A ReImplementation of AlphaGo-Zero on a game of Gomoku

Abhishek Srivastava, Linde Yang
srivastava.abh@husky.neu.edu, yang.lind@husky.neu.edu
Khoury college of Computer Science
Northeastern University
Boston, MA 02115

Abstract

One of the long standing goals of Artificial Intelligence is to be able to learn on its own just like humans. Recently a lot of work has been done in this field and with the creation of AlphaGo-zero which has the ability to defeat the best Go players in the world, we are heading towards this goal. In this paper we are implementing this same algorithm on a different game called Gomoku to test the effectiveness of the algorithm over different environments.

Keywords- *Alphago-zero, MCTS, deep neural network(DNN), Gomoku*

Introduction

Perfect information games are theoretically solvable and has a very good real world use in environments which are deterministic and certain, such as Chess and Go.

But in practice, the problem is not as easy as it looks. Though the game is of perfect information and can be solved by alpha-beta tree search or minimax algorithm but due to very large state space, this task becomes unrealistic in real time.For example let's take the case of Go. The number of legal board positions in Go has been calculated to be approximately 2×10^{170} which is vastly greater than the number of atoms in the universe, estimated to be about 1×10^{80}[1]. Solving a problem of this size is very challenging in space and time complexity and almost impossible with our current resources.

But recent advancements in this field have developed many good algorithms and one of them is AlphaGo-zero which is successfully able to beat the best Go players in the world. Because of this success of Alphago-zero we want to study and emulate its methods on a different game called Gomoku.

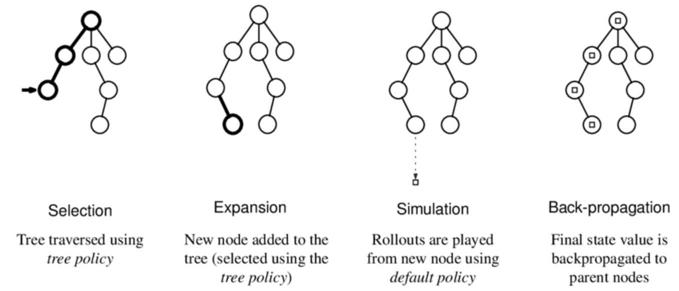
Gomoku[16], also called Five in a Row, is an abstract strategy board game. It is traditionally played with Go pieces (black and white stones) on a Go board. It can be played using the 15 by 15 board or the 19 by 19 board. Because pieces are typically not moved or removed from the board, Gomoku may also be played as a paper-and-pencil game. The game is known in several countries under different names.

Players alternate turns placing a stone of their color on an empty intersection. The winner is the first player to form an unbroken chain of five stones horizontally, vertically, or diagonally.

In our version of Gomoku, we are playing it on a 5 by 5 board and winning condition is to make a chain of 4 stones. The complexity of this game is around 1×10^{25}. Which is big enough to test our underlying method and also computationally solvable with our resources.

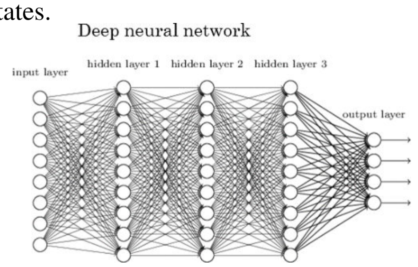
The primary method we used is of Alphago-zero.It consists of a monte carlo tree search as a search tree and a deep neural network for value and policy approximation. The neural network is trained by self-play reinforcement learning, starting from random play, without any supervision or use of human data. It only uses the current state of the board as input feature. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte- Carlo rollouts. To achieve these results, we incorporate lookahead search inside the training loop, resulting in rapid improvement and precise and stable learning. Further technical details would be in Project description.

