Learning to Play Go

Yun Chen*, Wenyuan Zeng*, Yutian Feng†, Kaihua Sun*

December 2021

Abstract

Go is known as the most challenging classical game for artificial intelligence because of its complexity. Prior to deep learning methods, modern computer Go systems rely on processing millions of possible future positions to play well, which is slow and inefficient. In this paper, we train a neural network to parameterize the target policy. Two learning paradigms are used: behavior cloning and self-play. We compare our learned policy with a number of baselines and show significantly better performance.

1. Introduction

The ultimate goal of artificial intelligence (AI) is to build a general intelligent actor that is capable of learning and understanding any intellectual tasks. Among many possible candidate, board game playing seems to be one of the most ideal test case for an artificial intelligent actor because of several reasons. First, it is commonly believed that even for human, one needs high intelligence and long time training in order to perform well. Second, board game can happen in a cyber world with abstracted representations. This helps focusing on developing high-level intelligence such as reasoning rather than low-level intelligence such as perceiving the world[1]. Third, board game by nature involves interaction between two actors, which is very similar to the Turing Test.

In fact, building a good board game AI is indeed the focus of many research in past decades. An early milestone is Deep Blue (3), which shows super-human performance on chess for the first time. Despite its impressive performance, it is also arguably true that chess is not the most difficult board game. Instead, go is usually considered as the most complex board game in the world due to its tremendous branching factor. Top go players are always super talented. For a long period, many people believed that there will not be an AI that can beat human on go. However, in 2016, AI take the last pearl on the crown. AlphaGo (14) exhibit super human performance by combining neural networks and monte-carlo tree search. One year later, AlphaZero (15) significantly outperforms its elder brother utilizing self-play learning technique. Notably, AlphaZero can learn from scratch without any supervisions from human players! These breakthroughs show how powerful a neural network based AI can be with well designed learning algorithms.

In this paper, we aim to learn a neural go AI player similar to AlphaGo and AlphaZero. We compare two learning paradigms: behavior cloning and self-play and use a neural network to parameterize the target policy. The spirit of this paper is not focusing on providing a novel approach, but rather a faithful implementation of existing method and summarizing insights/results we gain during our experiments. Due to the computation budget, we restrict the go board to a smaller size and accordingly adjust model and other hyper-parameters. We compare our learned policy with a number of baselines and show significantly better performance measured by win-rate.

2. Related Work

Building artificial intelligence for board games has always been a fascinating and active research area. Many great work has been proposed and AIs have shown dominant performance in many games, including checker (13) in 1992, othello (2) in 1998 and chess (3) in 2002. Most of these methods rely either on hand crafted features or manually designed expert system. However, for more sophisticated game such as go, it is almost impossible to come up with effective AI system in such ways. For a long time, many people believe go is simply intractable for AI systems.

These beliefs got challenged in 2010s with the surges of deep learning. Deep learning is a machine learning technique which can automatically learn effective features from large amount of data and has impressive model capacity. In the recent decades, it has shown tremendous successes in many complicated tasks, such as image classfication (9),

* {yun, wenyuan, kaihua}@cs.toronto.edu, † yutian.feng@mail.utoronto.ca

[1]This by no means claim the low-level intelligence is not important. In fact, many of the recent advances for AI in both academia and industry surges from low-level vision community.

object detection (12), natural language processing (7). This becomes the key to solve the last stubborn board game: go. In 2016, AlphaGo (14) first exhibits super-human performance on go game. It uses traditional monte-carlo tree search to choose the next action. Meanwhile, AlphaGo employ behavior cloning and neural networks to make some "fuzzy inference", which significantly reduces the otherwise intractable searching time. Later, AlphaZero (15) further improves the performance via self-play.

In this project, we follow the line of AlphaGo and AlphaZero, and implement our neural go AI for a smaller board size (9×9). We borrow some of open-sourced implementation of the go game environment which are not related to the core algorithms, such as the game simulator and UI frontend etc. Please refer to our Code/README for all the third-party codes we're borrowing.

