ChokoZero

Eugene Francisco
Department of Mathematics
Stanford University
eugenef@stanford.edu

Bodo Wirth
Department of Mathematics
Stanford University
bodow@stanford.edu

Naveen Kannan
Department of Mathematics
Stanford University
naveenkc@stanford.edu

Abstract

This project investigates the application of Proximal Policy Optimization (PPO) to train an agent at learning the two-player board game Choko. Choko is a board game played on a five-by-five grid in alternating turns between two players. The game, popular in The Gambia, has a rich history dating back over two hundred years. While there is an extensive body of research that tackles the use of deep reinforcement learning (RL) for the purpose of two-player board games like Chess, Checkers, and Go, Choko presents a unique challenge because of its dynamic rule set that combines placement phases and capture phases throughout the entire course of the game. Despite its strategic depth and longstanding history, Choko has remained virtually unexplored in the context of deep reinforcement learning. We applied both Deep Q-Learning (DQL) and PPO for the task of learning Choko. Using the actor and critic from PPO, we implemented a Policy Guided Tree Search (PGTS) which used the actor's action choices and the critic's value estimates to search for optimal actions. These methods achieved high performing results, with a search algorithm that acted orders of magnitude faster than minimax agents with comparable performance and comparable depth. Our ultimate PGTS fairs competitively against human players and outperformed all the other deep RL agents we trained, beating most of them in a majority of matches.

1 Introduction

Deep RL has achieved remarkable success in the realm of two-player, perfect-information games, establishing itself as the most powerful framework for teaching computers how to play complex strategic games. Perhaps most famously, DeepMind's development of AlphaGo, which used deep neural networks to estimate win probabilities and generate high-value moves, defeated world champion Go player Lee Sedol in 2016 (Silver et al., 2016). Building on the work of AlphaGo, AlphaZero developed a unified architecture for training deep RL agents at strategic board games, ultimately surpassing state-of-the-art performance in Chess, Shogi, and Go, without relying on any human feedback(Silver et al., 2018). These advances, especially the capabilities of deep RL agents at learning complex strategic environments only through self-play, made us curious and optimistic about the applications of these methods to a new board game like Choko.

1.1 Rules of Choko

Choko is played on a 5 by 5 grid between two players in alternating turns. The board begins empty and each player is given 12 pieces, either blue or red to denote piece ownership. Blue has the first move and may place any one of their pieces anywhere on the board. Because Blue is the first to move, Blue receives *drop initiative*, which means that as long as Blue keeps on placing pieces on the board, Red is forced to also place a piece on the board. During any one of Blue's turns, Blue may choose to move any blue piece on the board, at which point Red is allowed to either place a red piece or move an existing red piece, and the *drop initiative* is removed from Blue. Should either player decide to

place a piece again, that player receives the *drop initiative*, forcing the other player to place pieces until the *drop initiative* is removed.

Movement is orthogonal and pieces can only be moved to empty spaces. To capture pieces, players can jump over opponent's pieces. If, say, Blue is capturing a red piece, Blue's piece must be directly adjacent to the intended red capture piece and the landing zone opposite Red's captured piece must be clear for the jump. Similarly if Red was capturing a blue piece. After a jump, the player performing the capture may capture any other opponent's piece from the board, at which point their turn ends. Jumps are counted as a type of movement, so they may only be executed if the drop initiative is removed or by the player who has the drop initiative. Captured pieces are completely eliminated from the game.

We say the game is in a *drop-phase* if some player owns the drop initiative, and that the game is in a *capture phase* if neither player has the drop initiative. The game ends when one player captures all the opponents' pieces, in which case the capturing player wins. The game draws if one player is not able to move (though we include several extra draw conditions in our implementation so that games don't run too long).

1.2 Novelty and Problem Statement

The combination of drop-phases and capture-phases allows for some incredibly unique strategic positioning. For example, Blue's piece placement may be vulnerable to captures from Red but as long as Blue retains the drop initiative, Red won't be able to capture until Blue moves for the first time, or one player runs out of pieces to place. In practice, this has the effect of splitting many games into a drop-phase dominated opening where players compete for defensive positioning and then a capture-phase dominated end game where players compete in alternating skirmishes.

