Building a Reversed Othello Agent with Actor-Critic Method

Mengxuan Wu
Department of Computer Science and Engineering
Southern University of Science and Technology
12212006@mail.sustech.edu.cn

***Abstract*—This project explores the development of an agent for the game of Reversed Othello by employing the Actor-Critic method, a reinforcement learning approach that combines the benefits of both policy-based and value-based methods.**

I. Introduction

Reinforcement learning (RL) is a machine learning paradigm that enables an agent to learn how to behave in an environment by performing actions and receiving rewards. The Actor-Critic method is a popular RL algorithm that combines the benefits of both policy-based and value-based methods. In this project, we aim to develop an agent for the game of Reversed Othello using the Actor-Critic method.

II. Preliminary

A. Background

1) Reversed Othello: In the classic game of Othello, two players take turns placing discs on an 8×8 board. The discs are black on one side and white on the other. When a player places a disc on the board, any discs of the opposing color that are sandwiched between the newly placed disc and another disc of the same color are flipped to the new color. The game ends when no more moves can be made, and the player with the most discs of their color on the board wins.

In Reversed Othello, the rules are the same as in the classic game, but the goal is to have the fewest discs of your color on the board at the end of the game.

2) Actor-Critic Method: The Actor-Critic method is a reinforcement learning algorithm that combines the benefits of both policy-based and value-based methods. The Actor is responsible for selecting actions based on the current policy, while the Critic evaluates the value of the chosen actions. The Actor-Critic method uses a neural network to approximate the policy and value functions.

B. Notation

In the context of reinforcement learning, the problem is often formulated as a Markov decision process (MDP), defined by the tuple $(\mathcal{S}, \mathcal{A}, p, r)$, where $\mathcal{S}$ denotes state space and $\mathcal{A}$ denotes action space, and the unknown state transition probability $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0; 1)$ represents the probability density of the next state $s_{t+1} \in \mathcal{S}$ given the current state $s_t \in \mathcal{S}$ and action $a_t \in \mathcal{A}$. The environment emits a bounded reward $r : \mathcal{S} \times \mathcal{A} \to [r_{min}; r_{max}]$ on each transition. We will use $\rho_\pi(s_t)$ and $\rho_\pi(s_t, a_t)$ to denote the state and state-action marginals of the trajectory distribution induced by a policy $\pi(a_t|s_t)$.

III. Methodology

This project follows the same methodology used in the AlphaZero algorithm for the game of Go [1]. That is, we will use a combination of Monte Carlo Tree Search (MCTS) and neural networks for both training and playing the game.

A. General Workflow

1) Training: The training process consists of three main steps: self-play, neural network training, and self-pitting.

Algorithm 1: Training Algorithm

```
Initialize the neural network parameters $\theta$ randomly;
for iteration $i = 1, 2, \dots, N$ do
    for episode $j = 1, 2, \dots, M$ do
        Run MCTS to generate training data;
    end
    Train the neural network using the training data;
    Self-pit the neural network against the previous
      best version;
    if the new version wins by a large margin then
        Save the new version of the neural network;
    end
end
```

2) Playing: The playing process is based on the MCTS algorithm, except that the neural network is used to guide the search. To be more specific, we modify the UCT formula to include the neural network's policy output and value output. The modified formula is as follows:

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

where $Q(s, a)$ is the action-value function, $\pi(a|s)$ is the possibility of taking action a in state s, and $N(s, a)$ is the visit count of action a in state s. And c_{puct} is a hyperparameter that controls the exploration-exploitation trade-off.

B. Detailed Implementation

1) Neural Network: We use a single neural network to approximate both the policy and value functions. The neural network takes the current state s as input and outputs a policy vector $\vec{\pi}(s)$ that represents the probability distribution over actions, and a scalar value $v(s)$ that represents the value of the current state.

In each episode, we run MCTS to generate training data, which are in the form of $(s, \vec{\rho}, z)$ tuples, where s is the state, $\vec{\rho}$ is an estimate of policy (will be covered in the next subsection), and z is the outcome of the game in the perspective of the current player. The neural network is then trained to minimize the following loss function:

$$L = \sum_t (v(s_t) - z_t)^2 - \vec{\rho_t} \cdot \log(\vec{\pi}(s_t)) \tag{2}$$

For the first term, we want to minimize the difference between the predicted value and the actual outcome, by using the mean squared error. For the second term, we want to adjust the policy to match the estimated policy, by using the cross-entropy loss. Its underlying idea is that the neural network will learn to predict the value of the current state and learn a policy that predicts the correct action to take in that state.

