Machine Learning State Evaluation in Prismata

Rory Campbell

M.Sc. Thesis

Memorial University of Newfoundland

March 24, 2020

Abstract

Strategy games are a unique and interesting testbed for AI protocols due their complex rules and large state and action spaces. Recent work in game AI has shown that strong, robust AI agents can be created by combining existing techniques of deep learning and heuristic search. Heuristic search techniques typically make use of an evaluation function to judge the value of a game state, however these functions have historically been hand-coded by game experts. Recent results have shown that it is possible to use modern deep learning techniques to learn these evaluation functions, bypassing the need for expert knowledge.

In this thesis, we explore the implementation of this idea in Prismata, an online strategy game by Lunarch Studios. By generating game trace training data with existing state-of-the-art AI agents, we are able to use a Machine Learning (ML) approach to learn a new evaluation function. We trained several evaluation models with various parameters in order to compare prediction time with prediction accuracy. To evaluate the strength of our learned model, we ran a tournament between AI players which differ only in their state evaluation strategy. The results of this tournament demonstrate that our learned model when combined with the existing Prismata Hierarchical Portfolio Search system, produces a new AI agent which is able to defeat the previously strongest agents. A subset of the research presented in this thesis was the subject of a publication in the Artificial Intelligence and Interactive Digital Entertainment (AIIDE) 2019 Strategy Games Workshop [1].

Acknowledgements

The work in this thesis was supervised by Dr. David Churchill whose consistent mentor-ship proved invaluable in all stages of the research described. The other members of the AI and Games Lab were all instrumental in this work but, in particular, I acknowledge the help of Caroline Strickland and Richard Kelly, both of whom are tremendous researchers and cherished friends. I would like to thank my parents for their guidance and support and also my wonderful girlfriend Morgan, whose partnership and encouragement has meant so much to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research into Artificial Intelligence (AI) for games has passed numerous milestones, defeating the most advanced human players in classic turn-based boardgames such as Checkers [2] and, more recently, Go [3]. Strategy video games have become milestones for AI play thanks in part to their large state and action spaces and detailed rule-sets, which make them interesting test-beds for AI research. Many AI techniques have been applied to the challenge of playing video games at an expert level and this thesis attempts to implement some of these modern AI development strategies and test their success against other relevant approaches.

1.1 Motivation

1.1.1 Improvements to Gameplay

In many single-player and multi-player games, players frequently interact with non-player characters (NPCs), which perform various functions within the game world. In some games these NPCs play critical roles in a narrative, and a player's evaluation of these interactions can effect their perception of the quality of that narrative. In other games, the NPCs may be obstacles to the player in some way and the challenge they provide can become a critical detail in the player's perception of the game's systems. Interaction with AI in these ways can effect the critical and consumer reception of a game [4] and potentially have an impact on a game's commercial performance. Games with better overall AI systems are more fun and engaging for players, resulting in tangible effects such as better reviews and more game sales.

Strategy games such as *Starcraft* [1] and *Empire: Total War* [2] have been criticized for having AI which provided insufficient challenge, a feature which has on occasion been mitigated by allowing the AI to break game rules in an attempt to balance a competitive encounter. By developing AI agents which are more skilled opponents, players engaged in both the single and multiplayer modes would benefit.

[1] `https://starcraft.com/en-us/`
[2] `https://www.mobygames.com/game/empire-total-war`

1.1.2 Improvements in Game Development

Quality Assurance (QA) testing is a critical part of game development in which QA testers are hired to identify and report bugs to game developers for correction [5]. Many modern games publish content consistently after release and this schedule creates difficulties when the time-consuming process of QA testing must be undertaken with each additional batch of content. Games such as Rare's Sea of Thieves[3] have used AI to automate certain QA testing protocols [6]. A QA tester may be required to test the persistence of in-game boundaries, which may involve many attempts to break the boundaries by colliding with them with various trajectories and velocities, a task which is better suited to an AI testing protocol. By assigning some of the bug testing responsibility to automated testing, QA teams can be free to evaluate the quality of other aspects of the gameplay experience.

1.1.3 General Applicability

Work in artificial intelligence is a prominent component of modern research in self-driving cars [7], fraud detection [8], recommendation systems for popular destinations on the web [9], and, according to a study from The University of Hull, "Artificial intelligence techniques have the potential to be applied in almost every field of medicine" [10]. Strategy games serve as interesting test-beds for AI protocols due to the complexity resulting from their complicated rule-sets in conjunction with their large state and action spaces. By developing AI agents which

[3] https://www.seaofthieves.com/

can play strategy games at an advanced level, we are showcasing the new methods' ability to cope with these complex problems and are advancing the science of AI as a whole, potentially impacting our ability to solve AI problems in other fields.

1.2 Prismata AI

Prismata is a complex turn-based strategy game which features an amalgamation of game-play features found in other strategy games. Prismata's ruleset and complexity make it an interesting testbed for AI agents, which will be outlined in greater depth in Chapter 4. The current AI system for Prismata is based on a technique called Hierarchical Portfolio Search (HPS), introduced in [11]. **The main goal of this thesis is to improve the performance of HPS by using Machine Learning (ML) to replace components of the system that were crafted by expert knowledge**.

1.3 Thesis Outline

Chapter 2 of this thesis will discuss related work in the field of AI for video games, specifically in the field of heuristic search, followed in Chapter 3 by detailed outline of Prismata including an overview of its gameplay rules. Chapter 4 will give a description of the workings of its current AI agent, paying particular attention to state evaluation. **Chapter 5 introduces the original research contributions of this thesis** and we will go over the frameworks and general strategies involved in

training a model to perform state evaluation. Chapter 6 demonstrates our most relevant experiments and results pertaining mostly to observing the effects of key variable changes on the performance of our new state evaluation strategy. We will then make some concluding remarks in Chapter 7 focused mostly around potential directions for future work on this research.

Chapter 2

Background

2.1 Games as an AI Testbed

Games serve as a good testbed for AI research given their complicated rule-sets and large state and action spaces [12]. Many strategy board games have served as testbeds for AI development and the success of AI players against world champions has been a historic milestone in the development of these AI players. In 1992, Chinook, a checkers playing AI program was narrowly defeated by Dr. Marion Tinsley [2], widely considered to be the greatest checkers player in history [13]. A rematch took place in 1994 during which Dr. Tinsley had to withdraw due to health concerns and conceded his title to Chinook, which would go on to defend its title twice by 1996 [14]. Members of the team which developed Chinook would later go on to solve the game of checkers [15], which means that it achieved perfect play. In 1997, Deep Blue II, an AI chess player developed by IBM, successfully

defeated then world champion Gary Kasparov [16]. Deep Blue II was the culmination of many previous attempts at a world-class chess AI and was developed with the assistance of numerous chess Grandmasters [16].

Casino card games such as Poker have also been the subject of AI research, with researchers at the University of Alberta suggesting in 1998 that Poker served as a "better testbed for machine intelligence research related to decision making problems" [17]. AI research in Poker is currently ongoing; with AI players such as Libratus, developed at Carnegie Mellon University, defeating four top professional players in a tournament in 2017 [18].

More recently, the team at Google's DeepMind developed an AI Go player named AlphaGo, which defeated then world champion Lee Sedol [3], the process of which is the subject of a documentary [19]. DeepMind would go on to introduce AlphaZero, an AI player designed to function on a broader set of game rules, which demonstrated its effectiveness in both Chess and Shogi by defeating high-level AI players in both games and also defeating AlphaGo [20]. In the world of PC strategy games, StarCraft is one of the most prominent test beds of AI research. Although a StarCraft AI has yet to defeat a world champion, there are numerous active StarCraft AI tournaments, with SCAIT[1] and AIIDE[2] being of particular note. DeepMind has also been working on the development of a StarCraft bot, known as AlphaStar, [21], whose performance in the game of StarCraft II was the subject of a demonstration in early 2019[3]. AlphaStar makes alterations to the AI system

[1] https://sscaitournament.com/
[2] https://www.cs.mun.ca/~dchurchill/starcraftaicomp/
[3] https://www.youtube.com/watch?v=cUTMhmVh1qs

of AlphaGo and was ultimately able to defeat a professional player [22]. Dota 2, a popular multiplayer online battle arena (MOBA), has also served as a testbed for AI research. OpenAI Five, a Dota 2 player developed by OpenAI, became the first AI player to defeat the Dota 2 world champion team in a tournament held in 2019[4].

2.2 Heuristic Search and Games

2.2.1 Game Tree Search

The process of playing a game from start to finish can be represented as a sequence of *states* and *actions*. The state of a game is an instantaneous snapshot of the current game configuration (for example, the configuration of the pieces on a Chess board). Actions are performed by players which transition one game state to another (for example, moving a Chess piece and capturing an enemy). If we then consider all possible paths through a particular game, we can represent this collection of paths as a *game tree*, with nodes in the tree representing game states, and edges representing actions. Game tree search is the process of searching through a game tree to find the best action that a player can perform (within a given time limit). Heuristic search (applied to game trees) is the study of algorithms which are used to explore the game tree in an attempt to most efficiently find these optimal actions. Due to the large number of possible states for most interesting games, game trees are often too large to search completely to the end of the game (leaf nodes).

[4]https://openai.com/blog/openai-five-finals/

Search Strategies in Game AI

There are a variety of algorithms designed which can be used to search through the game states in a game's search tree. The minimax algorithm [23] can take a substantial amount of time to examine a game's search tree, time which can be saved with the help of an optimization like alpha-beta pruning. Alpha-beta pruning [23] is an added improvement to minimax which does not alter the end result but it speeds up the process by not exploring branches of the tree that can be shown to have no effect on the search result. Alpha-beta pruning has been used to play two-player strategy board games, in particular it has been used extensively in chess programs [24].