Through our exploration of various deep RL algorithms applied the game of Choko, we aim to create an agent capable of not just playing coherently, but of revealing novel, non-obvious strategies that go beyond the surface level and reveal deeper strategic insight.

2 Related Work

Proximal Policy Optimization (PPO) (Schulman et al., 2017) serves as the primary reinforcement learning algorithm in our project. We adopt PPO for its stable and efficient on-policy updates, and use an actor–critic architecture with convolutional encoders suited to Choko's spatial board layout. PPO has proven effective across a wide range of tasks, including board games, and provides a reliable foundation for learning in environments with complex strategic structure. All training data is generated through self-play, allowing the agent to improve iteratively without expert supervision.

As a value-based comparison, we also implement Deep Q-Learning (DQL), which estimates action values through temporal-difference updates (Mnih et al., 2015). DQL has seen success in large discrete action spaces, most notably in Atari games, and offers a contrasting approach to policy-gradient methods. In our experiments, DQL provides a baseline for learning dynamics in Choko, particularly during the early drop phases when board control and piece placement dominate.

To improve action selection during evaluation, we were inspired by Monte Carlo Tree Search (MCTS), using the trained PPO network to supply prior probabilities and value estimates. MCTS has been central to many of the strongest game-playing systems, including AlphaZero (Silver et al., 2018), where it complements network guidance with deeper lookahead. As such, we implemented a naive Policy Guided Tree Search which searches for the optimal action in a set of actor-selected rollouts (see Section 3.6).

3 Method

3.1 Environment Description

For use in all of our deep RL experiments, we implemented our own custom OpenAI Gym-style environment for our agent to interact with the board. For both of these experiments, we used the same state and reward reprentations. Observations s_t consisted of a flatenned 25 element long vector

where each entry represents the state of the corresponding tile on the board (empty, blue, or red). We gave agents a reward of $r_t = 0.1$ for each piece they captured, and a final reward of $r_t = 2$ if the agent wins or $r_t = -2$ if the agent loses (and a 0 reward if the agent draws). The draw conditions were slightly extended in this implementation to include any games which exceed 100 action in total (50 actions per player). The action space had 2625 elements and as such our action was represented by an integer $a_t \in \{0, 1 \ldots, 2625\}$. The first 25 elements of the action space correspond to simple placements. The next $25 \cdot 4 = 100$ elements corresponded to moves. The final $25 \cdot 4 \cdot 25 = 2500$ elements correspond to jumps and extra captures.

3.2 DQL Implementation

Our DQL critic network consisted of an input layer of 25 neurons for the board state, three hidden layers with 64 neurons each, and one output layer with 2625 Q-values corresponding to each action. We used a Q-learning replay buffer with 20,000 tuples of $(s_t, a_t, m_t, r_t, s_{t+5})$, where m_t is a stored action mask to identify allowable actions for the feedforward step during training. The replay buffer is initialized before training with self-play experiences, where actions are taken ϵ greedy according to the outputted Q-values of the critic network. Because of the sparse reward environment, we used a 5-timestep TD gap to populate the replay buffer and allow for rewards to flow farther through the critic.

After this, the critic is trained with standard DQL training cycles for 200,000 iterations, where we use a frozen target network Q'_θ to compute the bootstrapped Q-networks that are adjoined to the summed observed rewards:

1. **Sample:** sample a single mini-batch of 64 experiences from the replay buffer.
2. **Targets:** calculate a target y_t for each tuple in the mini-batch, computed as
$$y_t = r_t + Q'_\theta(s_{t+5}, a_t)$$
where r_t is the sum of the previous five rewards, precomputed during experience generation.
3. **Gradient Descent:** perform one step of gradient descent using the loss function
$$\mathcal{L}_{\text{smooth } L_1}(y_t, Q_\theta(s_t, a_t)),$$
where $\mathcal{L}_{\text{smooth } L_1}$ is the smooth L_1 loss using a standard $\delta = 1$, defined as
$$\mathcal{L}_{\text{smooth } L_1}(x, y) = \begin{cases} \frac{1}{2}(x-y)^2 : & |x-y| < 1 \\ |x-y| - \frac{1}{2} & |x-y| \geq 1 \end{cases}.$$
4. **Rollout:** run one game of self-play and add the experiences to the replay-buffer, removing the oldest experiences to make space. Repeat until completion.

We used a learning rate of 0.001 with the Adam optimizer.