3. Method

Our goal is to learn to play Go with a neural network. Specifically, given an arbitrary state of a game, our model is asked to pick an action for the next step just like human players. One of the major difficulties is how to learn such a model. To this end, we propose and compare two learning paradigm: behavior cloning and self-play. The former one is known to be efficient but it is hard to outperform expert demonstration. On the contrary, the second one is difficult to learn but has great potentials to exhibit super-human performance. In the following sections, we will first describe our problem formulations, including the input representations and model architectures. We then explain our inference procedure using Monte-Carlo Tree Search (MCTS). Finally, we will describe how we utilize the behavior cloning and self-play for learning.

3.1. Neural Go Player

A board game, e.g., Go, typically involves two player who play in turns. At each timestep, one of the player choose an action $\mathbf{a}_t$ based on the current state of the board game $\mathbf{s}_t$ and her policy $\pi(\mathbf{a}_t|\mathbf{s}_t)$. Here, we use $\mathbf{a}_t$ to denote an arbitrary action allowed by the game, e.g., choose an empty entry on the board and place a go stone. The policy π is a deterministic / stochastic function mapping from a valid state $\mathbf{s}_t$ to an action $\mathbf{a}_t$ or a distribution of actions $p(\mathbf{a}_t)$. Note that for complicated games such as go, a good policy is typically super complex as well. Therefore, we propose to use one of the most powerful machine learning model for learning such a complex policy, namely parameterize π with a neural network. Next, we will explain the input representation and the model architecture.

Input Representation The first question is how to represent the game state $\mathbf{s}_t$ such that we can feed it into our neural network. Since $\mathbf{s}_t$ represents the state of the game, one straight-forward way is to simply take a screenshot of the board. Such an image of course encode all the information we need to play the game, as it is how human interact with the game. However, this might be unnecessarily complicate. It requires the model to learn to parse the scene in addition to play the game. Recall that our main goal of this project is learning to play go, we thus choose a more compact encoding.

Suppose the board is $N \times N$, where each (i, j) entry is a valid location for placing a go stone (at the beginning of a game). Within the scope of this project, we assume perfect knowledge of the world. This decouples the complexity of perception from our policy learning. More specifically, we define our input as a tensor with size of $N \times N \times C$ and each entry can take either value of 0 or 1. This 2D-image-like representation can be efficiently processed by convolutional neural networks and thus be compatible with our backbone model. The first T channels out of the total C channels represent the current and past configurations of black stones. For example, suppose the game has been played for a while and the timestep comes to t. We encode the configuration of black stone at t into a $N \times N$ matrix, where an entry (i, j) is 1 if and only if there is a black stone occupying that location on the board and 0 otherwise. Such a $N \times N$ matrix becomes the first out of those T channels. We can further encode more information by represent the black stone configuration at $t-1, t-2, \cdots, t-T+1$ respectively and concatenate them all together to form T channels. Although it seems duplicate information, we argue adding those history information can help our model to focus on a highlighted region in which area players are actively placing their stones. Next, we do similar things for white stone and result in another T channels. The last 2 channels out of C encode the turn of players, e.g., the first channel is all 1 and second is all 0 if it's turn for the black stone at t and vice versa. As a result, the total channel $C = 2T + 2$. We use $T = 8$ in practice and thus $C = 18$. This concludes our input encoding.

Model Given the input tensor with size $N \times N \times C$, our model will generate two outputs: an action distribution $p(\mathbf{a}_t|\mathbf{s}_t)$ and a scalar value $v(\mathbf{s}_t) \in [0, 1]$. The action distribution is a $N \times N + 1$ vector, where the first $N \times N$ values in this vector represent the probability of placing a go stone on the $N \times N$ board, and the additional 1 indicates "PASS action" – performing no action. All values in this vector sum to 1. The scalar value $v(\mathbf{s}_t)$ is also known as value function. It represents the goodness of the current state $\mathbf{s}_t$ from the perspective of the current player. More specifically, this value predicts the win rate of the current player given the current state $\mathbf{s}_t$. Both of these two outputs will be used during inference as we will explain in Section 3.2.

