Learning from Failure: Introducing Failure Ratio in Reinforcement Learning

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

Abstract

Deep reinforcement learning combined with Monte-Carlo tree search (MCTS) has demonstrated high performance and thus has been attracting much attention. However, the learning convergence is quite time consuming. In comparison, learning by playing board games with human opponents is more efficient because skills and strategies can be acquired from the failure patterns. We assume that failure patterns contain much meaningful information to expedite the training process, working as prior knowledge for reinforcement learning. To utilize this prior knowledge, we propose an efficient tree search method that introduces the use of a failure ratio that has a high value for failure patterns. We tested our hypothesis by applying this method to the Othello board game. The results show that our method has a higher winning ratio than a state-of-the-art method, especially in the early stage of learning.

1 Introduction

Recently, reinforcement learning (RL) has been attracting much attention, especially for robot control and game applications [Mnih *et al.*, 2015; Silver *et al.*, 2017; Wu and Tian, 2017; Pathak *et al.*, 2017; Kimura *et al.*, 2018; Lillicrap *et al.*, 2015; Kimura, 2018; Mnih *et al.*, 2016; Silver *et al.*, 2016]. AlphaZero [Silver *et al.*, 2017] demonstrated astonishing performance by defeating professional Shogi (Japanese chess) players after only 24 hours of training, thereby reducing the tremendous amount of training data previously required. This achievement was attained by integrating into the training procedure an adversarial setting called “self-play.” The learning process of AlphaZero consists of two phases: self-play using Monte Carlo tree search (MCTS) to make training data and deep network parameters updating using the obtained data.

However, AlphaZero has a huge computational cost; it requires 5,000 TPUs for self-play and 64 GPUs for updating the network parameters. The huge cost makes it almost impossible to train the model unless such a high performance computational resource is available. Therefore, a more efficient way to train such models is needed.

When people learn and acquire skills, learning from failures is important. This helps prevent repetition of the same mistake. For example, there may be moments when a person playing a board game would like to cancel the last move and try a different move. In a practice environment, such canceling is called ‘undo.’ These situations are very likely to be important in determining the winner. Hence, we assume that learning from failure is effective and works as prior knowledge in reinforcement learning. Leveraging this prior knowledge, we made a hypothesis that weighting of exploration of moves right before the agent fails will improve the efficiency of learning. The effectiveness of such weightings is higher in the beginning of training, when the algorithm does not have much training data.

Therefore, we propose integrating a framework that encourages the exploration of critical situations that lead to failure with the state-of-the-art AlphaZero algorithm. In our method, critical situations for winning are prioritized by applying a weighting function to a tree search during training. The failure ratio is defined as a difference between the Q-value (the quality of a state-action pair) of the current move and the previous move. Its usage encourages the agents to prioritize exploration of situations that are important in determining the winner, which facilitates the learning process.

We evaluated our proposed method by using an 8×8 Othello game and compared the performance of our method with that of the state-of-the-art AlphaZero algorithm [Silver *et al.*, 2017]. Our study offers the following three contributions:

- A prior knowledge about learning from failure is formalized as “failure ratio”, the gap between the current predicted value and the value of the previous move in the MCTS architecture, to direct agents to explore in the proximity of failed actions;
- The self-play process is made more efficient by weighting critical situations for winning using the failure ratio;
- The proposed method is evaluated using the Othello game with two board sizes, and a higher winning ratio than that of the state-of-the-art method is achieved in the early stage of learning.

2 Related Work

The MCTS [Browne *et al.*, 2012] search algorithm combines the precision of a tree search with the generality of random

