

ShabuShabu: Learning Shobu By Self-Play Through Reinforcement Learning

Ayushman Choudhury, Brandon Gong, Seowon Chang

May 3, 2025

1 Introduction

Shobu [4] is a relatively nascent (2019) board game which has not yet been well-explored using computational methods. Like chess, it is a two-player, sequential, deterministic game with perfect information. Shobu possesses a unique movement mechanic in which players’ moves occur in two phases, *passive* and *aggressive*, and opponent stones are simply displaced rather than immediately captured. This results in a dynamic, ever-shifting board state wholly distinct from many of its board game predecessors.

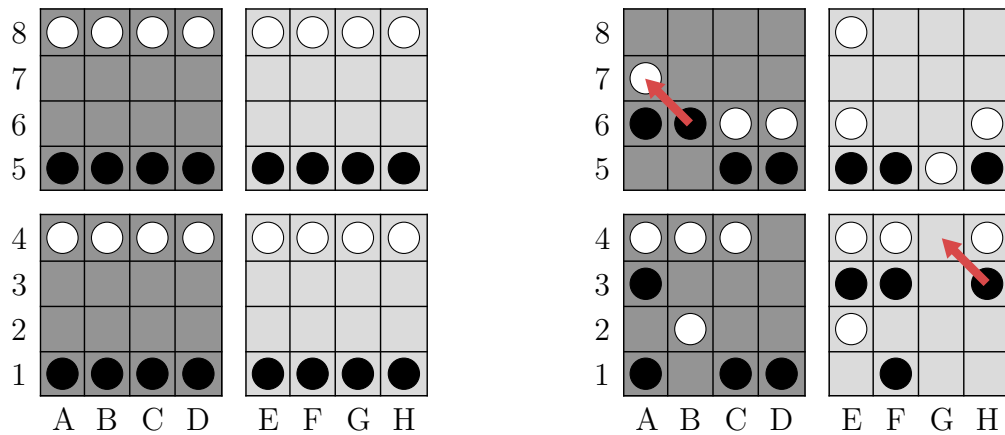


Figure 1: Left: (a) The Shobu starting position, from black’s point of view. The lower two subboards are black’s home boards, and the upper two are white’s. Right: (b) A game of Shobu in progress. Here, black can select their H3 stone on their right home board as the passive stone and their B6 stone (on a subboard of opposite color) as the aggressive stone, and move both one square northwest: h3b6NW (red arrows). After this move, white’s A7 stone will be pushed off the subboard and removed from play. Other legal moves include d1f5N2, f1a1NE, d1h5N2, and f3b6SE.

The game is played on four 4×4 “subboards”—two light and two dark—each starting with four stones per player (Fig. 1a). On their turn, a player first chooses a *passive* stone on either “home board,” moving it one or two spaces in any cardinal or ordinal direction.

They must then follow up by moving an *aggressive* stone on a subboard of the opposite color by the same direction and distance; in this phase only, they may “push” a single opposing stone (Fig. 1b). Stones pushed off the edges of a subboard are removed from play, and the game is won when a player eliminates all of their opponent’s stones from any one board.

Due to the recency of Shobu’s creation and lack of a developed player base, not much is known about what a “good” strategy may be. For example, while keeping more stones on a subboard may prevent loss in the short-term, it also may restrict mobility and attacking options. In this case, reinforcement learning (RL) via self-play and exploration is an attractive approach because of the model’s flexibility to make discoveries and evaluations on its own rather than adhering to possibly-flawed manually designed heuristics.

Because of the various directions and distances to move a stone and the myriad of passive and aggressive stone pairing possibilities, Shobu’s branch factor is quite large. We estimate the average number of legal moves to be 62.2, with 232 possible moves in the starting position. In contrast, the branch factor of checkers is 2.8, and the branch factor for chess is 35.

We hypothesize the exponential cost of Shobu’s significantly larger branch factor may make traditional minimax search intractable, especially on consumer hardware, although we leave a quantitative evaluation of minimax performance to future work. As a result, we only examined RL strategies that involve playing without search—i.e., a proximal policy optimization model, discussed in Appendix A—and an RL model based on Monte Carlo tree search (MCTS) named ShabuShabu, which reduces the search space by primarily considering moves believed to be “good.”

Of course, no discussion of an MCTS-based RL agent for games is complete without mentioning the contributions of AlphaGo [7] and subsequently AlphaZero [8]. These models were an early proof of concept in applying RL to games and served as a source of inspiration and ideas for our model architecture (Section 4).

AlphaZero used in total 5,064 first-generation TPUs for 34 hours to train their Go model. We must get by with 64 CPU cores from Oscar [2]. To that end, we incorporate various enhancements over the original AlphaZero algorithm inspired by KataGo [9], and implement an efficient Shobu board representation and a highly parallelized training setup to maximize training speed with limited hardware. We discuss these optimizations in Section 4.5.

Finally, in addition to the baseline model implementation, we develop a toolkit of modular utilities to diagnose model behavior and evaluate model performance, both qualitatively and quantitatively. These tools and our evaluation metrics using them are covered in Section 5.

2 Related Work

2.1 AlphaZero

AlphaZero was the first deep reinforcement learning method to play strategy games like chess and Go solely through self-play. Given a game environment (Go, chess, Shogi), AlphaZero takes in the game board state as a multi-channel image and feeds it through a convolutional neural network with multi-head outputs. The model outputs a value and a policy, where the value is a regression on the predicted outcome of the game and the policy is a distribution of next moves given the board state. By generating millions of samples and training the model

based on the value and policy outputs, AlphaZero was able to generalize to superhuman levels of play. We take the general ideas from this paper and adapt it for Shobu. We highlight the following key differences from this paper in our work:

- Given the computational limitations and smaller game complexity compared to Go, we reduce the number of blocks and number of filters in the residual net backbone. For additional details see Section 4.3
- Unlike chess and Go where there are multi-move game mechanics, Shobu moves in a given board state are independent of past board states. Thus, we remove board history in the representation of the board state, which also eases computational burden during training and inference time.
- We use a smaller minibatch size (256) and upweight the value function (weight coefficient of 1.5) to further ease computational burden and accelerate learning of the value function.

2.2 KataGo

KataGo introduced optimizations for self-play reinforcement learning for Go, introducing training schemes and architectures to make training these types of models more efficient. KataGo reduced the number of samples and time needed to train a strong Go model, and provided detailed experiments for trying different training regimes, curriculum learning, and architecture designs. Given our limited compute power, we leveraged a number of discoveries from this work to improve our Shobu self-play model. These include:

- **Playout cap randomization:** By reducing the playout cap for some random moves during an episode, we increase the number of game results for the value network to learn from while maintaining a substantial number of full depth MCTS searches for the policy network to reference.
- KataGo showed reduced model sizes and minibatch sizes still reached strong levels of play in Go, further reinforcing that reducing the backbone complexity should not hinder the theoretical cap of our model.
- KataGo showed the importance of global pooling in accelerating learning, which we incorporated into our architecture.

3 Data

Like AlphaZero and KataGo, training samples are collected during self-play by the model, which are generated continuously [8,9] in parallel with training steps. To parallelize sample generation, we assign a worker for each CPU core to simulate the game using a copy of the current model. For each game, the worker pulls the latest version of the model weights to launch a game. The results of the game (transitions, rewards, target distributions from MCTS) are then appended to a shared memory queue, which is accessed by the train worker

to update the model. The latest training samples (rolling window of 50K samples) are kept for sampling minibatches to train on.

4 Method

Here we outline the details of ShabuShabu, an MCTS-based RL model for playing the game Shobu.

