Report of Final Project in Neural Networks and Deep Learning

Peng Lu
14307110169
Fudan University
14307110169@fudan.edu.cn

Heyuan Gao
14307110093
Fudan University
14307110093@fudan.edu.cn

Abstract

Reinforcement learning shows better property than supervised learning on specific conditions, especially where sequential decision is required and human experience is not easily available, such as chess games. In this project, we use the AlphaGo Zero self-play algorithm to train an agent which masters the game of Reversi. In the experiments, we examined the trained agent, finding it performs better than alpha-beta algorithm with max search depth 6. We also compare the performance of agents that have difference detailed setting.

1 Introduction

Recently, many achievements in the field of machine learning are made by supervised learning systems which require plentiful labelled data to train themselves. However, labelled data are not easily available in some cases. For example, when a robot is learning to walk on rough ground, developers are not able to provide it with proper movement at every position. Also, in some computer games, such as chesses or video games, there are too much different states to label if we use supervised learning system to train an agent, and human knowledge might not be optimal. In these cases, we can make agent interact with environment, collecting experience and learning from it. This method is named Reinforcement Learning (RL).

One of the most remarkable progress toward Reinforcement Learning in the past year is AlphaGo Zero, an algorithm that enables agents to learn to play ancient Chinese game Go without any human data and domain knowledge [1]. The method I use in this project is originated from AlphaGo Zero, and it shows great power on the game of Reversi.

Reversi is a two-player game on a 8×8 square. The game begins with a particular state where each player has 2 pieces placed in the center of the board (Fig. 1). The black and the white player place at alternate turns one piece at a time and a valid move must make at least one piece of opponent enclosed on a horizontal, vertical or diagonal line. Afterwards, these enclosed pieces are flipped. The game ends when neither player has a valid move and the player with more pieces on the board is declared winner. When the number of pieces of each color are equal, the game ends in a draw. The game of Reversi has simple rules but large number of potential states, and it has been a popular benchmark for computational intelligence methods [2].

Because of the sequential decision nature of the game of Reversi, it is suitable to use Reinforcement learning to train agents that master the game. Reinforcement learning is learning what to do - how to map situations to actions - so as to maximize a numerical reward signal.[3] In a Reinforcement learning system, an agent takes action $a \in \mathcal{A}$ according to the state of the environment $s \in \mathcal{S}$ based on its policy $\pi(a|s)$, and the state of environment changes afterwards: $s' \sim p(s'|s, a)$. The agent will receive rewards from environment $r(s, a, s')$. The trajectory of n^{th} trial is

$$\tau^n = s_0^n, a_0^n, s_1^n, a_1^n, r_1^n, \cdots, s_{T-1}^n, a_{T-1}^n, s_{T_n}^n, r_{T_n}^n$$

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

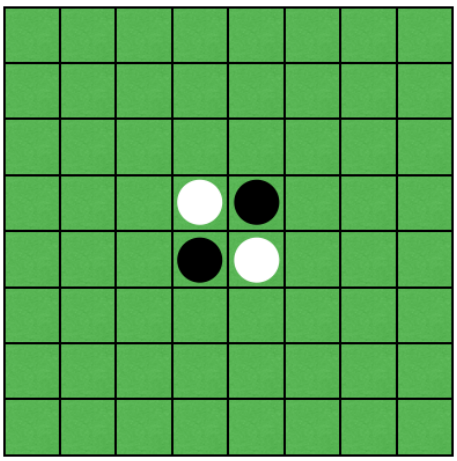

Figure 1: The begining state of game Reversi

where T_n is the number of steps in this trial and $r_t^n = r(s_{t-1}^n, a_{t-1}^n, s_t^n)$. And we also assume that the trials are Markov decision processes, which means the state of time $t+1$ (s_{t+1}^n) is only related to the last state and action s_t^n, a_t^n:

$$p(s_{t+1}^n | s_t^n, a_t^n, \cdots, s_0^n, a_0^n) = p(s_{t+1}^n | s_t^n, a_t^n)$$

Thus the probability of trajectory τ^n can be written as:

$$p(\tau^n) = p(s_0^n) \prod_{t=1}^{T_n - 1} \pi(a_t^n | s_t^n) p(s_{t+1}^n | s_t^n, a_t^n)$$

Also, the return of a trial is the total reward agent receives:

$$G(\tau^n) = \sum_{t=0}^{T-1} r_t^n$$

Notice that r_t can be either positive or negative. When $r_t < 0$, we can regard it as penalty.