Background


Selection
Tree traversed using *tree policy*
Expansion
New node added to the tree (selected using the *tree policy*)
Simulation
Rollouts are played from new node using *default policy*
Back-propagation
Final state value is backpropagated to parent nodes

The method primarily consists of Monte Carlo Tree search and Neural Network. Monte-Carlo Tree Search (MCTS)[2], illustrated in Figure 1, is a best-first search technique which uses stochastic simulations. MCTS can be applied to any game of finite length. Its basis is the simulation of games where both the AI controlled player and its opponents play random moves, or, better, pseudo-random moves. From a single random game (where every player selects his actions randomly), very little can be learnt. But from simulating a multitude of random games, a good strategy can be inferred.

The algorithm builds and uses a tree of possible future game states.



The other important aspect of the code is Deep Neural Network[5]. A deep neural network (DNN) is an artificial neural network (ANN) with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship.

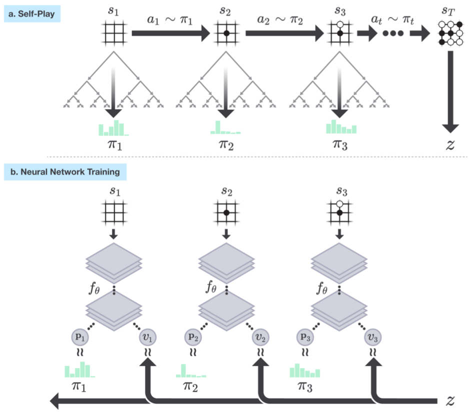
Related Work

There are multiple methods explored prior to AlphaGo-zero to solve the games with very large state space. One of the most prominent method is AlphaGo fan. Alpha Go Fan[4] utilised two deep neural networks: a policy network that outputs move prob- abilities, and a value network that outputs a position evaluation. The policy network was trained initially by supervised learning to accurately predict human expert moves, and was subsequently refined by policy-gradient reinforcement learning. The value network was trained to predict the winner of games played by the policy network against itself. Once trained, these networks were combined with a Monte-Carlo Tree Search (MCTS) to provide a lookahead search, using the policy network to narrow down the search to high-probability moves, and using the value network (in conjunction with Monte-Carlo rollouts using a fast rollout policy) to evaluate positions in the tree. The difference between Alphago Fan and Alphago zero is Alphago fan requires two different neural network to learn policy and values whereas Alphago zero requires just one to capture both. This is a very significant improvement from the perspective of computation as in Alphago zero we just need to train one neural network. The other significant difference is Alphago Fan requires supervised learning to train the neural network which limits its use to the environments where we are having sufficient amount of data to train whereas Alphago zero trains itself from a large number of self play games and hence can be applied to various environments without any prior experience in those environments.

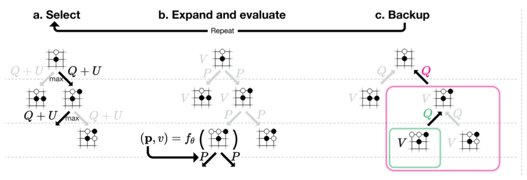
Project Description

This method uses a deep neural network f_w which is a function of w. This neural network takes as an input the raw board representation S_c with respect to current player and S_o with respect to opponent player and outputs both move probabilities and a value, $(p, v) = f_w(S_c, S_o)$. The vector of move probabilities p represents the probability of selecting each move (including pass), $Pa = Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position $S = (S_c, S_o)$. This neural network combines the roles of both policy network and value network[7] into a single architecture. The neural network consists of many residual blocks [11] of convolutional layers [8],[6] with batch normalisation [9] and rectifier nonlinearities [13] (see Methods)

The neural network in AlphaGo Zero is trained from games of self-play by a different reinforcement learning algorithm. In each position $S = (S_c, S_o)$, an MCTS search is executed, guided by the neural network f_w . The MCTS search outputs probabilities pi of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities p of the neural network [10], [14] $f_w(S_c, S_o)$; MCTS may therefore be viewed as a powerful policy improvement operator with search – using the improved MCTS-based policy to select each move, then using the game winner z as a sample of the value – may be viewed as a powerful policy evaluation operator. The main idea of our reinforcement learning algorithm is to use these search operators repeatedly in a policy iteration procedure [12],[15]: the neural network's parameters are updated to make the move probabilities and value $(p, v) = f_w(S_c, S_o)$ more closely match the improved search probabilities and self-play winner (pi, z); these new parameters are used in the next iteration of self-play to make the search even stronger. Figure below illustrates the self-play training pipeline.



