Double Prioritized State Recycled Experience Replay

Fanchen Bu, Dong Eui Chang
Korea Advanced Institute of Science and Technology
boqvezen97@kaist.ac.kr, dechang@kaist.ac.kr

Abstract

Experience replay enables online reinforcement learning agents to store and reuse the previous experiences of interacting with the environment. In the original method, the experiences are sampled and replayed uniformly at random. A prior work called prioritized experience replay was developed where experiences are prioritized, so as to replay experiences seeming to be more important more frequently. In this paper, we develop a method called double-prioritized state-recycled (DPSR) experience replay, prioritizing the experiences in both training stage and storing stage, as well as replacing the experiences in the memory with state recycling to make the best of experiences that seem to have low priorities temporarily. We used this method in Deep Q-Networks (DQN), and achieved a state-of-the-art result, outperforming the original method and prioritized experience replay on many Atari games.

Keywords: Deep reinforcement learning, Experience replay

1. Introduction

In online reinforcement learning, agents learn to change the parameters of the policy while interacting with the environment at the same time. Without remembering the previous experiences, agents are only able to update the parameters immediately after each single step, which may affect the efficiency of training process as some experiences can be rare but significant.

To tackle this issue, experience replay was introduced where the experiences are stored in memory and utilized more methodically. The prominent effect of experience play was proved by its application in Deep Q-Networks (DQN) [2] [3], on account of its capability to break the temporal correlations of the sequential experiences and palliate the non-stationary distribution problem. Generally, with experience replay, we can downsize the amount of the experiences required for the training process, and therefore reduce the main computational cost in most cases of Reinforcement Learning.

In the original version of experience replay algorithm, a uniform sampling strategy is used, which can hardly harmonize with the different significance of experiences, and therefore lose some efficiency of learning. Then, prioritized experience replay was developed to address this issue by directly and simply prioritizing the experiences with higher temporal difference (TD) errors when sampling experiences for training.

In this paper, we introduce double-prioritized state-recycled (DPSR) experience replay prioritizing the experiences by some standard both in sampling and replacing, as well as executing state recycling, which makes use of some old and likely useless experiences. Our key idea is to keep the experiences that are more useful in the replay buffer for a longer time and make them tend to be sampled more easily and frequently. By keeping a high-quality replay buffer, a Reinforcement Learning agent can waste less time and learn more effectively.

Specifically, the main contributions of our work are listed as follows:

1) We extended the previous prioritized experience replay and developed a novel experience replay algorithm, double-prioritized experience replay, where the experiences in replay buffer are prioritized in both sampling stage and replacing stage for training the agent.

2) We developed state recycling, a special technique to reuse and update the experiences based on old ones, and integrate it with the double-prioritized experience replay algorithm, eventually forming the double-prioritized state-recycled (DPSR) experience replay.

3) We applied and tested our DPSR experience replay on Atari games with Deep Q-Networks (DQN). We compared the performance of our method with both the original experience replay and prioritized experience replay. In most Atari games tested, DPSR experience replay outperforms both baseline methods and achieves state-of-the-art results.

2. Background

2.1 Problem Statement

Consider non-discount reinforcement learning (RL), which can be represented by a quadruple \((S, A, P, R)\), where \(S\) is the set of states, \(A\) is the set of actions, \(P : S \times A \rightarrow S\) is the state transition function, \(R : S \times A \times S \rightarrow \mathbb{R}\) is the reward function. At each timestep, the RL agent takes action \(a \in A\) in current state \(s \in S\) and observes the next state \(s' \in S\) with instant reward \(r \in \mathbb{R}\), which forms a quadruple \((s, a, r, s')\) called a transition, or an experience. Usually, the objective of RL is to make the agent learn a policy \(\pi : S \rightarrow A\) that maximizes the cumulative reward.
$R_c = E[\sum_t r_t]$ when the agent follows it to choose the actions.