4.1 Shobu Environment

To make the most of the hardware, we implement Shobu using bitboards [5], which have been favored in top game engines such as Stockfish and Lc0 over the “mailbox” implementation (in which pieces are stored in an array and moved from slot to slot).

Each Shobu board is represented with two 64-bit integers—one storing the positions of black stones and another storing the positions of white stones—and a boolean to indicate the next player. This remarkably dense representation format not only provides us with ease of copying and a small memory footprint (both great features to have when doing tree search with large rollouts), but is also well-suited for generating Shobu moves simultaneously and rapidly using simple bitwise operations.

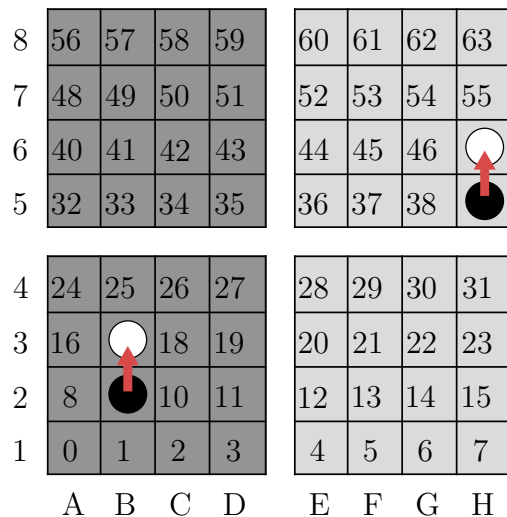


Figure 2: The Shobu bitboard layout with example aggressive moves b2N and h5N.

Figure 2 shows the Shobu bitboard layout and two example aggressive moves, each one a single step north. Note if we shift the black bitboard left by 8 bits, the bit at position 9 will be shifted up to position 17 (left red arrow), and the bit at position 39 will be shifted up to position 47 (right red arrow). We can then superimpose this with the unshifted white bitboard via a bitwise AND to find all white pieces which would be pushed by a northward move, `(black << 8) & white`. Bits 24-31 would be masked before this operation to prevent stones from jumping across subboards. Finally, to *serialize* this bitset back into individual moves, we can repeatedly remove the least-significant bit from the bitset and take cartesian product with legal passive moves (not shown in diagram) to generate a full list of moves.

A key feature of the Shobu implementation is the ability to flip the board between black and white players, effectively rotating the board 180° and swapping the colors. The bitboard representation also makes this straightforward; we simply reverse each 64-bit integer and exchange the white and black bitboards. By doing so, we can train a model which exclusively plays as black, thus halving the amount of board states it needs to “learn” and reducing the potential for model confusion.

4.2 Monte Carlo Tree Search

Due to the exponential costs of Shobu’s high branch factor (which we estimate to be around 62.2), and our inability to capitalize on minimax optimizations such as alpha-beta pruning due to an effectively random static evaluation function at the start of training as well as a lack of move generation order heuristics, we opted to proceed with Monte Carlo Tree Search (MCTS) with the PUCT algorithm. MCTS as a search algorithm for self-play models has been well-explored by our predecessors, and we generally follow the standard implementation, which we cover in more detail in [Appendix B](#).

We introduced and modulated the “exploration bonus” γ (which we later found was also employed by KataGo under the name c_{PUCT}) throughout training, ranging from 1.5 to 1.0, to encourage exploration and reduce the impact of noisy evaluation in early stages of training (epochs 4200-6900). This modification to the PUCT formula was due to the observation that early models would tunnel-vision on nonsensical candidate moves, wasting compute that could have been better spent visiting other nodes (Figure 3).

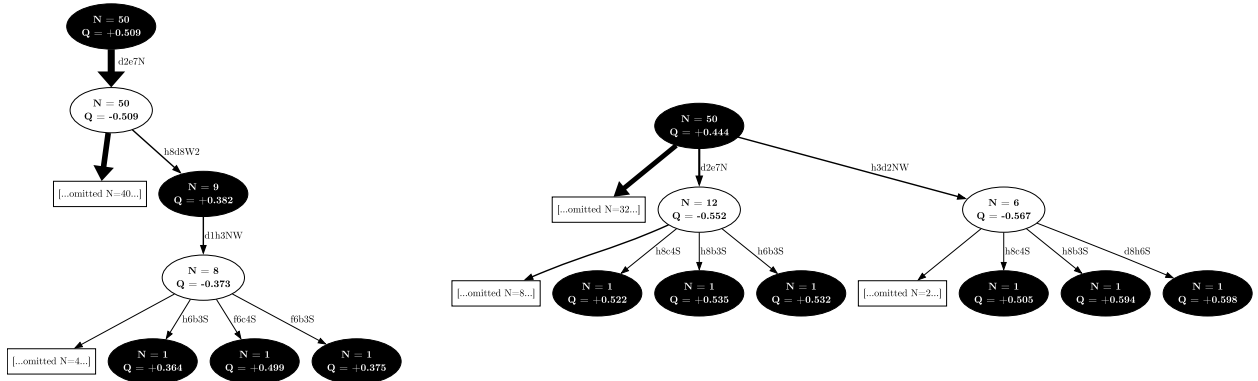


Figure 3: Illustrating the effects of γ , the exploration bonus. On the left is MCTS executed with the default PUCT formula ($\gamma = 1$), where the policies and values are provided with a very primitive, untrained model. Note that it only considers one (nonsensical) candidate move from the root. On the right, we apply an exploration bonus of $\gamma = 1.5$, resulting in a “bushier” search tree with many more candidate moves considered.

Like AlphaZero and KataGo, we apply Dirichlet noise to the priors generated at the root node to further encourage exploration. To estimate the Dirichlet noise parameter, we first observed the Dirichlet parameters used by AlphaZero and the branch factor of different games. We then fit an exponential function to the Dirichlet noise parameters chosen by AlphaZero based on the branch factor of the games it implements (chess, shogi, Go). This

gave us an empirical estimate of 0.22 for Shobu’s branch factor, which we used as the Dirichlet parameter during training.

To further reduce over-sharpness in the MCTS rollout (which translates to overly sharp policy distributions), we apply a temperature softmax at the root of each MCTS rollout. Katago found that adding a temperature at the root allowed the model to explore alternative, equally good moves that otherwise would not be reached from the standard MCTS. Therefore, we applied a root softmax with temperature value of 1.1.

During training time, self-play is implemented by running a rollout of MCTS (repeating the above three steps 100-800 times) and then sampling a move at random according to the distribution of visit counts of the current position’s children. Model behavior can be made “more random” by smoothing out this distribution according to a temperature parameter, τ , before sampling. We found that the $\tau = 1$ employed by AlphaZero and KataGo was too low, and caused the model to overfit, playing the same line repeatedly with only slight variations while not generalizing well to other less-studied positions. We thus set the temperature schedule

$$\tau(n) = \begin{cases} \infty & n < 1 \\ 3 & n < 3 \\ 1 & 3 \leq n < 5 \\ \frac{1}{5}(10 - n) & n \leq 10 \\ 0 & \text{otherwise,} \end{cases}$$

where n is the full-ply move number. We find this gives a good combination of forcing the model to explore a variety of positions while still allowing it to play at full strength as the game progresses.

4.3 Model Architecture

We briefly outline the architecture of ShabuShabu (Figure 4). First, the board is represented as an 8-channel image, where the first four channels represent the locations of the pieces of the current player on the four subboards, and the last four channel represent the pieces of the opponent player on the four subboards. The model leverages a deep residual convolutional neural network to extract features from the Shobu board state, and uses multiple heads to convert the learned features in a policy distribution over valid moves and a value representing the expected outcome of the game. Full details of the model implementation are located in [Appendix C](#).

