Reward Backpropagation Prioritized Experience Replay

Yangxin Zhong [1] **Borui Wang** [1] **Yuanfang Wang** [1]

Abstract

Sample efficiency is an important topic in reinforcement learning. With limited data and experience, how can we converge to a good policy more quickly? In this paper, we propose a new experience replay method called Reward Backpropagation, which gives higher minibatch sampling priority to those (s, a, r, s') with $r \neq 0$ and then propagate the priority backward to its previous transition once it has been sampled and so on. Experiments show that DQN model combined with our method converges 1.5x faster than vanilla DQN and also has higher performance.

1. Introduction

Reinforcement learning is a suitable framework for sequential decision making problem, where an agent explores the environment by making actions, observes a stream of experience with rewards and learn to make good decisions from the observation. One of the most popular reinforcement learning algorithms is Q learning. It requires a large amount of experience to estimate the value of taking a specific action under certain state, which is called Q value. By comparing Q values of different actions in the same state, the agent learns the optimal policy.

However, many real world problems have very large state spaces along with sparse and delayed rewards. In this case we don't usually have enough effective experience to get accurate Q value estimates, which makes even simple tasks might need days for training. Moreover, exploration and getting experience are pretty expensive in some real world problems such as robot and vehicle control. As a result, how to use and reuse the limited samples of experience to learn better policy in a short time becomes an important topic, which is called sample efficiency.

In this paper, we propose a new method called Reward Backpropagation Prioritized Experience Replay that aims to improve the sample efficiency of modern bootstrapping reinforcement learning algorithms such as Deep-Q Networks (DQN) (Mnih et al., 2015). The main idea of this approach is that in order to converge to accurate estimate of the Q values more quickly with limited experience, we should try to use those (s, a, r, s') transitions with non-zero reward to update the Q estimate model first, followed by using their predecessor transitions and so on by giving higher sampling priority from replay meormy. The motivation behind this idea is that non-zero reward transitions usually have more significant meanings than zero reward ones. They indicate short-term success or failure of action sequence in the near past. If we propagate the stimulus of non-zero reward backward to the actions before it, we can provide more accurate target Q values for those actions and thus update more efficiently.

For evaluation, we implement a DQN with Reward Backpropagation Experience Replay to learn to play the Atari game of Pong, Breakout and Ice Hockey. We then compare our method with the vanilla DQN baseline in terms of convergence speed and final performance. Experiments show that our method of adding Reward Backpropagation Experience Replay improve the baseline in converging 1.5x faster than vanilla DQN and also performing better at the end of 20-epoch training.

2. Related Work

In Q learning, we need to estimate Q values of every state-action pair, which could become a problem if we have large state and action spaces. A typical way to deal with this is to use function approximation, which extracts features from state-action pair and use them to approximate Q value by a model such as linear function.

Recently, the approach called deep-Q network (DQN), which uses the powerful deep convolutional neural network as function approximation model, has achieved promising results (Mnih et al., 2015). The model was trained and tested within the Atari 2600 computer games environment. It takes preprocessed gray scale images from game as states and inputs for convolutional neural network and approximates the Q values for actions using the neural network. (Van Hasselt et al., 2016) improved the DQN method by

[1]Stanford University, Palo Alto, California, USA. Correspondence to: Yangxin Zhong <yangxin@stanford.edu>, Borui Wang <wbr@stanford.edu>, Yuanfang Wang <yolanda.wang@stanford.edu>.

Source code link: https://github.com/yolandawww/DQN-cs234.git

changing the learning target of neural network with the idea of Double Q-learning. This reduces the problem of overestimation suffered by vanilla DQN. Both the Q-learning method with deep neural network can solve the problem of large state and action spaces, but still suffer from low training efficiency resulting from sparse and delayed rewards. To help address this, based on the DQN model, the DQ(λ)N model (Mousavi et al., 2017) further added the Q(λ) method (Watkins, 1989), which uses weighted average of Q values of all n-step backups to improve learning efficiency. However, in the training process, in order to implement multiple-step backups, the algorithm needs to sample a sequence of continuous transitions from the replay memory instead of independent transitions in each update step. This may cause the sample correlated issue mentioned in original DQN paper. Prioritized experience replay (PER) is another state-of-art in sample efficiency topic (Schaul et al., 2016). Since this work is highly related to ours, it will be discussed in details in section 3.2.