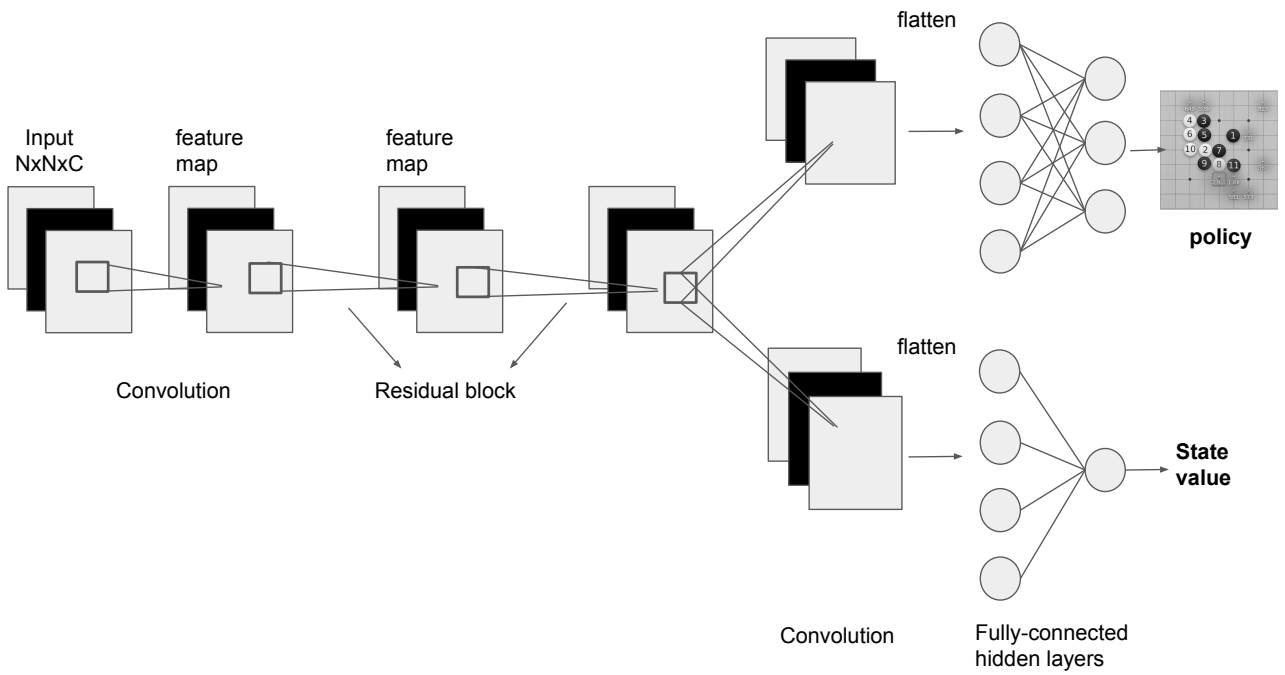

Figure 1. **Model Architecture**. Board position is converted to a $N \times N \times C$ tensor, followed with one convolution layer and two residual blocks. Two branches are appended to predict the state value and action respectively.

In order to process the input tensor and generate the outputs, we build a backbone network adapted from ResNet (9). More specifically, we first apply a convolution layer with kernel size of 3 and channel number of 256 on top of the input, followed by a Batch Normalization (10) layer and ReLU (11) activation layer. Next, two blocks of residual layers are applied. Each residual block is consisted of two convolutional layer with kernel size of 3 and channel number of 256. There are also batch normalization and ReLU activation layer after each convolutional layers. Besides, in addition to naively apply these layers sequentially, there is an extra skip-connection. Formally speaking, if the input tensor to a residual block is x and layers in the block is represented as $layers()$, then the output (incorporating the skip-connection) is $out = x + layers(x)$. Despite its simplicity, such a simple modification significantly ease the training process and enable training a really deep network (9). Note that we employ a small board size $N = 9$ due to limit computation budget, and thus there is no pooling (downsampling) layers in the network. In practice, if the input size is big, one can potentially utilize pooling layers to save computation and memory footprint.