3.3 PPO V1 Implementation

Our actor and critic shared an underlying trunk, which had a 25 dimensional input layer for the state representation and a 64-neuron hidden layer. From this hidden layer, the actor head splits directly to the 2625 dimensional output layer of logits. Similarly, the critic head splits directly to the single dimensional output representing the value $V(s_t)$ of the input state.

Our PPO training cycle alternated between rollout phases and training phases for 5000 iterations, where rollout and training phases worked as follows:

1. **Rollout Phase:** the rollout buffer is emptied if it already contained experiences and then refilled with self-play experiences from the most recent actor/critic. We used a rollout buffer of 4096 experiences, where each experience is represented by a tuple $(s_t, a_t, m_t, \pi'_t, A_t, R_t)$, where m_t are the action masks for the state s_t, $\pi'_t = \pi_\theta^{\text{old}}(a_t|s_t)$ is the probability for the network which generated this action to take action a_t at state s_t, A_t is the advantage estimate for this action, and R_t is the bootstrapped estimate for the total rewards starting from this state and taking this action.

The advantage estimates A_t are calculated recursively after each game finishes. Concretely, given a trajectory $\tau = (\tau_1, \tau_2, \ldots, \tau_T)$, where each τ_t represents a tuple (s_t, a_t, r_t), we calculate for each t

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t),$$

which can be thought of as the single step TD estimate of the advantage of the action at time step t. (Note that we append an extra value for $V(s_T) = 0$). Then, we calculate A_t as

$$A_t = \delta_t + \gamma\lambda A_{t+1},$$

which can be thought of as the aggregated weighted-advantages of taking action a_t at timestep t.

The returns R_t are calculated by bootstrapping the critic's value estimate onto the computed Advantage estimates:

$$R_t = A_t + V(s_t).$$

2. **Training Phase**: perform 6 epochs of gradient descent over the whole rollout buffer with minibatches of 64 experiences. The final objective function that we minimize in these gradient descent steps is a linear combination of three surrogate objectives. The first of these is the *clipped surrogate objective* which is meant to train the actor, defined as

$$\mathcal{L}^{\text{Clip}}(\theta) = \hat{\mathbb{E}}_t\big[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\big]$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_\theta^{\text{old}}(a_t|s_t)}$, π_θ^{old} being the probability that the actor in the experience replay takes action a_t when given state s_t. For our critic, we use a standard MSE surrogate objective

$$\mathcal{L}^{\text{Value}}(\theta) = \hat{\mathbb{E}}((V_\theta(s_t) - R_t)^2).$$

Finally, to encourage exploration, we include an extra surrogate entropy objective

$$\mathcal{L}^{\text{Entropy}}(\theta) = -\hat{\mathbb{E}}(H(\pi_\theta(\cdot|s_t)))$$

where $H(\pi_\theta(\cdot|s_t))$ is the entropy of the distribution given by π_θ. We combine these three *surrogate* losses into a single combined loss for the actor and critic:

$$\mathcal{L}(\theta) = \mathcal{L}^{\text{Clip}}(\theta) + C_1\mathcal{L}^{\text{Value}}(\theta) + C_2\mathcal{L}^{\text{Entropy}}(\theta)$$

where C_1 and C_2 are tunable hyperparameters. We chose value of $C_1 = 0.05$ and $C_2 = 0.01$.

We note that the output of the actor head $\pi_\theta(s_t)$ of the network is a set of logits of dimension 2625 with one logit per action. To retrieve the action probabilities, we first apply the action mask m_t to the logits and then apply the softmax to this masked result. In this way, we only consider probabilities for valid actions. Note that the probability distribution over which we calculate the entropy is this masked distribution. We used the Adam optimizer with a learning rate of 0.0001.

3.4 PPO V2 Implementation

For our second experiment PPO V2, we changed the rollout-phase of the training cycle to diversiy the kinds of opponents and strategies that the agent sees and trained the agent for 3000 iterations. Specifically, we initialize an empty list of agents at the start of training and add a frozen version of the current agent every 500 rollouts, working out to 6 total frozen *ancestors* that we store by the end of training. When playing out episodes to fill up the rollout buffer, we randomly sample an opponent from this list of ancestors and record only the states and actions that correspond to the current agent. We make sure to alternate the starting position in these turns so that the current agent gets experience beginning with and without the drop initiative. Additionally, we increased the rollout buffer size to 8192 experiences, we changed the batch size from 64 to 128, and we correspondingly doubled the number of epochs from 6 to 12.