4.4 Loss Function

Samples are obtained from the self play games, which consists of the board state and the associated game end results (draw, win, or loss). We denote the model as $f(\theta) = \hat{v}, \hat{\pi}$, where θ is the model parameters, \hat{v} is the model’s value function, and $\hat{\pi}$ is the model’s policy function. For a given state s , we optimize the loss function

$$L(s) = \beta_{value} * (\hat{v}(s) - r(s))^2 - \sum_m \pi(m) \log(\pi(\hat{m})) + \beta_{L2} ||\theta||^2$$

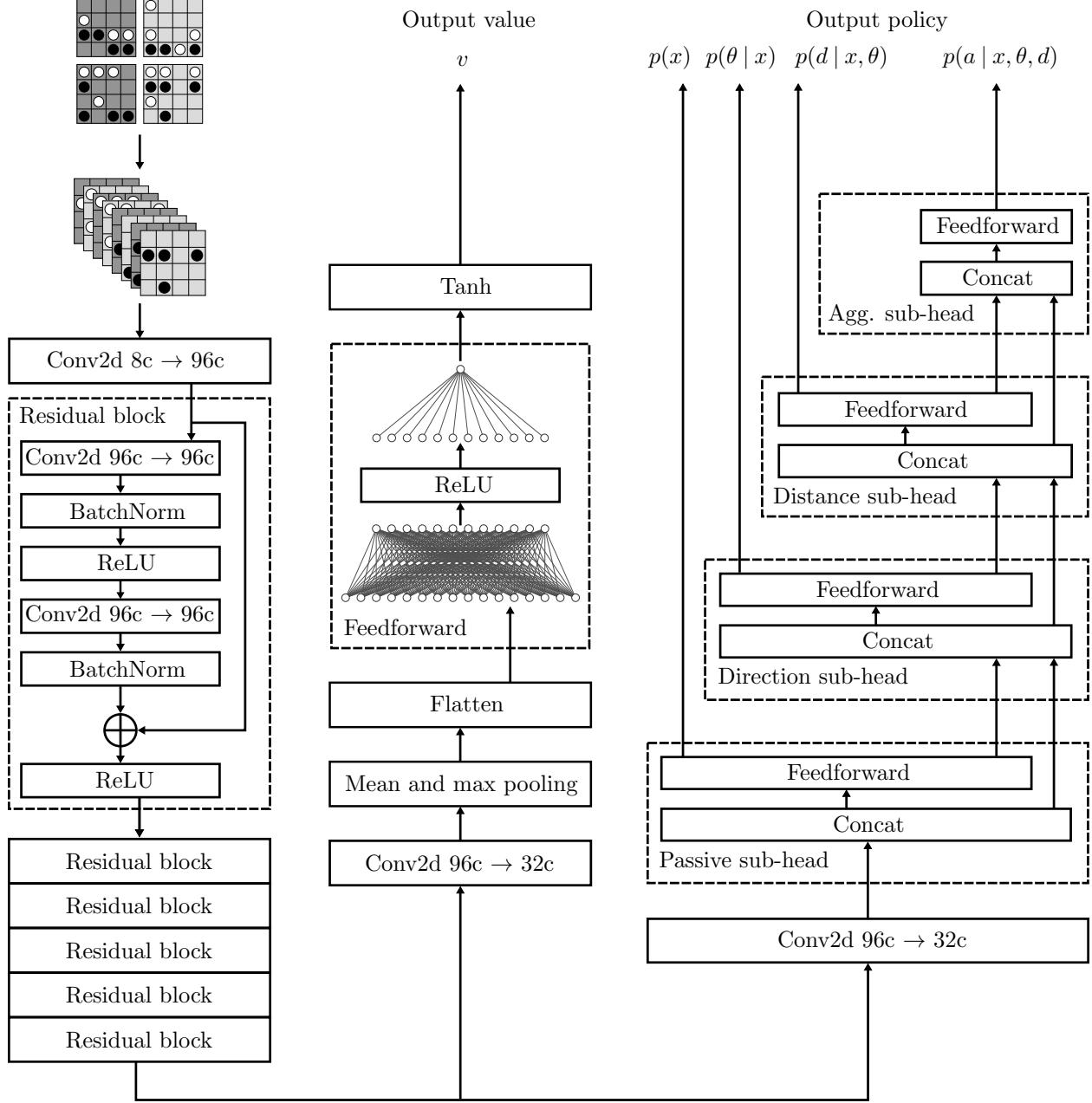


Figure 4: ShabuShabu architecture. On the left: input transformation and initial convolution, backbone. Center: value head. Right: Policy sub-heads, outputting conditional logits for passive piece x , direction θ , distance d , and aggressive piece a . Note that all residual blocks are identical; the first one is expanded to show internal structure. Likewise, all feed-forward blocks consist of two layers with an intermediate ReLU activation. Input, hidden, and output sizes vary based on usage.

where $r(s) \in -1, 0, 1$ is the outcome of the game where s was sampled from, m ranges over the valid moves for s , $\pi(m)$ is the target policy distribution derived from the MCTS, $\beta_{value} = 1.5$ is the scaling constant for the value function loss, and $\beta_{L2} = 3e-5$ weights the L2 penalty on the model weights θ . Essentially, we have a value function MSE loss which evaluates the model’s ability to predict the expected return of a board state and a policy function cross entropy loss which evaluates how closely the model predicts the target distribution from the MCTS.

4.5 Training Optimizations

4.5.1 Parallelized train/simulation

Given the computational burden of generating self-play game samples, we implemented a simultaneous train and self-play framework. Given n CPU cores, we allocated $n - 2$ cores for self-play generation and 2 cores for training. Each of the self-play worker cores continuously generated samples from simulating games using the latest version of the model weights obtained from training. After a sufficient warmup buffer of samples were generated, training was launched. In each epoch of the train loop, a training step was computed on a minibatch of 256 samples were randomly selected from the rolling window of self-play samples. This allowed us to generate as many samples as possible while training for sufficient number of steps. We set $n = 64$ on as Oscar CPU batch job (in our experiments, using GPU job was significantly less performant; we expect this to be due to the costs of MCTS). Each training run lasting up to 24 hours, starting from the last checkpoint of the previous run. The training runs were composed in a curriculum learning framework: early training runs encouraged exploration, middle training runs addressed any overfitting or underfitting issues observed in intermediate checkpoints, and later training runs focused on fine-tuning or strengthening of model strategies.

4.5.2 Playout cap randomization

We adopted an idea from KataGo [9], called playout cap randomization. The key motivation behind this idea is the tension between the value and policy networks, where the value network requires a large number of games to learn a robust function, which can be obtained by generating more games with a reduced playout during MCTS. However, the policy network requires larger playouts to learn meaningful move distributions. To ease this tension, the KataGo paper suggests using playout cap randomization. For a small percentage p of moves, the full playout N is used for the MCTS. For the remainder of moves, a reduced playout $n < N$ is used. Only the moves with a full playout are retained for training. This results in more games for the value network to train on, while retaining meaningful move distributions for the policy network to train on. For training, we set $p = 0.25$, $N = 400$, and $n = 100$.

4.6 Training Regime

We trained ShabuShabu for a total of 50K train steps over several Oscar CPU jobs. We periodically evaluated checkpoints using the `explorer` and `eval` detailed in Section 5. Using `explorer`, we identified signs of model overfitting to certain board states or moves, and we

evaluated checkpoint improvement using `n_game_match` from `eval`. To mitigate model overfitting or improve rate of improvement, we implemented incremental training adjustments as training progressed. This curriculum-learning training regime allowed model strength to improve over using limited time and compute.