The goal of Reinforcement learning system is to find a policy that enables agent to get relatively large return in every trials. Thus our object is expected return with regard of policy $\pi_\theta(a|s)$:

$$\mathcal{J}(\theta) = \mathbb{E}_{\tau \sim p_\theta}[G(\tau)]$$

where θ represents parameters of policy. The expected return is determined given a particular policy while return of one trial might remain random (but related to the policy).

2 Reinforcement Learning and Reversi

In the game of Reversi, at step t, state s_t represents the board, or position of all pieces; the player (our agent) can observe the state and make decision of next action according to its policy: $a_t \sim \pi_\theta(\cdot|s_t)$. Afterwards, the opponent places his/her piece, and our agent observes next state $s_{t+1} \sim p_\theta(s_{t+1}|s_t, a_t)$, which is determined by agent's policy and the opponent's policy besides s_t. As for the reward, in this case our agent do not receive immediate reward after each action but a return $G(\tau)$ after a trial τ ends.

2.1 Policy gradient

Since policy is mapping from state to action, we can use network to model it, and θ is the parameters of the network. Thus out policy $\pi_\theta(a|s)$ is differentiable on θ. We can use gradient ascend, optimizing parameter θ to maximize out object $\mathcal{J}(\theta)$. The gradient of parameters is:

$$\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \frac{\partial}{\partial \theta} \int p_\theta(\tau) G(\tau) d\tau = \int \left(\frac{\partial}{\partial \theta} p_\theta(\tau) \right) G(\tau) d\tau \tag{1}$$

$$= \int p_\theta(\tau) \left(\frac{G(\tau)}{p_\theta(\tau)} \frac{\partial}{\partial \theta} p_\theta(\tau) \right) d\tau \tag{2}$$

This representation shows that in order to maximize $\mathcal{J}(\theta)$, for a trajectory τ, if $G(\tau) > 0$, which means agent received positive return, then $\frac{\partial}{\partial \theta} p_\theta(\tau)$ should be positive, so the probability of appearance of τ increases. What is weird is that $G(\tau)$ is divided by $p_\theta(\tau)$. This implementation can be regarded as normalization. If $G(\tau)$ is not divided by $p_\theta(\tau)$, $G(\tau^1) > G(\tau^2)$ and $p_\theta(\tau^1) < p_\theta(\tau^2)$, since τ^2 will be sampled with more times than τ^1, and $p_\theta(\tau^2)$ might be enlarged more than $p_\theta(\tau^1)$. It is unreasonable because we want to sample τ^1 more times since it leads to greater return. This problem emits in equation (2). Notice that $\frac{1}{p_\theta(\tau)} \frac{\partial}{\partial \theta} p_\theta(\tau) = \frac{\partial}{\partial \theta} \log p_\theta(\tau)$ and by Markov property, we get:

$$\frac{\partial}{\partial \theta} \log p_\theta(\tau) = \frac{\partial}{\partial \theta} \log \left(p(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \right) \tag{3}$$

$$= \frac{\partial}{\partial \theta} \left(\log p(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right) \tag{4}$$

$$= \sum_{t=0}^{T-1} \frac{\partial}{\partial \theta} \log \pi_\theta(a_t|s_t) \tag{5}$$

so

$$\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \int p_\theta(\tau) \left(G(\tau) \frac{\partial}{\partial \theta} \log p_\theta(\tau) \right) d\tau \tag{6}$$

$$= \int p_\theta(\tau) \left(G(\tau) \sum_{t=0}^{T-1} \frac{\partial}{\partial \theta} \log \pi_\theta(a_t|s_t) \right) d\tau \tag{7}$$

$$\approx \sum_{n=1}^{N} \sum_{t=0}^{T-1} G(\tau^n) \frac{\partial}{\partial \theta} \log \pi_\theta(a_t|s_t) \tag{8}$$

The last approximation means we can play chess for N episodes, let $G(\tau^n)$ be the indicator of the result:

$$G(\tau^n) = \begin{cases} 1, & \text{our agent wins} \\ -1, & \text{out agent loses} \end{cases}$$

After that, maximize the pseudo-object $\widetilde{J}(\theta) = \sum_n [G(\tau^n) \sum_t \log \pi_\theta(a_t|s_t)]$.

So if we use policy gradient to train out agent, training process is: agent plays games and collects data, then updating the parameter of the policy network, next plays games again. The training of policy in the algorithm in this project is similar to policy gradient, and I will compare these two methods later.