Finally, we use two branch of headers to predict the $p(\mathbf{a}_t|\mathbf{s}_t)$ and $v(\mathbf{s}_t)$ respectively. For both of them, we apply one convolutional layer followed by a few fully-connected layers. We flatten the outputs from the convolutional layer into 1-D vectors before feeding to the fully-connected layers. The output dimensions of fully-connected layers are designed to match the dimensions of the final desired outputs. For instance, the final fully-connected layer for predicting $p(\mathbf{a}_t|\mathbf{s}_t)$ has a shape of N^2+1 while that of $v(\mathbf{s}_t)$ is 1. Besides, the normalization of $p(\mathbf{a}_t|\mathbf{s}_t)$ and $v(\mathbf{s}_t)$ can be achieved by softmax and tanh functions respectively. The overall model architecture is illustrated in Figure 1.

3.2. Inference with MCTS

Assuming we have a trained policy, how can we perform inference. The most straight-forward way is to directly utilize the $p(\mathbf{a_t}|\mathbf{s}_t)$ and sample action from such a distribution. However, the major limitation with this approach is that it can lead to high bias if the policy is not well trained. To tackle this, we employ Monte-Carlo Tree Search (MCTS) (6) similar to the AlphaGo (14). MCTS is a tree-based search technique that can improve a given policy during inference time. More specifically, it uses the given policy to simulate different future configurations and outcomes of the game. As one can imagine, if a configuration is closer to the end of a game, it is easier and more accurate to estimate the future outcome (win/lose) and next action. Therefore, based on these simulated results, the MCTS can have a better estimation of which action leads to better win rate and improve the initial given policy. In the following, we first define the tree structure critical to MCTS and how it relates to the final result. We then explain how to build

that tree.

Tree Definition The core of MCTS is a tree encoding information from different simulation runs. Suppose we are interested in generating the next action $\mathbf{a}_t$ for a given starting state $\mathbf{s}_t$. The root of the tree would be this $\mathbf{s}_t$. Each node in this tree is a valid configuration of the game, and there is an edge between node $\mathbf{s}_i$ and $\mathbf{s}_j$ if there is an action that can transit $\mathbf{s}_i$ to $\mathbf{s}_j$. For example, all nodes with depth equals 1 are $\mathbf{s}_{t+1}$ that can be directly reached from $\mathbf{s}_t$ by taking an action. For each edge, namely a state action pair $(\mathbf{s}, \mathbf{a})$, we store several information. First, the prior action distribution $p(\mathbf{a}|\mathbf{s})$ computed from our network. Second, a visit counter $N(\mathbf{s}, \mathbf{a})$ denoting how many times we have simulated this edge in our searching. Third, $Q(\mathbf{s}, \mathbf{a})$ denoting the expected win rate after taking this action. The latter two values will be computed during the tree construction stage introduced in Section 3.2. The final predicted action for the given starting state $\mathbf{s}_t$ will be

$$\mathbf{a}_t = \arg\max_{\mathbf{a}} N(\mathbf{s}_t, \mathbf{a}).$$

Tree Construction The tree starts with only one node $\mathbf{s}_t$ and we repeatedly expand and update values in it. At each iteration of the tree construction, we start from the root node. We then choose the next action that maximize the upper confidence bound $U(\mathbf{s}, \mathbf{a})$ of $Q(\mathbf{s}, \mathbf{a})$,

$$U(\mathbf{s}, \mathbf{a}) = Q(\mathbf{s}, \mathbf{a}) + c_{puct} p(\mathbf{s}, \mathbf{a}) \frac{\sqrt{\sum_{\mathbf{a}} N(\mathbf{s}, \mathbf{a})}}{1 + N(\mathbf{s}, \mathbf{a})}.$$

Here, c_{puct} is a hyper-parameter and we use 0.5 in our experiments. By repeatedly choosing the next action and transit to the next state given this action, we essentially simulate one episode of the game. After a few such steps, we may reach to the end of a game! A win or lose outcome is generated by game rules. In this case, we update all $Q(\mathbf{s}, \mathbf{a})$ and $N(\mathbf{s}, \mathbf{a})$ in a backward manner along all edges we have traversed in this simulation run. Recall that $Q(\mathbf{s}, \mathbf{a})$ denotes the expected win rate. The update rule (applied backwardly) can be summarized as

$$Q(\mathbf{s}, \mathbf{a}) = \frac{N(\mathbf{s}_a, \mathbf{a})}{\sum_{\mathbf{a}} N(\mathbf{s}_a, \mathbf{a})} \times Q(\mathbf{s}_a, \mathbf{a}),$$