The main training regime was conducted as follows:

- Early training with exploration coefficient (epochs 0 - 8000)
- Adding random positions and increasing temperature in opening moves to avoid overfitting (epochs 8000 - 32000)
- Introduced warm annealed restarts to learn new strategies by increasing exploration coefficient for a few hundred epochs, then continuing training (epochs 32000 - 50000)

To analyze the effects of different training parameter decisions, we performed the following ablations:

- Show strength of learned MCTS by comparing against a random agent, PPO agent, random MCTS agent.
- Show benefit of adding random positions during training (no random position training started from epoch 8000 from early training to 16000 epochs)
- Show benefit of annealed restart (no warm restart from epochs 32000 to 48000)
- Show benefit of keeping more full game trajectories (increased random positions training starting from epoch 34000 to epoch 48000)

5 Metrics

5.1 Qualitative metrics

Because we have a large number of hyperparameters to select (e.g. Dirichlet noise parameter, PUCT exploration bonus, learning rate, playout cap sizes, value loss scaling and regularization coefficients, move sampling temperature etc.) but not enough compute or time to choose parameters and allow training to proceed for a requisite number of epochs before evaluating differences in outcomes, we opted to use qualitative checks on the model outputs and MCTS behavior during training time to identify bugs and analyze model behavior. We then used this information to guide our decision making on how to further tune hyperparameters within our model.

We developed a tool called `explorer` which enables much of this qualitative evaluation of model performance. `explorer` is a REPL which allows us to interactively run MCTS from an arbitrary position and climb up and down the search branches, examining value and policy model outputs at each node. We used `explorer` to:

- (Bugfix) Detect a bug in which players were not being flipped properly at each level of the tree, causing rewards to be propagated improperly from leaf nodes.

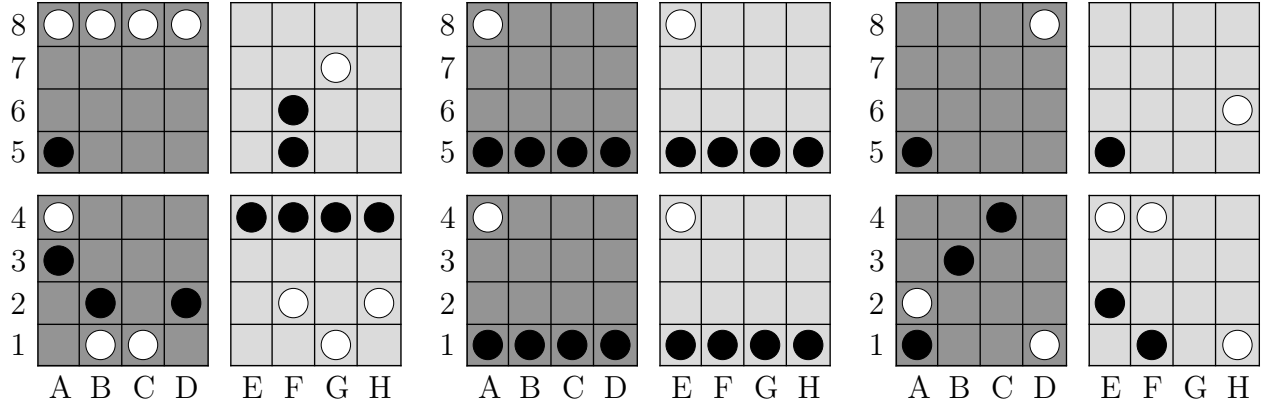


Figure 5: Shobu test positions. From left to right: (a) **BLACK_MATE_IN_1**: black has an immediate win available, b2f6NE2. (b) **MATERIAL_ADVANTAGE**: black has a large material advantage, but no immediate win. (c) **PUSH_OPTIONS**: Black has various captures available, and one stone on A1 which white is threatening to push off on the next turn.

- (Bugfix) Detect the presence of end-game reward sign issues causing the search to avoid winning states (Fig. 5a).
- (Hyperparameter tuning) Based on the insight that node visits were heavily skewed towards one move while completely skipping other interesting candidates, we introduced 1.5x exploration bonus in PUCT score computation for early training.
- (Hyperparameter tuning) Determine the effects of sampling temperature on move choice randomness, and choose an appropriate τ scheduler for sampling moves at train and evaluation time.
- (Model evaluation) Assess what positions a model believes to be good or not by examining value network outputs. For example, ShabuShabu after only 200 epochs gives the **MATERIAL_ADVANTAGE** test position (Fig. 5b) a close to drawing evaluation ($q = +0.065$), while after 3200 epochs, ShabuShabu seems to have learned to value material more with a strongly confident evaluation ($q = +0.871$).
- (Model evaluation) Assess how *effectively* a model finds a good move by examining the distribution of priors (policy network outputs) and visit counts: is the model considering all moves evenly? The correct move the most? Or focusing on an incorrect move? We found that over time, ShabuShabu becomes increasingly opinionated towards stronger moves.

These observations, while not being numerical metrics, offer us concrete insights into model performance and provide us with confidence in the correctness of our implementation. As such, they serve as an important evaluation metric particularly in the early stages of training.

5.2 Quantitative metrics

Evaluating the absolute performance of ShabuShabu is inherently difficult due to the lack of human experts, well-studied positions to serve as a test suite, or other rated engines to compare against. Instead, we compare ShabuShabu against (1) a primitive random play agent which simply samples a move uniformly from available legal moves, (2) the PPO agent, and (3) older versions of the model from saved checkpoints. We use an Elo rating system [1] as a metric of relative strength between the model and other agents as well as between the model’s various checkpoints as training progresses.

We developed a testing setup, `eval`, that runs games between any of these agents under a singular, generic interface. Any combination of human (via manual move input), random agent, PPO agent, and model checkpoint agent can be readily be used with `eval`. In early stages, ShabuShabu’s performance in games against the random agent was evaluated as a proof of concept of the model’s ability to learn at all. We quickly progressed to evaluating ShabuShabu’s performance against older checkpoints, and periodically play games against the model manually to check its strength relative to a human beginner. `eval` provides a number of different facilities (`n_game_match`, `round_robin`, and `tiered_arena`) which we discuss in further detail in [Appendix D](#).

We implement self-play Elo according to Elo’s *The Rating of Chessplayers, Past and Present* [1], in particular Section 3.33, “Rating a Group of Unrated Players” via a round robin special ratings tournament. Following Elo’s recommendation, we assume the skill level of agents are normally distributed with a mean of 1200 and standard deviation of 200, computing the z-score of each model’s raw score to obtain their Elo rating.

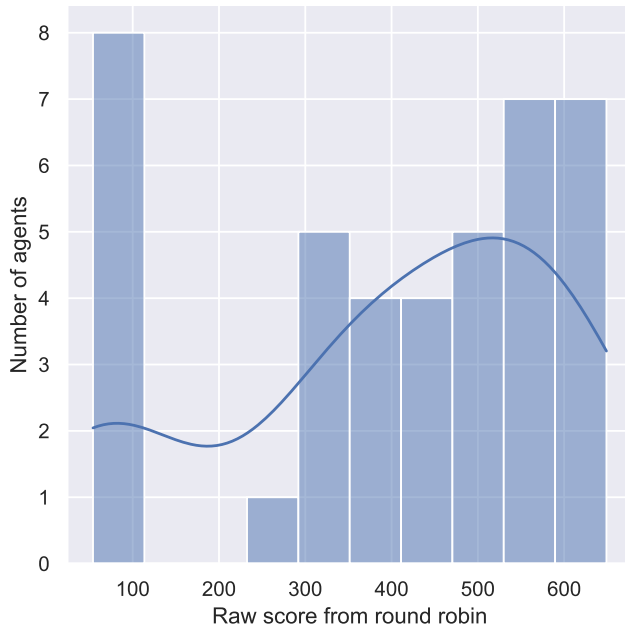


Figure 6: Distribution of raw scores from round robin evaluation tournament, sorted into 10 bins with KDE. We see very weak models collected into the lowest bin, while the remainder of the scores show a strong rightward skew as model strength improves.