Heuristic search strategies combine search algorithms like minimax with heuristic optimizations and they have been applied heavily to game AI. Monte Carlo Tree Search (MCTS) [25] is a relatively recent heuristic search strategy which assigns values to the nodes in a game's search tree based on the outcome of playouts which involve random actions. The term playout describes an instance of a game which is played from some starting point until the game ends. Once the end of a playout is reached, meaning that the game has ended in one of win, loss, or draw, all nodes along the path to the end are given updated weights based on the outcome. MCTS has been used to tackle board games such as Chess [26] and Shogi [26]. As mentioned earlier, Chess AI often employs alpha-beta [27] but DeepMind's AlphaZero protocol employs MCTS in conjunction with Reinforcement Learning (RL) and was able to achieve better performance than the previously known strongest agents which were based on alpha-beta [26].

MCTS has been applied to computer strategy games in both commercial and research settings. The Total War series of strategy games has received criticism throughout its history for the perceived weaknesses in its AI opponents, which at one time used a protocol known as Goal Oriented Action Planning [28], which had proven to be successful in first-person shooters such as F.E.A.R. [29], which was praised for its strong AI [30]. For Total War: Rome II, the developers attempted to implement MCTS to improve the AI opponents' ability to coordinate its decision making across the game's numerous mechanics [31] and found that the AI agent took a great deal more time than players to make its decisions [32].

In academic research, MCTS has been applied with success to computer card games such as Hearthstone [33] and Magic: The Gathering [34]. However, since playouts are random in MCTS and the search trees are so large, even MCTS is not able to completely search the entire tree of many games. This was demonstrated at the General Video Game-AI Competition at the 2014 IEEE Computational Intelligence in Games, where AI players target general performance across multiple games and agents using MCTS experienced losses due to time constraints [35].

2.2.2 Heuristic Search in Prismata

Hierarchical Portfolio Search (HPS) [11] is the heuristic search algorithm which forms the basis of the AI protocols in Prismata. A detailed description of HPS can be found in Chapter 4. HPS was designed to explore the large search trees found in many strategy video games by generating a set of actions fewer than those in the game's entire search tree and iterating over this smaller set of possibilities. The

details of state evaluation will be discussed in Chapter 4 and will be the basis of the machine learning improvements proposed and tested in this thesis.

2.3 State Evaluation

Since none of the discussed search algorithms can search an entire game tree within a feasible amount of time, we must have a strategy for evaluating non-terminal leaf nodes, a process called state evaluation. Historically, state evaluation used heuristics which performed calculations based on features which were considered most important in determining the relative advantage of players in a gamestate. These heuristics were hand-coded by experts and relied on domain-specific knowledge meaning that they were both time consuming to develop and also needed to be built from scratch for each new game that the protocol was applied to. Examples of hand-coded heuristics can be found in historic chess-playing agents, such as the Greenblatt chess AI, which assigned value to the various chess pieces and used these evaluations to create variables which were considered critical in the game [27]. More recently, in work on Starcraft, hand-coded heuristics have been constructed based on the differential in valuable resources and incorporated into MCTS [36]. Prismata's strongest AI agent attempts to improve on hand-coded heuristics by using a technique known as symmetric playout, which will be discussed in our section on Prismata's existing AI players.

2.4 Learning State Evaluation

Machine Learning has been used as a state evaluation technique in general game playing AI as a means to incorporate experience as a means of learning, as opposed to using a fixed heuristic [37]. Similar work has been completed in research on Hearthstone[5], where MCTS is supplemented with a network trained on games between AI players to predict which player is likely to win from a certain state [38] and as a state evaluation strategy for AlphaGo [39]. **The idea of supplementing a search technique with supervised learning is one of the central motivating ideas for the research in this thesis**. The existing Prismata AI will be described in detail in Chapter 4 and in Chapter 5 we discuss using deep neural networks for state evaluation, which is the approach tested in this thesis.

[5] `https://playhearthstone.com/en-us/`

Chapter 3

Prismata

3.1 Game Description

Prismata is a strategy game developed by Lunarch Studios which combines "concepts from real-time strategy games, collectible card games, and table-top strategy games". Before the name Prismata was given to the game, its internal working title was MCDS, which stood for: “Magic the Gathering, Chess, Dominion, StarCraft"’, the four games which inspired its creation and from which its gameplay elements are borrowed. Before we discuss any AI system for Prismata, it is first necessary to understand its basic game rules and game theoretic properties. Full game rules available on the official Prismata website[1], but we will provide all necessary background here.

Prismata has the following properties:

[1] http://www.prismata.net

1. Oppositional Two Player: Although there is single-player content featuring puzzle challenges and a narrative, the focus of the game and of this research is its competitive 1 vs. 1 game mode, where players compete against each other. 1 vs. 1 is also available against a set of AI players provided by the game.

2. Deterministic - Every game features the same 11 base-set units in combination with a random set of 8 units which is generated at the beginning of the game. Each unit type has a fixed total supply for each player. After exhausting this supply amount, no further units of that type may be purchased. These units are added to the group of purchasable units and both players may purchase any of the available units. No further randomization takes place throughout the game, as Prismata has no decks which can be shuffled or mechanics through the game which incorporate chance.

3. Alternating Move: Players alternate turns under the restriction of a time limit within each turn, although the game features no overall time limit. Once the time limit for a player's turn is reached, the turn is passed to the opposing player with no additional penalty. A player may also choose to pass the turn at any time.

4. Zero Sum - All games in Prismata end in win, loss, or draw with a player winning after destroying all the enemy units or after the enemy forfeits.

5. Perfect Information - Prismata does not employ a fog of war like mechanic to obscure information from either player and no such mechanics can be

employed in the game. Players can observe all of their opponent's units, including both units on the board and those which are buyable in the future.

Players in Prismata each control a number of units, with each unit having a specific *type* such as a Drone or an Engineer, similar to most popular RTS games such as StarCraft which has unit types such as Marine or Zergling. Prismata units are used to build an economy for purchasing further units, produce resources, attack opponents, and defend incoming attacks. Units are divided into the 11 found in the base set and the 8 random units, which are selected from a pool of approximately 100. The base set units are described in Table 3.1, with two sample units described in Figure 3.1.

Also similar to RTS, players start the game with a few economic units which can produce resources for the player. These resources in turn can be spent on purchasing additional units which come in one of 3 main flavors: economic units (produce resources), aggressive units (can attack), or defensive units (can block incoming attack). The resources in Prismata consist of Gold and Green, which accumulate between turns, and Energy, Red, and Blue which are depleted at the end of each turn. Additionally, Prismata has an attack resource which is calculated based on the cumulative total of attack points put forth by a player.

Unit	Type	Start count	Supply	Cost	Build Time	Ability
Engineer	Blocker	2	20	2 gold	1	Gain 1 energy (p)
Drone	Blocker	6/7	20	3 gold, 1 energy	1	Gain 1 gold (c)
Conduit	N/A	0	10	4 gold	1	Gain 1 green (p)
Blastforge	N/A	0	10	5 gold	1	Gain 1 blue (p)
Animus	N/A	0	10	6 gold	1	Gain 2 red (p)
Forcefield	Blocker	0	20	1 gold, 1 green	0	Prompt
Gauss Cannon	N/A	0	10	6 gold, 1 green	1	Gain 1 energy
Wall	Blocker	0	10	5 gold, 1 blue	0	Prompt
Steelsplitter	Blocker	0	10	6 gold, 1 blue	1	Gain 1 attack (c)
Tarsier	N/A	0	10	4 gold, 1 red	2	Gain 1 attack (c)
Rhino	Blocker	0	10	5 gold, 1 red	0	Prompt, Gain 1 attack (c)

Table 3.1: Prismata's Base Set, where the letter in parentheses for abilities is either 'p' for a passive ability or 'c' for an ability that activates when the unit is clicked. The Prismata term prompt means that the unit is able to block immediately.

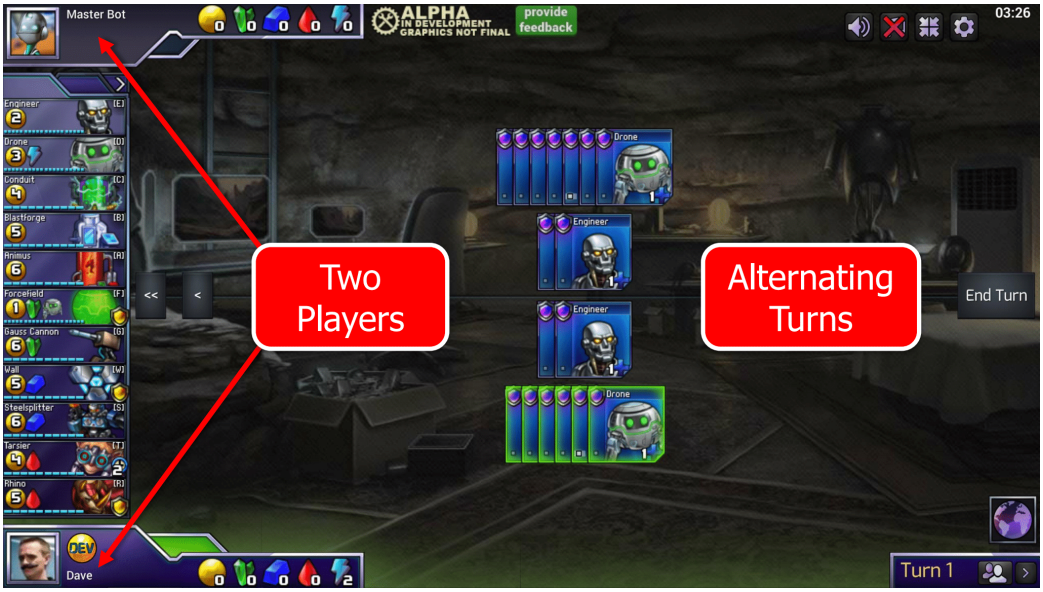
Figure 3.1: Two units from the Prismata Base Set. The cost of building these units is shown on the top right of their respective panels. The unit health is shown in the bottom right. Drones have a clickable ability, whose description we can read in the center.

