Efficient Exploration with Failure Ratio for Deep Reinforcement Learning

Minori Narita[1] **and Daiki Kimura**[2]
[1]University of Massachusetts, Amherst, United States
[2]IBM Research AI, Japan
[1]mnarita@umass.edu, [2]daiki@jp.ibm.com,

Abstract

A combination of Monte Carlo tree search (MCTS) and deep reinforcement learning has demonstrated incredibly high performance and has been attracting much attention. However, the convergence of learning is very time-consuming. When we want to acquire skills efficiently, it is important to learn from failure by locating its cause and modifying the strategy accordingly. Using the analogy of this context, we propose an efficient tree search method by introducing a failure ratio that has high values in important phases. We applied our method to Othello board game. In the experiments, our method showed a higher winning ratio than the state-of-the-art method, especially in the early stages of learning.

Introduction

Reinforcement learning (RL) has been attracting much attention recently. Deep Q networks (Mnih et al. 2015) and other algorithms have been proposed for various game applications (Pathak et al. 2017; Kimura et al. 2018; Silver, Huang, and et al. 2016; Silver et al. 2017). AlphaGo (Silver, Huang, and et al. 2016) was the first algorithm to beat a human professional player in a full-size game of Go. AlphaZero (Silver et al. 2017) has also demonstrated an astounding performance by surpassing professional Shogi (Japanese chess) players after only 24 hours of training, thus reducing the amount of huge training data required previously. These achievements were obtained by integrating an adversarial setting called "self-play."

The learning process of AlphaZero (Silver et al. 2017) consists of the following two phases: self-play to create training data by utilizing Monte Carlo tree search (MCTS), and the update of deep network parameters using the training data obtained in the previous step. MCTS is an algorithm that searches the action that gives high mean rewards by multiple simulations of the gameplay. In AlphaZero (Silver et al. 2017), it takes the balance between exploration and exploitation by taking the summation of the predicted winning ratio and the upper confidential bound that weighs the exploration of nodes less visited by the agent.

However, AlphaZero has a huge computational cost; it requires 5,000 TPU for self-play and 64 GPU for network updates. Recently, some methods such as prioritized experience replay (Schaul et al. 2015) have been proposed to overcome this issue by means of weighing exploration. In this method, data sampling is prioritized based on temporal difference (TD) to make the learning more efficient. However, as the method weighs training data to decide which samples should be drawn next, it contributes to the efficiency only after the agent has experienced an episode. Therefore, we need to improve the efficiency of the self-play process, which creates the training data used in the network updates.

On the other hand, when we humans acquire skills, it is important to learn from failure, finding its cause and modifying the strategy accordingly. The above process can help to avoid making the same mistake again. For example, when we play a board game, there are moments when we want to cancel the last action. In a practice environment, we can cancel the last move and try a different move, which is called 'undo'. These situations are most likely to be important for determining the winner. Using the analogy of this concept, we assume that the learning efficiency can be improved if the algorithm weighs the exploration of specific moves considered as failure moves.

Therefore, we propose a framework that encourages exploration of critical situations leading to failures. To weigh critical situations, we apply weighing to the tree search directly. We define the failure ratio as a difference between the Q-value (the value of the actions according to specific situations) of the current move and that of the previous move. It encourages the agents to do a prioritized exploration of situations that make the difference between winning and losing. We evaluate our method by using multiple Othello settings and compare it with the existing method (Silver et al. 2017). Our study offers the following contributions:

- Introducing a novel idea of "failure ratio", which utilizes the gap between the current predicted winning ratio and the previous ratio;
- Making the self-play process efficient by weighing the important phases for winning with the failure ratio;
- Evaluating our method on Othello and achieving a higher winning ratio than AlphaZero in the early learning stages.

