Reinforcement Learning – Self Play

Qing Zhou

May, 2020

1 Introduction

The motivation of this project is to understand how to achieve more table results in a more complicated state space, by implementation of the AlphaGo Zero code in the game of Hex. AlphaZero combines neural network and Monte Carlo Tree Search (MCTS) in an elegant framework to achieve the goal of stablised learning.

The neural network is parametrised by θ takes the state of the board s as input, and gives two outputs: the values of the board state $v_\theta(s) \in [-1, 1]$, and the policy $\vec{p}_\theta(s)$, a probability vector over all possible actions. When training the network, at the end of each self-play, the neural network is provided training examples o fthe form $(s_t, \vec{\pi}_t, z_t$, where $\vec{\pi}_t$ is an estimate of the policy from state s_t, and $z_t \in [-1, 1]$ is the final outcome of the game from the perspective of the player at s_t ($s_t = 1$ if the player wins, -1 if the player loses). The problem then becomes to minimize the following loss function[1]

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log(\vec{p}_\theta(s_t)) \tag{1}$$

So that given a state s, the neural network provides an estimate of the policy $\vec{p}_\theta$. To improve the accuracy of the estimates, MCTS is implemented. It makes use of the predictions from the neural network as a reference for simulating the best move of a game. These simulations are carried out by predicting the final outcome of playing a move in the current board. From explorations of some potential moves, MCTS chooses a move according to the probability distribution, then neural network uses the outcome provided by the MCTS to update itself. For the MCTS, each edge between two nodes $(i \to j)$ a Q-value ($Q(s, a)$) exists which takes the board state and actions as input, and gives the expected reward as output. An upper bound exists for the Q-value[1]

$$U(s, a) = Q(s, a) + c_{puct} P(s, a) + \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \tag{2}$$

where $P(s, \cdot) = \vec{p}_\theta(s)$, which is the prior probability of taking θ a particular action from state s according to the policy returned by our neural network. c_{puct} is a parameter which controls the degree of exploration. $N(s, a)$ is the number of times we took action a from the state s across simulations. The MCTree is initialised at the root. For a single simulation, the action a which maximises the upper bound $U(s, a)$ is calculated. If the next state s' exists, we can recursively call the MCTS on the state s'; otherwise it is added to the tree. Afterwards, $v(s')$ is back propagated along the path and all Q-valus get updated. After a few simulations, the $N(s, a)$ values at the root provide a better estimate for the policy. The improved policy $\vec{\pi}(s)$ is simply the normalised counts $N(s, \cdot)/\sum_b N(s, b)$.

2 Hex

Now we have everything at hand the question becomes to improve the training through self-play. This is done basically through iteration of policy. Each time we play a number of self-play games, and perform a fixed number of MCT simulations from the current state s_t. Afterwards a move is selected from the improved policy. At the end of this iteration, a training example is given, and the corresponding reward is computed. This example is then fed into the neural network, and we allow the old version to pit against the new version. If the new version wins more than a fraction of games, then the new version is accepted. The starting parameters we adopted for our network are listed as follows

Table 1: Hyperparameters used for the training. "numIters" gives the total iterations of the training; "numEps" is the number of complete self-play games to simulate during simulation; "updateThreshold" is the acceptance ratio for the new neural network; "maxlenOfQueue" is the number of games to train the neural network; "numMCTSSims" is the number of game moves for the MCTS to simulate; and "arenaCompare" gives the number of games to play during arena play to determine if the new network will be accepted.

Parameters	Values
numIters	37
numEps	100
tempThreshold	15
updateThreshold	0.6
maxlenOfQueue	200000
numMCTSSims	25
arenaCompare	40
cpuct	1

The tunable parameters here are "numEps", "tempThreshold", "updateThreshold", "numMCTSSims", and "arenaCompare". In the coming sections we will discuss the effects of those parameters. Two models were trained on one GPU for a total of 37 iterations.

3 Tournament

To see the training results of the two alphazero players, we let let the AlphaZero player 1 to pit against 3 other opponents: the ID-TT AlphaBetaPlayer, the MCTPlayer, and the AlphaZero player 2, in the arena. To show the stability and training result we plot the win ratio of the AlphaZero player 1 against other players. The result is shown in Fig. 1. The ID-TT player has a search depth of 3, and the MCT player has search maximum iteration of 15 moves. For each iteration, the win ratio is determined by 20 games. It can be seen, after ~ 6 iterations, AlphaZero player 1 was able to win over the ID-TT player with a percentage of 100%. The win ratio over the MCT player however, is oscillating between ~ 0.6 and 0.9, and needs extra time to stablise. The win ratio over another AlphaZero player with same parameters and training time, is oscillating around 0.5.

4 Hyperparameters

In order to see the effects of the tunable parameters on the outcome of the training results, we have trained other 5 models, each with one parameter different from those listed in Table 1. The changed parameter for each model is listed in Table 2.