Each turn of Prismata consists of a player performing a number of individual *Action*s, with a player's turn ending when they choose to stop acting and pass the turn (or if their turn timer expires). These player actions can be one of the following: purchase a unit of a given type, activate a unit ability, assigning incoming damage to a defending unit, assigning own damage to an enemy unit, or ending the current turn. Turns in Prismata are further broken down into an ordered series

of game *phases* in which only actions of a certain type can be performed: 1) defense phase - damage is assigned to defenders, 2) action phase - abilities of units are activated, 3) buy phase - new units are purchased, 4) breach phase - damage is assigned to enemy units. These phases are similar to other strategy games such as Magic: the Gathering's untap, upkeep, attack, and main phases, etc. A turn in Prismata therefore consists of an ordered sequence of individual actions, which in this thesis we will call a *Move*. Each player has their own pool of resources in Prismata, which are produced by unit actions. There are 6 resource types in Prismata: gold, energy, red, blue, green, and attack, which players can use in a variety of ways to perform actions such as the consumption of resources in order to purchase additional units or activate unit abilities.

A set of screen-shots demonstrating some basics of Prismata's board layout and select player actions can be seen in Figure 3.2.

Combat in Prismata consists of two main steps: Attacking and Blocking. Unlike games like Hearthstone, units do not specifically attack other units, instead a unit generates an amount of attack which is summed with all other attacking units into a single attack amount. Any amount of Attack generated by units during a player's turn must be assigned by the enemy to their defensive units (blocked) during the Defense phase of their next turn. When a defensive player chooses a blocker with h health to defend against a incoming attack: if $a \geq h$ the blocking unit is destroyed and the process repeats with $a - h$ remaining attack. If $a = 0$ or $a < h$ the blocking unit lives and the defense phase is complete. If a player generates more attack than their opponent can block, then all enemy blockers are



(a) The positioning of the two players on the Prismata board

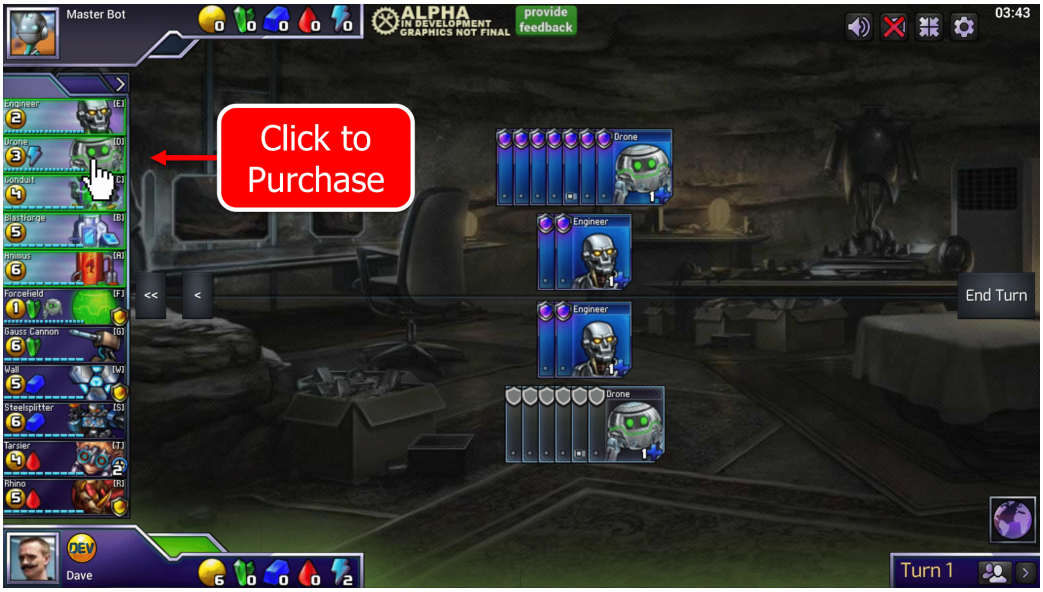
(b) The starting units of each player are seen here and are visible to both players

(c) The respective players' resource panels

(d) The player who currently has the turn can click units to use their abilities

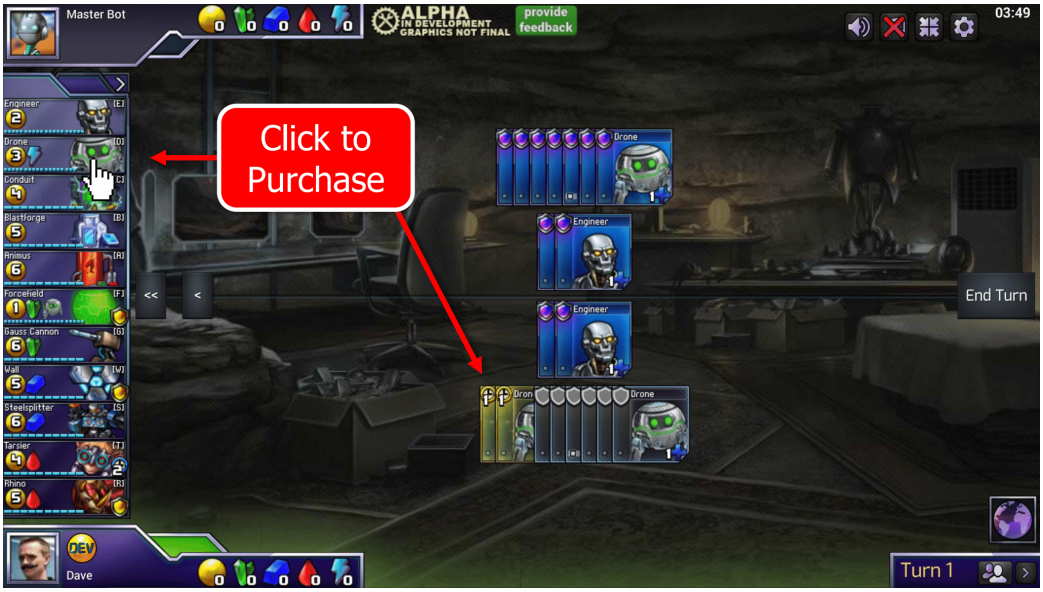
(e) The bottom player has clicked all 6 of their drones and used their ability. A drone gives the player 1 Gold when clicked, so the bottom player now has 6 Gold, as seen in their resource panel

(f) The left panel contains the shared buyable units. Visible here are the 11 base set units which are present in every game. Players can also click a button on this panel to see the shared buyable units which are not in the base set, and are randomly chosen at the beginning of the game



(g) Players can purchase units for which they have sufficient resources from the shared buyable pool by clicking on them. The cost of each unit is visible in the left panel

(h) The bottom player purchases a drone which costs them 3 Gold and 1 Energy, notice how their resource panel is decremented by this amount in the bottom left



(i) The bottom player still has more resources to spend

(j) The bottom player purchases another drone and is now out of resources to spend. They can click the button on the right to end their turn

Figure 3.2: A guided example of a basic set of Prismata actions

destroyed and the attacking player enters the Breach phase where remaining damage is assigned to enemy units of the attacker's choosing.

The main strategic decision making in Prismata involves deciding which units to purchase, and then how to most effectively perform combat with those units. The task of the AI system for Prismata is therefore to decide on the best move (ordered sequence of actions) to perform for any given turn. In the following chapter, we will introduce and discuss the AI system for Prismata.

Chapter 4

Prismata AI System

In this chapter we will introduce the AI challenges in Prismata, as well as introduce the existing AI system in Prismata which is based on a heuristic search technique known as Hierarchical Portfolio Search.

4.1 AI Challenges

Players in Prismata start with just a few units, and quickly grow armies that consist of dozens of units and resources by the middle and late game stages. Since a state of the game can consists of almost any conceivable combination of these units and resources, the state space of Prismata is exponentially large with respect to the number of purchasable units in the game. In order to gain a perspective on the size, consider that a game of Prismata features the 11 base set units and a random set of 8 units, which are selected from a pool of about 100, giving $\binom{100}{8} = 186087894300$

possibilities. There are 19 unit type options for each player, each of which has a total available supply, as described in Chapter 3, which, as we've seen, is 10 or 20 for every unit in the base set, but we will assume 10 for a conservative estimate in order to establish a lower bound. This means that there can be up to 10 units of each type in play so we have 10^{38} possible configurations of the units on the board once the random units have been selected, but we round up slightly to 10^{40} possibilities. Units also have properties that can vary during a game, such as variable health points or an active ability when clicked and although not every unit has one of these abilities, many units have more than 2 states, so we assume that each unit has 2 possible states, meaning that there are 2^{40} possible configurations of unit characteristics and abilities for the units in play and we assume that there are approximately 40 units in play. The following expression demonstrates an approximation for the lower bound of Prismata's state space.

$$186087894300 * 10^{40} * 2^{40} \approx 2 * 10^{63}$$

In addition to this, a player turn in Prismata consists of an ordered sequence of actions that are grouped into phases as described in our chapter detailing Prismata and each phase may consist of multiple actions which leads to a similarly exponential number of possible moves for a player on any given turn. These exponential state and action spaces pose challenges for existing search algorithms, which was why the Hierarchical Portfolio Search [11] algorithm was created by Dr. David Churchill specifically for the Prismata AI engine, and is explained in

the following chapter. The strategic difficulty of Prismata is not limited to its large state and action space and can be seen by examining a subset of its gameplay. In the defense phase of a turn, a player is responsible for distributing the attack points of the opponent amongst their own defenders, ideally in such a way as to minimize their loss. This process is equivalent to the bin-packing problem, in which a finite volume must be distributed among bins, each of which has some capacity. When the number of bins is known, as it is during the defense phase of Prismata, the problem is NP-complete. Such strategic decisions are a contributing factor to the difficulty of Prismata.