When using experience replay, at each timestep, the RL agent interacts with the environment and generates an experience $T_{new}$ which would be stored into the replay buffer $B$. When $B$ is full, some old experiences already in $B$ will be replaced (or recycled, in our method). Then at a certain frequency of training, the policy (parameters) of RL agent is updated by a batch of experiences $\{T_{i}\}$ sampled from $B$. The sub-problem we focus on is to learn the sampling mapping $\phi : B \rightarrow \{T_{i}\}$ and the replacing (and recycling) mapping $\tau : B \times T_{new} \rightarrow B'$, such that the cumulative reward $R_c$ is maximized when the agent samples and replaces experiences based on them.

![Figure 1: Problem overview](image)

### 3.1 Prioritized Sampling

In sampling, like prioritized experience replay, we compute the priorities of experiences based on TD errors and apply proportional prioritization. $T_{i}$'s priority $p_{i} = |\delta_{i}| + \epsilon$, where $\delta_{i}$ is the TD error of $T_{i}$, and $\epsilon$ is a small positive number used to make up the experiences with near-zero TD errors, then its probability of being chosen $PS_{i} = \frac{p_{i}^{\alpha}}{\sum_{j}p_{j}^{\alpha}}$, where $\alpha(\cdot)$ is the importance-sampling factor that can change over time, indicating how much the probabilities are affected by the priorities.

After the sampling of a batch is completed, we train the Q-network on every experience in the batch with corresponding weight $w_{i} = \frac{(N \cdot P(i))^{-\beta(t)}}{w_{max}}$, where $N$ is the size of replay buffer, $\beta(\cdot)$ is the bias-annealing factor that can change over time, indicating how strong the bias-annealing is, $t$ is the current timestep, and $w_{max} = \max_{i}(N \cdot P(j))^{-\beta(t)}$ is the max weight among all experiences in replay buffer currently at time $t$.

### 3.2 Prioritized Replacing

Both the original experience replay and prioritized experience replay replace the oldest experience in replay buffer with the latest one when it is full. However, an experience was generated early does not necessarily mean it is useless. Like human beings’ intuition, sometimes your first decision without any rational processes is surprisingly good.

Therefore, we propose a method to balance the oldness and usefulness, where first we select some candidates of replacing based on priorities and then find the oldest one among them and choose it as the experience to replace with the newly generated experience.

In detail, $T_{i}$’s possibility of being chosen to be a replacing candidates $PR_{i} = \frac{p_{i}^{\gamma(\cdot)}}{\sum_{j}p_{j}^{\gamma(\cdot)}}$, where $p_{i}$ is the same priority used in section 3.1 and $\gamma(\cdot)$ is the importance-replacing function on time.

When a new experience is inserted, its priority (TD error) is the TD error of $T_{i}$, $\epsilon$ is a small positive number, and $\gamma(\cdot)$ is the importance-replacing function on time.

When a new experience is inserted, its priority (TD error) is the TD error of $T_{i}$, $\epsilon$ is a small positive number, and $\gamma(\cdot)$ is the importance-replacing function on time.
a new one with the old state and a new action selected by the latest Q-network. To demonstrate it better, we introduce a motivating example which is simple but interesting.

![Illustration of the motivating example](image)

Figure 2: Illustration of the motivating example

As we can see, once the agent chooses a direction in the first step, it must follow the same direction till the end. In this environment, the agent can easily find a policy which keeps going right with cumulative positive reward. However, as we are omniscient, we know the best policy is find the big treasure in the left part, which can provide the highest reward as long as the depth of it d ≤ 500. With state recycling, the probability of finding this path is much higher.

In the common case, when it is time to do state recycling in replacing stage, before we replace an old experience already in replay buffer, we choose certain amount of state-recycling candidates according to their priorities. For every experience in the candidates, we keep its old state, and then input it into the latest Q-network to get a new action choice (if it is same in the candidates, we keep its old state, and then input it into probability of finding this path is much higher.