The Monte-Carlo tree search uses the neural network f_w to guide its simulations (see below Figure). Each edge(s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action-value $Q(s, a)$. Each simulation starts from the root state and iteratively selects moves that maximise an upper confidence bound $Q(s, a)+U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$ [3] , until a leaf node s is encountered. This leaf position is expanded and evaluated just once by the network to generate both prior probabilities and evaluation, $(P(S, .), V(S)) = f_w(S')$. Each edge (s, a) traversed in the simulation is updated to increment its visit count $N(s, a)$, and to update its action value to the mean evaluation over these simulations,$Q(s, a) = 1/N(s, a) \sum_{s'|s,a->s'} V(s')$, where $s, a \rightarrow s$ indicates that a simulation eventually reached s after taking move a from position s.



MCTS may be viewed as a self-play algorithm that, given neural network parameters w and a root position s, computes a vector of search probabilities recommending moves to play, $pi = \alpha W(s)$, proportional to the exponentiated visit count for each move, $pi \propto N(s,a)^{1/\tau}$, where τ is a temperature parameter.

The neural network is trained by a self-play reinforcement learning algorithm that uses MCTS to play each move. First, the neural network is initialised to random weights W_0. At each subsequent iteration $i1$, games of self-play are generated. At each time-step t, an MCTS search $pi_t = \alpha_{w_{i-1}}(s_t)$ is executed using the previous iteration of neural network$f_{w_{i-1}}$, and a move is played by sampling the search probabilities pi_t. A game terminates at step T when any one of the players win. Then the game is scored to give a final reward of $r_T \in \{1, +1\}$. The data for each time-step t is stored as (s_t, pi_t, z_t) where $z_t = \pm r_T$ is the game winner from the perspective of the current player at step t. In parallel, new network parameters W_i are trained from data (s,,z) sampled uniformly among all time-steps of the last iteration(s) of self-play. The neural network $(p, v) = f_w(S)$ is adjusted to minimise the error between the predicted value v and the self-play winner z, and to maximise the similarity of the neural network move probabilities p to the search probabilities pi. Specifically, the parameters w are adjusted by gradient descent on a loss function l that sums over mean squared error and cross-entropy losses respectively.

$(p, v) = f_w(S)$

$$l = (z - v)^2 - pi^T logp + c \parallel w \parallel^2 \tag{1}$$

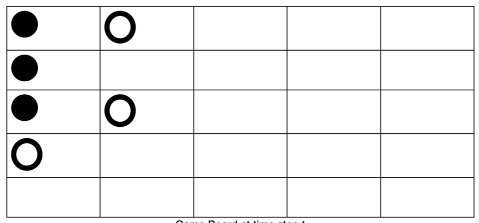
where c is a parameter controlling the level of L2 weight regularisation.

Methods

The three major components of this project are discussed below in detail.

Gomoku Environment

Gomoku environment is written in python. Important features of this environment is to return two boards, one from current players perspective and another from opponent's perspective. For example, current state of the board is as shown in figure.



Game Board at time step t

The above board will be interpreted by Neural Network as two boards one for all black stones and one for all white stones. Along with this, the game environment checks the terminal condition of 4 in a row and returns an appropriate flag.

MCTS

Monte carlo tree search uses 400 simulations for each rollout. Input to MCTS node is game environment and neural network. MCTS queries neural net to get policy pi and then runs rollout over it to get better pi which then is used in training of neural network.

