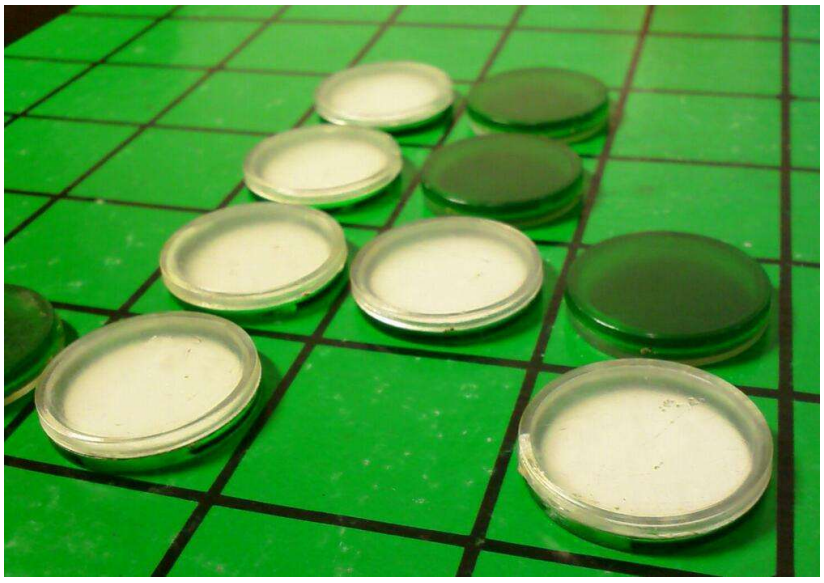
Othello Engine Based On Self-Play Reinfroced Method



Author : Qi Xiangyu(漆翔宇)
SID : 3170104557
Major : Computer Science and Technology
Date : 2019.04.18
Tel : 17342017090
Email : unispac@zju.edu.cn

1 Introduction

For a long time, Go game is considered as an unsolvable computing task for the quantity of the possible layouts of a $19 * 19$ board having an upper bound that can reach 3^{361} which is unimaginable. In 2016, AlphaGo, developed by deepmind, which uses recent deep reinforcement learning and Monto Carlo Tree Search methods, managed to defeat the top human player Sedol Lee and thus aroused a public attention.

Actually, the design of the AlphaGo system still relied on the expert domain knowledge to train the evaluation model. A large amount of historical chessing plays made by top human players were applied for the training. However, in 2017, deep mind again, released a new version of Go system, AlphaGo Zero, which was trained entirely without expert knowledge. It described an approach that reinforces the model with data purely obtained by self-play instead of human-play. And this new Go system even defeated the origin top-level Alpha Go. This result seems to reveal that the computers virtually may have a potentiality to outperform humans independent with any human assistance.

In this homework project, we designed a system to play the $8 * 8$ Reversi Chess based on the idea of AlphaGo Zero paper [1]. We have compared the system with those classic algorithms implemented in the example engines. The result turns out to be impressive : all the example engines are defeated. And soon after that, when testing the agent against human player, we are surprised to find that it is really smart and have actually leanrt some import strategies like occupying the corners and the margin sides which makes the agent easily beat us. Any way, completing such a system is a fascinating thing.

2 Methods

In this part, we will provide a overview of the self-play reinfroced algorithm. The core method is just from the origin AlphaGo Zero paper mentioned above.

The method is totally based on self-play algorithm. A Monte Carlo Tree Search (MCTS) is used in the decision-making process, which is responsible for enumerating the descendant nodes. And a value network is used to guide the tree policy and to replace the procedure of simulations. The core part is the self-play reinforcing method. The essence of this procedure is policy iteration. MCTS plays the role of guider which helps to improve the policy.

In the following sections, details of several components of the methods we use will be discussed.

2.1 Model

The model can be formulated as the form:

$$(\mathbf{p}, v) = f_\theta(s),$$

It accepts a tensor s which represents the state of the board and outputs a policy vector p and a value scalar v. The size of the policy vector p is as large as the size of action set and the value p[i] describes the possibility of the case that the action i is the best strategy. In term of v, it denotes the value of the state ranging from -1 to 1. The higher it is, the more the black side is likely to win. And we stipulate that, in the end state, the actual value of it is 1 when black wins, 0 when reaching a tie, -1 when white wins.