where $\mathbf{s}_a$ denotes the state transit from $\mathbf{s}$ after taking $\mathbf{a}$ and the leaf node is updated with our win-lose outcome. We may also end up with a new state that doesn't belong to our current tree. In this case, we add this new state to the tree, evaluating the neural network policy for computing the $p(\mathbf{a}|\mathbf{s})$. Similarly, we also update $Q(\mathbf{s}, \mathbf{a})$ and $N(\mathbf{s}, \mathbf{a})$ backwardly with the same rule, except that the win-lose outcome will be replaced by our model estimation, i.e., $v(\mathbf{s})$. Every such a forward simulation plus a backward update provides a new data sample of the futures and improve our estimation, which is precisely why this is called Monte-Carlo. In practice, we found repeating these iterations for 1600 times reaching a fairly accurate estimation while maintaining reasonable computation budget. For more complicated game such as 19×19 go game, it is possible more iterations and computations are needed.

3.3. Behavior Cloning Learning

Having discussed the model and inference, the remaining unaddressed question is how we learn the policy (neural network). The most straight-forward idea is to mimic how human would play the game (5). This is called behavior cloning and falls in the regime of supervised learning. Suppose we have a dataset recording human players' moves[2]. Each data is this dataset is a triplet of state, action and game outcome $(\mathbf{s}, \mathbf{a}, r)$, namely a sampled game configuration $\mathbf{s}$, human's action $\mathbf{a}$ and the game outcome $r \in 1, 0$ for the current player . Our total training objective is a multi-task loss considering both action distribution prediction and outcome prediction. Namely

$$\mathcal{L} = \mathcal{L}_p + \beta \mathcal{L}_v. \tag{1}$$

Here, the $\mathcal{L}_p$ is a cross-entropy loss between our predicted action distribution and the true data distribution. Suppose the size of the dataset is K, $\mathcal{L}_p$ can be written as

$$\mathcal{L}_p = \frac{1}{K} \sum_{k \in Data} \sum_{i,j=1}^{N} \hat{p}^k_{(i,j)} \log p^k_{(i,j)} + (1 - \hat{p}^k_{(i,j)})(\log(1 - p^k_{(i,j)})). \tag{2}$$

Here, the superscript k denotes the index of data. $\hat{p}_{(i,j)}$ is the ground truth probability mass, which is 1 if and only if human place a stone at location (i, j), and 0 otherwise. $p_{(i,j)}$ is the (i, j) entry value of our predicted probability from neural networks. For the other loss $\mathcal{L}_v$, we use ℓ_2 loss between the ground-truth outcome r and our predicted value function $v(\mathbf{s}_t)$.

3.4. Self-Play Learning

It is commonly acknowledged that behavior cloning has high training efficiency, e.g., high convergence speed and high training stability. However, one of the major drawback of behavior cloning is that it can hardly outperform the expert demonstrations. After all, the upper bound performance is to mimic the expert perfectly. To address this, we adopt the idea of self-play from AlphaZero (15) with some slight modifications to further improve our neural go player. Suppose we are given a current policy network and some existing training data, e.g., $(\mathbf{s}_t, \mathbf{a}_t)$, we randomly

[2]We use an open-sourced dataset for our setting from https://github.com/CGLemon/pyDLGO

Black \ White	GNU Go	Pachi	BC	Self-play
GNU Go	-	W	W	**W**
Pachi	B	-	W	**W**
BC	B	B	-	**W**
Self-play	**B**	**B**	**B**	-

Table 1. **Go playing results**. Each row shows results when one method takes black and plays against other methods. Each column shows results when one method takes white and plays against other methods. *BC* refers to the behavior cloning model. *W* means white player wins and *B* means black player.