The Prismata AI also has no means of violating the game rules. In certain strategy games, the AI is allowed to cheat the game's system by building units it does not have resources for or playing cards it does not actually have in its hand. Given Prismata's perfect information, any attempts to circumvent the rules are easily spotted.

4.1.1 Hierarchical Portfolio Search

Hierarchical Portfolio Search (HPS) [11] is a heuristic search algorithm that forms the basis of Prismata's existing AI system, and was designed specifically to tackle searching in environments with exponentially large action spaces. The main contribution of HPS is that it significantly reduces the action space of the problem by searching only over smaller subsets of actions which are created via a *Portfolio*. This Portfolio is a collection of sub-algorithms created by AI programmers or designers, which each tackle specific sub-problems within the game. An example

Defense	Ability	Buy	Breach
Min Cost Loss	Attack All	Buy Attack	Breach Cost
Save Attackers	Leave Block	Buy Defense	Breach Attack
	Do Not Attack	Buy Econ	

Table 4.1: A sample portfolio for Prismata

Portfolio can be seen in Table 2 - since a turn in Prismata is broken down into 4 unique phases, the portfolio consists of sub-algorithms (called *Partial Players*) of varying complexity which are capable of making decisions for each of these phases. For example, the defense phase portion of the Portfolio has an exhaustive search which attempts to minimize the number of attackers lost when defending, the buy phase contains a greedy knapsack solver which attempts to purchase the most attacking units given a number of available resources, while the ability phase has a script which simply attacks with all available units.

Once the portfolio has been formed, the turn moves are generated by simply taking all possible permutations of the actions decided by the partial players in the portfolio. In the example in Table I, this would result in a total of $2 \times 3 \times 3 \times 2 = 36$ total moves. This process is shown in the GenerateChildren function on line 3 of Algorithm 1. The final step of HPS is to then apply a high-level search algorithm of your choosing (Minimax, MCTS, etc) to the moves generated by the portfolio. The full HPS algorithm is shown in Algorithm 1, with NegaMax, a variant of minimax, chosen as the search algorithm for its compact description. The Prismata AI's implementation uses Alpha Beta as its search algorithm, which is functionally identical to NegaMax.

Algorithm 1 Hierarchical Portfolio Search

```
1:  procedure HPS(State s, Portfolio p)
2:      return NegaMax(s,p,maxDepth)
3:  procedure GenerateChildren(State s, Portfolio p)
4:      m[] ← ∅
5:      for all move phases f in s do
6:          m[f] ← ∅
7:          for PartialPlayers pp in p[f] do
8:              m[f].add(pp(s))
9:          moves[] ← crossProduct(m[f] : move phase f)
10:         return ApplyMovesToState(moves,s)
11: procedure NegaMax(State s, Portfolio p, Depth d)
12:     if (D == 0) or s.isTerminal() then
13:         return Eval(s) ← state evaluation
14:     children[] ← GenerateChildren(s,p)
15:     bestVal ← −∞
16:     for all c in children do
17:         val ← −NegaMax(c,p,d-1)
18:         bestVal ← max(bestVal,val)
19:     return bestVal
```

4.2 State Evaluation

As with all challenging environments, the state space of Prismata is far too large for the game tree to be exhaustively searched, and therefore we need some method of evaluating non-terminal lead nodes in our search algorithm. Decades of AI research has shown that more accurate heuristic evaluation functions produce stronger the overall AI agents [40], so the construction of this function is vitally important to the strength of the AI. The call to the evaluation function in HPS can be seen on line 13 of Algorithm 1, for which any method of evaluating a state of Prismata can be used - as long as it returns a larger positive number for states which benefit the

player to move, and a larger negative number for states which benefit the enemy player.

As discussed in our review of the history of AI in strategy games, these evaluation functions have been mostly hand-coded by domain experts using knowledge of what may be important to a given game state. The original heuristic used for the Prismata AI system was done in this way - the resource values for each unit owned by each player were summed, and the player resource sum difference was calculated, with the player having the highest sum being viewed as in a favorable position. This type of evaluation is flawed, as it fails to take into account the strategic position that those units may be in - an incredibly important piece of information that is left out.

After experimental testing, a better method of evaluation for Prismata was found: game playouts [41]. A simple scripted AI agent was constructed (called a Playout Player) and was used to evaluate a state. From a given state, both players were controlled by the same playout player until the end of the game, with the intuitive notion that if the same policy controlled both players, then the resulting winner was probably in a favorable position. This method then returns a value for who won the playout game: 1 if the player to move won, -1 if the enemy player won, or 0 if the game was a draw. Even though this method of evaluation was approximately 100x slower than the previous formula-based evaluation, resulting in fewer nodes searched by HPS - the heuristic evaluation was so much more accurate that the resulting player was stronger, winning more than 65% of games with identical search time limits. In many games, this delicate balance between the speed

of the evaluation function and its prediction accuracy plays a vital role in overall playing strength, and the overall effectiveness of the evaluation function can only be measured by playing the AI agents against each other with similar decision time limits.

In the past few years, several world-class game AI agents have been created which have made use of machine learning techniques for evaluating game states. For example, the DeepStack [42] and AlphaGo [43] systems were able to use deep neural networks to predict the value of a state in the games of Poker and Go, respectively. In the following chapters, we will discuss the main contribution of this paper: **using deep neural networks to learn to predict the values of Prismata states, and using this to construct an AI agent which is stronger than the current system**.

4.3 AI Players

Prismata's AI system includes AI players varying in strategic capability. In terms of the difficulty adjustment which is available to players choosing to compete against an AI opponent, Prismata offers the following options:

1. Master: HPS agent with a 1000ms time limit
2. Expert: Chooses same portfolio as Master Bot but has a fixed alpha-beta search depth of 1
3. Medium: Chooses randomly from the same portfolio as Master Bot

4. Easy: Medium bot, with less advanced purchasing strategy

5. Random: Chooses random actions until turn ends

The Master Bot with playout state evaluation will sometimes be referred to as HPS_P, particularly when comparing it to agents that also employ HPS but with a different state evaluation strategy. We will use the term HPS_R to refer to the HPS agent which uses the hand-coded resource-based heuristic state evaluation used in a previous version of the Prismata AI. Each resource was given a value (relative to the base gold resource) by Will Ma, one of the game's original designers and programmers. The evaluation for a player was then simply the sum of the resource costs of all units that a given player owned. If one player has a higher resource cost sum, then they are considered to be at an advantage by this heuristic.

Chapter 5

Learning State Evaluation

In this chapter we will discuss how we learned a state evaluation model for Prismata. We will discuss the overall learning objectives, the methods used for gathering the training data, the techniques and models used to do the learning, the state representation used to encode the Prismata states, as well as the implementation details of all methods involved. The selection, implementation, and testing of the techniques described here represent the main contribution of this thesis.

5.1 Learning Objectives

Our objective in learning a game state evaluation is to construct a model which can predict the *value* of any given input state. In our case, the value of a state is correlated to who the winner of the game should be if both players play optimally from that state until the end of the game. For simplicity, we will define the output

for our model to be a single real-valued number in the range [-1, 1]. Ideally, we want our model to predict the value of 1 for a state which should be a definite win for the current player of a state, the value of -1 for a state which is a definite loss for the current player (win for the enemy), and a value of 0 to a state which is a definite draw, assuming perfect play on both sides.

Learning state evaluation has advantages and disadvantages over the evaluation techniques discussed previously. The main advantages of learned prediction are: 1) learning can occur automatically without the need to specifically construct evaluation functions or playout player scripts, and 2) theoretically one can learn to predict a much stronger evaluation than hand-coded methods if enough quality training data is given. The main disadvantages are: 1) if the game is changed (rules or unit properties modified in any way) then we may have to re-train our models from scratch, and 2) learning requires access to vast amounts of high quality training data, as we cannot learn to predict anything more accurately than the samples we are given to learn from.

5.2 Data Gathering

The previously listed disadvantage of obtaining high quality training data poses a unique problem for complex games. Unlike traditional supervised learning tasks such as classification, in which we are typically given access to data sets of inputs along with their correctly labeled ground-truth outputs, it is difficult to obtain who the absolute winner should be from a given state of a complex game. After all, if

we were able to determine the true winner from a given state, then the AI task of creating a strong agent would have already been solved. Therefore, the best that we can typically do is create a model to predict the outcome of a game played by the best known players available at any given time.

Historically, when performing initial supervised learning experiments, game AI researchers have turned to human game replay data as the benchmark for strong input data - the outcomes of those human games would be used as the target label which a learned model would attempt to predict. For example, Google DeepMind initially learned on human replay data for both its AlphaGo and AlphaStar [44] AI systems - since at the time of initial learning, human players had a far greater skill level than existing AI systems for those games. The same is also true for Prismata, in which expert human players can easily defeat the current AI even on its hardest difficulty settings. Therefore, our best option for learning would be to use these expert human replays for our training data, however we first need to determine: 1) if they are available for use, and 2) whether there are enough games to train the models.