Neural Network

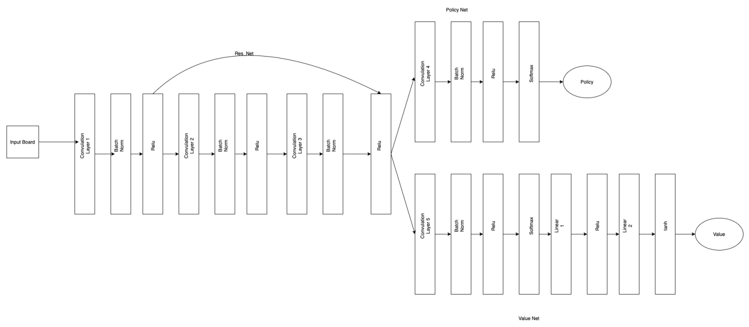


fig. 5 Neural Network Architecture for Gomoku

Input to the neural net is game board from current player and opponent's perspective. So in our case the input would be of shape 2 x 5 x 5.

As shown in above figure the neural net contains 3 common layers between policy and value net. During experiments we figured that neural network is performing better with residual net layer. So output of layer 1 is connected to input of layer 3.

The policy is calculated by adding one more layer and applying softmax over it. And the value is calculated by adding one more convulation layer and two linear layers. Kernel size and output filters of each layer are chosen based on experiments and in this case they are 3 and 32 respectively(this will change if board dimension changes).

Neural Network Layers : Conv1,Conv2,Conv3 in figure is of kernel size 3 x 3 and filters 32 with padding 1. Conv4 is of kernel size 1 x 1, padding 0 and output filters 2. And Conv5 is of kernel size 1 x 1, padding 0 and output filters 1. The L2 regularization constant is $1e-4$.

Training

We trained out reinforcement learning pipeline for around 5 days on microsoft azure. The training started with a random neural network and played more than 10000 games using 400 simulations for each MCTS. The parameters are updated from mini-batch of 32.

Experiments

During training we have collected 200 neural net weights over a course of 5 days. Below is empirical analysis of our results. For each weight we have played 10 games between our trained neural net to a random opponent and analysed its winning percentage.

NeuralNet as First Player

In this experiment we used trained neural net as first player and played it against a second player which is a random opponent. In above figure First player is the trained neural

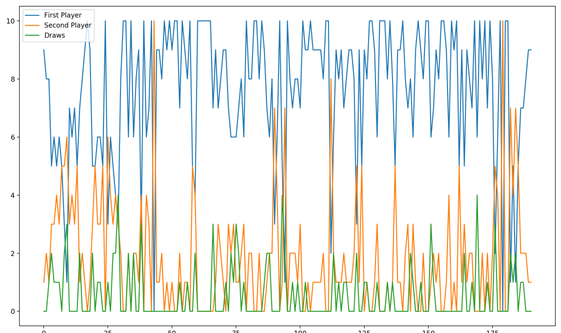


fig. 6 Games win analysis when NN is playing as first player

net and second player is a random neural net. The X axis is weights learned at time step t and Y axis is number of games won out of 10 by each player for given weights of first player.

From the analysis we can conclude that overall number of games won by the first player is more than the second player. This illustrates that our neural net against a random opponent is performing better. To further strengthen this argument let’s analyse average number of games won with respect to training.

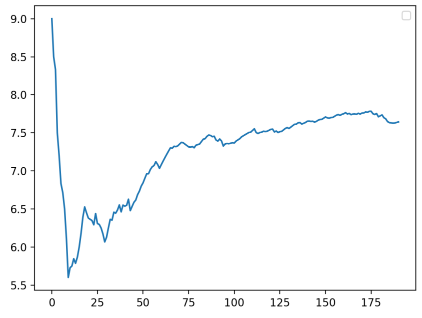


fig. 7 Games average win analysis when NN is playing as first player

we can see from above figure over time and more training our neural net is winning around 75% game. Also there is an initial drop in neural network performance. That is because initially neural net was exploring and figuring out best strategy which creates kind of randomness in initial phase but after sufficient training we can see the trend which is increasing.

Neural Net as Second Player

In this experiment we used trained neural net as second player and played it against the first player as a random opponent.

