Punisher: A Deep Reinforcement Learning Model Trained by Correcting Bad Actions

Jianyi Yang

Computer Science Department

International Technological University

Santa Clara, California, CA 95050, USA

yangjiany3465@students.itu.edu

*Abstract*—Deep Reinforcement Learning (DRL) is a deep learning (DL) network model that uses environmental feedback to train and make decisions. Expected value, as a powerful mathematical tool, is widely used in DRL network training. However, there are often deviations between the expected values and the actual values obtained from the environment. Additionally, accumulative deviations are also present in DRL networks. The accumulation of these deviations can result in a slower training speed and negatively impact the network's stability. To address these issues, this paper proposes a new DRL training method called Punisher. The principle behind Punisher is to identify the bad actions made by the DRL agent and correct only those actions during the training process. By focusing on correcting the bad actions, Punisher aims to improve the overall performance and stability of the DRL network. The experimental results demonstrate that the Punisher method exhibits excellent performance, faster training speeds and greater network stability, making it a promising approach for efficiently training DRL agents in various applications. Experiments presented in the main content of this paper have been posted at GitHub at https://github.com/Jimmyoungyi/Punisher.

Keywords-deep learning; punisher; reinforcement learning

# Introduction

Deep Reinforcement Learning (DRL) is an unsupervised model that performs autonomous learning through environmental feedback. This model type is currently used in many fields, such as gaming, chatbots, and robotics [1,2]. During the training process of the DRL network, it is common to calculate the expected value of each action and use the expected value to train the network [3,4]. However, there are deviations between the expected value and the actual value. Take the Deep Q-network (DQN) model as an example. DQN is one of the representatives of DRL models that train networks using expected values [5,6]. Since the DQN model uses the expected value to train the policy network, any deviation in the expected value will be inherited by the policy network. Current technology used in DQN also introduces deviations in predicting accurate numbers. The accumulation of these deviations can result in DQN not only having a slower training speed but also having relatively low predicting accuracy after training is complete, thereby affecting the stability of DQN.

This paper proposes a new DRL training method called Punisher. In the comparative experiments of this paper, Punisher used the same number and size of networks, the same loss function, and the same optimizer as DQN. The experimental results show that Punisher has obvious advantages regarding the average time-consuming per action, the number of episodes required to complete training, and the stability after training. Specifically, Punisher significantly reduces the average time-consuming per action and completes training with a smaller number of episodes. Additionally, Punisher demonstrates greater stability after training, resulting in improved performance compared to DQN.

The design of the Punisher architecture is based on two key features of deep learning (DL) and reinforcement learning (RL) [7]. Firstly, DL networks are trained according to calculating the gradient of the loss value. In ensuring the general gradient of the loss is correct, the larger the loss value obtained, the faster the training speed will be. Secondly, admitting that deviations are inevitable in DL network calculations and most RL environment models. However, it is possible to use a method that makes those deviations occur in an area that has less impact on the result. This method minimizes the impact of those deviations, thereby improving the overall stability of the DRL.

Punisher incorporates two innovative features:

Firstly, by adjusting the distribution of the value range of the state-critic network, the deviations generated by the state-critic network appear more in the range that has less impact on the results and less in the range that has a greater impact on the results. This ensures that the state-critic network provides a more accurate training data set for the policy network and improves the overall stability of Punisher.

Secondly, by adding an appropriate amount of random action, the deviations created by the two networks can be hedged. This innovation prevents the loss gradient from seriously deviating from the correct gradient after policy network training is completed and further increases the stability of Punisher.

# Method

## Preliminary Knowledge

The following is the basic structure of a DRL: First, there is an environment state. Based on this environment state, DRL uses a policy network to determine an optimal action. This action changes the environment to a new state, which has the identical structure as the old state. The environment module also returns this action's reward. This reward is used to calculate the loss value, which is then used to train the policy network. By looping this behavior, the policy network can determine actions more and more accurately.