3.5 PPO V3 Implementation

For our last PPO experiment, we changed our reward structure to encourage win-based learning, we incorporated the drop initiative into our state, and we deepened our actor/critic trunk. This last experiment trained for 4000 iterations. Specifically, while PPO V1 and V2 incorporated a 0.1 reward per piece captured, we scheduled the reward given for captures so that on the ith iteration out of 4000, the capture reward was

$$\max(0, 0.1 - 0.05 \cdot (i/4000))$$

so that rewards for capture would be 0 around half the way through training. We added an extra hidden layer to the shared actor/critic trunk with 64 neurons. Finally, while the state representation for V1 and V2 was just the flattened board representation, we appended an extra entry to the state which represents the drop initiative (none, blue, or red).

3.6 Policy Guided Tree Search

We used the actor and critic from the trained PPO V3 model to implement a naive Policy Guided Tree Search (PGTS) which searches for the optimal action by selecting paths using the actor and evaluating paths using the critic. Beginning at the root and considering game states as nodes with edges representing actions, we repeat the following recursive algorithm

1. **Selection:**

   If: the node we are on is terminal or deeper than a depth of $d = 5$, then query the critic for the value of this node (or retrieve the terminal value if the node is terminal) and return this value.
   Else: Query the actor for the top k legal moves to take from this state. For each of these actions, repeat from 1., storing the returned values from each of the exploration branches.
2. **Return:** the action associated with the largest observed value among the explored actions.

Like this, we recursively explore the tree to a depth of $d = 5$ and use the critic to evaluate the value of leaf nodes. We implemented a scheduled thinning of the branches with $k = 12$ in the root node, $k = 6$ in the depth 1 nodes, and $k = 3$ for all subsequent nodes.

4 Results

4.1 Experimental Techniques

Because there is no standardized elo system for scoring Choko players, we tested our agents by having each agent play against every other agent and recording the win, draw, and loss rate between different pairs. We also incorporated a non-RL Minimax agent which explored all possible legal moves at each state to a depth of $d = 3$ and estimated action value through the number of captures involved.

While we initially incorporated our own elo system to track the skill levels of these different agents, the small sample pool meant that elo ratings were highly variable and removed much of the nuance in the skill differences between agents.

The agent we trained with DQL was the weakest of all the experiments we tried. In twenty matches against the PPO V1 agent, the DQL agent won 30% of matches and lost 35% of matches, the remaining 35% being draws. In the same tournament against the PPO V2 agent, the DQL agent won 20% of games and lost another 50% of games, the remaining 30% being draws. These results motivated us to pursue a deeper implementation of PPO with V3 and PGTS instead of focusing on the DQL agent. A list the different win rates between pairs of agents is in Table 1, where the percentages listed correspond to the percent win rate of the agent in column against the agent in the row.

Every 10 rollouts when training our PPO agents and every 200 gradient steps when training our DQL agent, we pitted the current agent against a frozen ancestor from 10 iterations (respectively 200 games) ago. We allowed the ancestor to begin the game in the first half of these matches and the current policy to begin in the other half. These evaluation metrics became the primary way for us to gauge the performance of the models as they trained.

Table 1: Matchup Results, Percent of Column Wins over Row

Method	PGTS	PPO V3	PPO V2	PPO V1	Minimax
PGTS	-	0%	5%	0%	0%
PPO V3	50%	-	35%	25%	50%
PPO V2	25%	40%	-	20%	65%
PPO V1	30%	40%	35%	-	100%
Minimax	15%	0%	0%	0%	-