Figure 6 shows a potential caveat to our assumption, as the true distribution of agents we evaluated shows a bimodal distribution, with weaker (baseline and PPO) agents scoring extremely poorly and later agents getting high scores. This skew means that relative gaps between Elos in our calculations are not necessarily indicative of relative strength differences. However, comparisons between Elos still remains perfectly valid, meaning self-play Elo is still a useful metric in our case. We hypothesize that given more training time, the number of models which will score extremely high will begin to decrease, and discarding the lower bin would provide us with a better approximation of the normal distribution. To get a clearer picture of relative strengths between two models, we instead employ an alternative approach in which a large number of games are played and the win-draw-loss ratio is examined.

Finally, we must critically pay attention to the quality and variety of games played during `eval` matches. Because we wish to assess model quality playing under a large variety of positions and ensure that the model is not overfitting and simply getting better at playing only one particular line, models play the first move of the game using a high move selection temperature ($\tau = \infty$). This means each game starts with a mostly-random opening phase. After the opening concludes, we set $\tau = 0$, meaning each model plays what it believes to be the absolute strongest move. This temperature schedule provides us with a balance of variety and deterministic play that we believe to be a good indication of strength.

6 Results

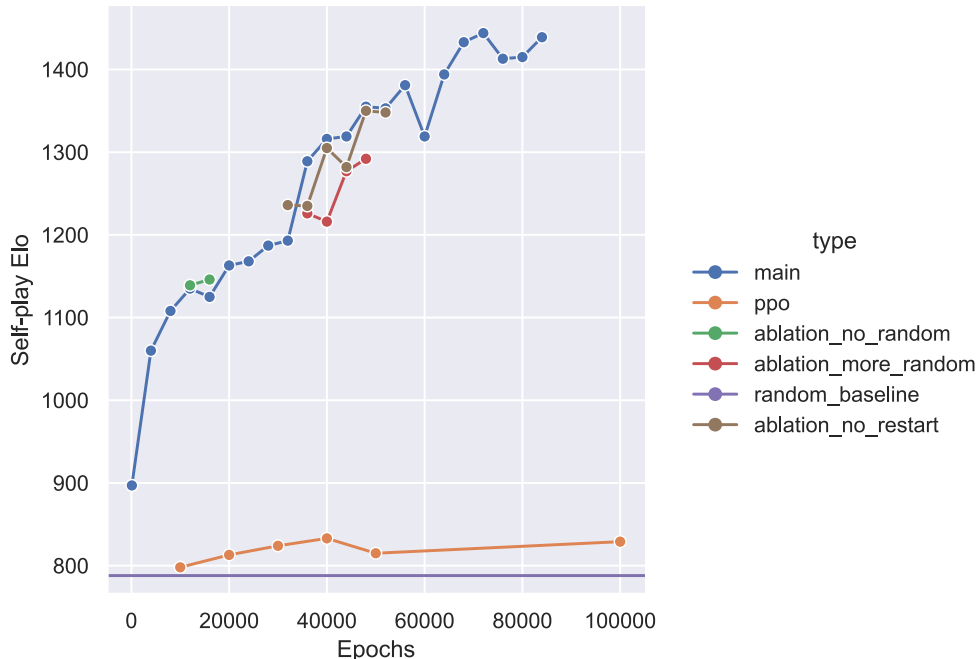


Figure 7: Elo over training steps. The x-axis represents training steps, and y-axis represents self-play Elo. The horizontal line represents a baseline random agent.

Figure 7 was generated from a round robin of 41 agents (22 checkpoints from the main

run, as well as other ablation tests). Each pair of agents played 20 games against each other (10 as black and 10 as white), with a max move cap of 32. The arena thus consisted of 16,400 games in total. We include model checkpoints from the main run at every 4K train steps, as well as ablations detailed in Section 4, a PPO model, and a random agent baseline. We see significant improvement in terms of Elo as training time increases, with significant strength over the PPO model and the random agent baseline. We note the following about the ablations:

- For the `ablation_no_random` run, we noted no significant difference in Elo compared to the main run. However, not incorporating the random position significantly slowed down training, where the time-equivalent checkpoint in the main run was at epoch 26K when the `ablation_no_random` run was at epoch 16K. Therefore, we stopped this ablation run.
- For the `ablation_more_random` run, we noted weaker performance over time. We reason that the increase in random positions caused the model to forget typical game trajectories, resulting in weaker performance.
- For the `ablation_no_restart` run, we noted no significant difference from the main run. We hypothesize that the warm restart was not necessary but not harmful, i.e. the model was not overfitting.

We also looked at the head to head performance with `n_game_match`. For each pair of agents A and B, we ran 100 matches between them, with a max move cap of 64, with the increase in number of matches reducing the variance in the game outcomes. We find a general trend in model improvement over time similar to the Elo results (Figure 8). We can also observe more fine-grained behavior. For example, the ablation run displayed in Figure 8 is the `ablation_no_restart` run - we see that the 48K and 48K ablation runs have similar H2H performance against each other, further confirming that the warm restart did not significantly impact model performance. Another interesting example is the performance of 56K versus 64K. 56K performs worse against 40K compared to 64K, but better against 44K, even though 64K had more training steps. We hypothesize that this could be due to the 64K having slightly different playing styles or adjusting its strategies to escape local optima, resulting in varied performance.

Lastly, we include one study of model behavior in practice by evaluating an example position, comparing model decision making between training epochs. We run a MCTS tree search of 800 simulations for each model, with no Dirichlet or root softmax. We see over time that the model is more effectively able to focus on critical candidate moves and thus search deeper on those moves (Figures 9, 10, 11).

7 Challenges

7.1 Handling limited compute

With the exploratory account on Oscar [2], we had the option of training on up to 2 GPUs and 4 CPU cores, or 64 CPU cores. We determined that the limiting factor in our compute was