choose one data sample $(\mathbf{s}, \mathbf{a})$ to begin our self-play. Starting from this $(\mathbf{s})$, we apply our policy network and the MCTS. After some updating steps of MCTS, we will have a fairly good estimation of $N(\mathbf{s}, \mathbf{a})$ and $Q(\mathbf{s}, \mathbf{a})$ at this node. We then construct a new dataset, each recording a triplet of $\mathbf{s}$, $N(\mathbf{s}, \mathbf{a})$ and $Q(\mathbf{s}, \mathbf{a})$. This dataset will be used for improving our current policy through Eq 1. However, instead of a 0-1 probability mass for ground-truth action distribution $\hat{p}^k_{(i,j)}$ in Eq 2 and 0-1 value network regression target, we now use a soft version ground-truth evaluated from MCTS. Namely $N(\mathbf{s}, \mathbf{a})$ with normalization for action distribution and $Q(\mathbf{s}, \mathbf{a})$ for value network. Intuitively, such a self-play iteration helps us distill the additional information gain from MCTS to our policy. Since MCTS usually improves bias over a give policy as we explained in Section 3.2, the newly generated dataset becomes a better expert demonstration. Therefore, we can continually improve our model.

There are a few notable differences between our implementations w.r.t. the original AlphaZero. First, our MCTS starts from a sampled state from an existing dataset, rather than from an empty board. This can possibly introduce some small bias since the sampled state comes from a worse policy, which might be sub-optimal in the first place. However, this greatly saves our computation budget since we don't need to build the tree from the very beginning of a game. Second, we only use the latest policy from behavior cloning to play with itself, rather than use the latest and the second last policy to compete with each other. We found in our experiments, this is enough to improve the model. Besides, in principal one can apply several round of self-play. Unfortunately, we didn't have time to explore that direction and only apply one round of self-play in our experiments.

4. Experiment

4.1. Experimental Details

Dataset: To train the policy net with supervisions, we use an SGF dataset containing about 40K games[3] played by human players on a 9×9 board on the KGS Go Server[4]. These game records consist roughly 1 million state-action pairs. There is also an outcome value associated with each state-action pair, denoting whether that player win the game. For example, if the current player (the one who takes the action) wins in the end, the outcome value is 1 and 0 otherwise.

Behavior Cloning: The network is trained with Adam optimizer with a batch size of 2048 on an Nvidia GeForce 1080ti GPU. The initial learning rate is set to 1e-3, decayed by a factor of 0.1 for every 128000 steps. We train our policy network for 400K iterations, which takes about 8 hours. Data augmentation like flipping, random rotation is applied.

Self-play training: Since we only apply one round of self-play training, we propose to dump those self-play game episodes to an offline dataset to accelerate training. To this end, we use the best model trained with behavior cloning to do self-play simulation for 1600 playouts. An self-play simulation has low a GPU utilization due to heavy game environment simulations on CPUs. Therefore, we run 16 self-play simulations simultaneously on 2 GPUs to batchify the neural network inference. This takes 24 hours to dump roughly 100K state-action pairs. During self-play training, we load the policy we learned through behavior cloning and fine-tune it with a learning rate of 1e-5 for 20k steps. Empirically, we observe that more training steps would cause overfitting issue. The same data augmentation techniques are applied similar to that of the behavior cloning.

4.2. Experimental Results for Playing Go

To evaluate our model, we test our models against other Go programs in Go board GUI sabaki[5] using GTP (Go Text Protocol), a text based protocol for communication with computer go programs (8). In the following, we first introduce the baselines, and then summarize the quantitative and qualitative results.

Baselines: We demonstrate the effectiveness of our neural go policy against the following baselines.

- GNU Go (4) has attracted researchers and hobbyists by providing the first open source program with a