3. Approach

3.1. Background of Q Learning

In a standard Q learning algorithm, we maintain a Q value estimate for every (s, a) pair (we can either use a table to store Q or use any function to approximate it). Once we get an experience (s, a, s', r), which means we took action a in state s and get reward r from environment and then transit to successor state s', we can create a new sample estimation for Q:

$$Q_{samp}(s, a) = r + \gamma V(s') = r + \gamma \max_{a'} Q(s', a'), \quad (1)$$

where γ is the discount factor. Then, we update estimate of $Q(s, a)$ in a running average way:

$$Q(s, a) := (1 - \alpha)Q(s, a) + \alpha Q_{samp}(s, a),$$

where α is called learning rate, because we can rewrite above formula into:

$$Q(s, a) := Q(s, a) + \alpha[Q_{samp}(s, a) - Q(s, a)] \quad (2)$$

Here we can see $Q_{samp}(s, a)$ as target and $[Q_{samp}(s, a) - Q(s, a)]$ as the error between sample and our former estimate. The state-of-art DQN model (Mnih et al., 2015) is a Q learning algorithm using a deep neural network as the approximate function of $Q(s, a)$. To improve sample efficiency, DQN uses a replay memory to store the old transitions and reuse them by uniformly sampling a minibatch of transitions for update at each learning step.

3.2. Prioritized Experience Replay

To further improve the sample efficiency of DQN, Prioritized experience replay (PER) assigns each transition a priority weight. While DQN uses a uniform sampling method for experience replay, PER samples each transition with a probability proportional to its priority weight, which is a function of the error between target Q and estimate Q value in last update. In this way, PER can replay important transitions more frequently, and therefore learn more efficiently.

However, using error as priority weight may not be a good solution in a sparse reward environment. Suppose in average, we have only one non-zero reward transition every one hundred zero reward transitions. Although the error of each of these zero reward transitions may be small, their sum can be very large, which makes our model always use zero reward transitions to update, which is bad for learning since zero rewards tell us nothing about whether the agent is acting good or bad.

Our method also uses a priority weight for each transition in the replay memory and sample them with a probability proportional to their weights. Different from PER which focuses directly in the training stage for improvement by using the error as sampling priority, we instead look more into the key issue of sparse reward environment, the large proportion of zero rewards. To raise the impact of those non-zero reward in learning process, we give the transitions with non-zero rewards higher priority weights in the very beginning, and once they are sampled, this higher priority will be propagated backward step by step to the previous zero reward transitions.

3.3. Motivation of Our Model

From equation (2), We find that the update rule of Q-value estimate depends heavily on target, whose definition is shown in equation (1). Basically we have two parts in definition of $Q_{samp}(s, a)$: 1) $\gamma \max_{a'} Q(s', a')$, we use this part to propagate the Q value estimate from successor state s' to current state s, and this can be seen as self-correction to make Q value estimate of current state consistent with successor state; 2) r, we use this part to propagate the impact of the reward "stimulus" from environment the Q value estimate in model.

Now suppose in the replay memory we have two adjacent transitions $(s_{t-1}, a_{t-1}, r_{t-1}, s_t)$ and (s_t, a_t, r_t, s_{t+1}), if the first transition is used for update before the second transition, then the reward stimulus r_t won't propagate to s_{t-1} in the first update. On the contrary, if we use second transition first, then impact of r_t can propagate to s_t in the first update and then to s_{t-1} in the second update, which is more efficient. Also, in the backward order, we can make Q value estimates of all the three states consistent by self-correction mentioned above, which can't be done in the forward order.