Hence, **our target is to train the arguments θ and to make the model give a better prediction of p and v.**

As the state of the board can be easily represented by a simple 3-values $8 * 8$ bitmap, CNN is considered to be an expressive tools. So we have used a 4-layers CNN to process the state map. And 2 full-connected layers are used to produce p and v.

Because of the lack of experience in designing a Convolutionary Neural Network (CNN), we refered to a simple tutorial written by Surag Nair in Stanford University [2] when dealing with the network construction. Some training tricks were involved in the network construction. Training is performed using the Adam [3] optimizer with a batch size of 64, with a dropout [4] of 0.3, and batch normalisation [5].

2.2 Monte Carlo Tree Search

Essentially, MCTS is just another type of minimax tree search. With depth-limit and a good evaluation function, the search can get a relatively reliable result in limited time resource. With alpha-beta pruning, some obvious bad cases will be avoided and the evaluation function plays a role of heuristic guider. Instead of alpha-beta pruning, MCTS takes use of the tools in probability and mathematical statistics theory and thus achieving an equivalent effect of pruning. But technically speaking, it is not pruning. Bootstrap seems a better term.

Unlike the common UCT algorithm, our MCTS takes a full use of the prediction model to guide the tree policy and to complish the simulation.

The upper bound confidence is defined as follow :

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

It is similar to the UCB1 which encourages approprite exploration to seldom-accessed nodes. But there is a facotr P(s,a) which is predicted by the model, denoting the possibility of the case that action a is the best decision in state s. Highly “recommened” actions by the model get more chances to be explored.

And instead of using quick decision-making strategy to simulate the game playing result, our algorithm just use

the scalar v predicted by model which is more quick and reliable.

2.3 Policy Improvement By Self-Play

We have known that alpha-beta-pruning minimax search is often implemented with a depth-limit and evaluation function. Since we have a evaluation function, why don't we just make decision according to the raw function? The reason is conspicuous. As the evaluation function can't be precise, it just plays a role of heuristic indicator which helps to save us the bother of discussing the obviously bad case. Thus, it is not strange that intuitively and also mathmatically, **the result gotten through search overmatches the one from raw function**..

We have also known that the model will give us a prediction of the value of each state and action. So, the model is essentially in another word, just the raw evaluation fucntion. As the search result is more reliable than the raw function result, in our context, it means that given a model, we can get a better prediction through search than the prediction given by the row model. In our method, MCTS is responsible for the improvement.

So far what we have talked about is just the classic search method. What makes our method outstanding is the ability to **accumulate the improvement** which the traditional search methods lack. Since the search method can take use of the model to give a better prediction and our target is just right to train the model making it produce better prediction, why not just letting the model to learn the better predicion made by the search method? That's the idea. We can let our agent play the game against itself. During the game playing, we record the prediction made by MCTS. And after enough datas are accumulated, we train the origin model with the better prediction examples thus improving the old model.

See the simple policy PI :

$\text{action} = \text{argmax}\{p[i]\}$,

p is the action vector in state s.

So in essence, this method is just a type of policy improvement. Searching, getting a better prediction, reinforcing the old model, getting a better PI.. That's it..

So far we have already talked about the core ideas of self-play reinforce. The last point need to be focused on is the training.

The training data is obtained purely by self-playing, and it is organized as the form :

$$(s_t, \vec{\pi}_t, z_t)$$

Note that z is the actual result of this game and π is the actual policy vector dereived from MCTS.

In the origin paper of Deep Mind, they give out the loss function below :

$$l = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c||\theta||^2$$

Intuitively, reducing the loss means a priciser state evaluation and policy prediction. The term $c * ||\theta||^2$ is used to regularize the loss, which is mainly on the purpose of avoid overfittng. But in our implementation, we simply ignore it for reversi chess is not as complex as Go game and there are some other effetive methods we take to achieve the goal of avoiding overfitting.

3 Experiments

3.1 Training

In our experiments, we have implemented the game agent system for $8 * 8$ reversi chess. When carrying out slef-play, MCTS will do 50 simulations per step to give out the action vector. Each training iteration is taken after 100 episodes of self-play and the datas collectd are used as training set to perform Stochastic Gradient Descent (SGD).

The training frame is implemented in pytorch and has been trained on a GPU (RTX2080TI) for about 3 days.

And as the chess board has an apparent feature of symmetry, each state can be represented by 8 equivalent forms (rotation : 4; mirror projection :2 ; 4*2=8). Thus, in practice, we split each training entry to 8 forms, bundle them together and sent to the learner. This trick ensures that the model can get the concept of symmetry.

