 1997.

Sebastian Thrun. Learning to play the game of chess. In *Advances in Neural Information Processing Systems*, 1995.

John Tromp and Gunnar Farnebäck. Combinatorics of go. In *International Conference on Computers and Games*, 2006.

John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *Transactions on Automatic Control*, 42(5), 1997.

Hado van Hasselt, Matteo Hessel, and John Aslanides. When to use parametric models in reinforcement learning? *Advances in Neural Information Processing Systems*, 2019.

Joel Veness, David Silver, Alan Blair, and William Uther. Bootstrapping from game tree search. In *Advances in Neural Information Processing Systems*, 2009.

Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3), 1992.

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

David J Wu. Accelerating self-play learning in go. *arXiv preprint arXiv:1902.10565*, 2019.

Timothy Yee, Viliam Lisỳ, Michael H Bowling, and S Kambhampati. Monte Carlo tree search in continuous action spaces with execution uncertainty. In *International Joint Conference on Artificial Intelligence*, 2016.

Kenny Young, Ryan Hayward, and Gautham Vasan. NeuroHex: A deep Q-learning hex agent. *arXiv preprint arXiv:1604.07097*, 2016.

Li Zhang, Wei Wang, Shijian Li, and Gang Pan. Monte Carlo neural fictitious self-play: Achieve approximate Nash equilibrium of imperfect-information games. *arXiv preprint arXiv:1903.09569*, 2019.