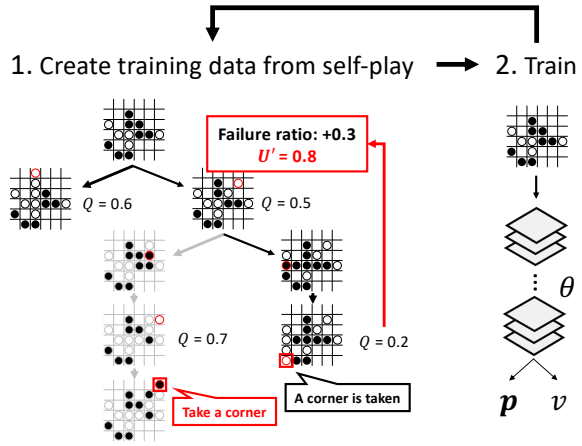

Figure 1: Overview of the learning architecture with failure ratio in Othello

Proposed Method

The overview of our proposed method is shown in Fig. 1. As in AlphaZero, the learning process consists of the combination of self-play to create training data and the parameters update of the policy and value networks. In self-play, the agent runs simulations of the game to generate training data. We integrated the notion of failure ratio into this self-play architecture. The second phase, or the parameter updates, follows the same way as in AlphaZero. In other words, using the training data obtained from self-play, the agent takes the board s_t as an input and the probability of choosing action $p = P(a|s_t)$ and the expected return $v \approx E[R_t|s_t, a_t]$ as outputs to learn parameter θ of the network $(p, v) = f(\theta)$.

In AlphaZero, the training data is produced by self-play using MCTS, and the parameters of the policy and value function networks are updated using this data. The MCTS algorithm searches a tree consisting of nodes, each of which corresponds to a different configuration of the board. Each node maintains the expected return for each state-action pair ($Q(s_t, a_t)$), and the number of times the agent visited a specific state-action pair in a simulation ($N(s_t, a_t)$). $Q(s_t, a_t)$ is the predicted winning ratio when the agent chooses action a_t in state s_t. Each node also holds a probability of taking action a_t from state s_t if the agent follows the current policy ($P(s_t, a_t)$). Then, AlphaZero expands the tree, maximizing the predicted winning ratio with the upper confidence bound $U(s_t, a_t)$, which is the summation of the predicted winning ratio $Q(s_t, a_t)$ and the upper confidence bound $b(s_t, a_t)$ that weighs less visited nodes. $b(s_t, a_t)$ is calculated from the above $P(s_t, a_t)$ and $N(s_t, a_t)$ values from each node. Hence, the agent goes down the tree from the root, expanding nodes by selecting the action that maximizes $U(s_t, a_t)$. $U(s_t, a_t)$ is calculated as follows:

$$U(s_t, a_t) = Q(s_t, a_t) + b(s_t, a_t), \tag{1}$$

$$b(s_t, a_t) = P(s_t, a_t) \frac{\sum_a N(s_t, a)}{1 + N(s_t, a_t)}. \tag{2}$$

In this study, we introduce a "failure ratio" and integrate it into the calculation for $U(s_t, a_t)$ (Eq. 1) for prioritizing important situations in self-play. We assume the failure ratio is calculated from the difference between the next predicted winning ratio and the current predicted winning ratio of the same agent. The next predicted winning ratio is the ratio of two time-steps ahead, because one step ahead is the opponent's turn. The definition of the failure ratio $f(s_t, a_t)$ is described in Eq 4. $Q(s_{t+2}, a_{t+2})$ denotes the predicted winning ratio at time $t + 2$ (the agent's next turn), two time-steps ahead of the current node at time t. When the failure ratio is high, the action a_t at time t will correspond to the failure action because the winning ratio will decrease compared to the previous value. In contrast, if the failure ratio is low, the action a_t is regarded as a good move. Hence, our method encourages agents to select good actions and discourages failure moves during the exploration in self-play.

The failure ratio at time t is updated when the Q-value of two time-steps ahead is given, like in SARSA (Rummery and Niranjan 1994). We define the weighted winning ratio used for tree exploration as a summation of the failure ratio and $U(s_t, a_t)$ in AlphaZero.