The calculation of expected value is a commonly used method of calculating loss value from an action's reward. Take the DQN model as an example. In DQN, every time an action occurs, the corresponding state, action, new state, and reward are stored in memory. Each action activates a policy network training process. During each training process, a certain number of random samples are selected from the memory. Then, the maximum expected value of these samples is calculated through a target network, which has an identical structure to the policy network. Then, use this expected value to calculate the loss value and train the policy network. After the policy network is trained, the policy network's parameters and the target network's parameters are mixed in a certain proportion to become the target network's new parameters. It can be seen from this process that the DRL policy network completely inherits the deviation in the expected value [8,9].

## Overview of Punisher

Based on the previous feature analysis of DL and RL, this project proposes the Punisher architecture. Like DQN, Punisher consists of two networks. The first network is a state-critic network, which is used to evaluate the score of each environment state. By comparing the state's score with the new state's score, bad actions can be identified. The second network is a policy network, which is used to determine the action of each state. Theoretically, correcting bad actions will produce the maximum loss value with the correct loss gradient direction. This loss is then used to train the policy network.

## The First Netwrok of Punisher

For the environment used in this paper, the states of this environment are continuous values, but its rewards are discrete values, 0 and 1. In order to identify bad actions, it is necessary to redefine the rewards from discrete values to continuous values. The first network is used here: the state-critic network. The input of the state-critic network is a state, and the output is the score of this state. Punisher stores the last 11 states of each game. If the game is truncated, start a new game, and clear the state memory. If the game is terminated, the index of the last state is 0, the index of the second last is 1, and the index of the eleventh last state is 10. The score of each state is calculated as the square of its index. Then, these data are used to train the state-critic network. This results in a state-critic network that can evaluate the score of each state. In this network, the closer the state is to failure, the more sensitive the score is. In this way, the deviations calculated by the state-critic network appear more at the range farther from the failure and less at the range closer to the failure. This method will be discussed in detail in the subsequent ablation study.

With the continuous score based on the continuous state, the reward of each action can be redefined by calculating the difference between the new state's score and the state's score. When the reward is negative, the action is identified as a bad action.

Like DQN, Punisher also uses a policy network to determine the action of each state and sets up a memory to store the state and action for training. The difference is that Punisher stores the corresponding action and state in memory and activates the policy network training process only when a bad action occurs. Not every action will be saved and activate the training process. In addition, during the training process, it will be judged whether the previous bad actions have been corrected by the current policy network. If the action is corrected, the memory will be skipped, and the loss value will not be calculated. If the action is not corrected, the memory will be used again to calculate the loss value. In this way, Punisher increases the operation speed, guarantees the maximum loss value, and maintains the accuracy of the direction of loss decline.

Under the inference stage of this model, compared with DQN, Punisher greatly reduces the operation time and the number of episodes required to complete the training. However, the test also exposed a problem, which is also one of the goals of this paper: How to prevent severe decay in a trained Punisher network? In other words, how to improve the stability of the trained Punisher?

## The Second Netwrok of Punisher

As mentioned earlier, a common issue in DRL is the occurrence of deviations when the policy network determines actions. Additionally, for the environment used in this paper., this environment has a certain tolerance for bad actions. As long as there are no consecutive bad actions or no bad actions at key positions, the game can be recovered and continue. The solution to this problem is adding random actions. After testing, the optimal frequency is every 20 actions, with the first 19 actions determined by the policy network and the last action selected randomly. This method significantly enhances Punisher's stability compared to Model 1 and even outperforms DQN in terms of stability. This method will be discussed in detail in the subsequent ablation study.

# Result

## Environment

The environment of this paper uses the classic Cartpole game. In this game, “A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart, and the goal is to balance the pole by applying forces in the left and right direction on the cart.” The state consists of four values: cart position, cart velocity, pole angle, and pole angular velocity. The action consists of two values: 0 and 1 represent left and right forces respectively [10].

