Learn to play Go

Albert Liu
Department of Computer Science
Stanford University
albertpl@stanford.edu

1 Introduction

The ancient Chinese game of Go is considered the hardest classic board game and until recently it had been a grand challenge for AI. It has relatively simple rules but vast possibilities of strategies, which typically requires decades of study to master. Recently AlphaGo [1], AlphaGoZero [2] and AlphaZero [3], developed by a DeepMind team, has rocked the world of AI and Go by soundly beating the best Go human players in the world.

The goal of this project is to train a agent that can win games against our oracle without incorporating lots of domain knowledge of the game. We formulate Go as a turn-taking, two player, zero-sum game of perfect information. The state space is 8 recent boards of possible placements of the stones and the player who plays that turn, represented as either black stone or white stone. The state is fully observable as input. The action space is any legal position of a stone, given the current state, and a pass action. We set the reward to +1 if current player wins, -1 if loses. No reward otherwise. Figure 4 shows a concrete example of this game and white stone is making a move at $P15$.

Generic search methods, such as Minimax search, are intractable due to enormous search space ($\sim 10^{170}$) and large number of legal move per state(~ 250). The other challenge comes from the need for massive computational resources. For example, AlphaGoZero use 64 GPUs and 19 CPUs to play 5 millions games to achieve superhuman level [2].

2 Related Work

Monte Carlo Tree Search (MCTS) is a common ingredient for recent Go programs. Browne *et al.* gives a great survey on MCTS [4]. Kocsis *et al.* develops Upper Confidence Tree [5] algorithm by applying UCB1 [6] as tree policy, which greatly improves the state of the art for Go programs. There are various improvements. RAVE [7] distributes the updated values of the simulation results to larger set of nodes than the sampled path from root to the leaf. And it applies AMAF [8] (all-move-as-first), which keep previous simulations and apply it whenever we examine the same (s, a) edge and then combines it with regular simulations. We explored vanilla MCTS with light rollout and didn't engage any variations more than what is specified in [2].

Deep Neural Network (DNN), especially Convolutional Neural Network, has been used to predict the next move and as the basis for strong Go program, such as [9] and [10]. The first Go program that defeats top professional player is AlphaGo [1], which combines DNN, supervised learning, MCTS and reinforcement learning. Subsequently, the same DeepMind team develops AlphaGoZero[2] without human data or guidance beyond the game rules, which learned exclusively through self-play reinforcement learning. Then they generalize it to more diverse game setting and build AlphaZero [3]. Two of our approaches are based on ideas in [1] and [2] but with much less computational resources, as described in 4.5.

3 Dataset and Environment

We collect experiences, i.e. game records, for our reinforcement learning based approaches. The standard approach is to have agents play against themselves through simulated games, i.e. self-play. We record the board position at each step, the move, the final reward and network outputs.

As illustrated in Figure 5, our simulator includes a Go gameplay engine, which is based on Pachi [11], an open source Go framework, and parts of the environment is adopted from OpenAI gym [12]. We choose Pachi mainly because of the optimal gameplay time. For each player, the input is a 2D numpy array (e.g. 9×9), representing current board with each element encoding the color of stone, 0 for empty, 1 for black stone and 2 for white stone. The output of each player is the next move to play, which is a scalar that either represents encoded board position (e.g. 50 represents "E4") or a pass action (e.g. 81). Each player can persist the board representation, moves and etc., to disk, which are used as experiences for further reinforcement learning. Figure 5 gives one concrete example on the input/output of the system. We use Keras [13] as our deep learning framework.

4 Approaches

We restrict our attention to 9×9 board to cut down computational complexity. Minimax search or AlphaBeta pruning, however, is still intractable. Instead, we start with UCT [5]. Then we will explore policy gradient based methods. Finally we will combine tree search and function approximation approach, following AlphaGoZero [2]. But first, we discuss the baseline and oracle.

4.1 Baseline

For baseline, we write a random player which selects legal move randomly, except that it won't commit suicide, i.e. filling in its own eye, which is an empty position where all adjacent positions and three out of four diagonally adjacent positions are its stones (or edges).

4.2 Oracle

We use Pachi built-in *UCT* engine as our oracle which is said to achieve highest amateur expect level (KGS 7 dan) on 9×9 board with reasonable hardware, where dan is expert Go ranking, from 1 to 7 (low to high). Technically the *UCT* engine implements RAVE [7], instead of classic UCT. This engine incorporates lots of heuristics and 3×3 pattern, e.g. self-atari detector and ladder testing [11].

4.3 Monte Carlo Tree Search

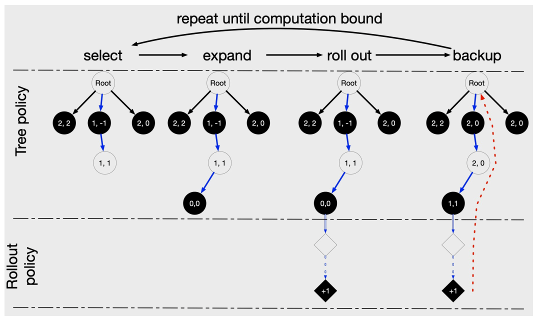


Figure 1: MCTS

We apply MCTS approach to the game of Go. Each node of the search tree is a game state, which is tuple of board representation and player to play. And there is an edge between node s and node s' if and only if there exists some move a such that $s' = \text{Succ}(s, a)$, where $\text{Succ}(s, a)$ is function that returns the next game state s', which is the alternated player and a new board produced by playing the move a on the board of s. For each s, a, we maintain visit count $n(s, a)$ and value $q(s, a)$.