Moreover, in a sparse reward environment, usually a reward can only be achieved after a sequence of actions and it is

Algorithm 1 DQN with Reward Backpropagation Prioritized Experience Replay

Parameters: non-zero reward priority $\beta \gg 1$, priority decay rate λ, learning rate α, minibatch size n.
Initialize DQN parameter θ with random values
Initialize replay memory M with capacity N
for each episode **do**
  Initialize state s
  for each step in the episode **do**
    choose an action a according to ϵ-greedy policy
    take action a, observe reward r and next state s'
    if $s' =$ terminal **or** $r \neq 0$ **then**
      $p \leftarrow \beta$
    else
      $p \leftarrow 1$
    end if
    store transition (s, a, r, s') with priority p in M
    $s \leftarrow s'$
    $b \leftarrow$ sample a minibatch of transitions in M with probabilities proportional to their priority p
    $e \leftarrow 0$
    for each transition (s_i, a_i, r_i, s_i') with p_i in b **do**
      if $s_i' =$ terminal **then**
        $Q' \leftarrow 0$
      else
        $Q' \leftarrow \max_a Q(s_i', a; \theta^-)$
      end if
      $e \leftarrow e + [r_i + Q' - Q(s_i, a_i; \theta)] \frac{\partial Q(s_i, a_i; \theta)}{\partial \theta}$
      $(s_j, a_j, r_j, s_j'), p_j \leftarrow$ *Predecessor*$[(s_i, a_i, r_i, s_i')]$
      if $s_j' \neq terminal$ **and** $r_j = 0$ **and** $p_i > 1$ **then**
        update $p_j \leftarrow p_i$ in M
      else if $p_i > 1$ **then**
        $(s_k, a_k, r_k, s_k'), p_k \leftarrow$ the first transition after (s_j, a_j, r_j, s_j') with $s_k' =$ terminal **or** $r_k \neq 0$
        update $p_k \leftarrow \max(\lambda p_i, 1)$ in M
      end if
      update $p_i \leftarrow 1$ in M
    end for
    $\theta \leftarrow \theta + \alpha \cdot \frac{e}{n}$
  end for
end for

an indicator of success or failure of the action sequence. Thus, updating backward from the states with rewards can encode the short term effect of those actions into their Q value estimates smoothly.

Inspired by the two insights above, we propose that we should set high priority to use non-zero reward transitions for update first and then their previous transitions and so on. Since in this way, the stimulus of a reward can be propagated backward efficiently to a sequence of actions just before it. We call this strategy Reward Backpropagation.

3.4. Reward Backpropagation PER

In the case of DQN with experience replay, our method is shown in Algorithm 1. Compared with vanilla DQN algorithm, we have two extra parameters: β and λ, which are priority weight for non-zero reward and priority decay rate when we start a new round of propagation. In our method, each transition observed from environment will be stored in replay memory along with a priority weight p. If the observed transition is terminal or has a non-zero reward, then $p = \beta \gg 1$; otherwise, $p = 1$. When we sample a minibatch of transitions in every update step, each transition in replay memory will be chosen with a probability proportional to its priority weight.

In our algorithm, once a transition with priority $p > 1$ is sampled, we propagate that high priority backward identically to its previous transition. Suppose a non-zero reward transition or a terminal transition, denoted by t_k, has prior-