The rewards have two values. The game is terminated when the cart's position exceeds a certain range, or the pole's angle exceeds a certain range. The game is truncated when the score reaches 500. As the game progresses, each action gets a reward of 1 and a score of 1. When the game is terminated, the last action gets a reward of 0 and a score of 0.

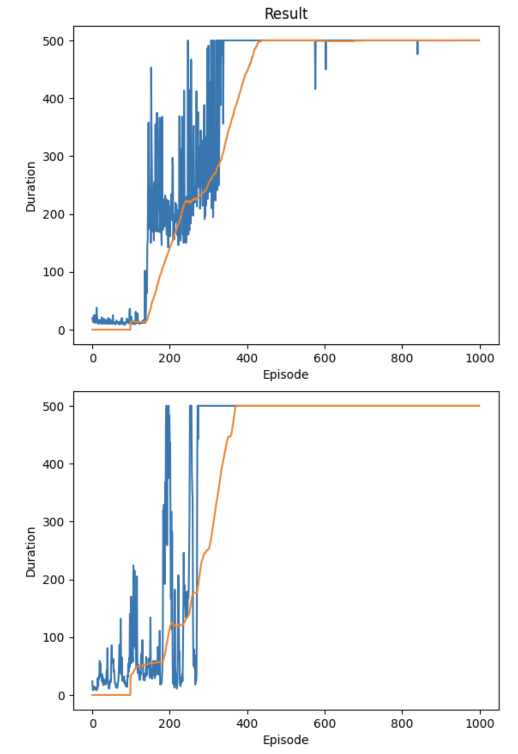
## Comparision Between DQN and Punisher

In the comparative experiment between DQN and Punisher, the DQN model was selected from the Pyrotch official website. For Punisher's model, this work used the same number and size of networks, and the same loss function and optimizer as DQN. In the experiment, each model was run three times; each time, 1000 episodes (games) were played. This work took the average of the three records. This work recorded and calculated the average time spent on each action, the number of times to reach the highest score (500) within 1000 episodes, the first episode to reach 500, the longest duration maintaining 500, and the average score per 100 episodes from 300-1000 episodes. The experimental results are demonstrated in Table I.

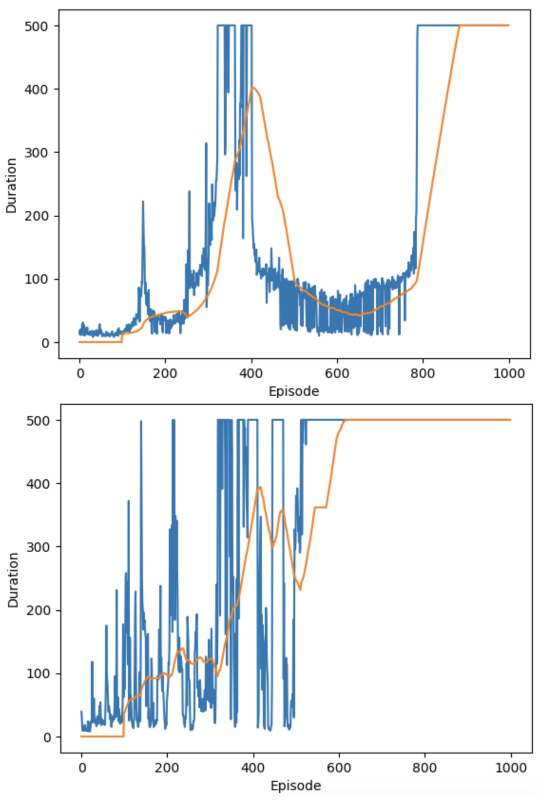
1. COMPARISON RESULT BETWEEN DQN AND PUNISHER

|  |  |  |  |
| --- | --- | --- | --- |
|  | DQN | Punisher | Comparison |
| Time/Action(1/1000s) | 2.839 | 2.304 | -0.188 |
| First 500 | 333.667 | 199.333 | -0.403 |
| Total 500 | 439.667 | 698.333 | 0.588 |
| Longest Duration | 201.667 | 529.667 | 1.626 |
| 301-400 ave | 310.800 | 448.193 | 0.442 |
| 401-500 ave | 304.933 | 416.137 | 0.365 |
| 501-600 ave | 347.773 | 493.163 | 0.418 |
| 601-700 ave | 343.653 | 490.310 | 0.427 |
| 701-800 ave | 305.650 | 500.000 | 0.636 |
| 801-900 ave | 454.770 | 500.000 | 0.099 |
| 901-1000 ave | 500.000 | 500.000 | 0.000 |