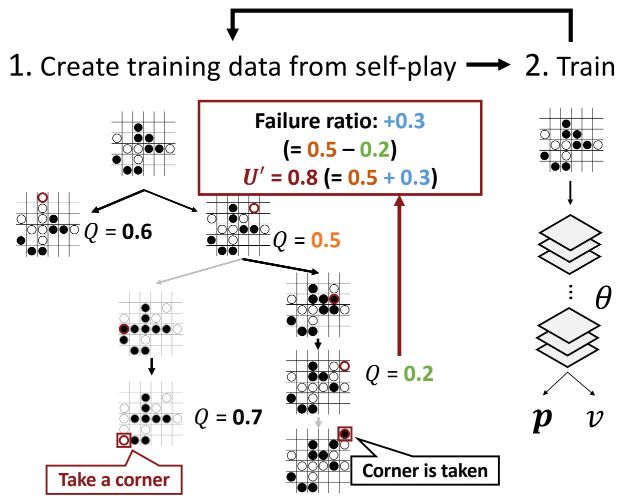

Figure 1: Overview of using proposed method in Othello game. Values denoted in the figure are examples.

sampling. Its high efficiency has been shown especially in the context of playing Go [Silver *et al.*, 2016]. Since it was first reported in 2006, various approaches for improving its performance have been proposed [Osaki *et al.*, 2008; Bjarnason *et al.*, 2009; Browne *et al.*, 2012]. Temporal difference learning with Monte Carlo simulation (TDMC) [Osaki *et al.*, 2008] uses a combination of temporal difference learning and the probability of winning in each non-terminal position, which is similar to our approach. However, our approach not only uses the temporal difference as in TDMC and in Q-learning, but also utilizes the difference between the current Q-value and the Q-value two time-steps before (the previous action taken by the agent) to further facilitate exploration over the important phases.

Prioritized experience replay [Schaul *et al.*, 2015] is based on an idea similar to that of our method. It allows an agent to store experiences and to learn from training data sampled from the stored experiences rather than from online sampling. When the data is sampled, it is weighted so that "important" samples for learning are more frequently drawn from the buffer. This method is widely used in various deep reinforcement learning setups. While prioritized experience replay does not contribute to data collecting process (as there is no prioritization in exploration), our method makes the process of collecting data itself more efficient, reducing the total amount of required data.

3 Method

The learning process of AlphaZero consists of two phases: self-play to create training data and updating of the dual policy and value networks. In this study, we propose to introduce failure ratio into this self-play architecture. The flow of parameter updates is the same as that in the existing version of AlphaZero. Figure 1 shows an overview of our proposed method. In this section, we first explain the existing AlphaZero method and then describe our proposed method to integrate a failure ratio into the self-play architecture.

3.1 AlphaZero

Self-play

In self-play, training data is collected by playing games with itself. Each move is determined by MCTS, which returns a policy $\pi(s,a)$, through a series of simulated game-plays, extending a tree from root to leaf. Each node in the tree corresponds to a different configuration of the board. Nodes hold the predicted winning ratio ($= Q(s_t,a_t)$), and the number of times the node is visited during simulation ($= N(s_t,a_t)$). The search proceeds by selecting action a_t at state s_t based on the current neural network parameters. The agent expands the tree by taking the maximum value of the predicted winning ratio with upper confidence bound ($= U(s_t,a_t)$). This value is the summation of the predicted winning ratio and upper confidence bound $b(s_t,a_t)$, which weights less-visited nodes to facilitate the search over these nodes. Formally, $U(s_t,a_t)$ and $b(s_t,a_t)$ are calculated as

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

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

At the beginning of each game during self-play, the tree is initialized, so $b(s_t,a_t)$ and $N(s_t,a_t)$ are set to zero.

Parameter updating

The policy and value functions are approximated by deep neural networks, which takes state s_t as an input, returns the probability distribution over action a ($\mathbf{p} = P(a|s_t)$), and calculates expected return $v \approx E[R_t|s_t,a_t]$. Formally, the dual neural networks are defined as

$$(\mathbf{p}, v) = f_\theta(s) \tag{3}$$

Parameters θ of the networks are updated using the training data (history of self-play moves) obtained from the self-play. The terminal state z is equivalent to the end of the game; the value of the terminal state is $+1$ for *winning*, -1 for *losing*, and 0 for *drawing*. Parameters θ are updated to minimize the error between the final outcome z and the predicted value of the state v_t, with maximizing the similarity between the search probability distribution $\boldsymbol{\pi}_t$ and the policy vector $\mathbf{p_t}$. Therefore, the loss function l is defined as follows:

$$l = (z - v(s_t))^2 - {\boldsymbol{\pi}_t}^{\mathrm{T}} \log \boldsymbol{p}_t + c||\theta||^2, \tag{4}$$

where c is a parameter weighting for L_2 regularization. The mean-squared error is used for the value function, and cross-entropy losses are used for the policy function. The training data is augmented by generating eight symmetric configurations for each time-step. Parameters are updated by stochastic gradient descent when an iteration is over. The updated network parameters are used for the search process during self-play in the next iteration.