As described in Algorithm 1, when player is asked to make a move, it create a new tree with current game state s_0 as the root node. Then the player simulates multiple random games (each is called a rollout) to evaluate the moves. For each rollout, initially it follows tree policy to traverse the tree down and then expands new node once it reaches the leaf of the tree. From the newly expanded node, it will follow a rollout policy (we use random policy) to finish the game. The final reward is backed up to all the nodes along the path back to root node. Once all rollouts are done, the player selects the move with the maximum visit counts from the root node.

```
Input: root node $s_0$: current game state
Input: $c > 0$: parameter to control the degree of exploration
1  while within computation bound do
2      $s \leftarrow s_0$
3      $\Delta \leftarrow \emptyset$
4      // selection based on tree policy, e.g. UCB1 [6]
5      while $s$ is in the tree do
6          $a \leftarrow \arg\max\{q(s,a) + c\sqrt{\frac{\log \sum_{a'} n(s,a')}{n(s,a)}}\}$
7          $\Delta \leftarrow \Delta \cup \{(s,a)\}$
8          $s \leftarrow \text{Succ}(s,a)$
9      expand tree with the new node $s$
10     continue the game from $s$ with random policy and let $r$ be the reward
11     // update each node on the path $s_0 \to ... \to s$
12     for $s, a \in \Delta$ do
13         $n(s,a) \leftarrow n(s,a) + 1$
14         $q(s,a) \leftarrow q(s,a) + \frac{1}{n(s,a)}(r - q(s,a))$
15 return $\arg\max_a q(s_0, a)$
```

Algorithm 1: MCTS with UCB policy

Figures 6, 7, 8, 9 show concret examples for each MCTS step.

4.3.1 Other techniques

To increase exploration, we follow [2] and apply these techniques

- The second term in UCB1 can be viewed as uniform priors on uncertainty. Additional Dirichlet noise can be added to it, $(1 - \epsilon)\text{Uniform}(s, a) + \epsilon\eta_a$ and $\eta \sim \text{Dirichlet}(\delta)$
- Instead of greedy selection after each tree search, we select the action proportionally to the visit counts for the first τ moves of the games which allows more diverse moves at the beginning of the games. For the rest of the game, we switch back to greedy selection.

Note c, δ, τ are considered hyperparameters for our MCTS algorithm. And we use notation $\delta = 0.0$ to mean no Dirichlet noise is added.

4.4 REINFORCE with baseline

The policy gradient based methods, such as REINFORCEMENT [14], allow us to learn stochastic policy naturally, compared to using $\epsilon-$ greedy policy in Q-learning. REINFORCE method uses Monte Carlo method to sample return to compute an unbiased estimation of the gradient. But typically Monte Carlo methods have high variance and therefore produces slow learning. An extra baseline term, as suggested in [15], can greatly reduce the variance without any bias, because the added term has an expectation of zero. We now describe how we apply this method in this problem.

4.4.1 Network architecture

Let t be current move index of current player, S_t be state the player makes decision on and A_t be move the player will take. As shown in Figure 10, we have a two headed deep neural network to approximate both the policy function $\pi_\theta(A_t|S_t)$ and state value function $v_w(S_t)$. The input to this network is S_t, which is an binary array in the shape of $17 \times 9 \times 9$, where 9 is the board size and 17 is the number of channels and each channel is described as following

1. For $i = 0, ..., 7$, channel i is a 9×9 2D array and each element is an indicator if current player has a stone at the position, at time $t - i$.
2. For $j = 0, ..., 7$, channel $j + 8$ is a 9×9 2D array and each element is an indicator if opponent player has a stone at the position, at time $t - j$.
3. The last channel (16) encodes current player, 1 for black player and 0 for white player.

The history of the board is necessary because *Ko* (repetition is restricted) is not directly observable otherwise.

The outputs of the network are

- $\pi_\theta(A_t|S_t)$

  A 82-tuple of representing the probability vector for taking each action A_t for S_t for current player. The action space size is $9^2 + 1 = 82$.

- $v_w(S_t)$

  A scalar between -1 and +1, representing the predicated value for S_t, from the perspective of current player.

  Figure 11 shows an sample input/output for the neural network.

θ and w share most of the parameters (ResNet and two Fully Connected layers), except that θ includes a Softmax layer and w includes a Tanh layer.

The network architecture is similar to what is used in [2], except that we use shorter residual tower with less convolution filters. Table 5 describes the architecture in details.

4.4.2 Training procedure

We update parameters θ and w continuously through last 10 games gathered through self-play. Let α be the learning rate, G_t be return. Let s_t, a_t, r_t be specific training sample, representing player taking action a_t at state s_t and yielding final reward r_t.

For policy function and θ. The objective function is $J(\theta) = V_{\pi_\theta}(S_t)$ and update rule is

$$\theta_{t+1} = \theta_t + \alpha(r_t - v_w(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t)$$

For value function and w. The objective function is $J(w) = ||v_w(S_t) - G_t||^2$ and update rule is

$$w_{t+1} = w_t + \alpha(r_t - v_w(s_t))\nabla_w v_w(s_t)$$

4.5 Combination of tree search and DNN

Finally we will combine MCTS and function approximation with DNN, in the same way as in [2]. This method can be viewed as an general form of policy iteration [15]. And we reuse the same DNN as described in 4.4.1. On high level, we repeat the following two steps until it converges

- Self play with DNN guided tree search

  We use self play to gather experiences. The tree search is augmented by the DNN in the following way

  - whenever we expand a new node s, instead of playing out the rest of the game to estimate the value, we perform an inference over s and use $v_w(s)$ as our estimation of the final reward.
  - add a prior distribution $\pi_\theta(a|s)$ to the uncertainty item

$$a \leftarrow \arg\max\{q(s,a) + c\pi_\theta(a|s)\frac{\sqrt{\sum_{a'} n(s,a')}}{1+n(s,a)}\} \tag{1}$$

  Essentially MCTS yields a better policy (than our parameterized policy function π_θ) and produces a sample of state s_t, probability vector p_{s_t} (based on visit counts after each search) and final reward r_t (outcome of the game), for every move t in the self play.