[3] http://u-go.net/gamerecords/

[4] https://www.gokgs.com/

[5] https://sabaki.yichuanshen.de/

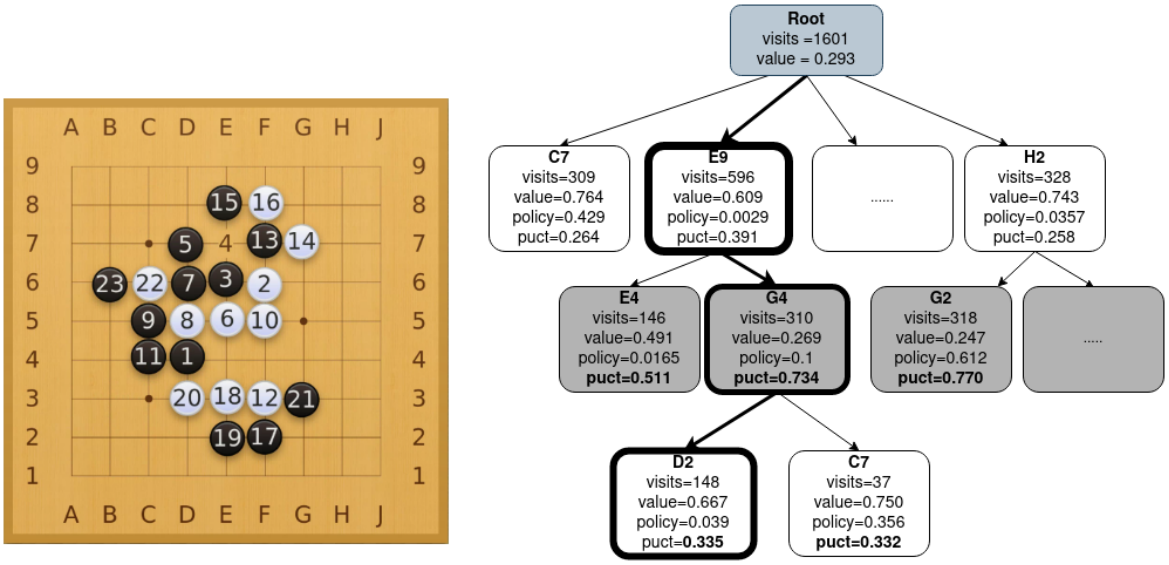

Figure 2. **MCTS tree structure.**. Each node indicates a position of the game. Root node indicates current position (shown in the left). The edges between root node and first-layer's nodes indicate actions for the root node (white to move), and the node in the first layer indicates the new position after taking the action. The 2nd layer indicates the action for the black stones in the position of first layer.

strength approaching the best classical programs. It has won 19x19 Go tournament in 2003. We use the default settings by GNU Go.

- Pachi(1) is a strong engine that has won a 7-stone handicap game against Zhou Junxun 9p. For 9x9, it can achieve 7d KGS strength. By default, Pachi currently uses the UCT engine that combines Monte Carlo approach with tree search. We set the playouts the same as our model.

Results: We test different pairs of methods twice, with one method taking black the other taking white and then reverse. Since all methods are deterministic[6] and we always start with an empty board, there is no need to play the games multiple times. We summarize the results in Table 1. As we can see, our behavior cloning method is strictly better than the baselines, as we always win regardless of taking white or black. This shows the effectiveness of our learned neural go player. Furthermore, self-play can help us learn an even better model. The overall go strength can be summarized as self-play $>$ behavior cloning $>$ Pachi $>$ GNU Go.

To gain more insights on how well different methods play, we show some terminal states in Figure 3 recorded at the end of different games. From the figure we can observe that our policy (self-play) can outperform GNU Go and Pachi by large margins. However, the self-play usually has a tight game w.r.t. behavior cloning policy. This might due to the fact that we only perform one round of self-play training instead of multiple rounds. Besides, simulating more self-play data can also potentially increase the performance. Because of the limited computation budget, we only generate 100k self-play state-action pairs, which is only one tenth of the original behavior cloning dataset. This also explains why the difference is not significant.

MCTS Tree Visulization: To analyze the effects of utilizing MCTS during inference, we also show the structure of one MCTS tree along with the statistics in Figure 2. Take this specific starting state (left figure) for examples, the action $E9$ has very low score based on the policy net prediction. However, MCTS simulation found that this move can actually make the situation better in terms of an higher winning rate. The root cause of this mismatch is our learned policy can sometimes show biases, while MCTS is able to reduce such biases through numerous Monte-Carlo simulations. This is also exactly the reason why self-play can improve the performance (distilling the better estimation from MCTS to the learned policy).

5. Conclusion

We implement an agent that learns to play Go on 9x9 board through behaviour cloning from human knowledge and then refined with self-play. Our agent convincingly beats all baselines including several famous Go programs.

[6]Recall that during MCTS, we always take the optimal action that maximizes $U(\mathbf{s}, \mathbf{a})$ to expand the tree, and therefore our polices are deterministic as well.