Algorithm 1 DQN with DPSR

**Input:** minibatch-size $k$, learning rate $\eta$, replay buffer size $N$, exploration function $\epsilon(\cdot)$, importance-sampling function $\alpha(\cdot)$, bias-annaealing function $\beta(\cdot)$, importance-replacing function $\gamma(\cdot)$, max priority set flag for state recycling $M$, common replacing candidates size $C_{s}$, state recycling candidates size $C_{r}$, target network updating frequency $F_{t}$, sampling frequency $F_{s}$, state recycling frequency $F_{r}$, total timesteps $T$

1: Initialize replay buffer $B = \emptyset$, $\Delta = 0$
2: Initialize the parameters of Q-network $\theta$ randomly, and initialize $\theta_{target} = \theta$
3: Observe $s_{0}$ and choose $a_{0} \sim \pi_{\theta}(s_{0})$ with $\epsilon(0)$ possibility to choose an action randomly
4: for $t = 1$ to $T$ do
5: Observe $s_{t}$, $r_{t}$
6: Assemble experience $(s_{t-1}, a_{t-1}, r_{t}, s_{t})$ with $p_{t} = \max_{c < t} p_{c}$
7: if $|B| < N$ then
8: Add the new experience at the end of the queue
9: else if $t \equiv 0 \mod F_{s}$ then
10: for $i = 1$ to $C_{r}$ do
11: Sample experience $(s_{t-i}, a_{t-i}, r_{t-i}, s_{t-i}) \sim P_{R}(t_{i}) = \frac{P_{i}^{-\gamma(t_{i})}}{\sum_{j} P_{j}^{-\gamma(t_{j})}}$
12: Input $s_{t-i-1}$ into the latest Q-network with $\theta$ and get its choice of action $a_{t-i-1}$
13: if $a_{t-i-1} = a_{t-i-1}$ then
14: Randomly choose an available action $\tilde{a}_{t-i-1}$ with $\tilde{a}_{t-i-1} \neq \tilde{a}_{t-i-1}$
15: end if
16: Observe $s_{t-i}', r_{t-i}'$
17: if $M = True$ then
18: $p_{t} = \max_{c < t} p_{c}$
19: else
20: Compute TD error $\delta_{t_{i}} = r_{t_{i}}' + Q_{target}(s_{t_{i}}', \arg \max_{a} Q(s_{t_{i}}', a)) - Q(s_{t_{i}-1}, a_{t_{i}-1})$
21: Update the priority $p_{t_{i}} \sim |\delta_{t_{i}}|$
22: end if
23: Assemble experience $(s_{t_{i}-1}, a_{t_{i}-1}, r_{t_{i}}', s_{t_{i}}')$ with new $p_{t_{i}}$ obtained above
24: end for
25: Let $\pi^* = \arg \min_{i \leq t \leq C_{r}} p_{t_{i}}$, and replace the corresponding experience $B_{t_{i}}$ with the new one
26: else
27: for $i = 1$ to $C_{s}$ do
28: Sample experience $(s_{t-i-1}, a_{t-i-1}, r_{t-i}, s_{t-i}) \sim P_{R}(t_{i}) = \frac{P_{i}^{-\gamma(t_{i})}}{\sum_{j} P_{j}^{-\gamma(t_{j})}}$
29: end for
30: Let $\pi^* = \arg \min_{i \leq t \leq C_{s}} t_{i}$, and replace the corresponding experience $B_{t_{i}}$ with the new one
31: end if
32: if $t \equiv 0 \mod F_{t}$ then
33: for $i = 1$ to $K$ do
34: Sample experience $(s_{t-i}, a_{t-i}, r_{t}, s_{t-i}) \sim P(i) = \frac{P_{i}^{\alpha(t_{i})}}{\sum_{j} P_{j}^{\alpha(t_{j})}}$
35: Compute importance-sampling weight $w_{i} = (N \cdot P(i))^{-\beta(t_{i})}$, then let $w_{i} \sim w_{i} / \max_{j} w_{j}$
36: Compute TD error $\delta_{i} = r_{t} + Q_{target}(s_{t}, \arg \max_{a} Q(s_{t}, a)) - Q(s_{t-i-1}, a_{t-i-1})$
37: Update the priority of experience $p_{i} \sim |\delta_{i}|$
38: Accumulate weight-change $\Delta \leftarrow \Delta + w_{i} \cdot \delta_{i}$
39: end for
40: Update weight $\theta \leftarrow \theta + \eta \cdot \Delta$, $\Delta \leftarrow 0$
41: if $t \equiv 0 \mod F_{t}$ then
42: $\theta_{target} \leftarrow \theta$
43: end if
44: end if
45: Choose action $a_{t} \sim \pi_{\theta}(s_{t})$ with $\epsilon(t)$ possibility to choose an action randomly
46: end for

4. Experiments Results

Now, we are going to show the performance of our method in realistic problem domains. We completed the implementation based on OpenAI Gym [5] platform and compared our method with the original experience replay and prioritized experience replay which are provided as baseline methods in the platform.