- Training of DNN

  We use experience gathered in self-play as training samples and train with the loss function

$$\mathrm{l}(\theta, w) = \sum_i (v_w(s_t) - r_t)^2 - \sum_a p_{s_t,a} \log \pi_\theta(a|s_t)$$

  This is a form of policy evaluation, i.e. we ask DNN to produce policy and value estimation as close as possible to statistics from the tree search.

We make the following variations to address the computation limitations

- Bootstrap weights θ, w by supervised pre-training, using 1000 game records played between oracles and training procedures 4.4.2.
- Simplify network architecture, as described in 5, to reduce run time for inference and training.

5 Experiments & Error analysis

5.1 Setup

All games are on 9×9 boards. *komi* is set to 0.5 to eliminate draw outcome. Evaluation results, including margin and win rate, are averaged over 100 games played between our agent and opponent, with our agent as black stone. The margin is the final score of our agent minus opponent's final score, taking *komi* into considerations. Positive margin means our agent wins the game, negative for loss. The win rate is defined as a fraction of the number of games our agent wins over total games.

All computation is done on one Linux box with one Nivida Titan X GPU. We use Adam [16] optimization with Cyclical Learning Rate scheduling[17]. The batch size is 32.

5.2 MCTS

First we want to understand how strong vanilla MCTS is. Like any Monte Carlo method, the accuracy for value estimation of each game state in MCTS is dependent on the number of rollouts. And therefore it is crucial to minimize the run time for each rollout so more rollouts can be used for each move. So we report median run time per game, on top of win rate, margin and number of rollouts per move.

Table 1: Results of MCTS v.s. opponents, $\tau = 20, c = 0.2, \delta = 0.0$

opponent policy	number of simulations per move	win rate	margin	time per game (second)
random	100	0.87	10.0	9.7
random	1000	1.0	11.5	94.7
pachi *UCT* engine	1000	0.03	-24.5	77.7

The number of rollouts certainly increases the strength. But we see our MCTS agent losses more games against pachi *UCT* engine even though it is based on tree search approach as well. We speculate that the reasons are

- Heavy rollout policy

  Unlike our MCTS, pachi entails more Go knowledge when selecting next move during the game simulation. This is called **heavy** or strong rollout policy, whereas ours is light rollout policy, i.e. no heuristics other than not filling eyes. Pachi's rule based simulation policy is handcrafted to mix in heuristics such as, if the last move has put its own group in *atari* we capture it; *Nakade*a move is played inside the eyeshape to prevent the opponent from splitting it into two eyes [11].

- Priors in expansion selection

  We expand a new node by randomly selecting a legal move. But Pachi apply again a set of heuristic function to this decision, equivalently adding a prior probability distribution. Specifically it takes the progressive bias strategy [7], which adds virtual simulations based on the applied heuristics with decreasing weights as the nodes are explored.

5.3 Exploration & Exploitation in MCTS

There are various hyperparameters that control our preference over exploration and exploitation. We are curious how they will influence the results.

1. After τ moves in each game, we would use search results in a greedy way. Until then, we sample proportionally to the visit counts. It appears that we can win more games by being less greedy at the beginning of the game.

Table 2: Impact of tau, with $c = 0.2, \delta = 0.0$

	$\tau = 7$	$\tau = 13$	$\tau = 20$	$\tau = 25$
win rate	0.0	0.0	0.03	0.03
90th percentile margin	-6.5	-6.3	-0.5	-3.4

2. Parameter c is the weight of the second term in selecting action 1 during tree search. We report average win rate, maximum search depth, defined as the maximum length of the path from root node to leaf node per game and min/max search breadth, defined as the ratio of visited children nodes over all children for root node. We see

Table 3: Impact of c, with $\tau = 20, \delta = 0.0$

	$c = 0.2$	$c = 0.6$	$c = 1.0$
win rate	0.03	0.01	0.0
max search depth	49.5	23.0	15
min search breadth	0.48	1.0	1.0

   that a bigger c value gives higher score to unexplored move and therefore greatly increases the search breadth ($\sim 100\%$), at the cost of shallow search though ($> 50\%$). And it turns out that the search depth is more important for our MCTS.

5.4 REINFORCE with baseline

Next we run experiments with our policy gradient based methods. Each training batch is sampled randomly from recent 100 games of self-play and then we train one epoch for each iteration. We evaluate every 10 iterations and report the results.

It appears that the learning is not making too much progress. The average margins various between -30 and -80 but doesn't show a clear trend. Further investigates show that the DNN output converges to almost deterministic policy, especially for early moves. This is detrimental to the learning as we are not exploring enough. We believe it converges too early to some less desirable local minimum.