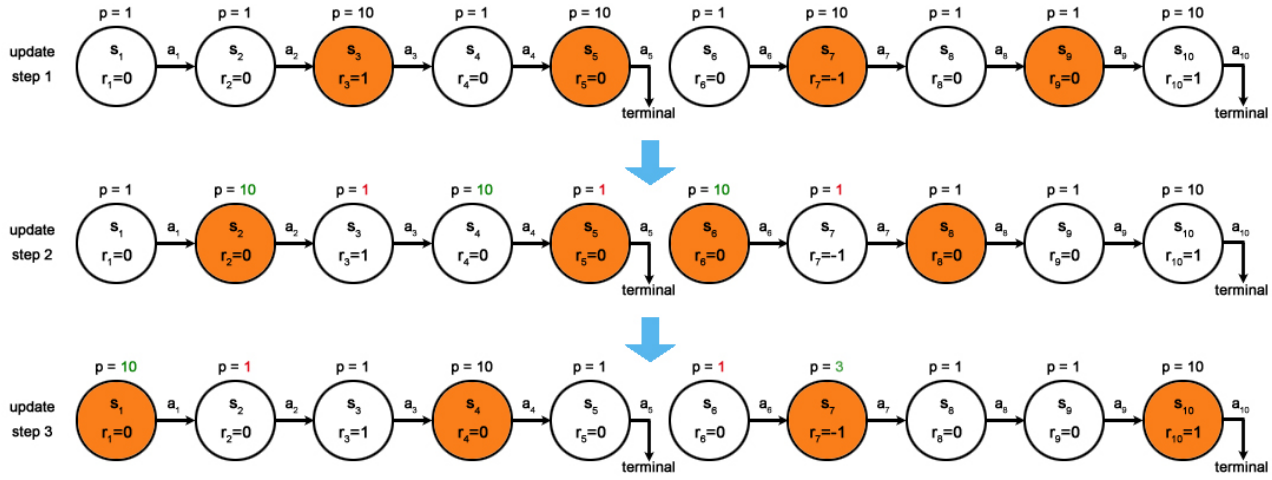


Figure 1. Illustration of priority backpropagation in our algorithm. $\beta = 10$, $\lambda = 0.3$. Suppose we have a replay memory with capacity $N = 10$. Each row denotes an update step. No new transition is added after step 1. The first row shows the initialization of priority p according to terminal and reward. The second and third rows show the priority change after propagation in their previous steps. Orange denotes the transitions that are sampled for update in each step. Minibatch size $n = 4$. Green denotes the priorities that were propagated to in the previous step. Red denotes the priorities that were set to normal weight, which is 1, after propagation in the previous step.

ity $\beta > 1$ at the very beginning. Then once t_k is sampled, its priority will be set to normal priority, which is 1, and transition t_{k-1} will get the high priority β. After that, once t_{k-1} is sampled, β will be propagated from t_{k-1} to t_{k-2}, and so on. This process will keep going until β is propagated to another non-zero reward transition or the head of episode. Then β will be propagated back to t_k and start a new round of propagation. But every time we start a new round, the high priority will be decayed by factor λ, until it goes down to normal priority 1. That is to say, in the second round, the high priority will become $\lambda\beta$, and in the third round will become $\lambda^2\beta$, and so on. In this loopy way, we propagate the high priority from non-zero reward transition backward to zero transitions again and again.

3.5. Demo of Reward Backpropagation

Figure 1 shows how the priorities change during the backpropagation algorithm. Note that only priority greater than 1 will be propagated and it will only be propagated when the corresponding transition is sampled. After propagation, the priority of this sampled transition will be set to 1. More details of parameter setting are introduced in caption of Figure 1.

3.6. Parameter Discussion

In our method, the high priority β and the priority propagation mechanism roughly ensure the backward order of transition using in update steps, which starts from non-zero reward to zero reward transitions. This allow the model to have the benefits we discussed in section 3.3. In practice, choosing β to be the ratio of non-zero reward transition frequency to zero reward transition frequency yields a good performance, since this setting makes the sampling probability of non-zero reward transitions approximately equal to probability of zero reward transitions at the very beginning. The decay rate λ controls how much we want to use the old transitions for backward updating or how many rounds of backpropagation we want. We found that a λ that makes it able to have 2 or 3 rounds of backpropagation usually yields a better performance. In our experiment, we set $\beta \in [10, 100]$ and $\lambda \in [0.1, 0.3]$.

4. Experiment Results

4.1. Experiment Setting

For evaluation, we test our algorithm in the Atari 2600 games. For the limitation of time and computation resources, Pong, Breakout and Ice Hockey are chosen in our experiments since they are less complex compared with other environments and training can converge much faster.