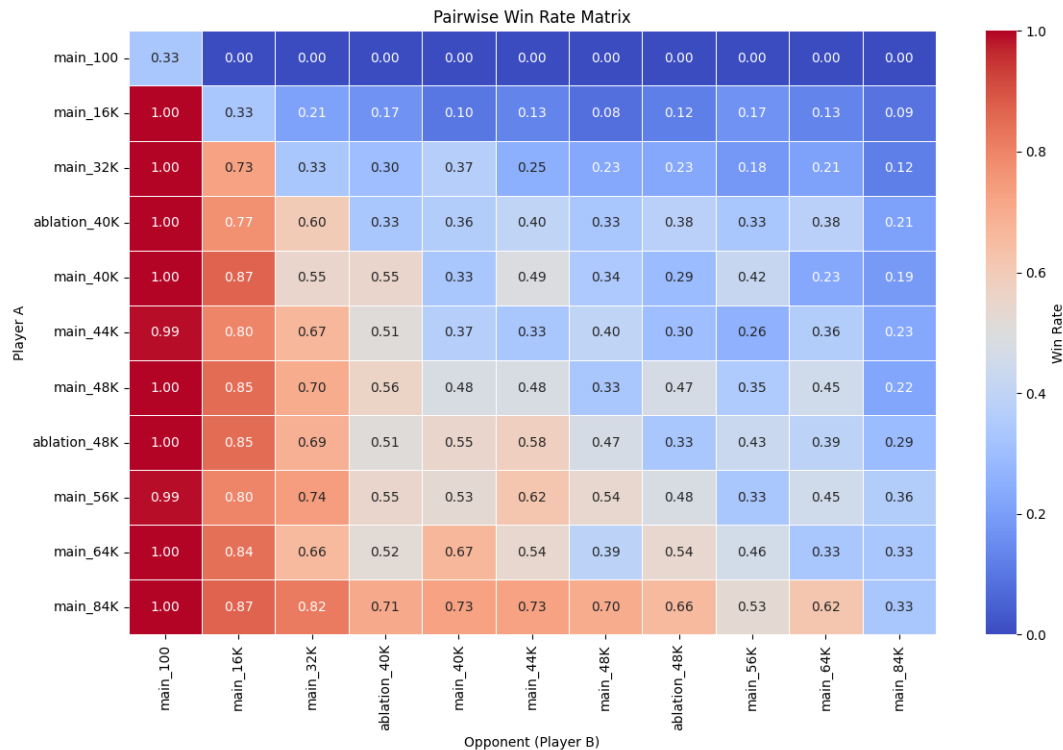


Figure 8: Pairwise win rate between key checkpoints (win rate of “Player A” over “Player B”; larger values means Player A won more). We notice overall improvement as training progresses.

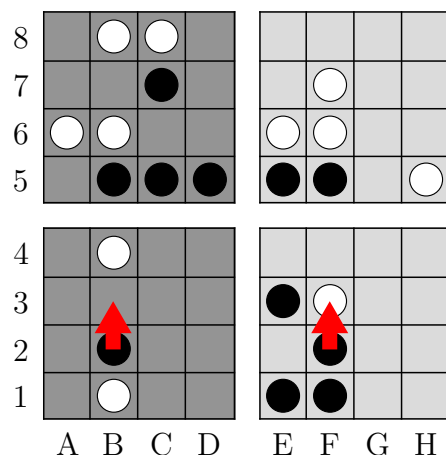


Figure 9: The test position. Black to move. The model’s top candidate move after 1000 training steps, b2f2N, is marked with red arrows. The position is believed to be slightly favored for black, with an evaluation of +0.245. Although the move itself looks logical (setting up b3e3NE#), the model shows a lack of awareness of this, exploring a completely nonsensical variation ...f7b8SE?? (allowing mate) 2. f1b5NE2?? (missing the mate) g6c7S.

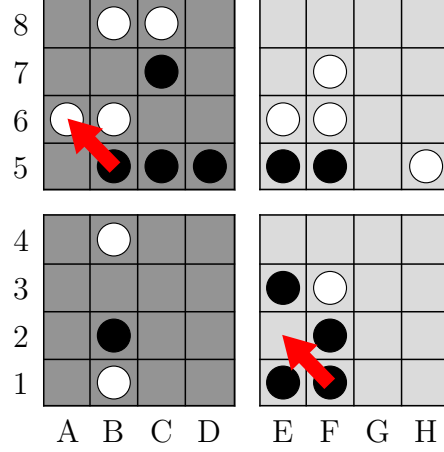


Figure 10: The model’s top candidate move after 40k training steps, f1b5NW, is marked with red arrows. Black is believed to be losing in this position, with an evaluation of -0.417 . This move is a stronger alternative to the previous candidate b2f2N, since in addition to still threatening mate in 1 it also removes white’s A6 stone, giving black a material advantage on the top-left board. The explored variation continues ...b6f3S! (avoiding mate) 2. e2a6NE h5b1W, with white alleviating its awkward doubled stones on the lower-left B file and preparing attacks against the black B2 stone.

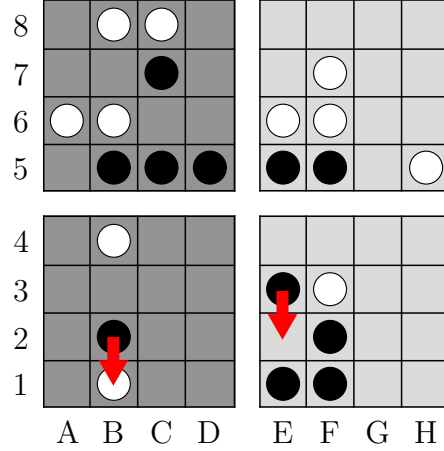


Figure 11: The model’s top candidate move after 84k training steps is e3b2S, finding a forced mate in 2 for black. White’s F3 square is in jeopardy following this move, and because of the flexible positioning of black’s B1 stone, white has nowhere to run (e.g., ...b6f3E2 is met by b1f1NE2#). Note for the old candidate move (f1b5NW), the 84k model finds the refutation ..e6b1N!, which prevents black’s mate threat by depriving black’s passive stone of the ability to move NE by 2 steps. After this move, white gains initiative on the top-right subboard with a double attack on black’s E5 and F5 stones.

game simulation, as the Monte Carlo tree search is computationally expensive. Therefore, we required a fine-grained multiprocessing setup to handle simultaneous game simulation and model training such that neither task was bottlenecked by the other. This required

- Explicitly assigning CPU cores for game generation vs. training.
- Syncing new model weights across workers at the beginning of each model generation.
- Storing training samples in a shared memory buffer and pulling the data locally for the training cores.
- Managing locking for workers and training cores to avoid race conditions, premature weight updating/sharing, or overwriting data in the shared memory buffer.

8 Reflection

- *How do you feel your project ultimately turned out? How did you do relative to your base/target/stretch goals?*

We feel that this project was successful, as we were able to train a fully self play agent for Shobu that we were not able to beat. Although we ourselves are beginners, since the model is stronger than us, we believe that we attained our stretch goal.

- *Did your model work out the way you expected it to?*

The strength of the final model matched our expectations going into the project. Given our limited knowledge of the game, the model identified moves we did not consider when evaluating test positions that we generated; and the final Elo rating as well as our own matches against the model matched our expectation that the model would exceed our playing skill.

- *How did your approach change over time?*

We first made the pivot from PPO to self-play MCTS, after observing instability with training a PPO model. During the training of the MCTS-based model, we made several key implementation changes, both in terms of engineering (for training efficiency) and design (for improving speed of learning). Some things that we adjusted during our training run included:

- The PUCT formula to encourage or discourage exploration.
- Temperature scheduling to increase or decrease the randomness of opening moves.
- Adding random positions into training.

As training progresses, we also built out our suite of evaluation tools and metrics to match our need for evaluating our model.

- *What kind of pivots did you make, if any?*

We made an early pivot from PPO to self-play MCTS, as we noticed that PPO was highly unstable given Shobu’s large branching factor and variable game state. We found this pivot extremely effective as the MCTS model was significantly more stable and responded well to our hyperparameter adjustments throughout training.

- *Would you have done differently if you could do your project over again?*
There are several engineering and implementations details that were important to effectively training the model, including but not limited to: multiprocessing setup for simulation and training; modifying the PUCT equation for MCTS; modifying the temperature scheduler to randomize openings; choosing to use direct cross entropy or KL divergence for the policy loss; trying to augment the training data. We either implemented these during training or were unable to allocate time to implement them; if we were to start the project again, we would address these details earlier to speed up the model development process.
- *What do you think you can further improve on if you had more time?*
We would run further ablations on different architectures for the value and policy heads, as well as evaluating the benefits increasing the depth of the residual backbone. We would have also liked to explore training data augmentation for better sample efficiency, as Shobu is a symmetrical game, and currently the model employs strategies which greatly focus on some areas of the board while potentially leaving weaknesses in others. Finally, although we briefly examined some model visualizations, a deeper dive into interpretability of the model is an interesting avenue of further exploration which we believe could unlock a better understanding of the game itself.
- *What are your biggest takeaways from this project/what did you learn?*
We realized that compute and validation metrics are crucial to training and evaluating these types of self-play agents. There is a necessary overhead of compute and engineering necessary to train these models within a reasonable amount of time. Furthermore, having some validation set, whether it be human qualitative evaluation, quantitative measures like Elo, or databases and previously developed engines (if they exist) are important to evaluate model strength and notice any signs of overfitting in an agent.