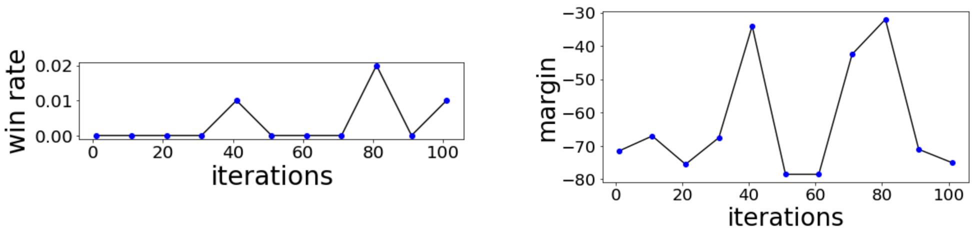


Figure 2: REINFORCE with baseline: 100 iterations self-play/training

5.5 Combination of search and DNN

Last we test the approach that combines MCTS and DNN, as we described in 4.5. Each training batch is sampled randomly from recent 10 games of self-play with 1000 MCTS simulations per move. We train one epoch in each iteration. Then we evaluate results by playing against our oracle and report results in Figure 3.

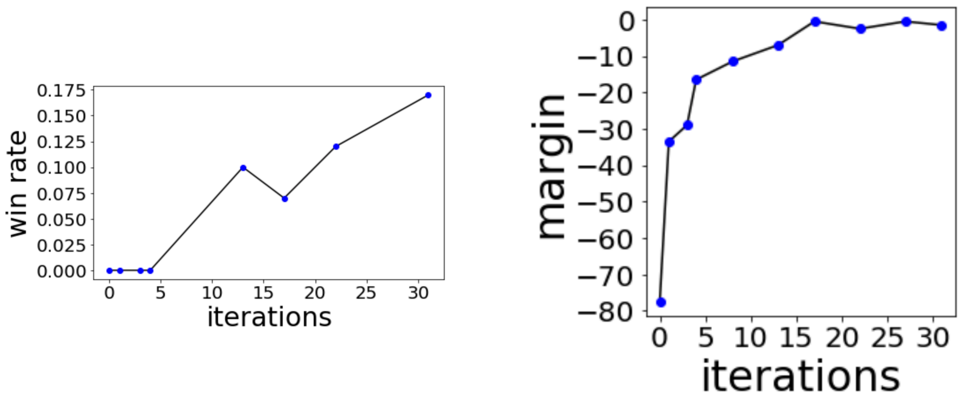


Figure 3: combination of search and DNN with 30 iterations self-play/training

It appears to win significantly more games than vanilla MCTS or policy gradient based approach after 8 iterations. We are then curious what contributes to the improvement.

Table 4: Tree search statistics with and without DNN

	win rate	margin	number of plys	max search depth	min search breadth
without DNN	0.03	-24.5	77	49.5	0.48
without	0.17	-1.5	34	9	0.26

We see in Table 4 that with the use of DNN predictions, the search is much more shallow (9 v.s. 49.5), probably due to much shorter game. And the search explores less moves for each root (0.26 v.s. 0.48). And the win rate and margin significantly improved. We speculate the search efficiency improves significantly, i.e. we select more promising move at each search step. One indirect evidence is it takes much less ($\sim 50\%$) plys on average to complete the game, which occurs when large territory of stones are captured and therefore the board is cleared. This matches our observation when we watched sample games played between our agent and oracle.

The reason for the quality improvements could be more accurate priors (than uniform priors in vanilla MCTS) when we expand a new node, therefore it can focus on moves more, instead uniformly exploring all moves.

6 Conclusion & Future Work

We explored MCTS, REINFORCE with baseline and the DNN guided tree search on the game of Go. The DNN guided tree search, based on [3] with some minor variations, yields best result. The next step is to scale up to reach optimal results. It is not trivial to combine MCTS and DNN effectively. Tree search typically runs much faster than DNN inference and some parallelism is required to reach the optimal throughput. And we have to greatly increase our efficiency and/or engage more powerful computational resources. For example, we could apply virtual loss technique [18].

Also we could use more advanced hyperparameter optimization approach, such Tree Parzen Estimator [19] [20], to select the best set of parameters. As we show in 5.3, they can have significant impacts on the game results.

Source codes

The source codes are hosted at `https://github.com/albertpl/cs221_final`.

Acknowledgments

We appreciate our project mentor William Bakst for his help and great advises throughout the project.

References

[1] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[2] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[4] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[5] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[6] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[7] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[8] Bruno Bouzy and Bernard Helmstetter. Monte-carlo go developments. In *Advances in computer games*, pages 159–174. Springer, 2004.

[9] Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. *arXiv preprint arXiv:1511.06410*, 2015.

[10] Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*, 2014.

[11] Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source go program. In *Advances in computer games*, pages 24–38. Springer, 2011.

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[13] François Chollet et al. Keras, 2015.

[14] Ronald J Williams. *Reinforcement-learning connectionist systems.* College of Computer Science, Northeastern University, 1987.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[18] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[19] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[20] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