We use vanilla DQN model as our baseline. The parameter setting of DQN is the same as original paper (Mnih et al., 2015): minibatch size n = 32, replay memory size N = 1000000, discount factor γ = 0.99, learning rate α = 0.00025, exploration rate ϵ = 1.0→ 0.1, update frequency = 4, target network update frequency = 10000, steps per

epoch = 250000, every action repeats 4 frames, and uses a momentum SGD optimizer. The original paper trained DQN model with 200 epochs (50 million steps). With limited computation resources, in our experiments we train all the model with 20 epochs (5 million steps), which means our agents explore and learn far less than the original paper. This causes lower final performance of our agents than the original's.

We also implement a DQN with Prioritized Experience Replay (PER) (Schaul et al., 2016) for comparison. We choose to use their rank-based PER model, which was claimed to be more robust than proportional PER. Furthermore, to increase the training speed, we implement the efficient version in the paper with a heap to approximate the perfect sorted array. The parameter setting of DQN part is the same as vanilla DQN. For the parameters of PER part, we try two different settings reported in their paper that achieve the best performance for rank-based model: $\{\alpha = 0.5 \rightarrow 0, \beta = 0, \eta = \eta_{DQN}/4\}$ and $\{\alpha = 0.7, \beta = 0.5 \rightarrow 1, \eta = \eta_{DQN}/4\}$, and then choose the ones perform better in our experiments as the final PER results: Pong and Ice Hockey used the former set and Breakout used the latter set.

For the implementation of our method, we use the same DQN parameters as the previous two baseline models, and settings of two extra parameters β and λ are: Pong, $\{\beta = 100, \lambda = 0.1\}$; Breakout, $\{\beta = 10, \lambda = 0.3\}$; Ice Hockey, $\{\beta = 10, \lambda = 0.3\}$. As mentioned above, We ran the training process for 20 epochs, and each epoch of our model took around 45 minutes using one NVIDIA Tesla M60 GPU.

4.2. Faster Convergence Speed

The curves of average evaluation scores per episode throughout the training process are shown in Figure 2. As shown in the figure, our method converges around 1.5x faster than vanilla DQN model in all of the three environments. For example, in Pong, our model achieved an average score greater than 12 after 7 epochs, while vanilla DQN model didn't accomplish this until epoch 11. We can find a rapid and stable boost in each of these environment at the beginning of training process. This suggests that the backward order of sampling use from non-zero reward transitions to their previous zero reward transitions can help the agent to learn a relatively good policy in a shorter time, that is to say, a warm start. After our method reached a relatively high performance, its performance kept increasing steadily on the whole in Pong and Breakout environment, while in Ice Hockey all the three methods show a sign of plateau.