References

- [1] Arpad Elo. The rating of chessplayers, past and present, 1986.
- [2] Center for Computation and Brown University Visualization. Oscar. <https://docs.ccv.brown.edu/oscar>.
- [3] Zhuang Liu, Xuanlin Li, Bingyi Kang, and Trevor Darrell. Regularization matters in policy optimization-an empirical study on continuous control. In *International Conference on Learning Representations*, 2020.
- [4] Jamie Sajdak and Manolis Vranas. Shobu. <https://www.smirkanddagger.com/product-page/shobu>, 2019. Accessed April 14, 2025.
- [5] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.
- [6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
- [7] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.
- [8] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.
- [9] David J. Wu. Accelerating self-play learning in go, 2020.

Appendix A: Proximal policy optimization

Another popular method of reinforcement learning is proximal policy optimization [6]. Proximal policy optimization (PPO) aims to find a policy distribution through interaction with the environment, stabilizing the policy updates through clipping and learning a value function to reduce variance. Given Shobu’s complex state space and variable environment and PPO’s balance between ease of implementation and expressive potential, we first tried training and evaluating a PPO model. The architecture of the PPO model was similar to the model described in Section 4.3, excluding the batch normalization layers as PPO can be sensitive to variance caused by batch normalization [3]. The PPO setup can be summarized as:

1. Generate m episodes of games using the current model θ , stopping at a max move cap M . The games states are stored in a memory buffer of max length L . For each episode, an opponent is drawn from the opponent queue, which has max size of o storing previous model checkpoints.
2. Perform the PPO update step, sampling over the entire memory buffer in minibatches of size b for batch number B times.
3. Empty the memory buffer.

Empirically, we observed the following drawbacks:

1. Shobu’s complexity, due to the passive and aggressive move game mechanic for each player turn, caused high variance in each episode. Thus, the model required several episodes before each training step to avoid overfitting to a particular variation of a game.
2. PPO’s high sensitivity to hyperparameters caused training instability (see Table 13 for hyperparameters). The best run resulted in a weak model capable of beating a random agent but was unable to improve beyond that (Figure 12); all other runs resulted in the failure to learn proper Shobu mechanics, with win rates below 20%.

The variance in training motivated the research into MCTS-based reinforcement learning methods, which requires 1) less hyperparameter tuning, 2) MCTS allows for more consistent and varied exploration during training, and 3) the generated games can be kept across train steps. These qualities addressed the observed issues with PPO. Thus, this work focuses on MCTS-based self play.

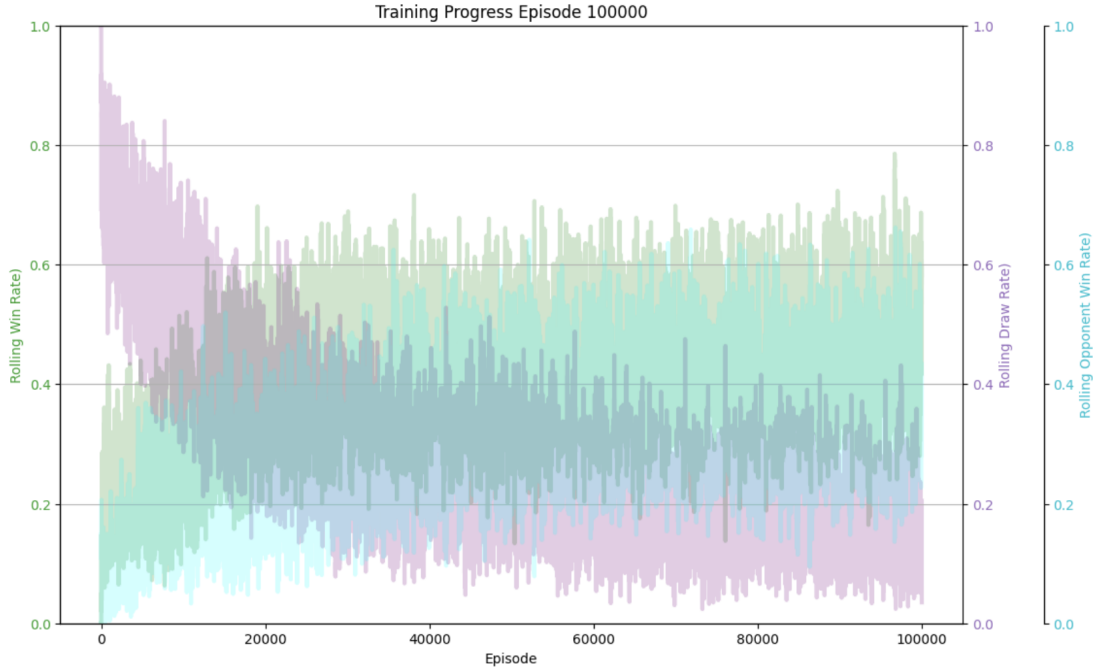


Figure 12: Rolling win rate graph from the best PPO training run. The x-axis represents the number of game episodes simulated. The green represents the win rate for the PPO model. The blue represents the win rate for the opponent, which is drawn from a pool of previous model checkpoints. The purple represents the draw rate. Because the opponent pool is updated periodically, model progress should be seen by the win rate increasing periodically as the model learns new strategies and counters. However, we only observe this behavior in the first 20K games but plateaus for the remainder of training.

Hyperparameter	Values
m	10, 20, 50, 100, 120
M	32, 64, 96, 128, 256
o	2, 4, 8, 12
b	32, 64, 128, 256
B	4, 6, 8, 10, 16, 20
L	50K, 75K, 100K
Value Loss Weight	0.2, 0.5, 1.0
Entropy Loss Weight	0.03, 0.01, 0.003, 0.001
Clip Parameter	0.1, 0.2, 0.3
Discount Factor	0.95, 0.98, 0.99, 0.995
Sparse Reward	1, 5, 10
Opponent Update	250, 500, 1K, 2K

Figure 13: Hyperparameter space for PPO. This is a reduced version for visualization purposes.

Appendix B: MCTS motivation and implementation

We generated a large number of Shobu positions at random ($\sim 10^6$) and counted the number of legal moves in each position. From this analysis, we estimate the branch factor (average number of legal moves in a position) of Shobu to be around 62.2. In contrast, chess has a branch factor of 35, and checkers has a branch factor of 2.8; Go is more complex with a branch factor of 250.

The time complexity of naive minimax search (the traditional tree search algorithm for sequential two-player games) given a branch factor b and a half-ply search depth d is $O(b^d)$, as for each node, we must search on average b elements. For example, to search 8 half-plies deep (4 full moves), we would need to evaluate $\sim 2.24 \times 10^{12}$ Shobu positions. Improvements such as alpha-beta pruning offer some speed-ups, but rely on complex move-order heuristics that are difficult to develop given our limited understanding of Shobu. Given that our value function is also extremely heavy (a large neural network), we opted to proceed with a Monte Carlo tree search (MCTS) instead.

