Learning Shobu By Self-Play Through Reinforcement Learning

Ayushman Choudhury, Brandon Gong, Seowon Chang

April 17, 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.

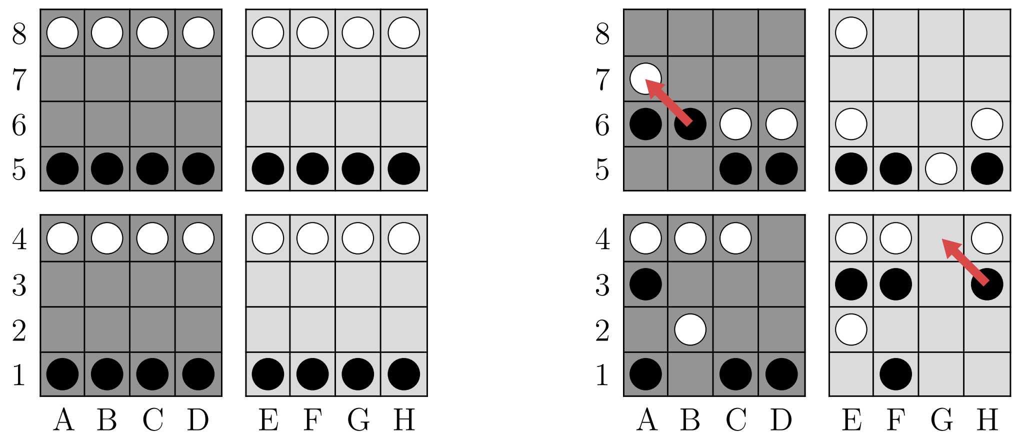

Figure 1: Left: (a) The Shobu starting position, from black's point of view. The lower two subboards are black's home boards, and the upper two are white's. Right: (b) A game of Shobu in progress. Here, black can select their H3 stone on their right home board as the passive stone and their B6 stone (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 Section 2.2—and a RL model based on Monte Carlo tree search (MCTS), discussed from Section 3 onwards, which reduces the search space by primarily considering moves believed to be "good."

Of course, no discussion of a 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 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. One extremely important utility in this set is `explorer`, an interactive REPL which effectively enables full inspectability and explainability of model behavior, which is otherwise a black box due to the inherent randomness and complexity of MCTS. These tools and our evaluation metrics using them are covered in Section 5.

2 Related Work

2.1 MCTS-based Reinforcement Learning

2.1.1 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.

2.1.2 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.