To ensure the fairness, we use the identical deep neural network architecture and most parameters of learning algorithm such as mini-batch size, learning rate, replay buffer size, exploration policy, importance-sampling and bias-annaealing factors, sampling frequency, total timesteps and so on, which are shown comprehensively in Table 2.

First, we tested our method in a simple environment cart-
pole, which is a classic control problem. As shown in Figure 3, our method learns the optimal policy much more efficiently than both the original method and prioritized experience replay.

Then we completed experiments on more complicated cases, Atari game environments. We train RL agents with the original experience replay, prioritized experience replay, and DPSR experience replay separately under basically same common parameter sets and get the test results shown in Table 1. In all the 24 games, our method wins 23 “gold medals” and 1 “silver medal”. The average and median of the performance improvement are 161.1% and 87.0% compared to the original method, while the numbers are 137.1% and 92.1% compared to prioritized experience replay (JourneyEscape and Zaxxon are excluded as both baseline methods get non-positive scores in these two games). Besides, we’d like to note that original experience replay outperforms prioritized experience replay in 14 games, and even wins one “gold medal” in SpaceInvaders, which is a little surprising.

5. Discussion

To get better performance, we tried many different sets of hyperparameters and found that the method with only prioritized sampling and prioritized replacing (state recycling disabled) can have quite bad performance in some games. We guess this phenomenon may be caused by the fact that both prioritized sampling and prioritized replacing would introduce bias, therefore when we only use them, the double-bias can cause some negative effect on performance. We use the same bias-annealing factor for prioritized experience replay and DPSR experience replay in order to maintain the fairness but we can rationally guess that DPSR experience replay may have better performance with stronger bias-annealing techniques because of the reason mentioned above.

When we decide state recycling frequency and two replacing candidates sizes, target network updating frequency and the computational cost for state recycling should be taken into consideration. That is why we keep these parameters in a small range.

6. Conclusion

In this paper, we proposed double-prioritized state-recycled (DPSR) experience replay, a method that can make RL agents learn more efficiently. We compared our method with original experience replay and prioritized experience replay in some simple environments and Atari game environments, achieving state-of-the-art results.

Acknowledgement

This research has been supported in part by the ICT R&D program of MSIP/IITP 2016-0-00563 [Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion], and MSIP/IITP 2019-0-01396 [Development of framework for analyzing, detecting, mitigating of bias in AI model and training data].
References

[1] L.-J. Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching,” Machine learning, vol. 8, no. 3-4, pp. 293–321, 1992.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” arXiv preprint arXiv:1312.5602, 2013.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[4] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” arXiv preprint arXiv:1511.05952, 2015.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.

[6] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, “Hindsight experience replay,” in Advances in neural information processing systems, pp. 5048–5058, 2017.

[7] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, “Dueling network architectures for deep reinforcement learning,” arXiv preprint arXiv:1511.06581, 2015.

[8] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, “Sample efficient actor-critic with experience replay,” arXiv preprint arXiv:1611.01224, 2016.

[9] G. Tesauro, “Temporal difference learning and td-gammon,” Communications of the ACM, vol. 38, no. 3, pp. 58–68, 1995.

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” arXiv preprint arXiv:1707.06347, 2017.

[11] A. L. Caterini and D. E. Chang, Deep Neural Networks in a Mathematical Framework. Springer, 2018.

[12] B. Seymour, J. P. O’Doherty, P. Dayan, M. Koltzenburg, A. K. Jones, R. J. Dolan, K. J. Friston, and R. S. Frackowiak, “Temporal difference models describe higher-order learning in humans,” Nature, vol. 429, no. 6992, pp. 664–667, 2004.

[13] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh, “Action-conditional video prediction using deep networks in atari games,” in Advances in neural information processing systems, pp. 2863–2871, 2015.