Prismata saves every game ever played by human players, with approximately 3 million total replays currently existing. This number, however, is deceptive, as Prismata undergoes regular game balance patches every few months with major changes to unit properties. Learning to predict game outcomes on replays which contain units with different properties than the current version of the game could yield results which are no longer valid. For example, certain actions which could be performed on an older patch such as purchasing a unit for a given number of

resources may no longer even be legal with the same number of resources on the current patch. Another factor limiting the usability of these replays is the rank of the players in the game. Since we would only want to use replays of high ranked players, this would cut approximately 60-80% of the replays from the data set as not being of high enough quality to use for training. On top of this, there is also a technical reason why these human replays could not be used for the training data in this research: the format in which they are recorded. In order to save storage space, Prismata replays are not stored as a sequence of game state descriptions, but are instead stored as an ordered sequence of player actions. Each action is of the format (TimeCode, PlayerID, ActionTypeID, TargetID), where the TargetID indicates the unique instance id of a unit in the game, which is assigned by the game engine based on some complex internal mechanism. When a player views a replay in the official game client, the client is able to simulate these actions from the beginning of the game to recreate a game state and display it for the user. Unfortunately, this process of recreating the game state by the official game engine is not usable by us in a manner that would allow for these game states to be written to a file to be used as a training data set, and the construction of such a system would not be possible within the time available. Based on all of these factors, the human replays cannot be used as a training set at this time.

Since it is not practical to train a model based on the best available human replays, we instead train a model using the best available AI players. By playing an AI agent against itself, we can generate as many game state traces as are required, with the learning target being the eventual winner of that game. The AI agents used

for the generation of the test data are agents that currently exist in the Prismata AI engine, namely: *ExpertAI* and *Master Bot*. Both of these agents use an Alpha Beta search implementation of HPS with a symmetric playout state evaluation, with the difference being that ExpertAI does a fixed depth-1 search, while Master Bot searches as many nodes as it can in a 3 second iterative-deepening Alpha Beta search. The main idea here is that the current playout player used by Master Bot is a simple scripted agent, meant to be fast enough to be used by the heuristic evaluation within a search. If we can learn to predict the outcome of a Master Bot game for a given state, then we can effectively replace the playout player evaluation by a learned Master Bot evaluation, resulting in a much stronger evaluation function, which hopefully leads to a better overall agent. We can leave these AI agents to play against themselves and generate game traces for as long as we want, providing ample data for learning.

5.3 Learning Method

In recent years, the vast majority of machine learning breakthroughs in AI for games have come through the use of Deep Neural Networks (DNN). AlphaGo, AlphaGo Zero[1], AlphaStar, DeepStack, and OpenAI Five[2] each make use of DNNs in their learning. Therefore, we have chosen to use DNNs for our supervised learning task. The details of this network will be given in Chapter 6.

[1] `https://deepmind.com/research/alphago/`
[2] `https://openai.com/five/`

5.4 State Representation

Before we can actually learn anything, we must first decide on the structure of the input and output to our supervised learning task. As we are using DNNs for learning, it is advantageous to devise a binary representation for our game states, which is the preferred input format for successful learning in most modern DNNs. It remains for us now to create a function which, given a game state, translates it into a binary sequence for input into a DNN. Also, as this data will be used as input to a neural network, the state representation must be of uniform length regardless of the state of the game, which may vary considerably in number of units, resources, etc.

For many AI agents learned on games, such as those trained by DeepMind to play Atari 2600 games, in some cases defeating expert human players, the game was summarized visually, using the raw pixels of the game's graphical output [45]. This approach lends itself to learning with convolutional neural networks (CNN), a popular tool in developing strong game AI, used in research on games as complex as Starcraft II. Unlike Atari 2600 games and Starcraft, Prismata lacks meaningful geometry; unit placement is fixed and has no effect on gameplay, and so CNNs are not appropriate for our task.

Several state representation systems were tested over the course of this research and through experimental trial and error, we arrived at a representation which appears to provide a good balance between representing the strategic nuances of a state and the size / complexity of the DNN required to learn on it effectively. Since

we are using this network as an evaluation tool in a search algorithm, the feed-forward prediction speed of the network is of vital importance as it will be called possibly thousands of times per search episode.

5.4.1 Unit Types and Resources

The state representation we will be using captures 3 main features of the state: the current resource counts for each player, the current unit type counts for each player, and the current player to move in the state. This encoding discards information such as which units may be activated, individual unit instance properties, among many others, but since the states are all recorded at the beginning of each turn when units are not yet activated, much of the effect of this information loss is alleviated. Our final binary representation is as follows:

$$[P, U1_1...U1_n, R1_1...R1_m, U2_1...U2_n, R2_1...R2_m] \tag{5.1}$$

where P is the current player to move at the given state (0 or 1), UX_i is the current count of unit type i for player X, and RX_i is the current count of resource i for player X. These counts are stored as one-hot encodings of their associated integer values with a maximum length of 40, a 1 in the index corresponding to the count, and a 0 everywhere else.

5.4.2 Unit Isomorphisms and Resources

As discussed in our section on Prismata game rules, Prismata units of the same type do not always have the same properties. Some unit types, including some defensive units such as the Forcefield, have variable health which means that not all Forcefields have equivalent consequence in the game. Our unit type representation ignores this context entirely, grouping together all forcefields, regardless of their current remaining health. In order to more completely represent a gamestate, we developed a state representation which more completely summarizes the relevance of active units. In mathematics, an isomorphism is a relationship defined between two objects under which they are considered equal, even though they may not truly be the same. Items which are isomorphic to one another are said to be members of the same isomorphism class. Recall that Prismata players break down a turn into four phases: Defense, ability, buy, and breach. At the beginning of our defense phase, an engineer that is built and an engineer that is under construction are both defensively irrelevant, thus having the same strategic consequence for the player and are members of the same isomorphism class.

Instead of counting unit types, as in our first representation, we instead define isomorphism classes within each unit type and count those. Each unit type will be subdivided into multiple isomorphism classes which adds significantly to the length of our state representation. The number of isomorphism classes within a unit varies significantly. Friendly engineers for instance have only 1 isomorphism class, while enemy Asteri Cannons have 16. The current resource counts for each player and the current player to move in the state are still vital pieces of information

which must be included. The numerical component of our representation, one-hot encodings of integers, remains the same and the state representation is as follows:

$$[P, I1_1...I1_j, R1_1...R1_m, I2_1...I2_j, R2_1...R2_m] \tag{5.2}$$

Although this representation is certainly a more complete summary of the game, it is also substantially more verbose and the networks required to learn on the data proved to be too large to perform feed-forward inference rapidly enough to compete with our unit type representation. For the remainder of this paper, all tests will be done using the unit type and resources state representation in expression 5.1.

5.5 Tensorflow and Keras

Many open-source libraries currently exist for the building and training of deep neural network models. Due to its popularity, ease of use, and GPU support, we chose to use Tensorflow[3] for the research performed in this thesis. Tensorflow was developed by Google and has an abundance of both official and unofficial documentation. On top of Tensorflow, we are using a high-level API known as Keras[4] which has a python wrapper making the coding of our network very clean. TensorBoard[5] visualizations are a set of useful graphical tools for observing a model's structure and the status of its learning metrics provided to us by Tensorflow.

[3] https://www.tensorflow.org/
[4] https://keras.io/
[5] https://www.tensorflow.org/tensorboard

5.6 Inference on the Trained Model

Since the Prismata AI engine is written in C++, we need to perform inference on our trained model within that C++ system. As of writing, Keras has no official C++ libraries, and so we needed to use additional libraries to perform the inference with C++. This section discusses approaches that were not selected for this research but all considered methods are presented for completeness.

5.6.1 Tensorflow C++

Tensorflow does officially support C++ so if we convert our Keras model into a basic tensorflow model, we might be able to write C++ functionality for inference. We were able to convert our Keras model to a standard TensorFlow model using open-source code found on the web [6] , but overall this task proved to be too difficult as C++ tensorflow is less commonly used and much of the available documentation is now deprecated.

5.6.2 FLASK Application

Since we have a working model which was constructed, trained in, and inferred upon using python, it is possible to develop a web application with a pythonic web development service such as flask [7] which loads the model and accepts POST requests for inference. We were able to successfully develop a FLASK application

[6] http://bitbionic.com/2017/08/18/run-your-keras-models-in-c-tensorflow/
[7] http://flask.pocoo.org/

which loaded and performed feed-forward prediction with a trained model, but integrating this with the existing C++ Prismata code significantly altered the architecture of our experimental setup and this procedure did not reach the required level of functionality in time to compare it with our selected approach.

5.6.3 Frugally Deep

Frugally Deep[8] is an open-source, header only library designed specifically for calling Keras networks in a feed-forward capacity from C++ and was ultimately chosen as the basis for the experiments described in Chapter 6. The downside to this approach is that frugally deep performs all its computations on a single-core of CPU and does not support GPU computation at all. Unfortunately, we could not find any way to perform a GPU implementation of the feed-forward of our network inside C++ with the Prismata AI engine, therefore we believe the results found in the next chapter could be improved by a significant margin once we overcome this hurdle.

[8] `https://github.com/Dobiasd/frugally-deep`

Chapter 6

Experiments and Results

In this chapter we will give the details of the methods and experiments that were performed in order to construct, train, and test our novel state evaluation model.