2.2 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

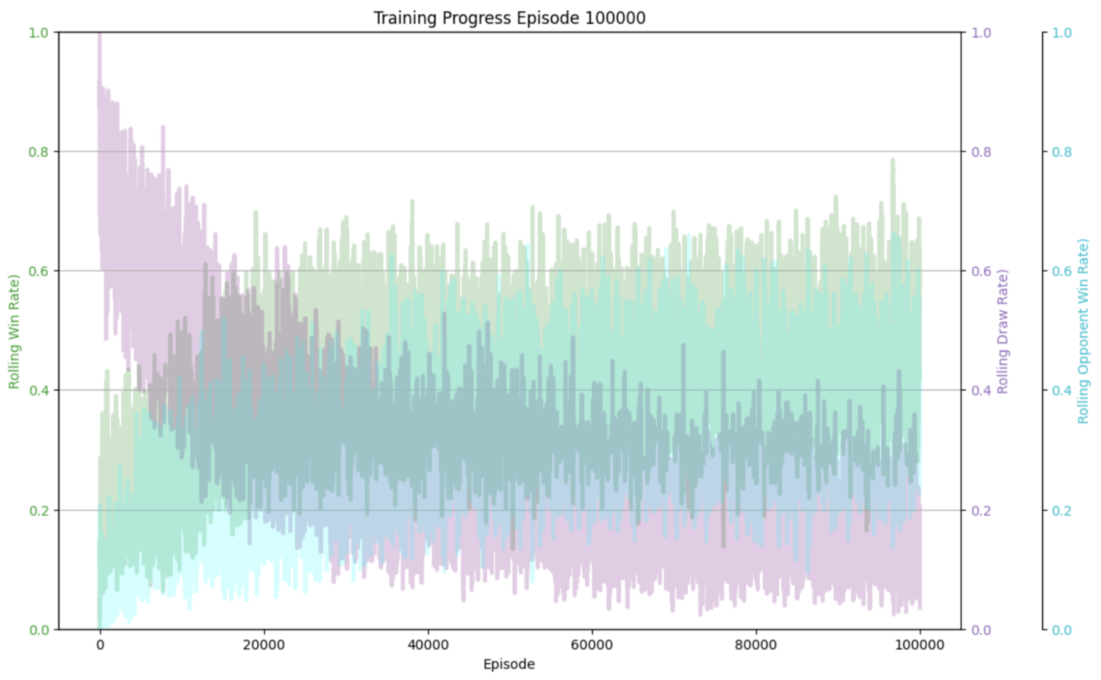

Figure 2: Rolling win rate graph from the best PPO training run. The x-axis represents the number of game episodes simulated. The green represents the win rate for the PPO model. The blue represents the win rate for the opponent, which is drawn from a pool of previous model checkpoints. The purple represents 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.

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:

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

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 3 for hyperparameters). The best run resulted in a weak model capable of beating a random agent but was unable to improve beyond that (Figure 2); all other runs resulted in the failure to learn proper Shobu mechanics, with win rates below 20%.

3. The on-policy nature of PPO caused training data inefficiency, where the simulated data was thrown out after each training step. For us, this was inefficient given our compute resources.

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. All of these qualities addressed the observed issues with PPO. Thus, this work focuses on MCTS-based self play.

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

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.

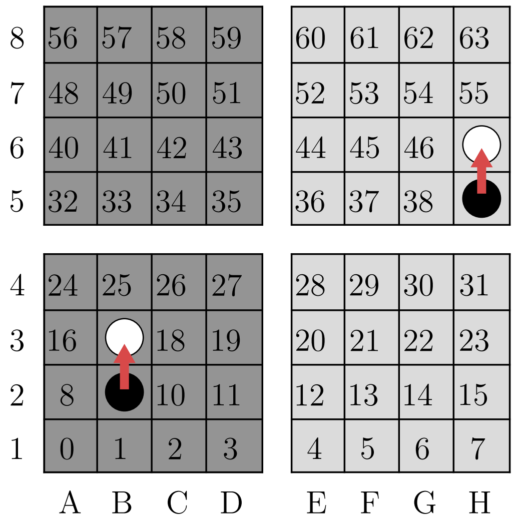

Figure 4: The Shobu bitboard layout with example aggressive moves b2N and h5N.

Figure 4 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

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 much search on average b elements. For example, to search 8 half-plies deep (4 full moves), we would need to evaluate $\sim 2.24 * 10^{12}$ 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.

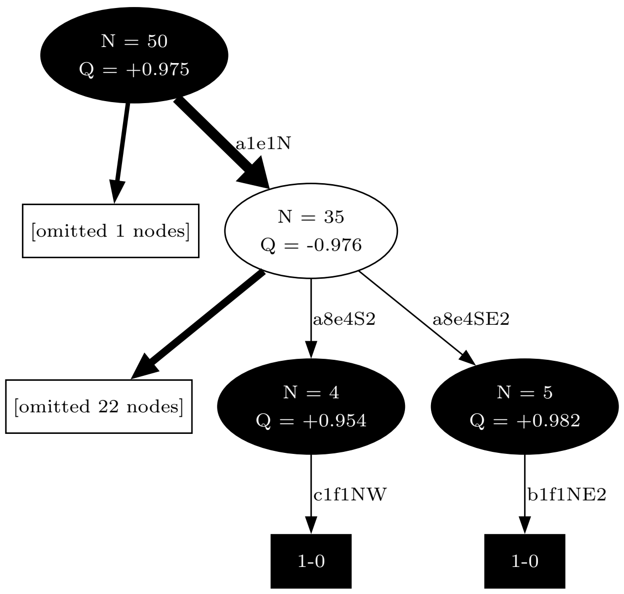

Figure 5: The model finds a forced mate in 2 in the `MATERIAL_ADVANTAGE` position (see Fig. 8b) using only 50 simulations. Note black has 232 legal moves in the initial position, so minimax with 50 nodes would not have even been able to fully search the first half-ply.

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 5 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 introduced a coefficient $\gamma = 1.5$, called the *exploration bonus*, 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 6).

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.