3.2 Proposed method

In our proposed method, we introduce a "failure ratio" into the calculation of $U(s_t,a_t)$ (Eq. 1) to prioritize important situations in self-play. Our assumption is that the failure ratio represents the degree of importance of its action. Therefore, the failure ratio is calculated from the difference between the next predicted winning ratio and the current predicted winning ratio of the agent. In a two-player board game, the next predicted winning ratio of the agent is the ratio of *two* time-steps ahead since *one* step ahead represents the opponent's turn. Equation 6 defines failure ratio $f(s_t, a_t)$. $Q(s_{t+2}, a_{t+2})$ 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, action a_t is considered to be a failure because the winning ratio decreased by taking a_t. Hence, the failure ratio encourages the agent to revisit a state where it selected a failure move to explore a better move.

The failure ratio is updated when the Q-value of *two* time-steps ahead is given, like in SARSA [Rummery and Niranjan, 1994].We define the weighted predicted winning ratio as the summation of the failure ratio $f(s_t, a_t)$ and the predicted winning ratio with upper confidence bound $U(s_t, a_t)$. In addition, we introduce a decay factor into this failure ratio. We hypothesize that although using the failure ratio is initially effective, its effectiveness decreases as learning proceeds and a sufficient number of failure cases have been attained. Hence, if the agent keeps focusing on learning failure cases, the parameters will be updated so that the agent comes close to failure patterns, which might not be optimal. Therefore, we decrease failure ratio in accordance with the progress in learning. The decay ratio reduces the weight of the failure ratio by an exponential order at each iteration. Ultimately, the predicted winning ratio ($U'(s_t, a_t)$) is

$$U'(s_t, a_t) = U(s_t, a_t) + \gamma^{n_{\text{ep}}} \alpha f(s_t, a_t) \tag{5}$$

$$f(s_t, a_t) = Q(s_t, a_t) - Q(s_{t+2}, a_{t+2}), \tag{6}$$

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

4 Evaluation

We evaluated our proposed method using 6×6 and 8×8 Othello environments. 6×6 and 8×8 mean the board size. We compared the performance of our method with that of AlphaZero [Silver *et al.*, 2017] by conducting matches at the end of each training iteration. One iteration is defined as a cycle consisting of self-play and training, as described in Fig. 1.

The Elo rating system is a method for calculating relative skill levels of players in board games. The expected score of *Player A* against *Player B* is calculated using

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}, \tag{7}$$

where R_A is the rating of *Player A*, and R_B is the rating of *Player B*. If *Player A* was expected to score E_A but actually scored S_A points, *Player A*'s rating is updated using

$$R'_A = R_A + K(S_A - E_A), \tag{8}$$

where K, the *K-factor*, is the maximum possible adjustment per game. In this experiment, we set $K = 32$ for all matches.

4.1 Network architecture

The policy network and the value network share most of their parameters. The shared part consists of four convolutional layers, followed by two fully-connected layers. The kernel size is 3, the filter size is 512, the stride is 1, and the padding is 1 for all layers. The output of the shared layers is connected to two separate networks, the policy network and the value network. Each layer is batch-normalized, except for the final layer of each network. The final layer of the policy network is activated by a softmax function while the value network is activated by a hyperbolic tangent function. All the other layers are activated by the ReLU function.