6.1 Experimental Setup and Performance Metrics

In order to evaluate the overall quality of the new learned evaluation method, we ran tournaments of AI vs. AI games using AI players with several different settings. For the purpose of these experiments, our agent using learned state evaluation will be referred to as HPS_L. Recall the players defined in chapter 4, section 4.3, where HPS_R uses a hand-coded resource-based heuristic state evaluation, HPS_P uses playout state evaluation, and HPS_L uses our learned state evaluation. These three agents all use HPS with Alpha-Beta search with a time limit of 1000ms each, differing only in state evaluation. Since the only variable we are altering is state eval-

uation, we can be sure that any variation in performance is caused by the methods we are testing. Each player is given a score at the end of the tournament which serves as the metric of the player's success, the score formula being the number of wins + draws/2, such that a score of 0.5 indicates both AI agents are of similar playing strength, with a score higher than 0.5 indicating a winning average.

The performance of our learned player can be decomposed into two primary components: the structure of the neural network and the quality of the training data.

6.1.1 Network Structure

Network structure encompasses all the parameter choices made in developing the code for our model. For the purposes of recreating our results, our network used a flattened input layer with dimensions of (842,1), with the unit and resource count arrays described in section 5.4.2 as the input. Our experiments implemented a variable number of dense hidden layers with variable numbers of neurons in each layer. The neurons in these dense hidden layers contain ReLU activation functions. Our output layer was a dense layer with a single neuron, defined with a sigmoid activation function.

To design a model we must select a learning rate. A learning rate serves as a scaling factor for adjusting the weights in a neural network. The higher our learning rate, the more the weights are adjusted within the network in order to approximate a solution. If the learning rate is too high, the network may over-adjust the weights and miss a strong solution, a process referred to as overshooting. All the networks

tested use a conservative learning rate of $1 * 10^{-5}$ in order to mitigate the possibility of overshooting. To supplement the lower learning rate, our network also uses an Adam optimizer [46], which implements the procedures of Adaptive Gradient Algorithm (AdaGrad) [47] and Root Mean Square Propagation (RMSProp) [48], meaning that Adam will adjust the learning rate for us, a process which has been shown to outperform other automatic learning rate adjustment protocols.

Experiments 1 and 2 were designed to test the effect of network size on the quality of our learned agent.

Network size, for the purposes of our experiments, refers to the depth (number of layers) and the width (number of neurons per layer) of a network. As we lessen the size of the network, we are making the process of feed-forward inference faster and, by extension, allowing our search algorithm to explore more nodes in the search tree. However, increasing the size of the network can improve its accuracy, leading to a trade-off between speed and accuracy. The effect of a faster feed-forward inference is explored in Experiment 1, which demonstrates the effect of network size against the number of nodes an agent can evaluate in the game's search tree given a time constraint. Experiment 2 focuses on the relationship between network size and accuracy, by comparing the testing accuracy of models which vary in size. The networks examined in these experiments are identical in every way except size. We will also demonstrate the speed vs. accuracy trade-off.

6.1.2 Quality of Training Data

As discussed in Chapter 5, it is best to provide our neural network with data from games featuring the best available AI players, which in our case is Prismata's Master Bot. In order to test the effect that quality of training data has on ultimate tournament performance, we generated data from 200,000 AI vs. AI games featuring AI of various game skill levels, where AI players were played against AI players of their same difficulty level. Our learned models were then trained on this data. The effect of using training data from AI players of varying skill is one of the details explored in Experiment 4, as learned models trained on this data are entered into our AI vs. AI tournament.

6.2 Experiment 1: Evaluation Speed

To test the effect of network structure on speed of inference, we trained multiple networks with identical training data (Master Bot games) and identical network properties, with the exception of size. The metric for speed will be the number of nodes per second explored by a set of AI agents with the only difference between agents being the model which they are calling upon for state evaluation. It is also useful to measure these speeds against those of certain existing AI agents. The agents tested on the criteria of evaluation speed are the following:

1. HPS_R: HPS using a hand-crafted formula state evaluation function
2. HPS_P: HPS using a hand-crafted playout simulation for state evaluation. This method was the previously best existing AI for Prismata, and was used in the retail version of the game
3. HPS_L-2-64: Our trained neural network model feed forward prediction time (2 layers, 64 neurons per layer).
4. HPS_L-2-128: 2 layers, 128 neurons per layer.
5. HPS_L-2-256: 2 layers, 256 neurons per layer.
6. HPS_L-2-512: 2 layers, 512 neurons per layer.
7. HPS_L-2-1024: 2 layers, 1024 neurons per layer.
8. HPS_L-3-1024: 3 layers, 1024 neurons per layer.

Each evaluation model was used in a sample Alpha Beta HPS player with a one second time limit per turn. Figure 6.1 shows the results for how many evaluations

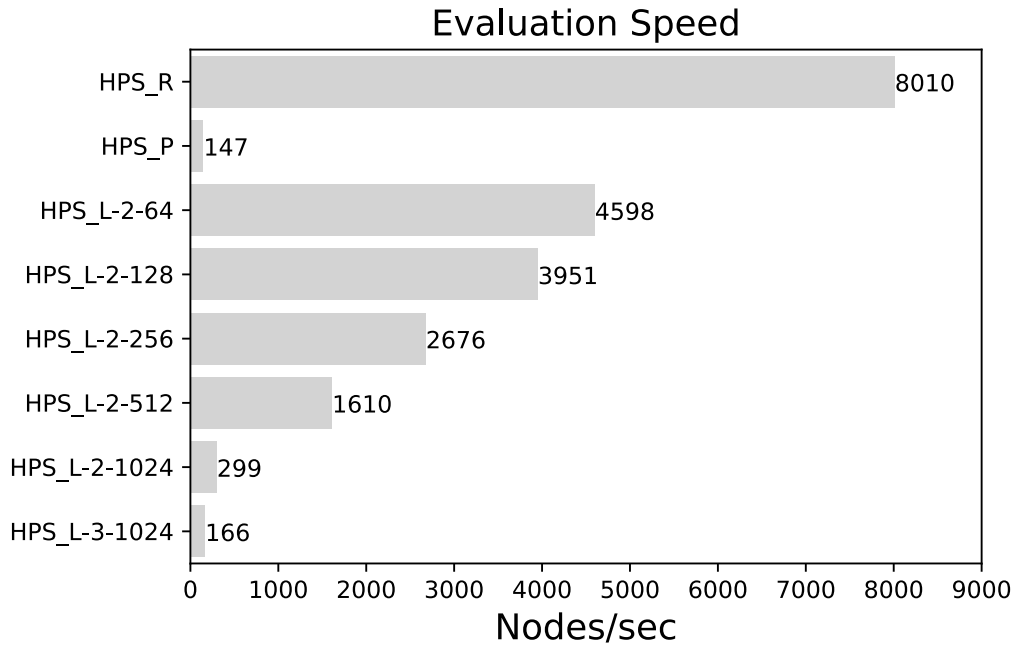


Figure 6.1: Evaluations per second of each method. HPS_R is the fastest, but also the least accurate. HPS_P represents the previous best-performing state evaluation method, which was also the slowest to calculate

were performed on average. From these results we can see that the Resource formula is by far the fastest evaluation, but it might later prove to be the least accurate. The Playout evaluation was the previous best evaluation method, but far slower than the resource heuristic. Our learned models lie in between these two in terms of speed, with speed decreasing as the model gets larger. If the accuracy of our learning is sufficiently high, these models would yield an overall stronger HPS player than with the other evaluation methods, as we will explore in the next section.

6.3 Experiment 2: Training Accuracy

Experiment 1 demonstrated that constructing a larger network reduces speed. Experiment 2 was designed to demonstrate how a larger network improves accuracy. These results are demonstrated here in Figure 6.2 by testing all the learned models from Experiment 1, visualizing their learning accuracy in the form of data recorded by Tensorboard which depicts learning over time and plotted using Matplotlib[1] and Seaborn[2], a data visualization library running on Matplotlib. HPS_R and HPS_P do not need to be examined here as they are not learning agents and do not have a training accuracy. The x-axis represents 1000 equally time-spaced accuracy reports, based on elapsed real time while the y-axis represents the percentage accuracy of training. The raw data could not be meaningfully displayed due to overlapping data points and so a regression of order 3 for each network architecture is shown, using built-in functionality in Seaborn. As we would expect, a larger network results in learning which is more accurate on average, increasing

[1] `https://matplotlib.org/`
[2] `https://seaborn.pydata.org/`

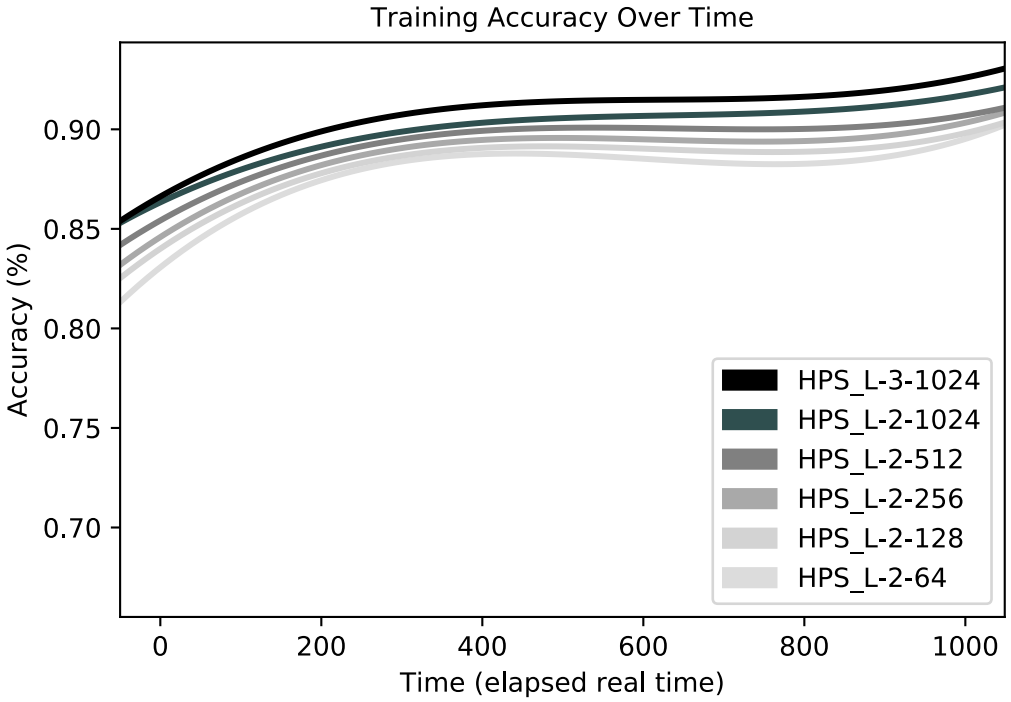


Figure 6.2: Training Accuracy of networks with varying width and depth over time

Network	Average Accuracy	Maximum Accuracy	Standard Deviation
HPS_L-2-64	0.88	0.91	0.01749
HPS_L-2-128	0.88	0.94	0.01667
HPS_L-2-256	0.89	0.92	0.01598
HPS_L-2-512	0.89	0.92	0.01522
HPS_L-2-1024	0.90	0.95	0.01564
HPS_L-3-1024	**0.91**	0.94	0.01600