Appendix

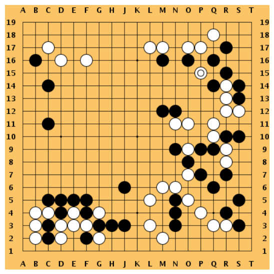
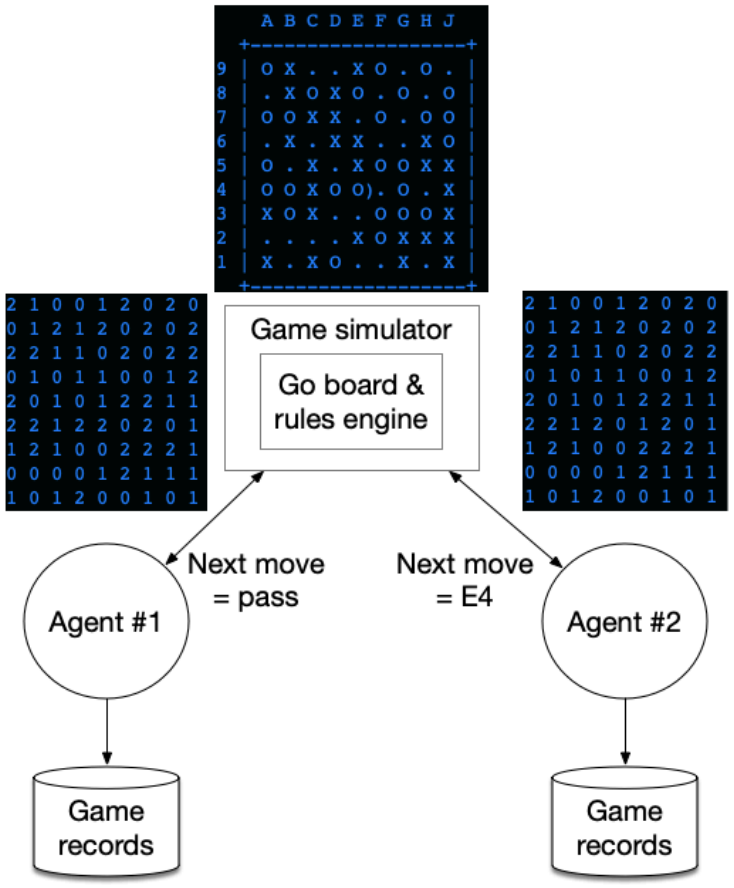


Figure 4: a snapshot of the Go board



```
    A B C D E F G H J
  +-------------------+
9 | O X . . X O . O . |
8 | . X O X O . O . O |
7 | O O X X . O . O O |
6 | . X . X X . . X O |
5 | O . X . X O O X X |
4 | O O X O O). O . X |
3 | X O X . . O O O X |
2 | . . . . X O X X X |
1 | X . X O . . X . X |
  +-------------------+
```

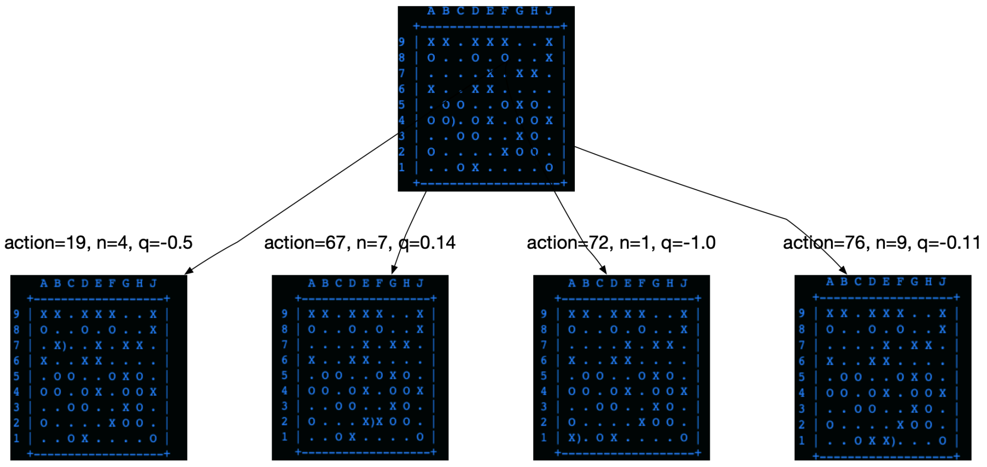
```
2 1 0 0 1 2 0 2 0
0 1 2 1 2 0 2 0 2
2 2 1 1 0 2 0 2 2
0 1 0 1 1 0 0 1 2
2 0 1 0 1 2 2 1 1
2 2 1 2 2 0 2 0 1
1 2 1 0 0 2 2 2 1
0 0 0 0 1 2 1 1 1
1 0 1 2 0 0 1 0 1
```

```
2 1 0 0 1 2 0 2 0
0 1 2 1 2 0 2 0 2
2 2 1 1 0 2 0 2 2
0 1 0 1 1 0 0 1 2
2 0 1 0 1 2 2 1 1
2 2 1 2 0 1 2 0 1
1 2 1 0 0 2 2 2 1
0 0 0 0 1 2 1 1 1
1 0 1 2 0 0 1 0 1
```

Figure 5: Environment setup. For Go boards drawn in all the figures, 'x' represents black stone and 'o' represents white stone. For numpy arrays, '0' means empty, '1' represent black stone and '2' represents white stone. All examples assume 9 × 9 board.

Table 5: Neural Network Architecture

Number	Layer description	Layer input
1	Input layer	N/A
2	Conv2D 32 3 × 3 filters with batch normalization and ReLU activation	1
3	Conv2D 32 3 × 3 filters with batch normalization and ReLU activation	2
4	Conv2D 32 3 × 3 filters with batch normalization	3
5	Add and then ReLU activation	2 & 4
...	repeated (3,4,5) 7 times	...
27	Conv2D 32 3 × 3 filters, batch normalization and ReLU activation	26
28	Conv2D 32 3 × 3 filters, batch normalization	27
29	Add and then ReLU activation	26 & 28
30	Conv2D 2 1 × 1 filters, batch normalization and ReLU activation	29
31	policy output, dense layer 162 × 83 with Softmax activation	30
32	Conv2D 1 1 × 1 filters, batch normalization and ReLU activation	29
33	hidden layer, dense layer 128 × 81	31
34	value output, scalar, dense layer 81 × 1	31



```
   A B C D E F G H J
  +-------------------+
9 | X X . X X X . . X |
8 | O . . O . O . . X |
7 | . . . . X . X X . |
6 | X . . X X . . . . |
5 | . O O . . O X O . |
4 | O O). O X . O O X |
3 | . . O O . . X O . |
2 | O . . . . X O O . |
1 | . . O X . . . . O |
  +-------------------+
```