4.2 Experimental setting

We prepared 20 agents with different random seeds for each method and conducted matches for all combinations in a round-robin manner, meaning 400 combinations in total. Moreover, we executed multiple matches for each combination and calculated the averaged winning ratio for these matches. For the 6×6 environment, we executed 50 matches for each combination ($= 20,000$ matches in total for each iteration) and for 8×8, we executed 20 matches for each combination ($= 8,000$ matches for each iteration). The winning ratio was defined as $n_{\texttt{win}}/n_{\texttt{lose}}$; we ignore the number of draws. Multiple hyperparameter settings, such as the weight of the failure ratio and the decay factor were tested to determine stability for the effectiveness of using failure ratio over time. We used the NVIDIA Tesla V100 GPU. It took approximately 5 hours to train one agent in both environments.

4.3 Results

Matches using 6×6 Othello environment

We evaluated the performance of our proposed method against that of AlphaZero [Silver *et al.*, 2017] by calculating the transition in the winning ratio in the 6×6 Othello environment. Figures 2 shows the transition results with failure ratio weight α set to 0.7 and decay ratio γ) set to 0.9, our method achieved an average ratio of around 54% in the early stage of learning. In fact, the proposed method with decay factor set to 0.9 maintained a higher winning ratio over 20 iterations. Its performance settled down to around 50% once the training converged. The decay factor plays an important role; without a decay factor ($\gamma = 1.0$), our method could only keep winning over ten iterations, and starts losing afterward.

Matches using 8×8 Othello environment

Figure 3 shows the transition in the winning ratio of our proposed method and Elo rating with $\alpha = 1.5$ and $\gamma = 0.8$ against AlphaZero. Our method achieved a higher winning ratio over the first 50 iterations. Moreover, it achieved a 66.4% winning ratio at around the 7^{th} iteration, which is much higher than the opponent's ratio. We also tried different hyperparameters such as $\alpha = 1.5$ and $\gamma = 0.8$, and the results were nearly the same. This indicates that our method performs stably well with different hyperparameters. As for Elo rating, the maximum difference between the proposed

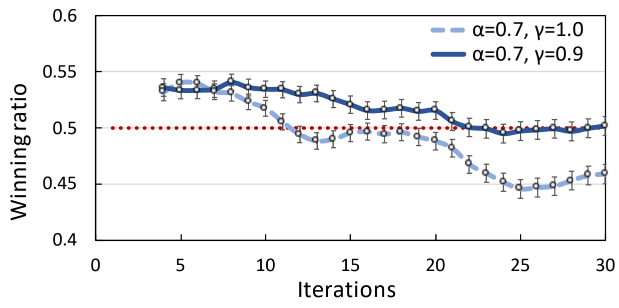

Figure 2: Winning ratio for proposed method against AlphaZero in each epoch using 6×6 Othello environment with failure ratio weight (α) set to 0.7 and decay ratio (γ) set to 1.0 or 0.9. We created 20 agents for each method and conducted all-play-all. We conducted 50 matches for each combination, so the result is the average for 20,000 matches. Error bars indicate standard error for all matches. Each point is moving average for four iterations.

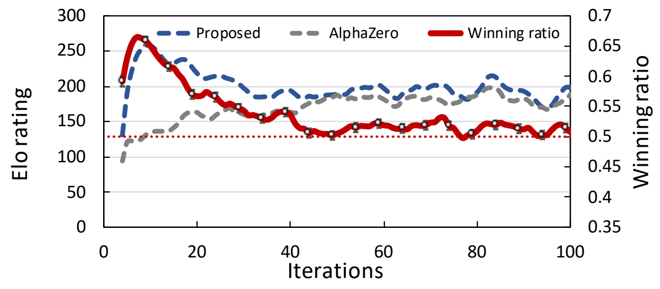

Figure 3: (Left axis, dashed lines) Elo rating for each method. (Right axis, red solid line) Winning ratio for proposed method against AlphaZero in each iteration. The failure ratio weight (α) was set to 1.5 and decay ratio (γ) was set to 0.8. We created 20 agents for each method and conducted all-play-all. We conducted 20 matches for each combination and updated the Elo rating in accordance with the results of all matches in each iteration. The winning ratio is the average for $8,000$ matches. Error bars indicate standard error for all matches. Each point is moving average for four iterations.

method and AlphaZero is 124.30 for $\alpha = 1.5, \gamma = 0.8$. The result shows that the proposed method maintained a higher Elo rating over the whole period. Our Elo ratings may look much smaller than those in the original AlphaZero paper [Silver *et al.*, 2017], but this is because our Elo ratings were calculated purely from the proposed method and AlphaZero, and the ratings for AlphaZero are not fixed to a constant value.