For a more intuitive comparison, Figure 1 and Figure 2 respectively demonstrate the best and worst experimental results of DQN and Punisher:



1. Best experimental results of DQN (up) and Punisher (down) (Figure credit: Original)



1. Worst experimental results of DQN (up) and Punisher (down) (Figure credit: Original)

The following conclusions can be drawn from the above comparison chart and experimental data: First, Punisher runs faster than DQN. As can be seen from the comparison, Punisher spends 18.8% less time per action on average than DQN. This is because, although the single training process calculation in Punisher is larger than DQN, Punisher does not train in every action. Second, Punisher requires fewer episodes to complete training than DQN. As can be seen from the comparison, in 1000 episodes, the total number of times Punisher reaches 500 is 58.8% more than DQN; it reaches 500 for the first time 40.3% faster than DQN. Third, Punisher is more stable than DQN. As can be seen from the comparison, Punisher's maximum duration of maintaining 500 is 162.6% longer than DQN. In summary, Punisher is superior to DQN overall. By comparing the average scores for each 100 episodes between 300 and 1000, it could be observed that Punisher's score in every section is greater than or equal to DQN's average score.

## Ablation Study of the Distribution and Size of the State-Critic Network Value Ranger

In this ablation study, two different comparative experiments were performed. As in the previous experiment, each model ran three times; each time, 1000 episodes were played; the average of the three records was taken. This work recorded the first episode reaching 500, the number of times reaching 500 in 1000 episodes, the longest duration maintaining 500, and the average score per 100 episodes from 600-1000 episodes.

In the first comparison experiment, the state-critic network was given different scores: i, ixi, and ixixi, as demonstrated in Table II.

1. PERFORMANCE OF DIFFERENT FORMS OF SCORES I

|  |  |  |  |
| --- | --- | --- | --- |
|  | i | i x i | i x i x i |
| First 500 | 125.667 | 199.333 | 396.333 |
| Total 500 | 500.333 | 698.333 | 317.000 |
| Longest Duration | 231.333 | 529.667 | 275.667 |
| 601-700 ave | 353.330 | 490.310 | 372.833 |
| 701-800 ave | 362.753 | 500.000 | 433.533 |
| 801-900 ave | 359.193 | 500.000 | 397.217 |
| 901-1000 ave | 325.067 | 500.000 | 380.910 |

The following conclusions can be drawn from the above experimental data. First, in i, i x i, and i x i x i, the number of episodes required to reach 500 for the first time increases sequentially. It can be inferred that as the number of squares of i increases, the training speed becomes slower and slower. Second, the overall performance of i x i x i is poor. The reason is that the model did not complete training within the 1000 episodes range in one of the three experiments, resulting in a significant drop in various values. Finally, i x i performs best in terms of balance between stability and training speed, as it is generally higher than the other two comparison sets in terms of various comparisons.

From the first set of experiments, it could be concluded that ixi performs better than i. The second set of experiments is a supplement to the first one.

In the second comparison experiment, the state-critic network's score was set to be i, ix10, and ixi. Since the i range is 0~10, ix10 equals expanding the value range of i proportionally to the value range of ixi. This experiment could compare the effects of changing the value range and the value distribution. The experimental results are shown in Table III.