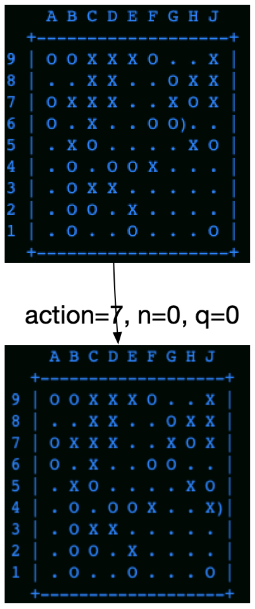
```
   A B C D E F G H J
  +-------------------+
9 | X X . X X X . . X |
8 | O . . O . O . . X |
7 | . X). . X . X X . |
6 | X . . X X . . . . |
5 | . O O . . O X O . |
4 | O O . O X . O O X |
3 | . . O O . . X O . |
2 | O . . . . X O O . |
1 | . . O X . . . . O |
  +-------------------+
```

```
   A B C D E F G H J
  +-------------------+
9 | X X . X X X . . X |
8 | O . . O . O . . X |
7 | . . . . X . X X . |
6 | X . . X X . . . . |
5 | . O O . . O X O . |
4 | O O . O X . O O X |
3 | . . O O . . X O . |
2 | O . . . X)X O O . |
1 | . . O X . . . . O |
  +-------------------+
```

```
   A B C D E F G H J
  +-------------------+
9 | X X . X X X . . X |
8 | O . . O . O . . X |
7 | . . . . X . X X . |
6 | X . . X X . . . . |
5 | . O O . . O X O . |
4 | O O . O X . O O X |
3 | . . O O . . X O . |
2 | O . . . . X O O . |
1 | X). O X . . . . O |
  +-------------------+
```

```
   A B C D E F G H J
  +-------------------+
9 | X X . X X X . . X |
8 | O . . O . O . . X |
7 | . . . . X . X X . |
6 | X . . X X . . . . |
5 | . O O . . O X O . |
4 | O O . O X . O O X |
3 | . . O O . . X O . |
2 | O . . . . X O O . |
1 | . . O X X). . . O |
  +-------------------+
```

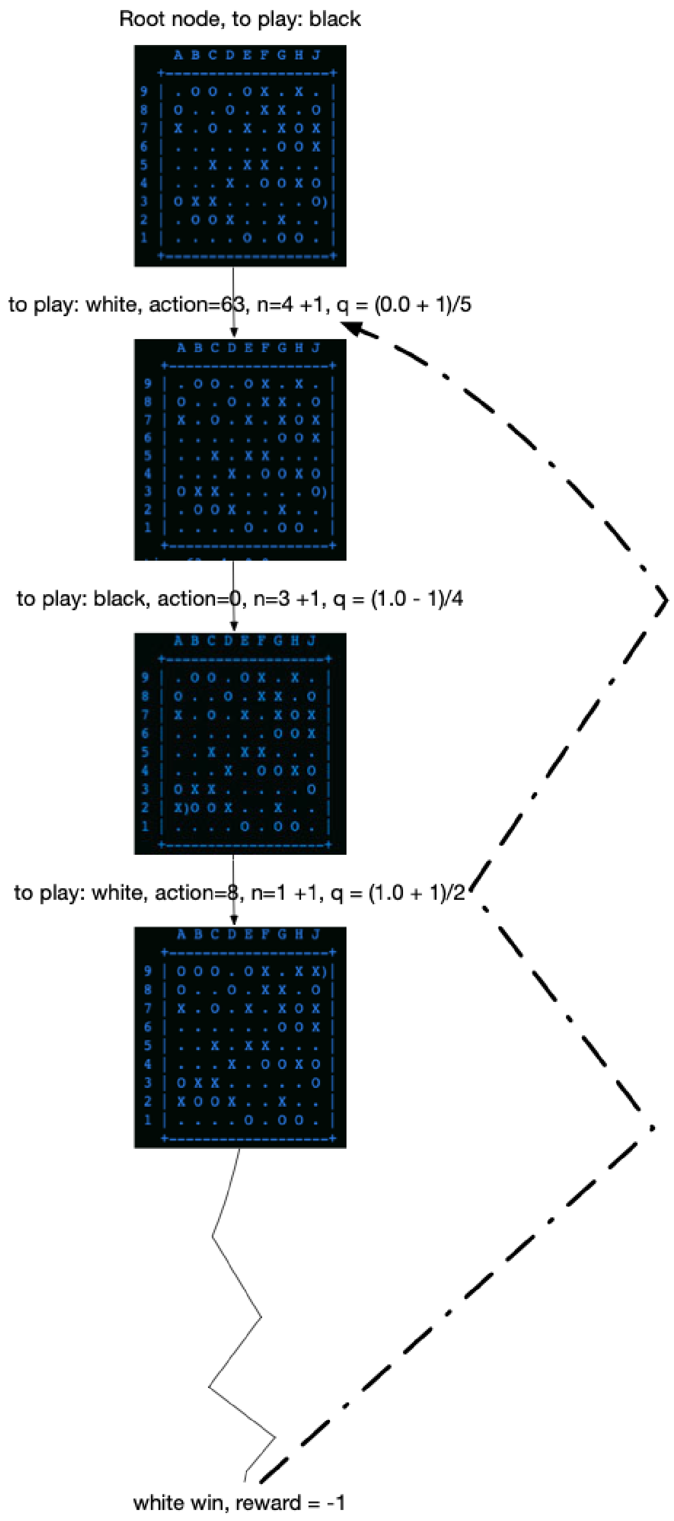
Figure 6: MCTS node selection via tree policy: shows 4 successor states only for brevity. $a = 67$ is selected. n, q is visit count and value for each node respectively. $a = 76$ has more visit counts but lower value.



```
   A B C D E F G H J
  +-------------------+
9 | O O X X X O . . X |
8 | . . X X . . O X X |
7 | O X X X . . X O X |
6 | O . X . . O O). . |
5 | . X O . . . . X O |
4 | . O . O O X . . . |
3 | . O X X . . . . . |
2 | . O O . X . . . . |
1 | . O . . O . . . O |
  +-------------------+
```

```
   A B C D E F G H J
  +-------------------+
9 | O O X X X O . . X |
8 | . . X X . . O X X |
7 | O X X X . . X O X |
6 | O . X . . O O . . |
5 | . X O . . . . X O |
4 | . O . O O X . . X)|
3 | . O X X . . . . . |
2 | . O O . X . . . . |
1 | . O . . O . . . O |
  +-------------------+
```