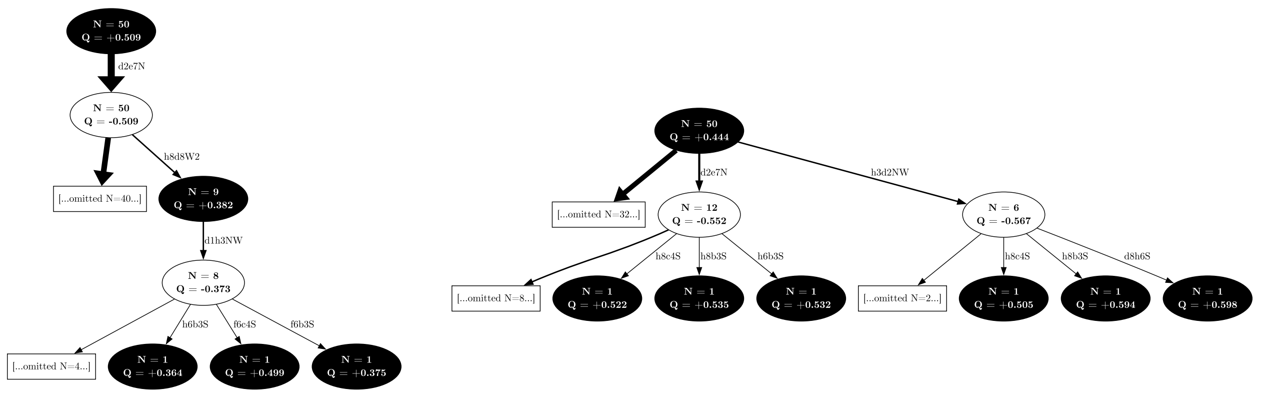

Figure 6: Illustrating the effects of γ, the exploration bonus. On the left is MCTS executed with the default PUCT formula ($\gamma = 1$), where the policies and values are provided with a very primitive, untrained model. Note that it only considers one candidate move from the root. On the right, we 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.

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

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

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

Here we outline the architecture of the model. At a higher level, 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.

4.3.1 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.

4.3.2 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:

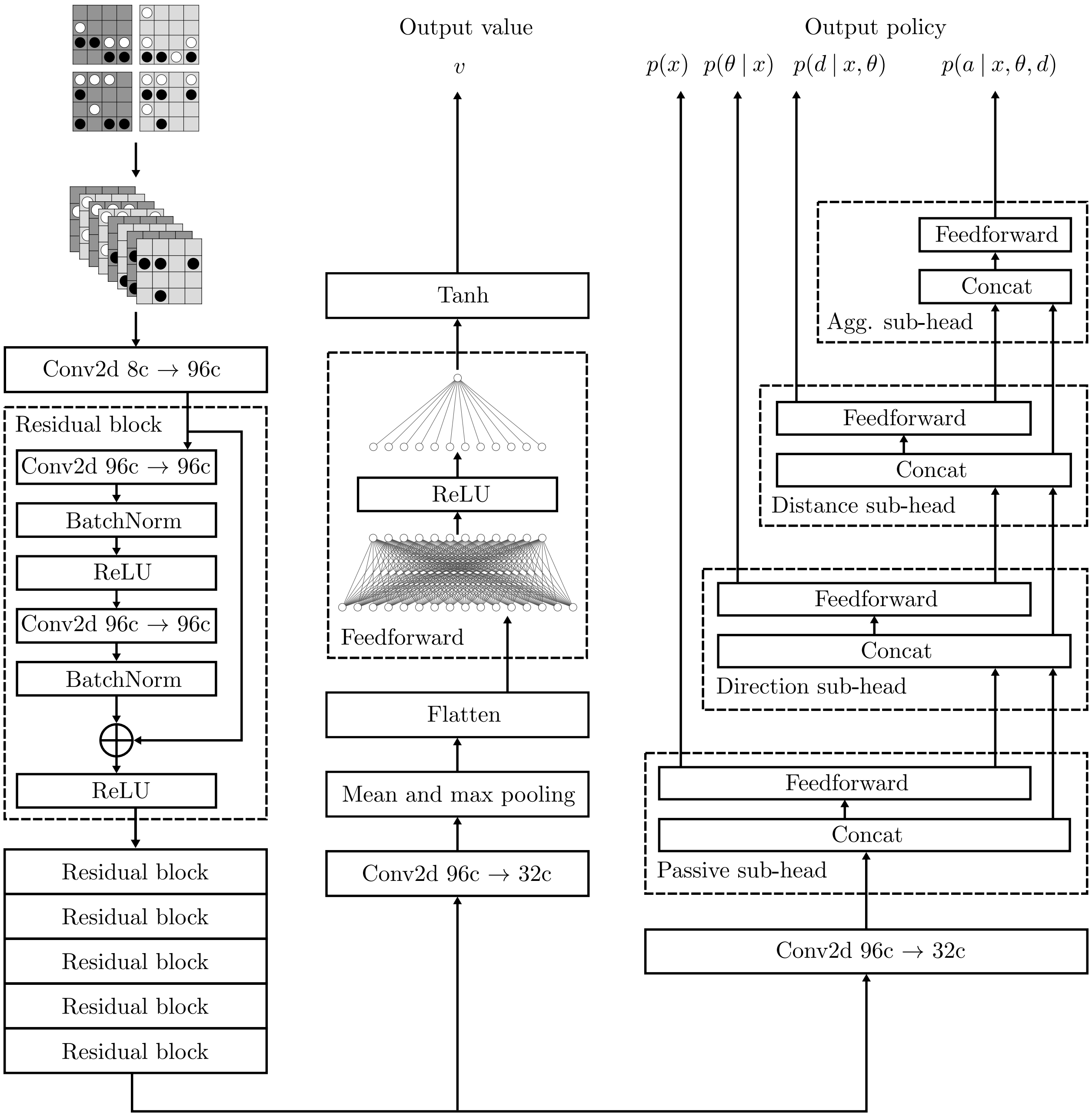

Figure 7: The model architecture. On the left: input transformation and initial convolution, backbone. Center: value head. Right: Policy sub-heads, outputting conditional logits for passive piece x, direction θ, distance d, and aggressive piece a. Note that all residual blocks are identical; the first one is expanded to show internal structure. Likewise, all feedforward blocks consist of two layers with an intermediate ReLU activation. Input, hidden, and output sizes vary based on usage.

- 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 ouput of the batch normalization layer to the input tensor.
  - A ReLU activation.

4.3.3 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.

4.3.4 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 $B \times 64$

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

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

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 $\beta_{L2} = 3e\text{-}5$ 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 quality 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 $n = 100$.

5 Metrics

5.1 Evaluation

5.1.1 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.

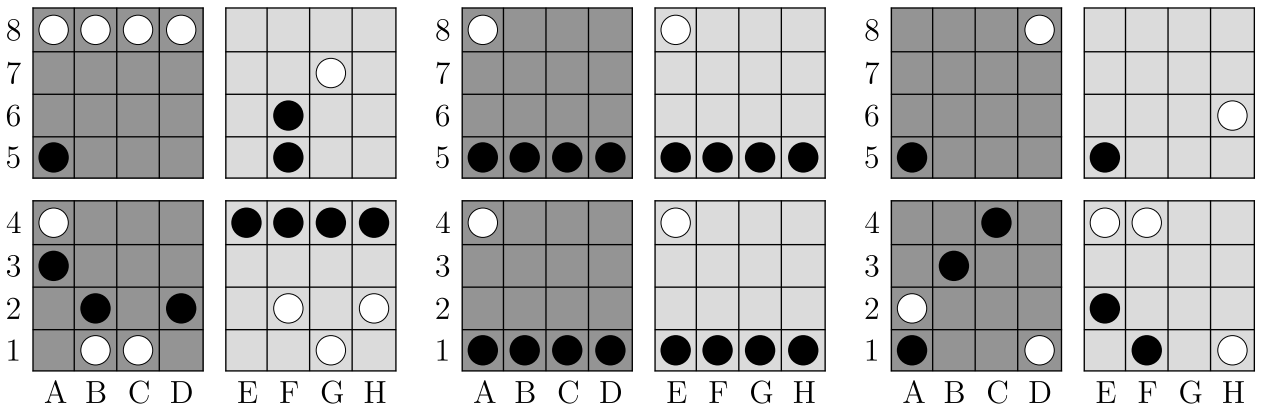

Figure 8: Shobu test positions. From left to right: (a) `BLACK_MATE_IN_1`: black has an immediate win available, b2f6NE2. (b) `MATERIAL_ADVANTAGE`: black has a large material advantage, but no immediate win. (c) `PUSH_OPTIONS`: black's best move is a1f1E2, pushing off white's H1 stone while dodging white's threat of h6a2S. Other pushes are available, but none avoid the trade on A1.

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.
- (Bugfix) Detect the presence of end-game reward sign issues causing the search to avoid winning states (Fig. 8a).
- (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 randomess, 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, the model after only 200 epochs gives the `MATERIAL_ADVANTAGE` test position (Fig. 8b) a close to drawing evaluation ($q = +0.065$), while after 3200 epochs, the model 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, the model becomes increasingly opinionated towards stronger moves. For example, in the `PUSH_OPTIONS` test position (Fig. 8c), the 200 epoch model considers the correct move a1f1E2 only 5 times out of 800 simulations, with more time wasted on nonsensical moves such as e2a5NE2 (70/800 visits). The 3200 model finds b3f1E2, a more reasonable but suboptimal move. In contrast, the 4700 epoch model snaps onto the strongest move readily, visiting it 473 times out of 800 simulations.