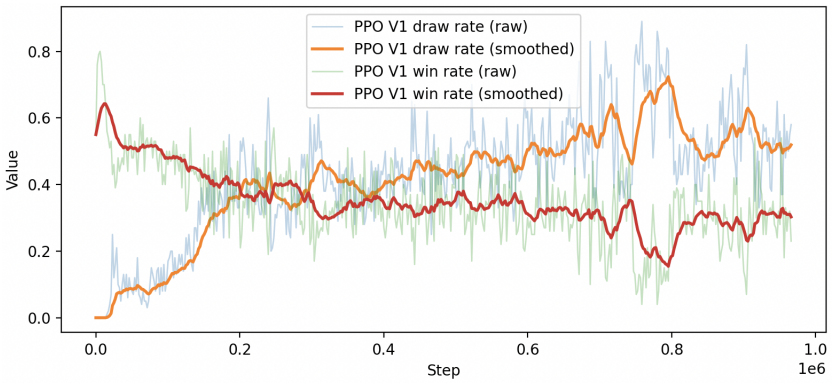

Figure 1: PPO V1 Win Rates and Draw Rates

4.2 Quantitative Evaluation

These head to head matches against frozen ancestors provided us a way to measure if the model was still improving over time. After 5000 iterations of training our PPO V1 model, which was trained using only current-policy self play, we noticed that the model's improvement had slowed substantially. The win rates and draw rates over time for PPO V1 is included in Figure 1. While win rates stayed ahead of loss rates throughout training, their steady fall indicated that the policy was learning to draw against itself instead of improve its current policy.[1]

We suspected that the descending win rates might be caused by stagnant policy improvements towards the end of training. Because the agent only sees episodes from self-play rollouts using the most recent model, we worried that later matches lacked the variety in playstyles needed to push the agent to learn more complex strategy. These results motivated us to add a more diverse self-play system in PPO V2 which created rollouts that pitted the current model against a variety of frozen ancestors from throughout training history. The win rates for PPO V2 in Figure 2 show substantial improvements, displaying a slight growth in win rate over time against the frozen agent. At the end of training, PPO V2 won 140% more matches than it lost against its evaluation agent.

While training, we also recorded the proportion of probability ratios $r_t(\theta)$ that were clipped to compute the clipped surrogate loss. Higher clip percentages generally indicate that the policy is changing more between iterations since more probability ratios $\pi_\theta(a|s)/\pi_{\theta^{\text{old}}}(a|s)$ fall outside the allowable ϵ range. The clip percentages for our PPO V2 experiment were on average higher than those for our PPO V1 experiment with clip percentages of 3.41% and 4.1% respectively, indicating greater movement and learning in our policy.

We began our experiment with PPO V3 under the suspicion that the model would do better if we increased model complexity and appended the drop initiative to the state. Our results from PPO V3 were very strong. Like V2, we saw a steady rise in win rates past around a quarter of the way through training (see Figure 3), though these curves plateaued around halfway through training. This told us that our policy was improving, albeit slowly. V3's loss rates grew as well during training, though

[1]Note that loss rates are not shown in the figure but are the difference between win and draw rates.