Figure 7: A new node Succ$(s, a = 7)$, randomly chosen, is expanded with zero visit count and value.


Root node, to play: black
to play: white, action=63, n=4 +1, q = (0.0 + 1)/5
to play: black, action=0, n=3 +1, q = (1.0 - 1)/4
to play: white, action=8, n=1 +1, q = (1.0 + 1)/2
white win, reward = -1

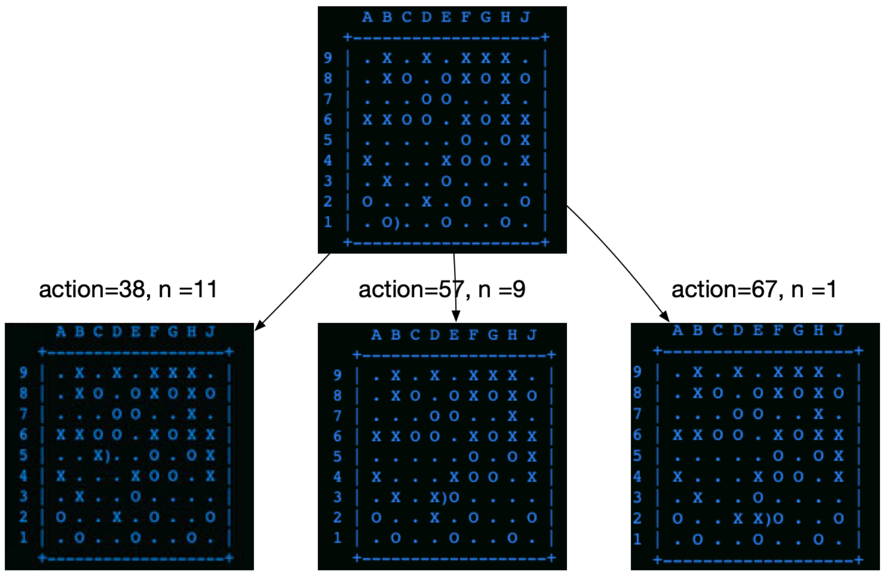
```
    A B C D E F G H J
  +-------------------+
9 | . O O . O X . X . |
8 | O . . O . X X . O |
7 | X . O . X . X O X |
6 | . . . . . . O O X |
5 | . . X . X X . . . |
4 | . . . X . O O X O |
3 | O X X . . . . . O)|
2 | . O O X . . X . . |
1 | . . . . O . O O . |
  +-------------------+
```

```
    A B C D E F G H J
  +-------------------+
9 | . O O . O X . X . |
8 | O . . O . X X . O |
7 | X . O . X . X O X |
6 | . . . . . . O O X |
5 | . . X . X X . . . |
4 | . . . X . O O X O |
3 | O X X . . . . . O)|
2 | . O O X . . X . . |
1 | . . . . O . O O . |
  +-------------------+
```

```
    A B C D E F G H J
  +-------------------+
9 | . O O . O X . X . |
8 | O . . O . X X . O |
7 | X . O . X . X O X |
6 | . . . . . . O O X |
5 | . . X . X X . . . |
4 | . . . X . O O X O |
3 | O X X . . . . . O |
2 | X)O O X . . X . . |
1 | . . . . O . O O . |
  +-------------------+
```

```
    A B C D E F G H J
  +-------------------+
9 | O O O . O X . X X)|
8 | O . . O . X X . O |
7 | X . O . X . X O X |
6 | . . . . . . O O X |
5 | . . X . X X . . . |
4 | . . . X . O O X O |
3 | O X X . . . . . O |
2 | X O O X . . X . . |
1 | . . . . O . O O . |
  +-------------------+
```

Figure 8: After the simulation is done. The result is backed up to all nodes on the path. Note the reward is from black player's perspective and needs to be adjusted for white player.



```
    A B C D E F G H J
  +-------------------+
9 | . X . X . X X X . |
8 | . X O . O X O X O |
7 | . . . O O . . X . |
6 | X X O O . X O X X |
5 | . . . . . O . O X |
4 | X . . . X O O . X |
3 | . X . . O . . . . |
2 | O . . X . O . . O |
1 | . O). . O . . O . |
  +-------------------+
```

```
    A B C D E F G H J
  +-------------------+
9 | . X . X . X X X . |
8 | . X O . O X O X O |
7 | . . . O O . . X . |
6 | X X O O . X O X X |
5 | . . X). . O . O X |
4 | X . . . X O O . X |
3 | . X . . O . . . . |
2 | O . . X . O . . O |
1 | . O . . O . . O . |
  +-------------------+
```

```
    A B C D E F G H J
  +-------------------+
9 | . X . X . X X X . |
8 | . X O . O X O X O |
7 | . . . O O . . X . |
6 | X X O O . X O X X |
5 | . . . . . O . O X |
4 | X . . . X O O . X |
3 | . X . X)O . . . . |
2 | O . . X . O . . O |
1 | . O . . O . . O . |
  +-------------------+
```

```
    A B C D E F G H J
  +-------------------+
9 | . X . X . X X X . |
8 | . X O . O X O X O |
7 | . . . O O . . X . |
6 | X X O O . X O X X |
5 | . . . . . O . O X |
4 | X . . . X O O . X |
3 | . X . . O . . . . |
2 | O . . X X)O . . O |
1 | . O . . O . . O . |
  +-------------------+
```