Another trick we took when doing the experiment is model evaluation. Altough we have talked about how the better prediction made by MCTS can be used to improve the model, there can always be some problems in practical training as the data may not be sufficient etc… So what we do is to let the new model play against the original one. Only when the new model can reach a win rate greater than 60% will we accept the new model and otherwise we put the formal training data in the history data stacks and restore the model with the old version. The training data will be accumulated untill the new model is accepted.

3.2 Evaluation

And we have run the system to play game against the classic example engine : greedy, random, eona, nonull, order and noorder. (The MCTS will do 200 simulations for each step in our evaluation experiments.)

Example engine	winrate
Greedy	30/30
Random	30/30
Eona	28/30
Nonull	27/30
Order	29/30
Noorder	30/30

Result is shown above. It is really impressive : just with 200 simulations, the agent easily defeats some baseline algorithms as greedy and random, and also outperforms the rest 4 types of engines implemented by traditional alpha-beta search.

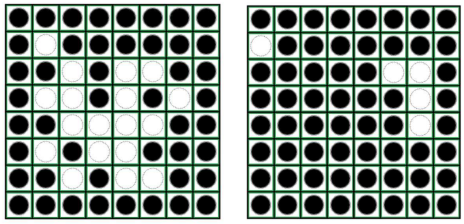


Figure 1.
Left : greedy(white) vs agent
Right : random(white) vs agent

3.3 Analysis

When observing the playing process, we found that the agent have learnt to occupy the corner and always tries to avoid giving the opponents the chance of taking the initialtive to get the corner position. That's smart, as the 4 corner positions is one of the most crucial factors for the game result.

But we fond that the decisions made by the agent at the early phase were week…

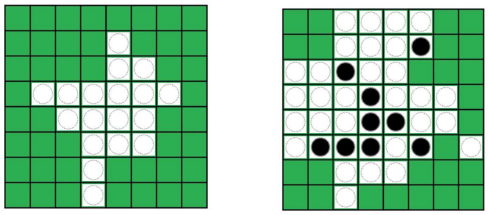


Figure 2 : eona(white) against agent(black)

Although during the game playings against eona, our agent have won 28 rounds out of 30, the early behaviors of the agent are usually not so outstanding. You can see that in figure 2, our agent lose the game very early in the left case.. But the final result of the right case is surprising:

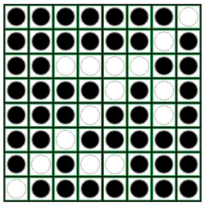


Figure 3 : agent(black) defeats eona(white)

It turns out that our agent did really well in occupying the corner and taking full use of the corner. (See that in the right case of figure 2, it shows the four corners were avoided by both the algorithm. But obviously our agent outperforms the eona in taking use of the corner.) But when dealing with the ordinary margin places, it did badly in the frist several steps so that it lost the early superiority.

It shows that the prediction tends to be more pricise in the late stage and less reliable at the beginning. It is probably out of the reasons that the training is not adequate and the speed of the convergence itself slows down sharply..

3.4 Potential Improvement

Based on the analysis above, at last we proposed 3 ways to improve the agent :

1. Taking more time to train and enhancing the acceptance threshold of the new model. It may improve the quality of our agent. But as the time is limited we didn't have the chance to verify it.

2. In the frist 5 or 6 steps, taking a random strategy based on the probability attribution predicted by MCTS. The idea is simple : Since the early probability predictions for actions are not very different, for example, the best action getting a value 0.31 and the second best getting 0.30, and the early prediction itself is also not so reliable, it makes no sense that we need to always perform the action that having the highest evaluation. Actually, this method is verified by our experiments. The agent with early random policy has a higher winrate when playing against with the original agent.

3. Using some expert domain knowledges to guide the first several steps's strategy.

Reference

[1]. [Silver et al. 2017a] Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2017a. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. ArXiv e-prints.

[2]. [surag 2017] Surag.N; 2017.
A Simple Alpha(Go) Zero Tutorial
web.stanford.edu/~surag/posts/alphazero.html

[3]. [Kingma and Ba 2014] Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[4]. [Srivastava et al. 2014] Srivastava, N.; Hinton, G. E.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. Journal of machine learning research 15(1):1929– 1958.

[5]. [Ioffe and Szegedy 2015] Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International Conference on Machine Learning, 448–456.