We also introduced a decay factor to this failure ratio. We hypothesized that although the failure ratio is effective at first, its effectiveness decreases as the learning proceeds and as enough failure cases are stored to be learned. If the agent continues learning these failure cases, the parameters will be updated and the agent goes to failure patterns. Therefore, the failure ratio should be decreased according to the progress of learning. The decay factor decreases the weight of the failure ratio by exponential order at each iteration. The predicted winning ratio in our method ($U'(s_t, a_t)$) is as follows:

$$U'(s_t, a_t) = U(s_t, a_t) + \gamma^{n_{\text{ep}}} \alpha f(s_t, a_t) \tag{3}$$

$$f(s_t, a_t) = Q(s_t, a_t) - Q(s_{t+2}, a_{t+2}), \tag{4}$$

where α is a weight that controls the effectiveness of the proposed failure ratio, γ is a hyperparameter for the decay factor, and n^{ep} denotes the current epoch index defined as starting from zero. Our proposed architecture collects training data using MCTS with $U'(s_t, a_t)$.

Experiments

We evaluate our proposed method in 6×6 and 8×8 Othello environments. Note that 6×6 denotes the board size. To check the transition of the winning ratio over iterations, we compare our method with the state-of-the-art method (Silver et al. 2017) by conducting matches in each iteration of the training phase. One iteration is defined as a cycle that consists of self-play and training, as described in Fig. 1.

Since reinforcement learning is not stable in some random seeds, we prepare 20 agents with different random seeds for each method and then conduct matches in all combinations (round-robin); hence, each setting has 400 combinations. Besides, even if we use the same combination, the matching results will be different, so we execute 50 matches for each. We therefore show the averaged winning ratio for 20,000 matches in all of the following results. The definition

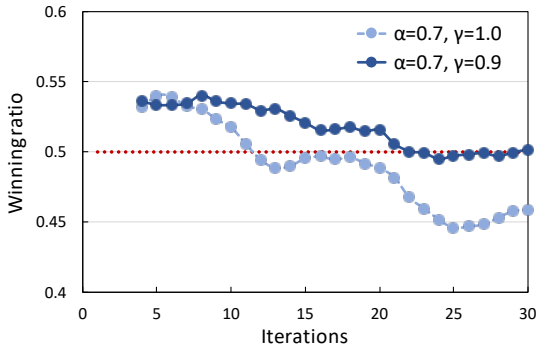

Figure 2: Winning ratio for each epoch of the proposed method against AlphaZero in the 6×6 Othello environment. We created 20 agents for each method and conducted all-play-all. The winning ratio is averaged for 20,000 matches. α is the weight for failure ratio, and γ denotes the decay ratio. Moving averages over four iterations were performed.

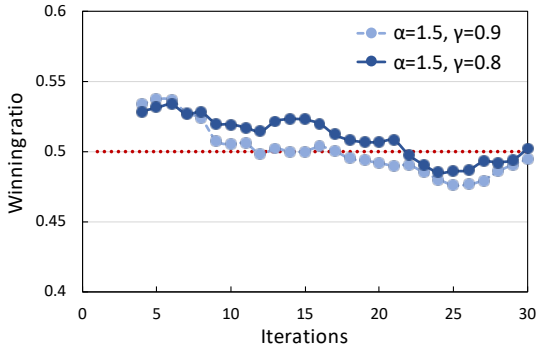

Figure 3: Winning ratio for each epoch of the proposed method against AlphaZero in the 6×6 Othello environment. We created 20 agents for each method and conducted all-play-all. The winning ratio is averaged for 20,000 matches. Moving averages over four iterations were performed.

of the winning ratio in this paper is $n_{\mathtt{win}}/n_{\mathtt{lose}}$, and we ignore the number of draws. We train 30 iterations in all experiments as we assume that our method is especially effective in the early stage of training. We try some hyperparameter settings in the experiments, and then we also confirm the stability of effectiveness of our failure ratio. In this experiment, we use NVIDIA Tesla V100; it takes about 5 hours to train one agent.