Our opponents can be either other agent or agent itself. If our agent plays against itself, which means in the first turn it acts as black players, and in the next turn, it acts as white player. This mechanism can be easily implemented by flipping all pieces, then the white player can use same policy as the black player. Thus in one trial we can collect two trajectories and two reversed return, which sky-rockets the efficiency. Also, AlphaGo Zero[1] has proven that agent that learns with itself can achieve even higher performance than those trained with human experts.

2.2 Monte Carlo Tree Search

Typically, Monte Carlo methods for sampling can be implemented by following steps: (1) given a initial state s_0 (2) using the policy to simulate a trial τ (3) and collect the return. At each state s_t, the agent selects the followed action simply according to policy. In the AlphaGo Zero algorithm, Monte Carlo Search Tree (MCTS) is used as a movement recommender, since MCTS provides stronger moves than the raw probability. [1] Here stronger means that the movement generated by MCTS is more likely to lead to winning.

Next part illustrate the MCTS algorithm used in AlphaGo-Zero. If our agent observes state s, how to simulate next action a? In the search tree, each node represents a specific state s and contains edges $\{(s, a); a \in \mathcal{A}(s)\}$ that is valid given s. Edges linked with node s represent valid actions. An edges representing action a contains a set of statistics:

$$\{N(s, a), W(s, a), Q(s, a), P(s, a)\}$$

where $N(s, a)$ is the visit count, $W(s, a)$ is the total action value, $Q(s, a) = W(s, a)/N(s, a)$ is the mean action value and $P(s, a)$ is the prior probability of selecting that edge which is computed by our policy $\pi_\theta(\cdot|s)$. The distribution $\widetilde{\pi}(a|s)$ can be generated by iterating the followed 4 steps:

Selection: select an action to explore. Next action:

$$a = \arg\max_b (Q(s, b) + U(s, b))$$

where $U(s, b) = c_{puct} P(s, b) \frac{\sqrt{\sum_e N(s,e)}}{1+N(s,b)}$, which is a variant of the PUCT algorithm. And c_{puct} is a constant determining the level of exploration. Search tree with greater c_{puct} are more prone to explore unexplored actions. This search control strategy selects actions with higher prior probability and low visit count at beginning, and prefers actions with high Q-value in the subsequent search [1].

Expansion: after taking action a, the search tree gets state s' (only in the search process, no effect on the real agent.) If s' has been explored, then search tree uses same selection method on s', otherwise expands s' to the tree and initialize its statistics (find valid moves and for each valid move a': $N(s', a') = W(s', a') = Q(s', a') = 0, P(s', a') = \pi_\theta(a'|s')$). Afterwards, using simulation to get the value of state s': $V(s')$.

Simulation: typical MCTS algorithms use rollout to investigate the value of state s'. Rollout means we start from state s' and use policy π_θ to simulate until an end of the game and use the return as $V(s')$. AlphaGo Zero uses another network V_ϕ to approximate the value function if s' is not an end state. After simulation, we get the value of state s', and then backforward it through coming path.

Backpropagation: Suppose that the path from s to an expanded (or end) node s' is:

$$s \xrightarrow{a} \cdots \xrightarrow{a_{(i-1)}} s_{(i)} \xrightarrow{a_{(i)}} \cdots \to s'$$

Then for every node $s_{(i)}$ on the path, we add the value of s' to total action value $W(s_{(i)}, a_{(i)}) = W(s_{(i)}, a_{(i)}) + V_\phi(s')$ and compute its mean action value $N(s_{(i)}, a_{(i)}) = \frac{N(s_{(i)}, a_{(i)})}{N(s_{(i)}, a_{(i)})}$.

The process of these four steps is a search. In order to get the recommended distribution $\widetilde{\pi}(a|s)$, we have to do several searches to expand the search tree and optimize the parameters of each edge. The number of search, which is denoted by n_s, is a hyperparameter that controls the balance of search depth (parameter accuracy) and search efficiency.

After n_s searches that begins on state s,

$$\widetilde{\pi}(a|s) = \frac{N(s, a)^{1/k}}{\sum_b N(s, b)^{1/k}}$$

where k denotes temperature. High temperature leads to high entropy. For example, when $k \to +\infty$, $\widetilde{\pi}(a|s)$ is uniform distribution over all valid moves given s. Contrarily, $\widetilde{\pi}(a|s)$ collapse to the Dirac delta function as $n \to 0$. On the early stage in training process, I set $k = 1$ to encourage the agent to explore different strategies. After 15 steps, the temperature is reduced to 0 to make moves stronger so that agent can learn how to beat opponents in the middle of a game.

Since the recommended distribution generates stronger moves than policy network (which is verified in experiments), our target is to make policy network able to mapping a state s to $\pi_\theta(s)$ which is close to distribution given by MCTS.