Table 6.1: Average, Maximum, and Standard Deviation of Training Accuracies of Models of Varying Size. The largest and slowest network has the highest average accuracy, but the faster networks are only slightly behind on training accuracy

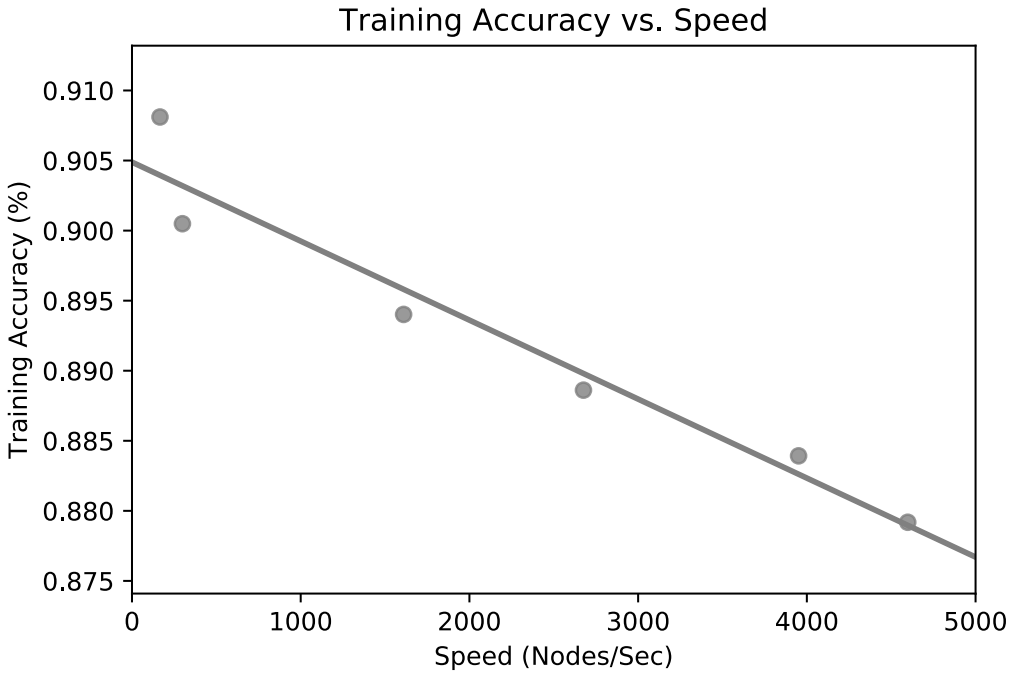


Figure 6.3: Tradeoff between training accuracy and speed. The y-axis measures training accuracy as a percentage against an x-axis measurement of nodes per second explored by our examined network architectures. The dots represent the mean training accuracies of each architecture and the line is a linear regression on these mean values

with each tested increase to size in both network width and network depth but the increases in performance appear to be minimal, while the results from Experiment 1 show vast time penalties in evaluation speed for the construction of larger networks. We can observe the tradeoff between accuracy and speed in Figure 6.3. This Seaborn visualization performs a linear regression on the mean training accuracies of each of our studied architectures from experiment one.

6.4 Experiment 3: Training Data Accuracy Impact

We trained state evaluation models using several training data sets obtained via playing various AI vs AI agents against each other. With any machine learning accuracy test, there are variables that we must account for: the input training set, and the target test set. In Figure 6.2 we showed training accuracy over time where the input training data was the same as the target output data. We wanted to also investigate how similar each of the learned models was, so we performed an experiment which varied both the input training set and target output set across all available data sets.

The results of this experiment can be seen in Table X, where table cell i,j represents the accuracy of the model when trained with data set i and tested against a target from data set j. As expected, the highest accuracy occurs when $i == j$, but we also notice relatively high accuracy values for very different training / target data sets. For example, a model trained with the Medium AI data set was able to predict the Master Bot target data set with 82% accuracy. Intuitively, this means that the models learned were quite similar, so the outcomes of the AI vs AI games that were played to generate this data must have also been quite similar. While this does not impact our final results, one potential benefit of this finding is that in the future, we could use faster running AI agents (such as Medium AI) to generate the model in far less time than slower running AI agents such as Master Bot, due to the high accuracy values.

Model/Dataset	Random	Easy	Medium	Expert	Master
Random	0.88	0.78	0.70	0.71	0.69
Easy	0.80	0.86	0.70	0.74	0.73
Medium	0.80	0.73	0.85	0.82	0.82
Expert	0.79	0.74	0.82	0.87	0.85
Master	0.78	0.74	0.81	0.84	0.89

Table 6.2: Trained Models tested on AI Games of Varying Quality. The value at row i, column j represents the testing accuracy of a model trained on data from *i* and tested on data from *j*.

HPS_R	Agent with hand-coded resource-based heuristic state evaluation
HPS_P	Agent with playout state evaluation
HPS_L_Mas	Our learned state evaluation on Master Bot data
HPS_L_Exp	Our learned state evaluation on Expert Bot data
HPS_L_Med	Our learned state evaluation on Medium Bot data
HPS_L_Easy	Our learned state evaluation on Easy Bot data
HPS_L_R	Our learned state evaluation on Random Bot data

Table 6.3: Players in the AI vs. AI tournament

6.5 Experiment 4: AI vs. AI Tournament

To test the effect of variable training data on ultimate tournament performance, we set up a round-robin tournament with the final score of each player being the score metric described in section 6.1. In order to speed up the tournament process, we placed all the players using any of our learned state evaluations in the same group so they would not have to play games against each other. The following players competed in the tournament of 13741 games: Note HPS_L_R, HPS_L_Easy, HPS_L_Med, and HPS_L_Exp all use the 2 layer, 64 neurons per layer structure which was found in experiment 1 to be the fastest network in terms of its ability to

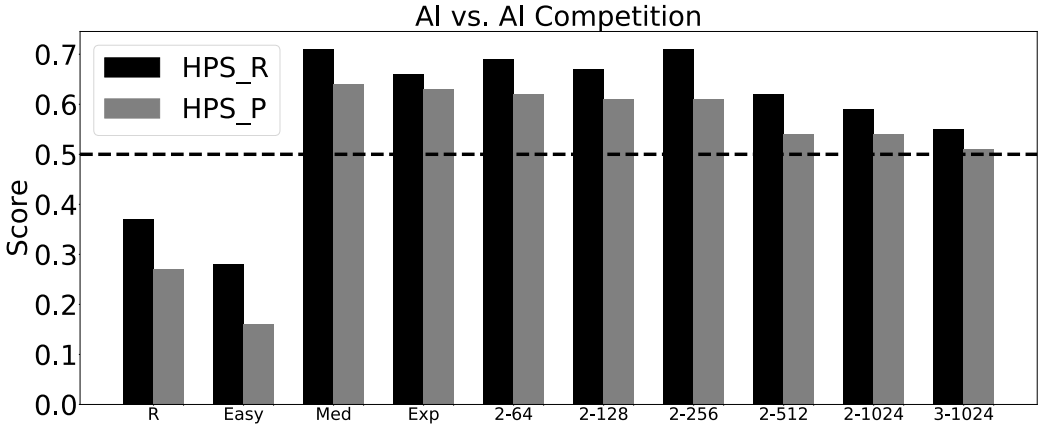


Figure 6.4: AI vs. AI Tournament Results. Each set of bars represents the success of a learned agent, labeled on the x-axis, against two previously developed Prismata AI agents, with HPS_P being the previous strongest bot. The y-axis represents the score of the learned agent against the respective existing AI agent, where score = number of wins + draws/2. A score of 0.5 indicates both AI agents are of similar playing strength, with a score higher than 0.5 indicating a winning average

Player	HPS_R	HPS_P
HPS_L_R	0.37	0.27
HPS_L_Easy	0.28	0.16
HPS_L_Med	**0.71**	**0.64**
HPS_L_Exp	**0.66**	**0.63**
HPS_L_Mas-2-64	**0.69**	**0.62**
HPS_L_Mas-2-128	**0.67**	**0.61**
HPS_L_Mas-2-256	**0.71**	**0.61**
HPS_L_Mas-2-512	**0.62**	**0.54**
HPS_L_Mas-2-1024	**0.59**	**0.54**
HPS_L_Mas-3-1024	**0.55**	**0.51**

Table 6.4: The numerical tournament results represented in Figure 6.4. A score > 0.5 means that our method for machine learning state evaluation outperforms the existing strongest Prismata AI agent, HPS_P. All such scores are represented in bold

visit more nodes in the search tree per second than our other learned agents. Figure 6.4 and Table 6.4 show the results of this tournament.