Results

Matches on 6×6 Othello We evaluate the performance of our proposed method against AlphaZero by calculating the transition of the winning ratio in 6×6 Othello environment with several hyperparameter settings. Figs. 2 and 3 show the transition results. Fig. 2 shows that our method achieved around 54% ratio in the early stages of learning. The winning ratio of our method with a prioritized ratio $\alpha = 0.7$ and a decay rate $\gamma = 0.9$ settles down to around

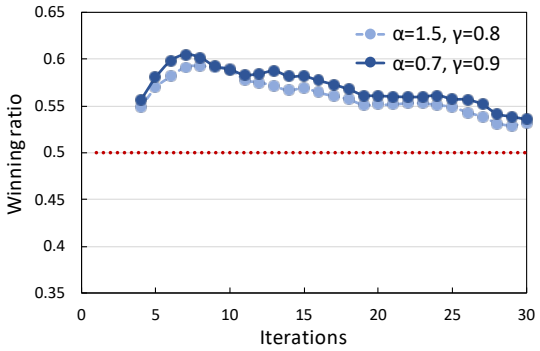

Figure 4: Winning ratio for each epoch of the proposed method against AlphaZero in 8×8 Othello environment. We created 20 agents for each method and conducted all-play-all. Winning ratio is averaged for 20,000 matches. Moving averages over four iterations were performed.

50% after the training converges, while at $\alpha = 0.7$ and without the decay factor ($\gamma = 1.0$) it is inferior to AlphaZero. The result indicates that the decay factor plays an important role in the proposed method; otherwise, the performance against AlphaZero worsens.

Fig. 3 shows the transition of the winning ratio using different hyperparameters. Both models achieve a better winning ratio than AlphaZero in the early stages of learning, but the model with $\gamma = 0.8$ gets a better result than with $\gamma = 0.9$, while $\gamma = 0.9$ was the best parameter when $\alpha = 0.7$. It indicates that the same decay ratio may not work well for a bigger weight for failure ratio α.

Matches on 8×8 Othello Fig. 4 shows the transition of the winning ratio of our proposed method against AlphaZero in 8×8 Othello environment over time. Our method demonstrates a higher winning ratio than AlphaZero in the first 30 iterations. Moreover, it shows a 61.5% winning ratio against AlphaZero at around 5^{th} iteration at $\alpha = 0.7$ and $\gamma = 0.9$, which is much higher than the opponent's. The results for both models are almost as good as in AlphaZero. This indicates that our method is stable with different hyperparameters in 8×8 Othello environment.

Discussion

Figure 2 indicates that the weight for failure ratio without the decay factor harms the performance of the agent in the latter part of the training. This is because it leads the agent to learn how to approach the states that are close to a serious mistake in the latter part of the training, which is not an optimal strategy. Introducing the decay factor helps the agent to learn efficiently only in the early phase of training and does not interrupt the learning process near convergence.

From Figs. 2 and 3, we see that the model with $\alpha = 1.5$, $\gamma = 0.9$ performs much worse than the model with $\alpha = 0.7$, $\gamma = 0.9$, although we are using the same decay factor γ. It indicates that $\gamma = 0.9$ is too big when $\alpha = 1.5$. In this sense, we understand how to tune these hyperparameters; for example, when we increase the bigger weight, it is better

to decrease the decay ratio.

We notice that the difference in performance between AlphaZero and our method gradually decreases over time for all cases in our experiments. This indicates that AlphaZero also learns various patterns, including failure cases, when it is given a sufficient amount of iterations. However, more importantly, our method takes much less time than AlphaZero to make the learning process converge.

As can be seen, the agents show much better results in the 8×8 Othello environment than in the 6×6 environment. A possible reason is that the learning policy in 8×8 is much more complicated than in 6×6, hence the more prominent effectiveness of the failure ratio. This implies that the effectiveness of the failure ratio is more significant in more difficult tasks such as Go or Shogi.