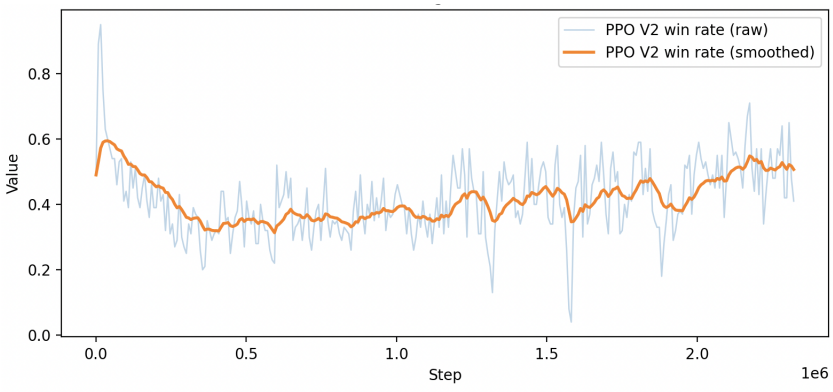

Figure 2: PPO V2 Win Rate

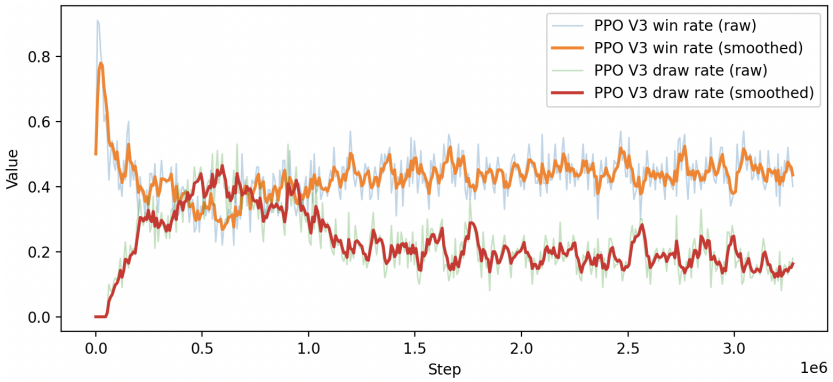

Figure 3: PPO V3 Draw and Win Rate

they stayed below V3's win rates over the course of training demonstrating improvement. At the end of training, our evaluation showed a 44% win rate and a 42% win rate, suggesting that our agent's training was approaching equilibrium and our model's learning had been saturated.

4.3 Qualitative Analysis

One of the things that most surprised us is how quickly the agents converged to a consistent opening style. Corners prove to be highly defensive positions since they cannot be captured unless forced to move. As such, all agents that we trained opened with corner positions and slowly moved inward.

We also noticed how consistently our PPO agent learned to combine capture and defense. Perhaps the simplest and best example of this is what we call a *secure capture*. Figure 4 shows a match between one of the authors (Eugene, playing blue) and the PPO V2 agent (red). In move 12, Red moves `B1` to `D1` capturing `C1`. In doing, it is now vulnerable to capture from `E1` but the agent recognizes this and takes `E1` using its extra capture, thus completing its secure capture. We tested scenarios like this across a dozen boards like Figure 3 in various positions and our PPO V2 agent consistently eliminated threats after committing to a capture. We emphasize that a jump + extra capture are not treated as separate moves but rather as a single action within the action space.

We were curious at how competent our agent was at executing defensive plays to evade capture. All of our PPO agents consistently identified direct dangers on the board and acted correspondingly. Skirmish 1 (Figure 5) shows one scenario where PPO V2 (Red) aptly chooses to evade Blue's capture by moving `B3` to `B4`. We saw this behavior consistently throughout all of our PPO agents.

One weakness we noted in our PPO agents was a lack of foresight at combining capture and defense to defend more than one piece. Shown in Skirmish 2 (Figure 6) is one such example, where PPO V2 plays red and it is red's move. `A2` is under threat from `A1`. At the same time, `D4` has the chance to take `C4`. The optimal move then would likely be to move `D4` to `B4` and then take `A1` using the extra

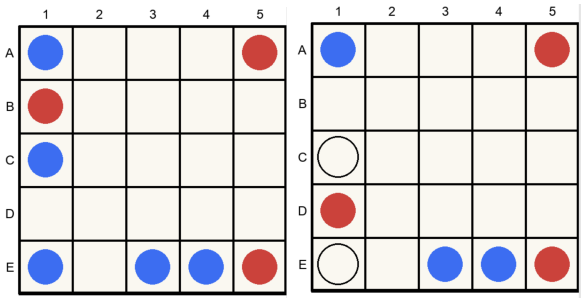

Figure 4: PPO V2 (Red) v Human (Blue), Move 12

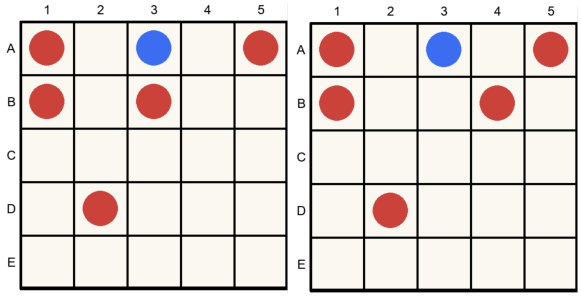

Figure 5: Skirmish 1

capture. This optimal move shows up as the fourth most probable move in PPO V3's policy, with a percent play of only 1%. The optimal move does not even show up in PPO V2's actor. Instead, the most probable move according to both policies is to move `D4` to `B4` and capture `C2` (illustrated in Skirmish 2, Figure 6 as well) with a 79% probability.

Once we applied PGTS to PPO V3, this lack of longer term strategic awareness disappeared. PGTS identified the optimal move in Skirmish 2 (Figure 6) despite it only appearing with a 1% chance in the underlying policy. This is particularly important because both the optimal and less optimal moves (middle and right in Figure 5) result in the same number of captured pieces, the only difference being the value assigned to those states from the critic, demonstrating a nuanced and accurate valuation of board states by the critic. In general, PGTS could consistently identify how to leverage captures so that its position after the capture was not compromised.