2.3 Networks

We have two networks, policy network π_θ maps a state s to a distribution $p_{\pi_\theta}(\cdot|s)$ and value network maps a state s to its value function $V_\phi(s)$, expected return. Policy network works as base of MCTS and value network simplified MCTS by replacing rollout. These two networks share most of layers, and branch off at the last layer. This setting makes networks learn faster because the weights of front layers are optimized with more times.

AlphaGo Zero uses Resnet instead of typical CNNs and achieves great progress. But the advancement of Resnet emerges only when the network is very deep. Because of the limitation of computer, I simply use a CNN followed by fully connected layers.

Also, the input of network used by AlphaGo Zero contains several history states, which is not implemented here.

2.4 Training Pipeline

Initialization: We initilize the parameters of the network to (θ_0, ϕ_0).

Simulation: We simulate N trials. In n^{th} trial, the agent acts as black player at first, using MCTS to generate a recommanded distribution and action

$$(s_1^n, \widetilde{\pi}(a|s_1^n), a_1^n)$$

Then the agent acts as white player and observed a flipped board as state s_2^n. Also, using MCTS we collect

$$(s_2^n, \widetilde{\pi}(a|s_2^n), a_2^n)$$

At end of the trial, the collected tuples are

$$\{(s_t^n, \widetilde{\pi}(a|s_t^n), a_t^n); t = 1, \cdots T\}$$

If the black player wins, then we can append each tuples with return

$$\{(s_t^n, \widetilde{\pi}(a|s_t^n), a_t^n, r_t^n = (-1)^{t+1}); t = 1, \cdots T\}$$

otherwise the ultimatly collected data is

$$\{(s_t^n, \widetilde{\pi}(a|s_t^n), a_t^n, r_t^n = (-1)^{t}); t = 1, \cdots T\}$$

Training: With data collected in the last step, we can train our network. The object is

$$L = \sum_n \sum_t \left[(V_\phi(s_t^n) - r_t^n)^2 - \widetilde{\pi}(a|s_t^n) \log \pi_\theta(a|s_t^n) \right]$$

The second item is the cross entropy between the recommanded distribution and our policy. AlphaGo Zero uses L2-normalization, but to get rid of finding the regularization parameter since limited computation resources, I use dropout as regularization.

Competing: After training, we get a new network $(\pi_{\theta'}, V_{\phi'})$. Let new network compete against old network, if it performes better, then we keep it and use it as new base of MCTS, otherwise we abandon it.

2.5 Comparison with policy gradient

In policy gradient algorithm, the object is

$$\widetilde{J}(\theta) = \sum_n [G(\tau^n) \sum_t \log \pi_\theta(a_t|s_t)] = \sum_t \left(\sum_n G(\tau^n) \right) \log \pi_\theta(a_t|s_t)$$

Notice that $\sum_n G(\tau^n) \propto Q^\pi(s, a)$.

In the AphaGo Zero algorithm, the object for policy network is

$$\sum_a \sum_t \sum_n \widetilde{\pi}(a|s_t^n) \log \pi_\theta(a|s_t^n) = \sum_a \sum_t \sum_n \frac{N(s_t^n, a)}{\sum_b N(s_t^n, b)} \log \pi_\theta(a|s_t^n)$$

and $\frac{N(s_t^n, a)}{\sum_b N(s_t^n, b)}$ is the visit proportion to action a. In the selection step in MCTS search, the visit frequency to edge a from node s_t^n is close to prior distribution $\pi_\theta(a|s_t^n)$ at first and asymptotically prefers actions with high action value. So $\frac{N(s_t^n, a)}{\sum_b N(s_t^n, b)}$ is strongly related to $Q^\pi(s, a)$.

3 Experiments

As mentioned in section 2, there are several hyperparameters in the algorithm. And I conducted experiments to investigate the effect of changing these hyperparameters on performance when the agent is competing with others. The effect of hyperparameters during training process should have been examined, but since the training procedure requires considerable computation resources, I did not implement this experiment.

I use eight different agents as player1, two of them use simple policy network with different temperature k, and the others use policy network together with a MCTS. I examined the effect of number of search n_s, constant c_{puct} and temperature k on agents with MCTS. Besides these eight agents, I also use a random choice agent and three alpha-beta agent with different max search depth as player2. The competition result is shown in Figure x.