4.4 Discussion

Decay factor

As shown in the experiment, weighting without a decay factor degrades the performance in the latter part of training. It indicates that the ratio rather leads the agent to learn how to approach states that are near a serious mistake, which is not the optimal strategy. Introducing the decay factor helps the agent learn efficiently only in the early phase of training and does not interrupt the learning process near convergence.

Convergence of exploration

The gradual decrease in the difference in performance between AlphaZero and our method over time indicates that AlphaZero also learns various patterns, including failure cases, when it has performed a sufficient number of iterations. However, most importantly, our method takes much less time than AlphaZero to make the learning process converge.

Board size

The much better results when using the 8×8 Othello environment could be because the learning policy for 8×8 is much more complicated than for 6×6, which increases the effectiveness of using the failure ratio. This indicates that the effectiveness of using the failure ratio is more significant for more difficult tasks, such as Go and Shogi.

Introducing a lower bound and success ratio

We conducted a preliminary experiment in which we introduced a lower bound in the calculation of the failure ratio and replaced the failure ratio with the success ratio (= opposite of the failure ratio) and tested two methods. We found that only the proposed method (Eq. (6)) worked well.

Failure ratio with a lower bound:

$$f(s_t, a_t) = \begin{cases} Q_t - Q_{t+2} & (f(s_t, a_t) \geq 0) \\ 0 & (\text{otherwise}) \end{cases} \tag{9}$$

Success ratio:

$$f(s_t, a_t) = -(Q_t - Q_{t+2}) = Q_{t+2} - Q_t \tag{10}$$

These two methods could not constantly achieve a higher winning ratio against AlphaZero. With Equation (9), exploration is prioritized only when the state-action pair at time t is a failure. Our method not only prioritizes the exploration of failed moves but also discourages the exploration of states where the agent performed well. The fact that Eq. (9) did not work indicates that we should also weight the successful states. With Equations (10), exploration is prioritized around successful states. While it may sound reasonable to focus on learning successful behaviors, it worked poorly. It indicates that encouraging the exploration of successful states leads the agent to learn similar successful patterns, which discourages agents from finding other better strategies.

5 Conclusion

We introduced a failure ratio in MCTS 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 states that will support beating the opponent. The experiments using Othello environments showed that the agents learned efficiently in the early stages of learning. We discovered that we need to reduce the degree of the failure ratio at each iteration so that it has an effect only on the early phase of training. Our method is more useful for complicated target tasks and can be combined with any RL algorithms that involve a self-play process. As future work, we plan to incorporate a prioritized experience replay method into our method to create stronger agents.

References

[Bjarnason *et al.*, 2009] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding klondike solitaire with monte-carlo planning. In *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'09, pages 26–33. AAAI Press, 2009.

[Browne *et al.*, 2012] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[Kimura *et al.*, 2018] Daiki Kimura, Subhajit Chaudhury, Ryuki Tachibana, and Sakyasingha Dasgupta. Internal model from observations for reward shaping. 2018.

[Kimura, 2018] Daiki Kimura. Daqn: Deep auto-encoder and q-network. *arXiv preprint arXiv:1806.00630*, 2018.

[Lillicrap *et al.*, 2015] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[Mnih *et al.*, 2016] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *https://arxiv.org/abs/1602.01783*, 2016.

[Osaki *et al.*, 2008] Y. Osaki, K. Shibahara, Y. Tajima, and Y. Kotani. An othello evaluation function based on temporal difference learning using probability of winning. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 205–211, Dec 2008.

[Pathak *et al.*, 2017] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

[Rummery and Niranjan, 1994] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

[Schaul *et al.*, 2015] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. 2015.

[Silver *et al.*, 2016] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[Silver *et al.*, 2017] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 2017.

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