1. PERFORMANCE OF DIFFERENT FORMS OF SCORES II

|  |  |  |  |
| --- | --- | --- | --- |
|  | i | i x i | i x i x i |
| First 500 | 125.667 | 147.667 | 199.333 |
| Total 500 | 500.333 | 672.000 | 698.333 |
| Longest Duration | 231.333 | 369.667 | 529.667 |
| 601-700 ave | 353.330 | 466.527 | 490.310 |
| 701-800 ave | 362.753 | 499.417 | 500.000 |
| 801-900 ave | 359.193 | 500.000 | 500.000 |
| 901-1000 ave | 325.067 | 500.000 | 500.000 |

The following conclusions can be drawn from the above experimental data. First, by comparing the data in column i and column ix10, it can be observed that expanding the value range reduces the training speed and improves the stability of Punisher. Second, by comparing the data in column ix10 and column ixi, it can be seen that with the same value range, adjusting the value distribution can further enhance Punisher's stability, but also results in a reduction in training speed. In conclusion, both expanding the value range and adjusting the distribution of values can reduce the training speed and enhance stability. Both methods can be used simultaneously to produce an overlay effect. Among this set of experiments, ixi still performs the best in terms of balancing stability and training speed.

It could be discussed that, firstly, although deviations in DRL calculations are inevitable, these deviations can be adjusted so that they occur within an acceptable range more frequently, thereby improving the overall effectiveness of DRL. Taking this paper as an example, changing the score from i to ixi geometrically changes a straight line into a curve. Various equations can be used to turn a straight line into a curve. These equations are of reference and trial value when designing DRL models in different environments.

Secondly, the excellent performance of using ixi in this paper is due to this specific environment. It can be seen from the previous description of the environment that the closer the state is to failure, the higher the accuracy requirements for the environment score. Therefore, there is no guarantee that ixi will perform well in all environments.

Finally, the state memory in this paper stores the last 11 states before failure. The number 11 was chosen because when the action is fixed, the game's score is 8~14, and the average score is 11. The 11 here is a hyperparameter. It can even be said that the entire state-critic network is a hyperparameter that needs to be adjusted differently according to different environments. Furthermore, the entire state-critic network can be omitted when the environment module can accurately identify bad actions.

## Ablation Study of Adding Random Actions

In this adding random actions comparative experiment, like the previous experiment, each model was run three times; each time, 1000 episodes were played; the average of the three records was taken. This work recorded the first episode reaching 500, the longest duration maintaining 500, and the average score per 100 episodes from 500-1000 episodes. The effects of no random actions, 1 random action per 20 actions, and 1 random action per 10 actions were compared. The experimental results are shown in Table IV.

1. PERFORMANCE ON DIFFERENT LEVELS OF RANDOM ACTIONS

|  |  |  |  |
| --- | --- | --- | --- |
|  | no random action | 1/20 random action | 1/10 random action |
| First 500 | 172.667 | 199.333 | 246.667 |
| Longest Duration | 359.333 | 529.667 | 325.333 |
| 501-600 ave | 437.110 | 493.163 | 403.213 |
| 601-700 ave | 432.513 | 490.310 | 500.000 |
| 701-800 ave | 437.483 | 500.000 | 494.207 |
| 801-900 ave | 457.507 | 500.000 | 469.003 |
| 901-1000 ave | 466.363 | 500.000 | 499.330 |

The following conclusions can be drawn from the above experimental data. First, it can be seen from the experimental results that the more frequently the random action is added, the more episodes are needed to reach 500 for the first time. It can be deduced that adding random actions slows down the training speed. Second, regarding stability, a model that adds one random action per 20 actions has the maximum duration to maintain 500. Both no random actions and too many random actions will reduce the stability of the Punisher.

Additionally, it can be conjectured that the decrease in stability may be related to the occurrence of deviations in the state-critical network. When the Punisher network training is complete and the game score reaches 500, states and new states far from the failure range will appear more frequently. As mentioned before, deviations from the state-critical network occur more often farther from the failure range, so correct actions are often erroneously stored in the bad action memory. These deviation data will make the direction of loss decline become increasingly stochastic. After adding random actions, bad actions will be generated in any range, including the range closer to failure. These data increase the possibility that the direction of loss decline is in the correct direction, hedging against the deviation data in the memory, thereby reducing stochasticity in the direction of loss decline. So, adding random actions will make Punisher more stable.