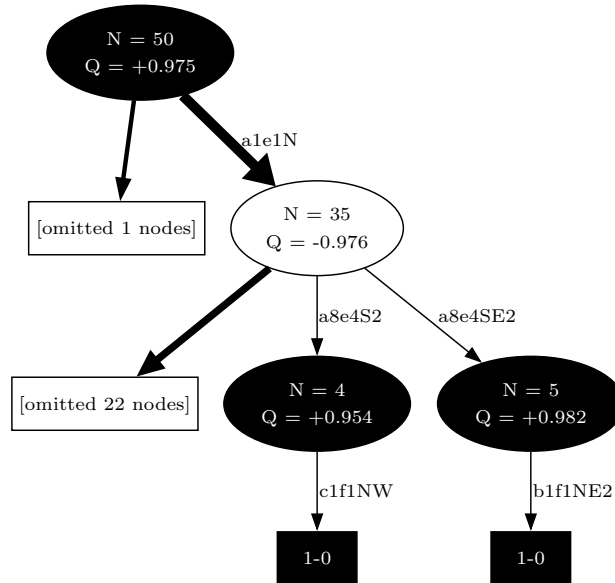


Figure 14: ShabuShabu finds a forced mate in 2 in the `MATERIAL_ADVANTAGE` position (see Fig. 8b) using only 50 simulations. Note black has 232 legal moves in the initial position, so minimax with 50 nodes would not have even been able to fully search the first half-ply.

The strength of MCTS lies in its ability to create highly imbalanced search trees, allowing significantly deeper searches given a certain number of node visits compared to minimax. Figure 14 shows this phenomenon on a smaller scale; the model is able to search three half-plies deep even with only 50 “rollouts” (visited child nodes), whereas minimax would not even be able to finish exploring the first half-ply in 50 visits (in practice, we search with 800 rollouts at evaluation time for improved play strength).

Like AlphaZero, we implement PUCT-guided MCTS (Predictor + Upper Confidence bound applied to Trees). Search proceeds as follows:

1. **Selection.** Starting from the root s and until we reach a leaf node with no children, we repeatedly select a child state c_t to visit according to

$$c_t = \arg \max_c (Q(c) + \gamma u(c)),$$

where $Q(c)$ is the average reward of all child nodes of c , and $u(s, a) = P(c) \frac{\sqrt{N(s)}}{1+N(c)}$, $P(c)$ being the child prior from the policy net and N being visit counts. This formula balances *exploitation*, encouraging revisiting promising nodes with its first term, and *exploration*, encouraging searching new nodes with the second term. We further control exploration by manipulating γ throughout the training phase, as discussed in Section 3.

2. **Expansion.** Move generation is run for the leaf node we arrive at (as long as we have not reached the end of the game, or the game has exceeded a maximum length cutoff), and we pass the position through the model to obtain a static evaluation $Q(n)$ as well as priors $P(c)$ for its children.
3. **Backpropagation** The static evaluation is passed up through all parent nodes to the root, updating $Q(n)$ for each node n along the path.

Appendix C: Model Architecture

We detail the specific implementation of ShabuShabu.

Model Input

The model receives the game state as a $B \times C \times H \times W$ image, where B is the batch size, $C = 8$ is the number of channels, and $H = W = 4$ is the board size. The first four channels represent the locations of the black pieces on the four subboards, where 1 indicates a piece and 0 indicates an empty square. The last four channels represent the locations of the white pieces on the four subboards, where 1 indicates a piece and 0 indicates an empty square.

Backbone

The backbone neural network extracts the features of the given board state. Similar to Alphazero and Katago, we employ a residual network. The backbone neural network consists of the following:

- A 3x3 convolution on the input board state image outputting $c_{backbone} = 96$ channels.
- A stack of $n = 6$ residual blocks. The blocks consist of:
 - A 3x3 convolution on the input tensor outputting $c_{backbone} = 96$ channels.
 - A batch normalization layer.
 - A ReLU activation.
 - A 3x3 convolution on the input tensor outputting $c_{backbone} = 96$ channels.
 - A batch normalization layer.
 - A skip connection adding the output of the batch normalization layer to the input tensor.
 - A ReLU activation.

Value Head

The value head predicts the future outcome of a board state, given the extracted features from the backbone. The value head consists of the following:

- A 3x3 convolution on the input tensor outputting $c_{head} = 32$ channels.
- A global pooling layer that concatenates the mean and maximum of each channels, outputting a vector of size $B \times 2c_{head}$
- A linear layer outputting $c_{value} = 48$ values.
- A ReLU activation.
- A linear layer outputting a scalar value.
- A TanH activation.

Policy Head

The policy head outputs the next move distribution given the features extracted from the backbone. The policy head consists of a policy feature extractor and four policy sub-heads.

- A 3x3 convolution on the input tensor outputting $c_{head} = 32$ channels.
- A flattening layer that outputs a $B \times (c_{head} * 4 * 4)$ tensor.
- A position sub-head that outputs logits on which piece to move.
 - A linear layer that take in a tensor of size $B \times 512$ and outputs a tensor of size $B \times c_{policy} = 256$
 - A ReLU activation.
 - A linear layer that take in a tensor of size $B \times c_{policy}$ and outputs a tensor of size $B \times 64$
- A direction sub-head that outputs logits on which direction to move. It takes in the backbone features concatenated with the position logits.
 - A linear layer that take in a tensor of size $B \times 576$ and outputs a tensor of size $B \times c_{policy} = 256$
 - A ReLU activation.
 - A linear layer that take in a tensor of size $B \times c_{policy}$ and outputs a tensor of size $B \times 16$
- A distance sub-head that outputs logits on how far to move. It takes in the backbone features concatenated with the position logits and direction logits.
 - A linear layer that take in a tensor of size $B \times 592$ and outputs a tensor of size $B \times c_{policy} = 256$
 - A ReLU activation.
 - A linear layer that take in a tensor of size $B \times c_{policy}$ and outputs a tensor of size $B \times 2$
- A aggressive move sub-head that outputs logits on which piece to move. It takes in the backbone features concatenated with the position, direction, and distance logits.
 - A linear layer that take in a tensor of size $B \times 594$ and outputs a tensor of size $B \times c_{policy} = 256$
 - A ReLU activation.
 - A linear layer that take in a tensor of size $B \times c_{policy}$ and outputs a tensor of size $B \times 64$

Appendix D: Quantitative evaluation tools

Besides one-off games, `eval` provides three similar but complementary facilities for playing larger numbers of games for a significantly improved picture of true model performance. `round_robin` plays a round robin tournament between an unbounded number of agents; each round, each agent plays twice against every other agent—once as black, and once as white—and tournaments may proceed for any number of rounds. Since the total number of games scales rapidly ($nt(n - 1)$, where n is the number of participants and t is the number of rounds), we run games in each round in parallel.

`eval` also provides `n_game_match`, which runs a large number of games between only two agents in parallel, and `tiered_arena`, which runs an arena-style tournament between a pool of unrated players. Although in `n_game_match` we cannot get an idea of the relative strengths of larger numbers of agents as in `round_robin`, we can get a much more precise picture of the relative strengths of two agents. This is useful especially when comparing two model checkpoints with a somewhat small number of epochs between them, as the improvement may only become apparent over a larger number of games.

Finally, `tiered_arena`, our primary evaluation mechanism, provides flexibility with scheduling evaluation jobs in that it maximizes use of available cores by removing the restriction that only games in a certain round may be parallelized and writes out all game results to non-volatile memory in a CSV format, allowing the tournament to occur incrementally over the course of several days as newer-epoch models are generated in the meantime.

`round_robin`, `n_game_match`, and `tiered_arena` return win-draw-loss statistics as well as an Elo rating for each player, which serves as our primary metric for evaluating model performance.