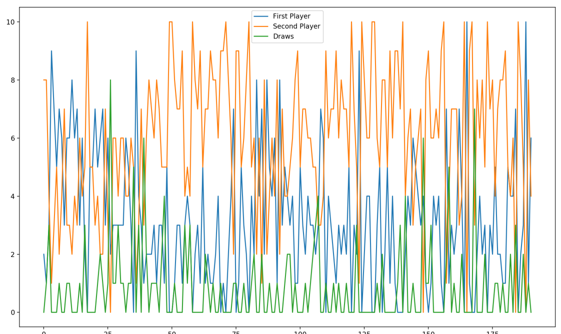


fig. 8 Games win analysis when NN is playing as second player

In below figure Second player is trained neural net and the first player is a random neural net. The X axis is weights learned at time step t and Y axis is number of games won out of 10 by each player for a given set of weights of the second player.

From this we can conclude that overall number of games won by the second player is more than the first player. This shows that our neural net is able to perform better than random player in both scenarios.

Also we analysed average number of games won with respect to training.

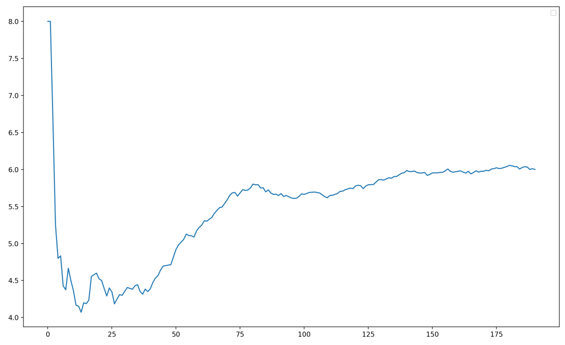


fig. 9 Games win analysis when NN is playing as first player

This shows that neural net as second player is winning around 60% games. Again there is an initial drop in network performance. That is because of initial randomness and uncertainty in neural net policy because the MCTS has not explored much by that time.

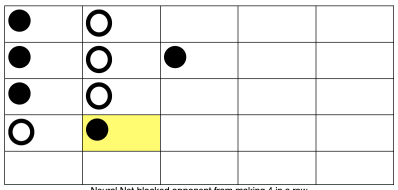
One of the important observation here is neural network as first player is winning 75% of games where as neural network as the second player is winning 60% of the games. This is because there is an inherent bias in Gomoku.

In Gomoku the person who plays first has more chances of winning. This bias can be removed either by constraining the movement of first player so that both players gets equal opportunity to win or by making both players to play equal number of games in a set of games.

During our training also we observed this issue. And because of this our neural net has learnt to play and win more as first player as compared to second player.

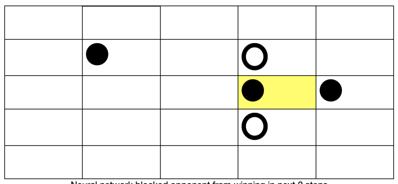
Policies Learnt

In this section we are going to discuss about some interesting strategies our neural net learnt after training.



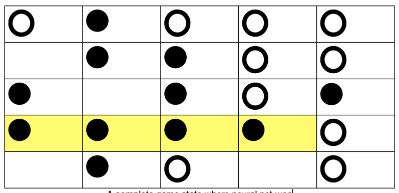
Neural Net blocked opponent from making 4 in a row

In above figure we can see that neural network is able to detect 3 in a row for opponent which is an immediate threat and it blocked it from making 4 in a row.



Neural network blocked opponent from winning in next 2 steps

In above figure the opponent would have won the game in next two steps but our neural net deduced that and blocked it from winning.



A complete game state where neural net won

Finally here is one of the comprehensive game which neural net has won.

Policies Failed

Though the neural net has learnt some very good strategies just after playing 10000 self play games. Still there is more room for improvement.

First strategies which neural net has learnt at upper part of the board does not apply at lower part of the board and because of this same strategy make neural network win at above part of board but make it loose at lower part. This is because at the time of training we just trained the neural net directly from the game state. Instead we could have rotated the board in all 4 directions and utilised the same game state for all parts of the board.

