Machine Learning State Evaluation in Prismata

Anonymous

***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 paper, 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 learn a more accurate evaluation function which when combined with the existing Prismata Hierarchical Portfolio Search system, produces a new AI agent which is able to defeat the previously strongest agents.**

I. Introduction

Developing sophisticated AI for computer strategy games has historically presented many interesting challenges due to the complexity in their decision making. These games often have state and action spaces which rival and frequently exceed that of their strategy board-game ancestors, such as Chess and GO. The amount of possible actions available in strategy games may be so large that classic tree search strategies and even state-of-the-art techniques such as Monte Carlo Tree Search (MCTS) cannot be used in real time.

Lunarch Studio's turn-based strategy game Prismata implements a search technique called Hierarchical Portfolio Search (HPS), which reduces the action space of Prismata in an effort to mitigate the deficiencies of other searching algorithms. Even with the action space reduction, HPS is unable to search the entirety of the remaining tree and, as such, needs to perform an evaluation procedure on non-terminal leaf nodes which will be encountered at the end of a search. The process of evaluating leaf nodes, called state evaluation, was initially done using a hard-coded heuristic, but this heuristic was time-consuming to construct. It was found that a technique known as symmetric playout, which involved conducting games between players with identical policies from leaf nodes, produced better results than the hard-coded heuristic but developing players to perform these playouts was also a time-consuming process.

In this paper, in an effort to improve state evaluation in Prismata, we present a machine learning solution where, instead of playing out a full game from leaf node states, we perform inference on a neural network trained on data from games between the strongest available players. We test this approach against other advanced Prismata AI agents by entering them into a round-robin tournament and assigning them a score based on their win-rate. We conclude with a few possibilites for future research directions.

II. Prismata

A. 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. A more detailed description of the game theoretic properties of Prismata can be found in [1], with the full game rules available on the official Prismata website[1].

Prismata is a *Two Player*, *Deterministic*, *Alternating Move* game with *Perfect Information*. While Prismata does have some single player puzzle content, its main competitive form of play is one vs. one, in which players alternate taking turns with their opponent until the game is over, similar to traditional board games such as Chess or Checkers. Games can be played with or without a time limit, however unlike Chess the time limit is applied to individual turns instead of the full game. If the time limit expires for a given turn, instead of a game loss, the turn is simply passed immediately to the opponent. 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. 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 possible outcomes in Prismata are win, loss, or draw. Victory is achieved by destroying all of the units of the opposing player. Both players have access to all of the game information at all times, with all units and resources owned by both players being globally visible. There are no mechanics such as decks, hands, or "fog of war" to conceal the information of any player in the game.

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

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

Fig. 1: A sample opening state for a game of Prismata.

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 paper 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 screenshot of the beginning state of a game of Prismata can be seen in Figure 1. In this image, Player 1's units on the bottom half of of the game screen, with Player 2's units on the top half of the screen. A shared pool of buyable unit types is visible on the left hand side of the screen, with currently purchasable units highlighted in green. Each player's resources are visible at the far edge of their side of the screen. To activate the ability of a given unit, it is clicked with the mouse cursor. All actions can be undone (effects reversed) by clicking a second time on the same game element.

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 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 do decide on the best move (ordered sequence of actions) to perform for any given turn.

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

TABLE I: A sample portfolio for Prismata

III. Prismata AI System

In this section 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.

A. 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. A conservative lower bound of 10^{74} possible states was calculated for Prismata [1], with a much higher number being much more realistic. In addition to this, a player turn in Prismata consists of an ordered sequence of actions, leading 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 algorithm was created specifically for the Prismata AI engine, and is explained in the following section.

B. Hierarchical Portfolio Search

Hierarchical Portfolio Search (HPS) [1] 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 Portfolio can be seen in Table I - 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.