Perhaps the best example of the longer term strategic thinking of PGTS comes from Move 22 in a match between PGTS (red) and our non-RL minimax benchmark (blue), shown in Figure 7. The clear move is `A3` to `C3`, but there are several strong choices for the extra capture. After this move, `B4` is threatened by both `A4` and `B5`. One candidate capture would be to remove one of these pieces, though this doesn't remove the threat altogether. Instead, PGTS chooses to capture `E3`, sacrificing `B4` but forking `E2` into a guaranteed capture. The move, which requires deep and critical strategic foresight, demonstrated the strong potential of this combined PPO/tree-search model.

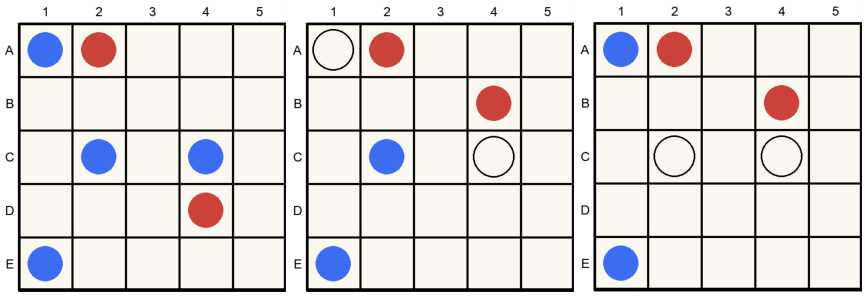

Figure 6: Skirmish 2. Left to right: pre-move, ideal-move, PPO V2-move.

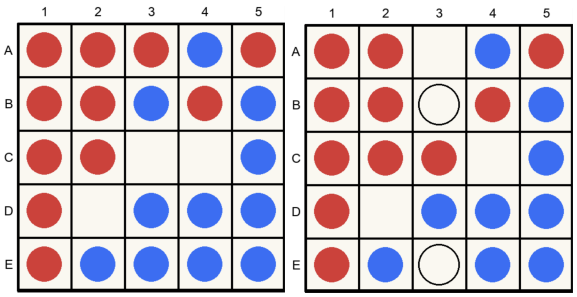

Figure 7: PGTS v Minimax Agent, Move 22

5 Discussion and Conclusion

On its own, our strongest PPO agent (PPO V3) showed critical understanding of the game environment and a strong tactical awareness of how to capture safely. At the same time, the model lacked longer term strategic foresight. Standard minimax tree searches benefit from their power to look ahead and see possible actions but are weak in regimes where the action space is so large. PGTS leverages the benefits of both worlds, creating a skilled policy for evaluating strong short term candidates and evaluating those options using a strong board critic.

One other benefit that PGTS provided when compared to out minimax baseline was the speed at which it could search through board states because of the actor's action pruning. Our PGTS algorithm could search up to 7 layers deep in the search tree in the same time (around 3 seconds) that the original minimax agent took to search just 3 layers. At the same time, PGTS required only moderate levels of compute and represented a relatively small model, taking 23 hours to train on a 2021 Apple M1 Pro GPU for a model with 180,738 trainable parameters. In this sense, PGTS offered us a relatively quick way to train a small yet powerful model for Choko. Finally, all of the authors were most impressed with how much Choko strategy they learned along the way while working on this project.

Our original intention when we began this project was to make an agent who could act with long term strategic foresight in non-trivial ways and we feel strongly that PGTS is such an agent. For future work, it would be interesting to consider larger models for the underlying trunk in the PPO implementation since these models might be able to absorb more of the experiences from the environment. Along these lines, our PPO V2 and V3 implementations both used a form of fictitious self play to diversify the experiences on which the model trains; in the future, a more diverse selection of opponents in this buffer could perhaps speed up training by giving the model a richer set of strategies in the environment.

References

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, and et al. 2015. Human-level Control through Deep Reinforcement Learning. *Nature* 518, 7540 (2015), 529–533.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. In *arXiv preprint arXiv:1707.06347*.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484–489.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2018. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play. *Science* 362, 6419 (2018), 1140–1144.