[14] H. Van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” in Thirty-eighth AAAI conference on artificial intelligence, 2016.

[15] Y. Hou, L. Liu, Q. Wei, X. Xu, and C. Chen, “A novel ddpg method with prioritized experience replay,” in 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 316–321, IEEE, 2017.

[16] S. P. Singh, T. Jaakkola, and M. I. Jordan, “Reinforcement learning with soft state aggregation,” in Advances in neural information processing systems, pp. 361–368, 1995.

[17] M. Humphrys, “Action selection methods using reinforcement learning,” From Animals to Animats, vol. 4, pp. 135–144, 1996.

[18] H. R. Tizhoosh, “Reinforcement learning based on actions and opposite actions,” in International conference on artificial intelligence and machine learning, vol. 414, 2005.

A. Implementation Details

We completed our implementation and experiments based on the deepq module on OpenAI Gym platform. The environments of Atari game we used are in NoFrameskip-v4 version (e.g. The environment name of the SpaceInvaders game is SpaceInvadersNoFrameskip-v4). There are some common hyperparameters (some may not used in original method and prioritized method) and also some inconstant hyperparameters for our method. See Table 2 for more details.

Table 2: Hyperparameters settings

| Hyperparameter | Value (Range of values) |
|----------------|-------------------------|
| k              | 32                      |
| η              | 0.0005                  |
| N              | 50000                   |
| ε(t)           | max(1 − 9.8t/T, 0.02)   |
| α(t)           | 0.6                     |
| β(t)           | 0.4 + 0.6t/T            |
| γ(t)           | 0.1, 0.2, 0.3, 0.4, 0.5, 0.6 |
| Cc             | 128, 256                |
| Cr             | 8, 16, 32, 64           |
| Fι             | 500                     |
| Fs             | 1                       |
| Fp             | 100000, 20000           |
| T              | 1000000                 |

Besides, as shown in Table 2 for different game environments, the best performance of our method may be achieved by different setting of parameters. But we can still find some settings of parameters which have relatively good performance in most games as shown in Table 2.
Table 3: Best parameters in different games

| Game name  | Parameters achieving the top 3 scores \((\gamma, C_c, F_r, C_r)\) |
|------------|---------------------------------------------------------------|
| AirRaid    | (.5, 128, 10k, 16), (.2, 128, 20k, 8), (.2, 128, 10k, 8) |
| Alien      | (.5, 256, 20k, 16), (.6, 256, 0, 0), (.2, 256, 20k, 64) |
| Amidar     | (.1, 128, 10k, 8), (.3, 256, 0, 0), (.1, 128, 10k, 16) |
| Assault    | (.6, 256, 10k, 64), (.1, 128, 20k, 8), (.5, 128, 10k, 8) |
| Asterix    | (.2, 128, 10k, 16), (.1, 256, 20k, 64), (.3, 128, 20k, 8) |
| BeamRider  | (.4, 256, 10k, 8), (.1, 128, 0, 0), (.3, 128, 10k, 8) |
| Bowling    | (.6, 256, 10k, 16), (.6, 256, 20k, 64), (.6, 128, 20k, 8) |
| Breakout   | (.1, 128, 10k, 64), (.3, 128, 10k, 8), (.2, 128, 20k, 32) |
| Carnival   | (.1, 128, 10k, 32), (.1, 256, 10k, 16), (.6, 128, 20k, 8) |
| Enduro     | (.6, 256, 10k, 64), (.6, 256, 10k, 64), (.6, 128, 10k, 16) |
| Freeway    | (.1, 256, 10k, 8), (.3, 128, 20k, 16), (.3, 128, 20k, 64) |
| Frostbite  | (.1, 256, 20k, 8), (.4, 256, 20k, 16), (.6, 128, 10k, 16) |
| Hero       | (.5, 128, 10k, 32), (.6, 256, 10k, 64), (.4, 256, 10k, 64) |
| JourneyEscape | (.6, 256, 10k, 64), (.3, 128, 10k, 8), (.2, 128, 20k, 64) |
| Krull      | (.5, 256, 20k, 16), (.1, 256, 20k, 64), (.2, 128, 20k, 64) |
| KungFuMaster | (.6, 128, 20k, 32), (.4, 128, 20k, 16), (.3, 256, 20k, 64) |
| MsPacman   | (.3, 128, 10k, 32), (.1, 128, 10k, 32), (.4, 256, 20k, 8) |
| Phoenix    | (.2, 256, 10k, 16), (.3, 128, 10k, 8), (.6, 256, 20k, 16) |
| Qbert      | (.6, 256, 10k, 32), (.5, 128, 10k, 16), (.6, 128, 10k, 16) |
| Riverraid  | (.5, 256, 10k, 32), (.1, 256, 20k, 64), (.6, 128, 10k, 32) |
| SpaceInvaders | (.3, 256, 10k, 8), (.1, 128, 10k, 16), (.1, 256, 20k, 64) |
| StarGunner | (.3, 128, 10k, 16), (.2, 128, 10k, 16), (.3, 128, 10k, 8) |
| VideoPinball | (.1, 128, 0, 0), (.1, 128, 20k, 16), (.4, 256, 20k, 8) |
| Zaxxon     | (.1, 256, 20k, 64), (.3, 128, 10k, 64), (.1, 256, 20k, 32) |