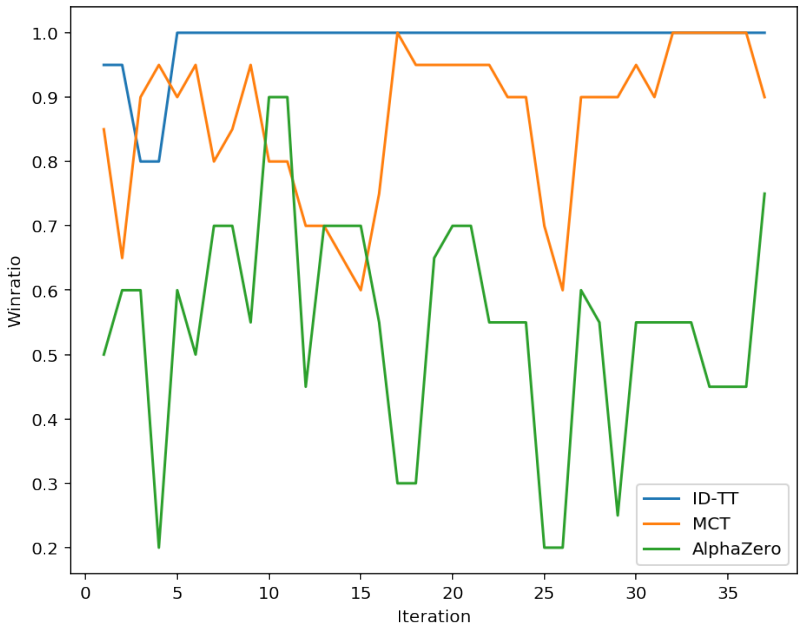

Figure 1: Win ratio of AlphaZero player 1 against other 3 players, for a total of 37 iterations.

Table 2: Changed parameters for the 5 extra model trained, in order to see the effects of the hyperparameters.

Model	Parameters	Values
Model 1	numEps	10
Model 2	tempThreshold	5
Model 3	updateThreshold	0.8
Model 4	numMCSSims	50
Model 5	arenaCompare	60

To see the performances of the different models, we allow each of them to pit against the MCT player, since the latter is the best player apart from the AlphaZero player. They were all trained for a total of 37 iterations, enough for us to see converged results. For each iteration, the AlphaZero player using the current model is to play with the MCTPlayer for 20 games, and at the end of each iteration, a win ratio will be calculated. The win ratios of the 5 models vs. the MCT player can be seen in Fig. 2. It can be seen, after $\sim$ 20 iterations, all models were able to beat model 0, the original model with parameters listed in Table 1. The purple line representing model 4 is seen to be stablised at around 0.975 most quickly, after $\sim$ 18 iterations. The second best model is model 5 represented by the brown line, stablising after $\sim$ 20 iterations, at around 0.95. After $\sim$ 26 iterations, model 2 reaches equilibrium, with minor fluctuations. Model 3 represented by the red line is seen to reach equilibrium after $\sim$ 27 iterations, then fluctuating around 0.95. And finally, the orange line representing model 1, has the largest fluctuations amongst all, is seen surpassing the original model at $\sim$ 25 iteration, and oscillating around 0.8 thereafter. Hence, to improve the performance of the trained model, we can tune the hyperparameters in the following way: i. decrease the value of "tempThreshold"; ii. increae the threshold at which we accept the new neural network; iii. use more number of simulation per MCTS; and iv. play more games between the new network and the old network at arena.

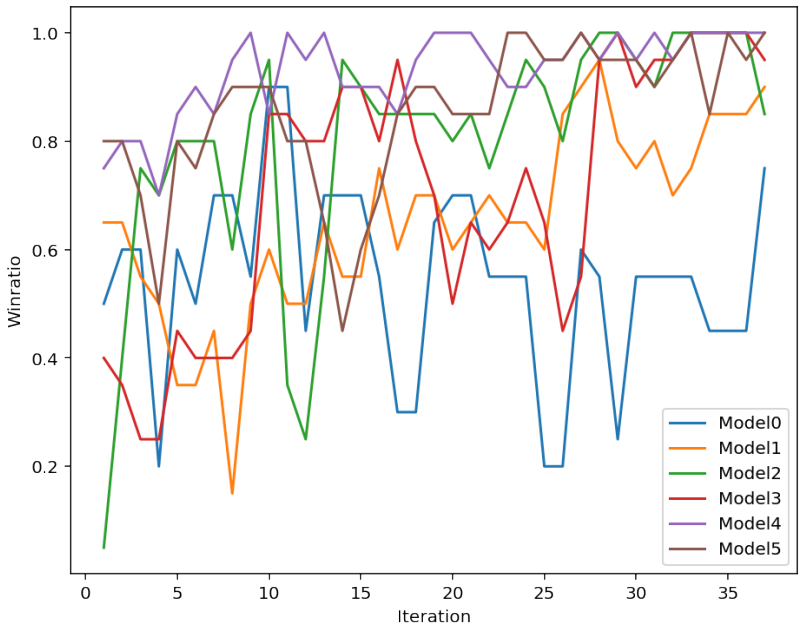

Figure 2: Win ratio of AlphaZero players using the above mentioned 5 different models, pitting against the MCT player, for a total of 37 iterations.

5 Conclusion

In this project, we have learnt the core of self play, by implementing the AlphaZero code developed by Surag Nair et al. on the game of Hex. We have seen the superacy of alphazero algorithm, by playing the model trained against other players. It is seen that in a very short time the AlphaZero player can beat the other players (AlphaBeta player with ID-TT enhancement, MCTree player) with a win ratio higher than at least 60%. Latter on we have explored the possibilities of improving the model by hyperparameter tunning. Since our aim is not to find the best solution, which can be time consuming and computationally too demanding, we went about this by changing one parameter at a time, and see the effect of changing this parameter on the performance of the model. This is done by allowing the model to play against the MCT player, for 20 games per iteration. At last, we have proposed our ways of tunning the parameters, in order to improve the performance of the trained model in a relatively short training epoch.

References

[1] Nair, S. (2017, December 29) A Simple Alpha(Go) Zero Tutorial. Retrieved from: `https://web.stanford.edu/~surag/posts/alphazero.html`

[2] Aske Plaat. Learning to play—reinforcement learning and games, 2020.

[3] Jin, J. (2018, May 20) Training and Implementing AlphaZero to play Hex. Retrieved from: `https://notes.jasonljin.com/projects/2018/05/20/Training-AlphaZero-To-Play-Hex.html`