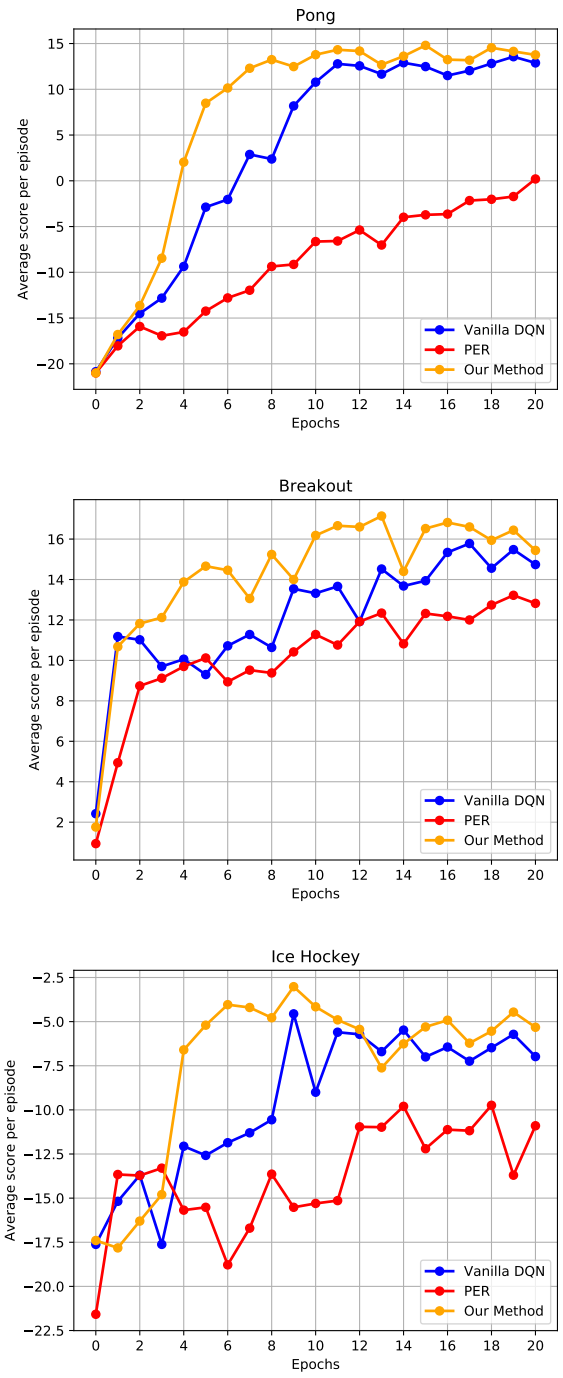
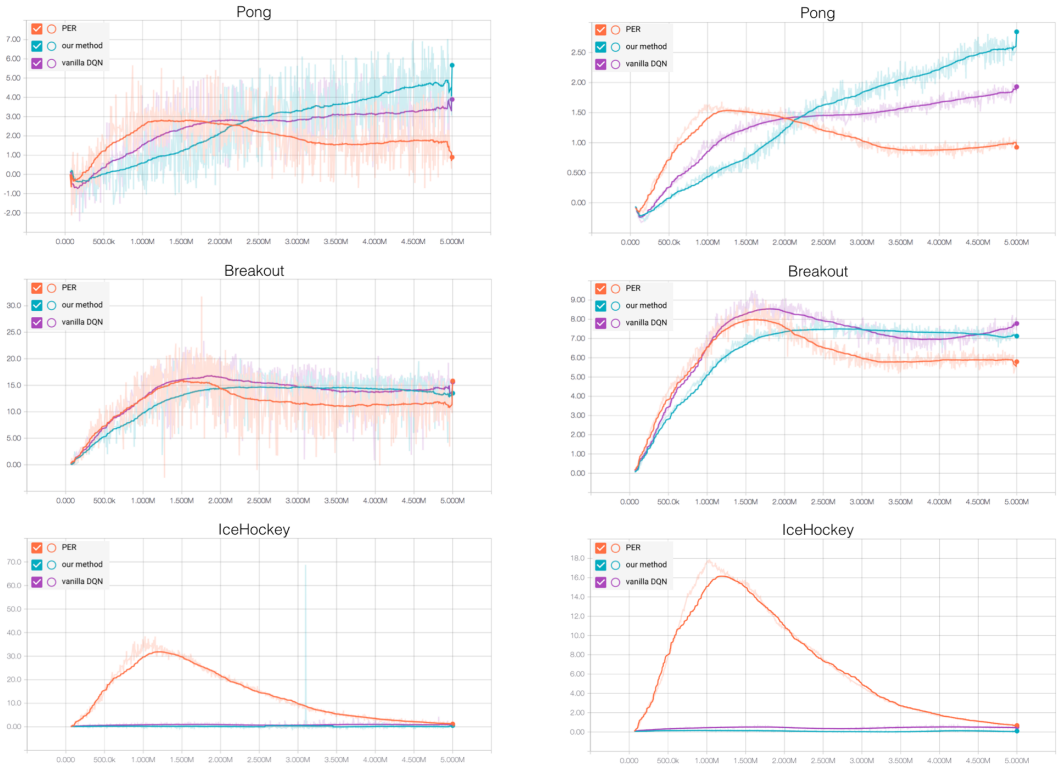


Figure 2. Learning curves of Vanilla DQN vs. Prioritized Experience Replay (PER) vs. Our method on Pong, Breakout, and Ice Hockey. Parameter settings in our method: Pong, $\beta = 100, \lambda = 0.1$; Breakout, $\beta = 10, \lambda = 0.3$; Ice Hockey, $\beta = 10, \lambda = 0.3$.