\* \(F_r = C_r = 0\) means state recycling is disabled

Table 4: Experiment results of some parameter sets

| Game name   | Original Prioritized DPSR0 \(*\) | DPSR1 | DPSR2 | DPSR3 | DPSR4 |
|-------------|---------------------------------|-------|-------|-------|-------|
| AirRaid     | 545.0                           | 602.5 | 2892.5 | 1285.0 | 632.5 |
| Alien       | 853.0                           | 917.0 | 995.0  | 673.0  | 942.0 |
| Amidar      | 155.0                           | 145.7 | 148.1  | 221.8  | 155.5 |
| Assault     | 882.5                           | 638.9 | 651.8  | 1525.6 | 2145.0 |
| Asterix     | 2055.0                          | 1435.0| 1600.0 | 2145.0 | 1795.0 |
| BeamRider   | 2101.8                          | 2442.0| 1525.6 | 1899.2 | 1086.0 |
| Bowling     | 29.0                            | 24.2  | 44.4   | 9.6    | 29.1  |
| Breakout    | 87.0                            | 136.4 | 134.5  | 80.6   | 91.5  |
| Carnival    | 3286.0                          | 2006.0| 3847.0 | 2704.0 | 860.0 |
| Enduro      | 612.3                           | 497.3 | 795.4  | 503.6  | 725.9 |
| Freeway     | 30.2                            | 29.5  | 30.7   | 31.7   | 30.5  |
| Frostbite   | 229.0                           | 998.0 | 382.0  | 664.0  | 217.0 |
| Hero        | 2891.5                          | 2585.0| 5937.0 | 2936.0 | 2838.5|
| JourneyEscape | -4150.0                         | -3350.0| -6520.0| -2130.0| -4290.0|
| Krull       | 4736.7                          | 5541.3| 5864.4 | 6418.8 | 6001.2|
| KungFuMaster | 14900.0                         | 19820.0| 14500.0| 16020.0| 13990.0|
| MsPacman    | 1682.0                          | 1666.0| 1926.0 | 1913.0 | 1768.0|
| Phoenix     | 3668.0                          | 2787.0| 3234.0 | 3122.0 | 3342.0|
| Qbert       | 2050.0                          | 1490.0| 870.0  | 1680.0 | 1440.0|
| Riverraid   | 5334.0                          | 4596.0| 5073.0 | 5349.0 | 4949.0|
| SpaceInvaders | 610.0                           | 136.5 | 5073.0 | 5349.0 | 4949.0|
| StarGunner  | 7313.0                          | 12025.1| 30337.0| 21596.9| 15689.0|
| VideoPinball | 0.0                             | 0.0   | 0.0    | 0.0    | 1270.0|

\* \(\gamma, C_c, F_r, C_r = (.6, 128, 20k, 8), (.1, 128, 10k, 16), (.2, 256, 20k, 8), (.3, 128, 10k, 8), (.3, 256, 10k, 16)\)