# Discussion

The first discussion is about training speed. A surprising phenomenon is that Punisher requires significantly fewer training samples than DQN but can complete training faster. Here is the author's speculation. DQN uses the expected value and all samples to train the module. When the action determined by the policy network is correct, there is no guarantee that the expected value points in the more correct gradient. Even if it is in the more correct gradient, the value of this loss is much smaller than the loss produced by correcting bad actions. On the other hand, all Punisher training samples are bad actions. Therefore, the gradient direction must be towards the correct direction, and the loss value obtained from Punisher must be much larger than that obtained from DQN. As a result, although the number of samples and training frequency used in Punisher training are smaller than those of DQN, the training speed is faster than that of DQN.

The second discussion is about stability. Stability and network degradation is a common problem in DRL. All network training depends on the correctness of the loss gradient. Here is the author's speculation. As the action determined by the policy network becomes more and more accurate, it is difficult to avoid that the gradient of loss becomes more and more stochastic. By learning this stochastic loss, the network is prone to degradation. Punisher also showed the same phenomenon. When the action determination becomes more and more accurate, the direction of loss decline becomes more and more stochastic. This phenomenon is ultimately counteracted by adding random actions. Adding random actions increases the number of real bad actions stored in the memory, thereby achieving an increase in stability, which is even better than the stability of DQN.

# Conclusion

This paper introduces a new DRL structure, Punisher, which directly discovers bad actions from state feedback and then trains only by correcting bad actions. This approach significantly accelerates the learning speed compared to DQN, which uses the expected value to train the policy network. Although this method brings stability issues, Punisher is successfully stabilized by addressing two key aspects: 1. adjusting the value range of the state-criticism network and 2. adding random actions. Through comparative experimental data, Punisher outperforms DQN in terms of the average training time per action, the number of episodes required to complete training, and module stability. Punisher is still in its relatively early stages, and there is ample room for further exploration. Experimental and research in this field will continue in the future.

##### References

1. N. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang,Y. C. Liang, and D. Kim. Applications of deep reinforcement learning in communications and networking: A survey. IEEE Communications Surveys & Tutorials, vol. 21(4), pp. 3133-3174, 2019.
2. A. Mosavi, Y. Faghan, P. Ghamisi, P. Duan., S. Ardabili, E. Salwana, and S. Band. Comprehensive review of deep reinforcement learning methods and applications in economics. Mathematics, vol. 8(10) , pp. 1640, 2020.
3. K. Arulkumaran, M.P. Deisenroth, M. Brundage, A.A.Bharath, Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine, vol. 34(6), pp. 26-38, 2017.
4. V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, An introduction to deep reinforcement learning. Foundations and Trends in Machine Learning, vol. 11(3-4) , pp. 219-354, 2018.
5. M. Roderick, J. MacGlashan, and S. Tellex, Implementing the deep q-network. arXiv preprint arXiv:1711.07478, 2017.
6. J. Fan, Z. Wang, Y. Xie, and Z. Yang, A theoretical analysis of deep Q-learning. In Learning for dynamics and control, pp. 486-489, 2020.
7. Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. nature, vol. 521(7553), pp.436-444 2015.
8. M. Li, A. Li, Y. Huang, and S. Chu. Implementation of deep reinforcement learning. In Proceedings of the 2nd International Conference on Information Science and Systems, pp. 232-236, 2019.
9. H. Li, R. Cai, N. Liu, X. Lin, and Y. Wang. Deep reinforcement learning: Algorithm, applications, and ultra-low-power implementation. Nano Communication Networks, vol. 16, pp. 81-90, 2018.
10. Reinforcement Learning (DQN) Tutorial, URL: https://pytorch.org/tutorials/intermediate/reinforcement\_q\_learning.html. Last accessed 2023/09/03