Temperature: we know that temperature controls entropy of a distribution. Higher temperature makes distribution more uniform, which enables agent to explore more potential actions in the training process. However, when we evaluate the ability of an agent, low temperature leads to better performance. The overall winning rate of agents with lower temperature over the agents with higher temperature (other parameters are same) is 0.75. This is because low temperature constrains the distribution of action given a state to the action that agent want to take most, and these actions generally lead to winning. However, as for agent with a policy network, its policy collapse to a determined policy which is just a mapping from a state to a specific action. At this time, if the opponent used determined policy, all games will generate same trajectories.

MCTS as policy improvement operator: As stated in [1], MCTS outputs distribution which leads to stronger moves.

Number of search: By intuition, more search leads to more accurate approximation of action value function, and outputs stronger moves. And it is supported by the experiment. The overall winning rate of agents with $n_s = 100$ over agents with $n_s = 50$ (and all others parameters are same) is 0.82.

Constant c_{puct}: This constant determines the level of exploration of a search tree. Large constant makes search tree prone to explore. In the training process, c_{puct} is always kept 1.0. In the experiment, we can find that when $n_s = 50$, agent with $c_{puct} = 1.5$ is more powerful, and the overall winning rate is 0.58. However, since the action preference evolutes with accordance of the time of search, so the effect of c_{puct} on performance is also influenced by n_s. Here we can only state that when $n_s \approx 50$, agent with $c_{puct} = 1.5$ performed better than that with $c_{puct} = 1$.

Other agents: I also use agents using alpha-beta algorithm policy as baseline and found that the performance of those agents do not strictly be positively related to the maximum search depth. As shown in Figure 2, max depth 4 enables agent win more games than max depth 6.

4 Discussion

The AlphaGo Zero self-play algorithm is a model-based Reinforcement learning method. It can be either on-policy or off-policy since we can easily change some hyperparameters of the model and might achieve better agent performance.

We find that our agent is weak when acts as white player, which might be due to that in the training process, when the agent acts as white player, we simply flipped the pieces and train it as black player. This implementation might lead to ambiguity since black player and white player might have different optimal policy.

References

[1] Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of go without human knowledge[J]. Nature, 2017, 550(7676): 354.

[2] Liskowski P, Jaśkowski W, Krawiec K. Learning to Play Othello with Deep Neural Networks[J]. arXiv preprint arXiv:1711.06583, 2017.

[3] Sutton R S, Barto A G. Reinforcement learning: An introduction[J]. 2011.

p2 \ p1	MCTS num:50, cpuct:1.0, temp:0	MCTS num:50, cpuct:1.0, temp:1.0	MCTS num:50, cpuct:1.5, temp:0	MCTS num:50, cpuct:1.5, temp:1.0	MCTS num:100, cpuct:1.0, temp:0	MCTS num:100, cpuct:1.0, temp:1	net player - temp:0	net player - temp:1	win rate of p2
MCTS num:50, cpuct:1.0, temp:0	0.46	0.21	0.6	0.26	0.84	0.41	0.06	0.07	0.636
MCTS num:50, cpuct:1.0, temp:1.0	0.79	0.49	0.86	0.56	0.95	0.79	0.53	0.32	0.339
MCTS num:50, cpuct:1.5, temp:0	0.42	0.18	0.4	0.2	0.62	0.36	0.02	0.06	0.718
MCTS num:50, cpuct:1.5, temp:1.0	0.74	0.5	0.83	0.47	0.85	0.68	0.35	0.26	0.415
MCTS num:100, cpuct:1.0, temp:0	0.22	0.08	0.27	0.14	0.36	0.23	0.02	0.01	0.834
MCTS num:100, cpuct:1.0, temp:1	0.52	0.31	0.54	0.36	0.71	0.46	0.28	0.22	0.575
net temp:0	0.94	0.56	0.94	0.73	1	0.69	0.5	0.36	0.285
net temp:1	0.96	0.74	0.94	0.71	1	0.84	0.68	0.56	0.19625
random player	1	0.99	0.99	0.98	1	1	0.96	0.96	0.015
alpha beta depth:2	1	0.84	0.98	0.78	0.9	0.84	0	0.62	0.255
alpha beta depth:4	0.89	0.62	0.85	0.56	0.7	0.76	0	0.38	0.405
alpha beta depth:6	0.7	0.51	0.79	0.48	0.82	0.73	1	0.24	0.34125
win rate of p1	0.72	0.502	0.749	0.519	0.812	0.649	0.366	0.338	

Figure 2: Results of competition between 8 player1 and 12 player2. The title of columns represent the type and hyperparameters of player1 (num is n_s, count is c_{puct}, temp is temperature); the title of rows represent the type and hyperparameters of player2. In each square is the winning rate of player1 over player2. The last column records the winning rate of corresponding player2. All winning rate are generated by 100 games.