These results demonstrate that our new method for state evaluation outperforms the previous strongest state evaluation strategy. Generally, tournament performance seems to increase as quality of training data increases but this effect is minimal between Medium, Expert, and Master data although it is significant between Medium AI training data and Easy or Random AI training data. With a model learned on Master Bot data, performance tends to decrease as the size of the model increases, which is to be expected given the severe drop in nodes explored per second as demonstrated in Experiment 1.

Chapter 7

Conclusions and Future Work

In this thesis we have introduced a neural network model to learn state evaluations in Prismata. We trained this model on game traces generated by the existing best AI agent for the game: Master Bot, which uses Hierarchical Portfolio Search with a playout evaluation. Using state evaluation models trained on AI vs. AI games, we were able to produce a state evaluation strategy which was capable of evaluating up to 31 times more nodes per second as the previous state-of-the-art playout-based approach. This new model was faster even though our experiments were performed only with CPU computation for network predictions, which could be sped up even further in the future by utilizing GPU computations. A variety of network structures were tested and we demonstrated through experimentation that smaller networks had faster evaluation speeds, and were able to maintain a relatively high level of training accuracy compared to larger, slower networks. Finally, we played a AI agent tournament in which our newly proposed learned state

evaluation method was able to defeat the existing state-of-the-art Master Bot with playout evaluations in up to 71% of games. We can therefore conclude that this new method for learning state evaluations using deep neural networks resulted in an AI agent that played stronger than any previously existing agent, while also running faster, and relying on less hand-coded knowledge from the game's developers.

7.1 Iterative Improvement

In the future, we have several ideas on how to further increase the strength of the learned evaluation agent. First, we can continue to make improvements to both the network topology and state representation in order to produce a smaller, more accurate model, which will result in both more evaluations per second, and and overall better AI agent. Next, we believe that this process can be iterated: now that we have a stronger AI agent than the original Master Bot, we can train a model based on this new agent, which should produce a better overall evaluation function, which in turn should produce a better agent. We feel that eventually this may yield to diminishing returns, but it should work in the short term to produce a stronger agent overall. Lastly, we would like to improve our agent even further by learning of policies for the entire game of Prismata, not limiting ourselves to mere evaluation functions.

7.2 Card Buy Learning

Recall that in the sample portfolio presented in Table 4.1, there were 3 different heuristics for purchasing units during the buy phase of a Prismata turn, meaning that the action of buying multiplies the search space by three. In theory, we could train a model in much the same way we did for state evaluation but instead of learning the outcome of a game from a given state, we could learn the purchasing decisions of Master Bot from a given state and while combining heuristics to form a portfolio, we would not have to search over buying heuristics and the search could explore more nodes in the search tree, presuming that the feed-forward inference is faster than iterating over all possible combinations of incorporating the purchase heuristics.

The training data would be much the same as with our state evaluation problem, using the unit type count and resource state representation in a one-hot format, the only difference now being that instead of training our model on a set of binary win/loss labels, we would train it to predict an array of values which represent the cards which would be purchased by our most advanced AI agent.

We performed some preliminary tests on this question but difficulties emerged when the AI agent with learned purchasing behaviour was put to the test. The model was attempting to return an array of units to be purchased, which was one-hot encoded just as the state representation was and was often quite long. A single incorrectly predicted one-hot value could lead to a vastly different choice of which unit to purchase and advantage in Prismata is highly sensitive to mistakes, meaning

that an otherwise advanced agent could often be rendered useless.

Were the process of training a model to learn Master Bot's purchase decisions more successful it could, in theory, be extended to be a substitute for any other phase of the turn, greatly improving the speed of the search.

7.3 Deep Reinforcement Learning

Future work on Prismata could feature the development of an AI agent similar to those developed by DeepMind which uses Deep Reinforcement Learning, which is a reinforcement learning approach supported with deep neural networks, or some related deep learning tool. Such an agent could learn to actually play the game of Prismata instead of just performing state evaluation, by training a policy network similar to the one trained in AlphaGo or AlphaStar. While this would be a more complex problem to solve, we feel confident that it would yield a strong AI agent, due to the promising results found in this thesis.

Bibliography

[1] R. Campbell and D. Churchill, "Machine learning state evaluation in prismata," 2019. [Online]. Available: https://skatgame.net/mburo/aiide19ws/paper-3.pdf

[2] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, "A world championship caliber checkers program," *Artificial Intelligence*, vol. 53, no. 2-3, pp. 273–289, 1992.

[3] S. Borowiec, "Alphago seals 4-1 victory over go grandmaster lee sedol," *The Guardian*, vol. 15, 2016.

[4] R. McCaffrey, "Alien: Isolation review," https://ca.ign.com/articles/2014/10/03/alien-isolation-review, accessed: 2019-11-17.

[5] G. McAllister and G. R. White, "Video game development and user experience," in *Game user experience evaluation*. Springer, 2015, pp. 11–35.

[6] T. Thompson, "Automated testing for gameplay bugs | ai of sea of thieves (part 4)," https://aiandgames.com/seaofthieves4/, accessed: 2019-11-17.

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[8] A. Zakaryazad and E. Duman, "A profit-driven artificial neural network (ann) with applications to fraud detection and direct marketing," *Neurocomputing*, vol. 175, pp. 121–131, 2016.

[9] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*. ACM, 2016, pp. 191–198.

[10] A. Ramesh, C. Kambhampati, J. R. Monson, and P. Drew, "Artificial intelligence in medicine." *Annals of The Royal College of Surgeons of England*, vol. 86, no. 5, p. 334, 2004.

[11] D. Churchill and M. Buro, "Hierarchical portfolio search: Prismata's robust ai architecture for games with large search spaces," in *Proceedings of the Artificial Intelligence in Interactive Digital Entertainment Conference*, 2015.

[12] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *TCIAIG*, 2013. [Online]. Available: http://webdocs.cs.ualberta.ca/~cdavid/pdf/starcraft_survey.pdf

[13] J. Schaeffer, "Marion tinsley: Human perfection at checkers?" https://wylliedraughts.com/Tinsley.htm, accessed: 2019-11-17.

[14] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "Chinook the world man-machine checkers champion," *AI Magazine*, vol. 17, no. 1, pp. 21–21, 1996.

[15] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[16] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue," *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[17] D. Billings, D. Papp, J. Schaeffer, and D. Szafron, "Poker as a testbed for ai research," in *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer, 1998, pp. 228–238.

[18] N. Brown and T. Sandholm, "Superhuman ai for heads-up no-limit poker: Libratus beats top professionals," *Science*, vol. 359, no. 6374, pp. 418–424, 2018.

[19] "Alphago," https://www.imdb.com/title/tt6700846/, 2017, accessed: 2019-11-17.

[20] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[21] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, *et al.*, "Alphastar: Mastering the real-time strategy game starcraft ii," *DeepMind Blog*, 2019.

[22] K. Arulkumaran, A. Cully, and J. Togelius, "Alphastar: An evolutionary computation perspective," *arXiv preprint arXiv:1902.01724*, 2019.

[23] S. H. Fuller, J. G. Gaschnig, J. Gillogly, *et al.*, *Analysis of the alpha-beta pruning algorithm*. Department of Computer Science, Carnegie-Mellon University, 1973.

[24] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[25] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[26] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[27] R. D. Greenblatt, D. E. Eastlake, and S. D. Crocker, "The greenblatt chess program," in *Computer chess compendium*. Springer, 1988, pp. 56–66.

[28] T. Thompson, "Evolution of war the ai of total war part 2," https://www.gamasutra.com/blogs/TommyThompson/20180205/314171/Evolution_of_War__The_AI_of_Total_War_Part_2.php, accessed: 2020-03-24.

[29] J. Orkin, "Three states and a plan: the ai of fear," in *Game Developers Conference*, vol. 2006, 2006, p. 4.

[30] E. Brudvig, "F.e.a.r. review," https://https:ca.ign.com/articles/2006/10/25/fear-review-2, accessed: 2019-11-17.

[31] A. J. Champandard, "Monte-carlo tree search in total war: Rome ii's campaign ai," https://web.archive.org/web/20150302041541/https://aigamedev.com/open/coverage/mcts-rome-ii/, accessed: 2020-03-24.

[32] T. Thompson, "Revolutionary warfare the ai of total war part 3," https://www.gamasutra.com/blogs/TommyThompson/20180212/314399/Revolutionary_Warfare__The_AI_of_Total_War_Part_3.php.

[33] S. Zhang, "Improving collectible card game ai with heuristic search and machine learning techniques," 2017.

[34] C. D. Ward and P. I. Cowling, "Monte carlo search applied to card selection in magic: The gathering," in *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009, pp. 9–16.

[35] H. Park and K.-J. Kim, "Mcts with influence map for general video game playing," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2015, pp. 534–535.

[36] A. Uriarte and S. Ontanón, "Game-tree search over high-level game states in rts games," in *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2014.

[37] D. Michulke and M. Thielscher, "Neural networks for state evaluation in general game playing," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2009, pp. 95–110.

[38] M. Świechowski, T. Tajmajer, and A. Janusz, "Improving hearthstone ai by combining mcts and supervised learning algorithms," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018, pp. 1–8.

[39] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[40] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 11, no. 11, pp. 1203–1212, 1989.

[41] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for rts game combat scenarios," in *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[42] M. Moravčík, M. Schmid, N. Burch, V. Lisỳ, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, no. 6337, pp. 508–513, 2017.

[43] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, jan 2016.

[44] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, Y. Wu, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II," https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[46] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[47] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[48] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.