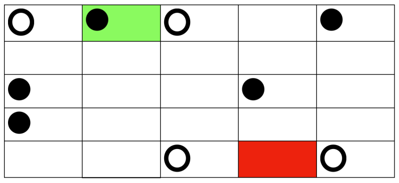


fig. 13 Policy learnt in upper part of board(shown in green) to same policy failed in lower part of board(shown in red)

Second neural net is still not able to see immediate win. For example if the there is some strategy by which neural net can win in 1 step but it will not choose that strategy and try to block all possible winning chances of the opponent. This is probably because neural network is giving more weightage to the policy which makes it to not lose than to win. Till now we have not figured out any solution for it but we plan to add some feature based method in current neural net which would consequently give a little more weightage to immediate win.

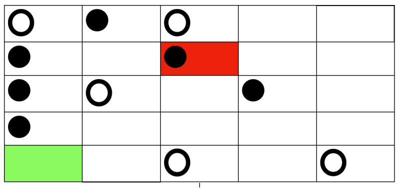


fig. 14 NN played a move shown in red instead could easily won the game by move shown in green

Conclusion

In this paper we discussed about Alphago-zero algorithm which is able to defeat the best go players in the world. Then we talked about its underlying methods and how we implemented those methods for Gomoku. We also analysed the winning percentage of our trained neural net against random opponent and observed inherit bias in Gomoku.We have also seen some good strategies that neural network has learnt and some important strategies which it still has not figured out.

This project gave us insight into implementation of reinforcement learning in real world scenarios. We learnt and implemented state of the art algorithms from scratch. At the time of development we faced multiple issues which motivated us to redo our maths and debug our code. This project also helped us in developing the skill of debugging in machine learning codes.

References

[1] In: (). DOI: `https://en.wikipedia.org/wiki/Go_(game)`.

[2] chaslot et al. *Monte-Carlo Tree Search: A New Framework for Game AI.* URL: `https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf`. (accessed: 01.09.2016).

[3] C Clark. *Training deep convolutional neural networks to play go. In 32nd International Conference on Machine Learning, 1766–1774 (2015).* (accessed: 01.09.2016).

[4] DeepMind. *Mastering the Game of Go without Human Knowledge.* URL: `https://www.nature.com/articles/nature16961`. (accessed: 01.09.2016).

[5] M. Enzenberger. *EVALUATION IN GO BY A NEURAL NETWORK USING SOFT SEGMENTATION.* URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.8391&rep=rep1&type=pdf`. (accessed: 01.09.2016).

[6] S. Gelly. *Combining online and offline learning in UCT. In 17th International Conference on Machine Learning, 273–280 (2007).* (accessed: 01.09.2016).

[7] C Kocsis L. Szepesvari. *Bandit based Monte-Carlo planning. In 15th European Conference on Machine Learning, 282–293 (2006).* (accessed: 01.09.2016).

[8] A. Krizhevsky. *ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems, 1097–1105 (2012).* (accessed: 01.09.2016).

[9] S. Lawrence. *Face recognition: a convolutional neural-network approach. IEEE Transactions on Neural Networks 8, 98–113 (1997).* (accessed: 01.09.2016).

[10] LeCun. *Nature 521, 436–444 (2015).*. (accessed: 01.09.2016).

[11] Campbell M. *Deep Blue. Artificial Intelligence 134, 57–83 (2002).* (accessed: 01.09.2016).

[12] C. J Maddison. *Move evaluation in Go using deep convolutional neural networks. 3rd International Conference on Learning Representations (2015).* (accessed: 01.09.2016).

[13] V Mnih. *Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).* (accessed: 01.09.2016).

[14] D Stern. *Bayesian pattern ranking for move prediction in the game of Go. In International Conference of Machine Learning, 873–880 (2006).* (accessed: 01.09.2016).

[15] I. Sutskever. *Mimicking Go experts with convolutional neural networks. In International Conference on Artificial Neural Networks, 101–110 (2008).* (accessed: 01.09.2016).

[16] Wikipedia. *Gomoku Game.* URL: `https://en.wikipedia.org/wiki/Gomoku`. (accessed: 01.09.2016).