Also, in the Ice Hockey environment, we find some abnormal observation that the performance of our model didn't increase a lot at the very beginning but increased rapidly after 3 epochs, while the other two methods first increased



(a) Average Q value

(b) Maximum Q value

Figure 3. Average and maximum Q values of vanilla DQN vs. Prioritized Experience Replay (PER) vs. our model during training in Pong, Breakout and Ice Hockey.

rapidly and then fell back a little and increased slowly. Moreover, the performance of our model also fell back a little after 10 epochs and sometimes got worse than vanilla DQN (epoch 13 and 14). We argue that these abnormal performance are due to the Ice Hockey environment itself. Since in this game the agent needs to control two ice hockey players against the other two to win by teamwork, it is more complex than Pong and Breakout. Usually some newly learnt "good" short-term policies may harm the long-term rewards and cause fall back in performance. Therefore, this environment requires more explorations and training, and is harder for the models to get a stable and good policy within a short time like 20 epochs. Actually, DQN and PER didn't do a very good job even after 200 epochs for Ice Hockey (Mnih et al., 2015; Schaul et al., 2016), they got average scores around 0 (our best is around -3 within 10 epochs).

4.3. Better Performance

Besides faster convergence time, Figure 2 also shows better performance at the end of 20 epochs of our method than the vanilla DQN and PER DQN. Although 20 epochs training is actually short compared with 200 epochs in the original papers, which means the performance in this figure is not the real final performance. The figure shows that at least our models can get higher performance in a short time by using the reward backpropagation prioritized experience replay strategy we proposed.

4.4. Poor Performance of PER

An unexpected observation shown in Figure 2 is that DQN with PER performed very bad in all the three environments. It not only shows a slower convergence learning/speed curve but also gets much lower performance after 20 epochs compared with vanilla DQN and our model. We have checked, debugged and tested our implementation of PER for several times to make sure it is correct. And we also tried several different parameter settings reported to be good in the original PER paper, but it still yields to a poor performance.

The only different between our experiments and original ones is that we trained for a shorter time (20 epochs vs. 200 epochs in the original paper). In the original paper, they only reported the performance at the end of 200 epochs, but the average scores for DQN with PER after 20 epochs were unclear. They did provide the learning curves for these three environments, but those were experiments for

Double DQN with PER, not vanilla DQN with PER. Also, for Breakout and Ice Hockey, the Double DQN model with PER actually performs poorer than the vanilla DQN model within 20 epochs in their learning curves (Schaul et al., 2016). All of these indicate that PER may be beneficial to the agent performance after a long time of training, but it might suffer from lower efficiency of sample used at the early stage of training and thus might learn slowly at the very beginning.

One of the explanation we found in our experiments is that PER often overshoots the real Q value estimates at the early stage of training. In Figure 3, we can find that the average/maximum Q values of the PER model often increase rapidly at the very beginning and then fall back to lower values afterwards. This is evident especially in Ice Hockey environment, where the Q value estimates of PER model exploded within 5 epochs. This may caused by the sampling strategy PER used. Since PER always tends to choose those transitions with large error in last update step as the new minibatch, it will boost the model to converge to the target Q values. However, at the very beginning of training, all of the target Q values are randomly initialized and totally incorrect. As a result, PER might make the model converge to a bad policy very quickly at the start. Fortunately, this can be fixed after a longer time of training as shown in Figure 3.