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.

5.1.2 Quantitative metrics

Evaluating the absolute performance of the model 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 the model 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, the model'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 the model's performance against older checkpoints, and periodically play games against the model manually to check its strength relative to a human beginner.

Besides one-off games, `eval` provides two 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. Although 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.

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

We implement a simple rating system with a rating-independent K-factor of $K = 24$ and no provisional rating. All agents begin the match with a starting Elo of 1200, and ratings are updated at the conclusion of each game.

The update formula is given as follows. Suppose player A has rating R_A and player B has rating R_B, and they play a game in which they respectively score S_A and S_B points, $S_A, S_B \in \{0, 1\}$, $S_A + S_B = 1$.

We have (using Elo's original assumption that player skill is normally distributed and scaled such that a 400 Elo difference means a player is approximately expected to win 10/11 games):

$$E[S_A] = \frac{1}{1 + 10^{(R_B - R_A)/400}},$$
$$E[S_B] = \frac{1}{1 + 10^{(R_A - R_B)/400}}.$$

Given these expected scores, we linearly adjust each player's Elo rating by our K-factor of 24:

$$R'_A = R_A + 24(E[S_A] - S_A),$$
$$R'_B = R_B + 24(E[S_B] - S_B).$$

Note that in each case, one of $E[S_A] - S_A$ and $E[S_B] - S_B$ will be negative, and the other will be positive; thus both ratings are adjusted in the appropriate directions, with greater differences between expected score and actual score leading to larger adjustments.

This system, though simple, has its drawbacks, as without provisional ratings a very strong agent matched against an extremely weak opponent can only improve their Elo by a diminishing number of points per game. Thus with our system, relative gaps in Elo may not be so indicative of true relative strength, though comparisons between Elo as comparisons of strength are still valid. More games should be played in order to allow the ratings to properly converge.

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 three moves of the game using a high move selection temperature ($\tau = 3$). 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.

5.2 Goals

1. Base goals

    - Create baseline agents to compare against: random agent, user agent, PPO (proximal policy optimization) agent
    - Train a self-play MCTS agents

    - Implement evaluation tools (ELO calculation, MCTS explorer tool)

2. Target goals
    - Implement curriculum learning framework to strengthen MCTS agent
    - Develop Shobu theory/strategies by observing model play

3. Stretch goals
    - Train an above-average (strong amateur level) agent that is competitive with human play

6 Ethics

- Why is Deep Learning a good approach to this problem?
  Like chess and Go, Shobu is a turn-based game which can be modeled as a Markov decision process. The evaluation of a board state and the decision process behind picking a move are complex functions that are not easily modeled by heuristics and involve a constantly changing environment affected by the players. Therefore, a reinforcement learning approach lends well to learning a model that can effectively play this game, and we leaned on prior work done on similar games (chess and Go) to guide our model design and evaluation.

- What are some environmental concerns about self-play algorithms, which require heavy compute to generate simulations?

  These self-play algorithms require a large amount of computational power to generate these simulations. In our case, we are training our model on Oscar, using 64 CPUs per person. In the cases of larger simulations, extensive amounts of GPUs or TPUs may be needed.

  The more processing units are needed, the more energy is consumed — both the direct electricity consumption to run these models, but also the indirect costs, like cooling technology and computing center maintenance if these computing resources are used remotely, as we have done. If this energy is drawn from fossil fuels, then it can be causing environmental damage, via the harmful effects of burning fossil fuels.

  There are many ways to alleviate these concerns. On a broad scale, alternative sources of energy, and more efficient uses of limited energy sources, can decrease the environmental footprint of a computing center. More relevant to us, we can also increase the efficiency of our code, by using more efficient data structures, decreasing function call overhead, and in general training smaller models.

7 Division of Labor

- Ayushman:

    1. MCTS implementation
    2. Model training/checkpointing
    3. Model evaluation

- Brandon (Capstone):
    1. Shobu environment implementation
    2. MCTS implementation
    3. MCTS performance optimization
    4. Environment and MCTS evaluation tools
    5. Model evaluation

- Seowon (Capstone):
    1. PPO model implementation
    2. PPO model training and evaluation
    3. MCTS neural network design and implementation
    4. MCTS performance optimization
    5. Model training/checkpointing
    6. Model evaluation

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, Dharshan 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.