2) Monte Carlo Tree Search: The MCTS algorithm is used to search for the best move in the game tree. And we will also use the result of the MCTS to improve the policy.

In the training phase, the MCTS algorithm will choose the best action to rollout based on the UCT formula for each step. After a certain number of simulations, the UCT values of all actions will be normalized to form a policy vector $\vec{\rho}$, which is seen as an estimate of the true policy and will be used to train the neural network.

3) Tricks for Better Performance:

- **Symmetry:** We can exploit the symmetry of the game to improve the training efficiency. For each sampled state, we will use flip and rotate operations to generate more training data.
- **Replay Buffer:** We can store the training data in a replay buffer and sample mini-batches from it to train the neural network.
- **Temperature:** We can use a temperature parameter to control the level of exploration in the MCTS algorithm. In our implementation, we will sample actions from the policy for the first 15 steps and then use the best action for the remaining steps.
- **Parallelism:** We can use multiple threads to run the MCTS algorithm in parallel to generate samples more efficiently. Every time we forward propagate through the neural network, we wait for all threads to finish and utilize the batched operation for optimal performance.

C. Analysis

The off-policy algorithm and the parallelism trick both help to improve the training efficiency. This is the most obvious when running on a GPU, as the batched operation reduce both the forward and backward propagation time.

Also, this method is extendable to other games, since the environment part (game simulation) can be easily extracted and replaced with another game, while the reinforcement learning part (neural network and MCTS) remains the same.

IV. Experiments

A. Setup

The source code I used for this project is based on the open-source framework *AlphaZero General* [2], a Python implementation of the AlphaZero algorithm. The framework provides a simple interface for building and training agents, and I implement the Reversed Othello environment and the neural network model based on the framework. However, this repository is not well-maintained (stopped updating 2 years ago), and lacks the support for parallel MCTS simulation. So I turn to a forked repository [3] that has implemented the parallel MCTS simulation and use Cython for better performance.

Since the training process is still computationally expensive, I use the GPU resource from Colab Pro to speed up the training process.

Because the competition platform only accepts a single python file with limited file size, I experiment with a network that is much smaller than the original one to reduce the model size (the original model has 4 convolutional layers and 2 fully connected layers, while my model has 2 convolutional layers and 2 fully connected layers with fewer neurons).

B. Results

The training process is fast and stable. For each iteration, it takes 3 minutes to run 100 games for sample generation, and 2 minutes for training. Generally, the program take about 6 minutes for each iteration, and the training process converges after 20 iterations (win rate stable at 0.5).

For the hyperparameters, I simply stick to the original values used for the classic Othello game in the *AlphaZero General* repository, due to the lack of time for tuning.

However, the final model is not as strong as I expected. The model can beat me easily, but it cannot beat the pure MCTS agent without neural network guidance.

C. Analysis

Since the model does converge, the problem may lie in the model size. The original AlphaZero paper uses 20 residual blocks as the neural network for the game of Go, and the *AlphaZero General* repository uses a 4-layer CNN with 512 channels followed by 2 fully connected layers with 1024 and 512 neurons respectively for a 6×6 classic Othello game, while my model only has 2 convolutional layers with 64 channels followed by 2 fully connected layers with 512 and 256 neurons respectively for a 8×8 Reversed Othello game. The reduced model size leads to a lack of representation power, which results in poor performance.

Also, the lack of hyperparameter tuning may also contribute to the poor performance.

Additionally, I noticed some errors in the framework (may be due to the lack of maintenance). The parallel MCTS simulation might not be seeded correctly sometimes, which leads to the same result for each thread.

V. Conclusion

This project is a great opportunity for me to implement my knowledge of reinforcement learning and neural networks, which I have newly learned in the past two months for my group project. It's also a good chance for me to practice implementing a famous algorithm and to learn how to use open-source frameworks.

However, the result is not as good as I expected. The poor performance may be due to the lack of model size and hyperparameter tuning, as well as some errors in the framework.

For future work, I will try to apply some hyperparameter tuning techniques to improve the performance of the model. For example, the Population-Based Training (PBT) algorithm can be used to automatically tune the hyperparameters during training.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017.

[2] S. Thakoor, S. Nair, and M. Jhunjhunwala, "Learning to play othello without human knowledge," 2016.

[3] Bobingstern, "Bobingstern/alphazero-general: A fast, generalized, and modified implementation of deepmind's distinguished alphazero in pytorch.."