Related Work

Deep reinforcement learning in games has been attracting more and more attention recently, and various algorithms to achieve human-level performance in games have been proposed (Mnih et al. 2015; Kimura 2018; Silver et al. 2017; Wu and Tian 2017). DQN (Mnih et al. 2015) has become one of the fundamental approaches for various gameplays. It achieves high performance by combining deep neural networks and Q-learning with key components such as experience replay and the target network. DAQN (Kimura 2018) introduced auto-encoder architecture in DQN. AlphaZero (Silver et al. 2017) has achieved incredible performance using a combination of deep neural networks and MCTS. These methods are useful in discrete action spaces, so various algorithms (Lillicrap et al. 2015; Mnih et al. 2016) have also been proposed to deploy deep reinforcement learning in domains with continuous action spaces.

Monte-Carlo tree search (Browne et al. 2012) is a search algorithm that combines the precision of the tree search and the generality of random sampling. After it was first introduced, various approaches to improve its performance have been proposed (Osaki et al. 2008; Bjarnason, Fern, and Tadepalli 2009; Browne et al. 2012). Temporal difference learning with Monte Carlo (TDMC) (Osaki et al. 2008) uses a combination of temporal difference learning and a winning probability. (Bjarnason, Fern, and Tadepalli 2009) applied a combination of upper confidence trees (UCT) (Kocsis and Szepesvári 2006) and hindsight optimization (Yoon et al. 2008) to a single-player stochastic game.

Prioritized experience replay (Schaul et al. 2015) employs an idea similar to the one used in our method. It allows the agent to store experiences and learn from the training data sampled from the stored experience rather than from the online sampling. Each data is weighed by the TD error, so that "important" samples are more frequently drawn from the buffer. It is widely used in various deep RL to improve the performance. It is also expected that combining these two approaches enables agents to learn even more efficiently.

Conclusion

We introduced a failure ratio into the Monte Carlo tree search to enable a more efficient exploration in the early stages of training for faster convergence of learning. We used the difference between the current predicted winning ratio and the previous ratio to encourage the exploration of important states to beat the opponent. We evaluated our proposed method in 6×6 and 8×8 Othello environments and showed that the agents learned efficiently in the early stages of the learning iterations. We discovered that we need to decrease the degree of the failure ratio at each iteration so that it only has an effect on the early phase of training. It should also be noted that our method is more useful for complicated target tasks. It can be used in various domains with a self-play process in the learning architecture. As future work, we plan to incorporate a prioritized experience replay method in our method to make stronger agents. We also plan to test our method in various domains, especially for tasks that are more difficult than Othello game.

References

Bjarnason, R.; Fern, A.; and Tadepalli, P. 2009. Lower bounding klondike solitaire with monte-carlo planning. In *ICAPS*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; and et al. 2012. A survey of monte carlo tree search methods. *T-CIAIG*.

Kimura, D.; Chaudhury, S.; Tachibana, R.; and Dasgupta, S. 2018. Internal model from observations for reward shaping. In *ICML workshop*.

Kimura, D. 2018. Daqn: Deep auto-encoder and q-network. *arXiv:1806.00630*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; and et al. 2015. Human-level control through deep reinforcement learning. *Nature*.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; and et al. 2016. Asynchronous methods for deep reinforcement learning. *ICML*.

Osaki, Y.; Shibahara, K.; Tajima, Y.; and Kotani, Y. 2008. An othello evaluation function based on temporal difference learning using probability of winning. In *IEEE Symposium On Computational Intelligence and Games*.

Pathak, D.; Agrawal, P.; Efros, A. A.; and Darrell, T. 2017. Curiosity-driven exploration by self-supervised prediction. In *ICML*.

Rummery, G. A., and Niranjan, M. 1994. *On-line Q-learning using connectionist systems*. University of Cambridge.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized experience replay.

Silver, D.; Hubert, T.; Schrittwieser, J.; and et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. In *arxiv:1712.01815*.

Silver, D.; Huang, A.; and et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529:484–503.

Wu, Y., and Tian, Y. 2017. Training agent for first-person shooter game with actor-critic curriculum learning. In *ICLR*.

Yoon, S.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *National Conference on Artificial Intelligence*.