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
```

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 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.

C. 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. If we had an oracle that could determine the winner of a game from any given state then we would only need to search to depth 1 to determine the optimal action, however this is obviously not the case, and so we need to devise some custom function which can evaluate a state. Decades of AI research has shown that more accurate heuristic evaluation functions produce stronger the overall AI agents [2], 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.

Throughout the history of games AI research, these evaluation functions have been mostly hand-coded by domain experts using knowledge of what may or may not be important to a given game state. For example, early Chess evaluations involved heuristics such as assigning point values to piece types and board positions, and then summing those values for a given player. The original heuristic used for the Prismata AI system was similar to this - 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 however is flawed, as it fails to take into account the strategic position that those units may be in - an incredibly important part of the equation that is left out.

After experimental testing, a better method of evaluation for Prismata was found: game playouts. 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 [3] and AlphaGo [4] 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 sections, 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.

IV. Learning State Evaluation

In this section we will discuss how we learned a state evaluation model for Prismata. We will discuss the overall learning objectives we want to accomplish, 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.

A. 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]. 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.

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.

B. 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 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 - with the outcomes of those games as the target output values. For example, Google DeepMind initially learned on human replay data for both its AlphaGo and AlphaStar[5] 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 - 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 accurate. For example, certain actions which could be performed on an older patch such as purchasing a unit for a given number of a 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 paper: 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 rules. 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 be well outside the scope of this publication. 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.

C. 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[2], AlphaStar, DeepStack, and OpenAI Five[3] each make use of DNNs in their learning. Therefore, we have chosen to use

[2]https://deepmind.com/research/alphago/

[3]https://openai.com/five/

DNNs for our supervised learning task. The details of this network will be given in a future section.

D. 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 DNN 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 string 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. 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 will not be as important to use in our task.

Several state representation were tested over the course of this research, however describing each of them is outside the scope of this paper. Through experimental trial and error, we arrived at a representation which appears to be 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 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.

The final representation we used for our experiments encodes 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, etc, 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:

$$[P, U1_1...U1_n, R1_1...R1_m, U2_1...U2_n, R2_1...R2_m],$$

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.

E. Network Structure

We constructed a network with 2 hidden layers, with 512 neurons in each layer. We chose an initial learning rate of 0.00001, as opposed to the default of 0.001. Using a higher learning rate can lead to a process called overshooting which lessens the effectiveness of training a model. To supplement the lower learning rate, our network also uses an Adam optimizer, which implements the procedures of Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propogation (RMSProp), meaning that Adam will adjust the learning rate for us, a process which has been shown to outperform other automatic learning rate adjustment protocols.

F. Tensorflow and Keras

There are a variety of open-source machine learning libraries which we could rely on for building and training our model and we have chosen Tensorflow for this project. 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 which has a python wrapper making the construction of our network very clean from a coding perspective. Using Tensorflow also gives us access to the TensorBoard visualizations, a set of useful graphical tools for observing a model's structure and the status of it's learning metrics.

G. 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 need to use additional libraries to perform the inference with C++. Frugally Deep[4] is an open-source, header only library designed specifically for calling Keras networks in a feed-forward capacity from C++. 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 all results could easily be improved by a significant margin once we overcome this hurdle.

V. Experiments and Results

A. Network Training

In order to train out network, data was generated from 500,000 games of Master Bot vs. Master Bot, with the game settings of Base Set units + 8 random units at the start of each game. With a typical game length of a approximately 30 turns, and one sample generated each turn of each game, this resulted in 15,090,199 training samples for our neural net in total. Figure 2 shows the learning of the model over time generated in gnuplot from a csv recorded using TensorBoard[5]. The grey data points represent all recorded accuracy metrics with a line of best fit drawn in red. The x-axis represents the 1000 equally time-spaced accuracy reports from TensorBoard, with a total training time of 18 hours 41 minutes.

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

[5]https://www.tensorflow.org/guide/summaries_and_tensorboard

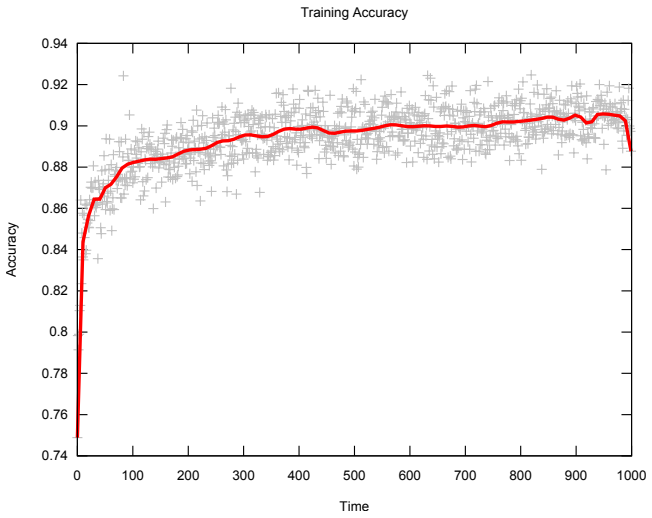

Fig. 2: Training accuracy over time

Evaluation Method	Eval / Sec
Resource	8010
Playout	147
Learned	1766

TABLE II: Evaluations per second of each method

B. Experiment 1: Evaluation Speed

The first experiment we performed was a simple test of the relative speed of each of the state evaluation methods available. Three evaluation methods were tested:

1) Resource: The hard-coded resource-based heuristic state evaluation used in the original version of the Prismata AI. Each resource was given a value (relative to the base gold resource) by one of the game's original designers and programmers Will Ma. 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.
2) Playout: Evaluation done by symmetric playout of the current game state to the end of the game.
3) Learned: Our trained neural network model feed forward prediction time.

Each evaluation model was used in a sample Alpha Beta HPS player with a one second time limit. Table II shows the results for how much evaluations were performed on average in each one second turn for each evaluation. From these results we can see that the Resoruce formula is by far the fastest evaluation, but also probably the least accurate. The Playout evaluation was the previous best evaluation method, but far slower than the resource heuristic. The Learned model lies in between these two in terms of speed, and if the accuracy is sufficiently high, should yield an overall stronger HPS player than with the other evaluation methods.

Opponent	Score
Resource	0.664
Playout	0.588

TABLE III: Shown is the score of an AI agent using the Learned evaluation function vs. agents using the Resource and Playout evaluation methods.

C. Experiment 2: AI vs. AI

In order to evaluate the overall quality of the new learned evaluation method, we ran a tournament of AI vs. AI games using AI players with several different settings. A total of 3 AI players were tested, each using HPS with Alpha-Beta search with a time limit of 1000ms each, each of which using one of the evaluation methods: Resource, Playout, or Learned. In total, 12800 games (6400 per matchup) were played, with the resulting scores displayed in Table III. The score formula was simply 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 the Learned player had a winning average against the other player.

Table III shows the score for the Learned evaluation vs. both the Resource evaluation and the Playout evaluation. Our new Learned evaluation had a win rate higher than 50% vs. the previous best evaluation methods, indicating that the new evaluation was indeed stronger overall than the previous best.

VI. Conclusions and Future Work

In this paper we have introduced a neural network model for learning to predict state evaluations in Prismata, an online strategy game. 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. By using the state evaluation model trained on these Master Bot games, we were able to use it to replace the previous Master Bot evaluation function, and produce an overall stronger AI agent than had previously existed.

In the future, we have several ideas 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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.