Figure 9: After all rollout are done. Select the next move with maximum moves. Shows 3 successor states from root state only. The action $a = 38$ is selected.

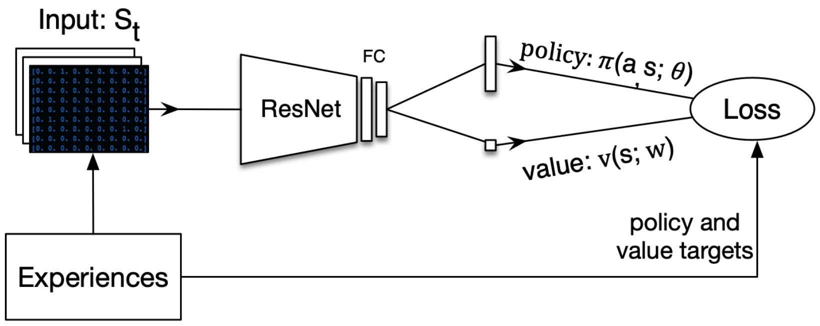
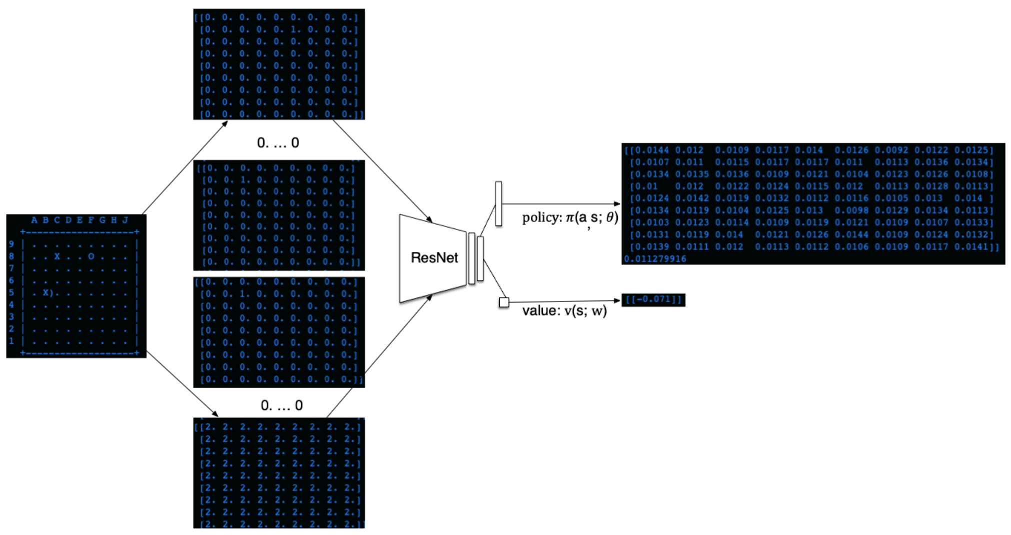


Figure 10: network architecture for policy gradients and NN guided MCTS approaches. The features and policy/value targets are gathered from self-play experiences. FC stands for Fully Connected layer. Features, targets and Loss are discussed in details in text.



```
   A B C D E F G H J
  +-----------------+
9 | . . . . . . . . . |
8 | . . X . . O . . . |
7 | . . . . . . . . . |
6 | . . . . . . . . . |
5 | . X). . . . . . . |
4 | . . . . . . . . . |
3 | . . . . . . . . . |
2 | . . . . . . . . . |
1 | . . . . . . . . . |
  +-----------------+
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

```
[[2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2. 2. 2. 2. 2.]]
```

```
[[0.0144 0.012  0.0109 0.0117 0.014  0.0126 0.0092 0.0122 0.0125]
 [0.0107 0.011  0.0115 0.0117 0.0117 0.011  0.0113 0.0136 0.0134]
 [0.0134 0.0135 0.0136 0.0109 0.0121 0.0104 0.0123 0.0126 0.0108]
 [0.01   0.012  0.0122 0.0124 0.0115 0.012  0.0113 0.0128 0.0113]
 [0.0124 0.0142 0.0119 0.0132 0.0112 0.0116 0.0105 0.013  0.014 ]
 [0.0134 0.0119 0.0104 0.0125 0.013  0.0098 0.0129 0.0134 0.0113]
 [0.0103 0.0123 0.0114 0.0109 0.0119 0.0121 0.0109 0.0107 0.0133]
 [0.0131 0.0119 0.014  0.0121 0.0126 0.0144 0.0109 0.0124 0.0132]
 [0.0139 0.0111 0.012  0.0113 0.0112 0.0106 0.0109 0.0117 0.0141]]
0.011279916
```

```
[[-0.071]]
```

Figure 11: example for an input/output for neural network. The input is $17 \times 9 \times 9$ arrays, created from the board position. There are two outputs, one 82-tuple of probability vector π_θ and one scalar $v_w \in [-1, 1]$. Refer to the text for more details.