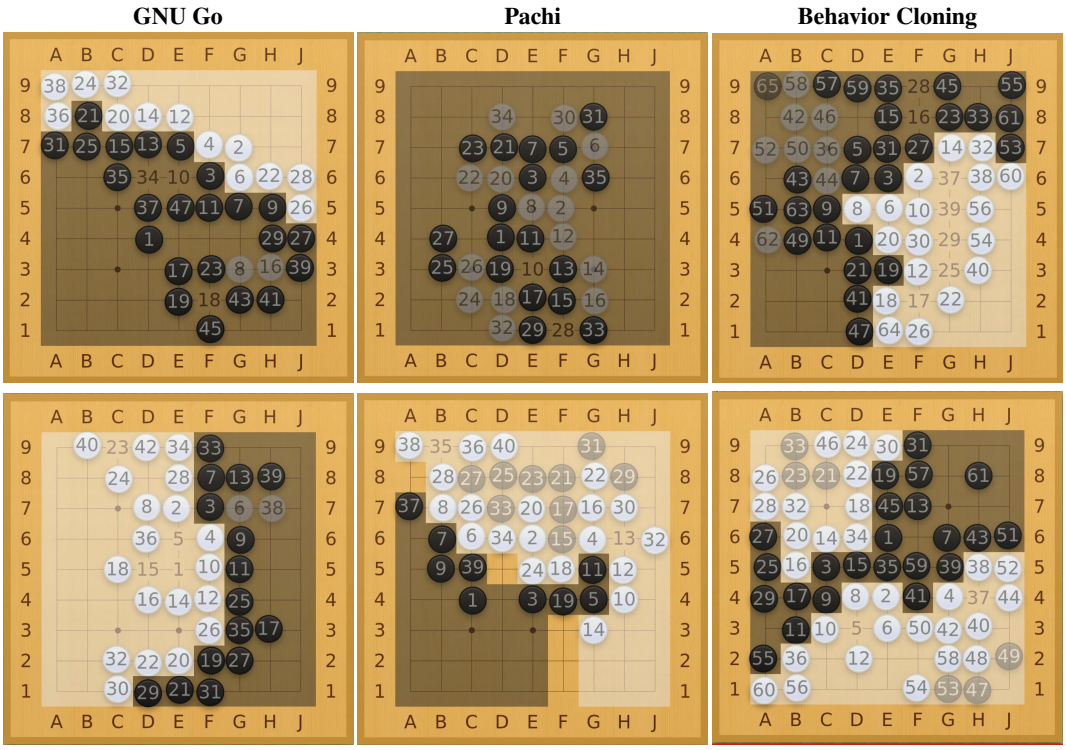

Figure 3. **Go-playing results between self-play model and other methods.** In the first row, our model (self-play) takes black stones and plays first. In the second row, our model takes white stones. In all games, the opponent resigns when win rate is fairly small. Tromp-Taylor rule is used for territory counting and self-play model show clear advantage.

The original implementation by DeepMind (14) uses orders of magnitude more raw computational power on industry hardware for several days. Due to the limited resource, we make a few small modifications to the original method to tradeoff computation and performance. Even with such compromises, the self-play training still shows improvements over the behavior cloning models. For the future work, we'll explore more efficient self-play training, as long as training with pure self-play – without any human knowledge.

6. Role of team members

Everyone gets involved in discussion of the projects as well as contributes to the writing of this report.

1. Yun: Core code developer, method design, writing (part of experiments).
2. Wenyuan: Principal writer, method design.
3. Yutian: Writing/figure (part of experiments), behavior cloning experiments running on cloud and documentation for code.
4. Kaihua: Baseline evaluations (pachi/gnugo), writing/figure (part of introduction, model and experiments).

References

[1] 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.

[2] Michael Buro. From simple features to sophisticated evaluation functions. In *International Conference on Computers and Games*, pages 126–145. Springer, 1998.

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[4] Ken Chen. Gnugo wins 19 x 19 go tournament. *ICGA journal*, 26(4):261–262, 2003.

[5] Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*, 2014.

[6] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] G FARNEBACK. Gtp-go text protocol. *http://www. lysator. liu. se/% 7egunnar/gtp/*.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[11] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[12] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99, 2015.

[13] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.

[14] 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.

[15] 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):354–359, 2017.