Different from PER, the average/maximum Q values of our Reward Backpropagation PER increased steadily in all the three environments unless it reached a plateau due to not enough explorations. Because our method doesn't choose the transitions with large error to replay, it won't lead to an overshot or fast convergence to bad policy at the beginning of training (even when vanilla DQN also shows a sign of overshooting in Breakout). Instead, our model focuses more on those more important transitions with non-zero rewards, and backpropagates their impact to the earlier actions, which turns out to be helpful for getting more accurate Q estimates steadily in a shorter time (see Figure 3). Note that in the IceHockey environment, the average performance are always negative within 20 epochs, which means that the more accurate Q values should be closer to 0 and even negative.

We can also find that extremely large Q value estimates occurred in our method in Ice Hockey after 3 million steps, but this outlier didn't impact our model much. In PER, this might become a problem since it is likely choose the corresponding transition to update frequently due to its extremely large error.

4.5. Slightly Longer Running time

Although our method converges 1.5x faster than vanilla DQN in terms of training steps, the running time of every step becomes longer (see Table 1). This is not because our model requires a longer time for optimization. Instead, the extra running time comes from the sampling priority operation. More concretely, in each step, we need to update, backpropagate the priorities in replay memory, and also need to normalize them for sampling by dividing them by their sum. In our final implementation, we used the running sum to save the time for sum calculation and used a pre-normalization method to infrequently normalize the whole priority array. After these optimizations, the running time of our method was only 10%-20% longer than vanilla DQN, which is shown in Table 1.

Table 1. Training time of 20 epochs for each method. (hours)

ENVIRONMENT	DQN	PER	OURS
PONG	14.42	18.51	15.56
BREAKOUT	10.84	17.96	12.93
ICE HOCKEY	12.53	19.48	13.66

However, in Table 1, we can find that the PER method cost even longer time than our method. In proportional PER, it also needs to sum over all the priorities and normalize them for sampling. Moreover, since PER uses the exponential function to project the raw error to final priorities and to calculate the importance sampling weights, it would cost much more extra time than our method. In their rank-based PER model, the extra time for sum over, exponential and normalizing operations are saved largely since they use a fixed rank-based priority array in each step (Schaul et al., 2016). However, they need to maintain a sorted array for all the errors of transitions in last update steps, where the sorting operation is highly time consuming. In their final efficient version (also our PER implementation), they use a heap array to approximate the perfect sorted array, which is reported to have a minor harm to the final performance but become much more efficient. Nevertheless, it still has a time complexity of $O(n \log N)$ for the heap updating in each step, where n is the minibatch size and N is the replay memory size, while our model only requires a time complexity of $O(n)$ for priority backpropagation in average. As a result, we can see in Table 1 that our method is much more efficient than PER in terms of running time.

5. Conclusion

To conclude, we proposed Reward Backpropagation Prioritized Experience Replay, which is a new method to improve the sampling efficiency for DQN model. Our method looks more into the issue of sparse reward in real world environment, which gives higher minibatch sampling priority to those more important transitions with non-zero reward and then propagate the high priority backward to their previous

zero reward transitions. Experiments show that the DQN model combined with our method converges 1.5x faster than the vanilla DQN and also has higher performance. Furthermore, our method shows better performance at the early stage of training and also faster running speed than the state-of-art PER model.

So far we have shown the benefits of our method to DQN model in a short training time. However, the long-term effect of our method is still unclear. In the future, we plan to run our model for longer training time to further evaluate its impact to the convergence speed and final performance of the DQN model. In addition, since our method does a better job than PER at the early stage of training but PER is proved to be beneficial after a long training time, we plan to combine the advantages of these two methods and propose a hybrid model in the future.

References

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518 (7540):529–533, 2015.

Mousavi, S. S., Schukat, M., Howley, E., and Mannion, P. Applying q (λ)-learning in deep reinforcement learning to play atari games. 2017.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. In *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*, 2016.

Van Hasselt, Hado, Guez, Arthur, and Silver, David. Deep reinforcement learning with double q-learning. In *AAAI*, pp. 2094–2100, 2016.

Watkins, C. J. C. H. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.