Integration of Efficient Deep Q-Network Techniques Into QT-Opt Reinforcement Learning Structure

Shudao Wei[1], Chenxing Li[1,2], Jan Seyler[2] and Shahram Eivazi[1,2]
[1]Department of Computer Science, University of Tübingen, Tübingen, Germany
[2]Advanced Develop. Analytics and Control, Festo SE and Co. KG, Esslingen, Germany

Keywords: Prioritized Experience Replay, Noisy Network, Mixed Policy, Distributed Reinforcement Learning, Q-Function Targets Via Optimization, Quantile Q Target Optimizer.

Abstract: There has been a growing interest in the development of offline reinforcement learning (RL) algorithms for real-world applications. For example, offline algorithms like qt-opt has demonstrated an impressive performance in grasping task. The primary motivation is to avoid the challenges associated with online data collection. However, these algorithms require extremely large dataset as well as huge computational resources. In this paper we investigate the applicability of well known improvement techniques from Deep Q-learning (DQN) methods to the QT-Opt offline algorithm, for both on-policy and mixed-policy training. For the first time, we show that prioritized experience replay(PER), noisy network, and distributional DQN can be used within QT-Opt framework. As result,for example, in a reacher environment from Pybullet simulation, we observe an obvious improvements in the learning process for the integrated techniques.

1 INTRODUCTION

For many years, reinforcement learning (RL) has received extensive attention, designed to describe and solve problems, in which agents learn strategies to achieve specific goals while interacting with the environment. Recently, RL has had fair achievements in real-world applications as such the implementation of more specific real-world applications requires, RL's learning efficiency become the goal to pursue. Given the vision-based data scope, the single-step duration and memory footprint required for training an agent are much more extensive than a value-based task. How to solve tasks in larger-scale environments with as few data and training steps as possible has become a focus of attention in RL.

A leading example for large scale real-world problem is QT-Opt (Kalashnikov et al., 2018) (Q-function Targets via Optimization) RL algorithm which is designed to provide generalization for different tasks and adaptation for compound action space. It meets the demand with a distributed asynchronous structure, a derivative-free optimization, and both online and offline training.

Main limitation of QT-Opt algorithms is that requires extremely large dataset as well as huge computational resources. To meet the demand for learning efficiency of RL in real-world in this paper we propose to apply well-known rainbow (Hessel et al., 2018) improvement techniques to the vanilla QT-Opt.

Hessel et al. (Hessel et al., 2018) empirically showed that prioritized experience replay(PER) (Schaul et al., 2015), noisy network (Fortunato et al., 2017), and distributional DQN (Bellemare et al., 2017; Dabney et al., 2018b; Dabney et al., 2018a) can improve overall performance of Deep Q-learning (DQN) algorithm. Prioritized experience replay (PER) (Schaul et al., 2015) in replay buffer samples training data non-uniformly under data potency, which exploits the available data more effectively; Noisy network (Fortunato et al., 2017) raises the randomness from the action selection level to the network level to generate new trajectory logic, which stimulates more complex explorations; Q-value in distributional perspective (Bellemare et al., 2017; Dabney et al., 2018b; Dabney et al., 2018a) provides more comprehensive information for action selection.

To our knowledge, there is no studies combining rainbow improvements in qt-opt structure. In this paper we implement these techniques in away to be compatible with non-distribution version of qt-opt. We conduct separate and integrated experiments in the vector-based robotic environments to provide baselines in the centralized PyBullet (Coumans and

Wei, S., Li, C., Seyler, J. and Eivazi, S.
Integration of Efficient Deep Q-Network Techniques Into QT-Opt Reinforcement Learning Structure.
DOI: 10.5220/0011715000003393
In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023) - Volume 3*, pages 592-599
ISBN: 978-989-758-623-1; ISSN: 2184-433X

Bai, 2021) simulated environments. Our contribution lists as follows:

- Benchmark of the QT-Opt performance under different basic robotic value-based environments.
- Attempts of PER Noisy network integration together with QT-Opt and distributional DQN (Q2-Opt), with the purpose to pursue training efficiency.

In this paper, we first introduce the current status and related work. Then the fundamentals of learning techniques and our methods are demonstrated. Finally, the experiments are analysed and compared.

2 RELATED WORKS

Reinforcement learning (RL) from a model-free perspective *Q-Learning* (Watkins and Dayan, 1992) is an algorithm that optimizes the *action state value* (Q-value) by iterative function repeatedly. To solve the exponential explosion of the action dimension, *Deep Q-Learning* (DQN) (Li, 2017) has been proposed, which approximates the q-value function by a deep neural network. Double Q-learning(Hasselt, 2010) algorithm was proposed, which utilizes two value functions to separate action selection and state evaluation processes to exclude the maximization process from the state evaluation. From this the double DQN (Van Hasselt et al., 2016) is derived, which maintains the separate state-action value evaluation network with an update delay, leveraging the target q-network from the original DQN. Clipped Double DQN (Fujimoto et al., 2018) is an extension algorithm to DQN, designed to address Q-value bias arising from the inherent estimation errors in Q-learning.

Cross-Entropy Method (CEM) (De Boer et al., 2005; Kalashnikov et al., 2018) replace the maximization process in action selection by a stochastic optimization over the actions. CEM is simple to implement, has robust properties around local optima in low-dimension problems, and no derivative operation, thus it is ideal for RL algorithms. QT-Opt (Kalashnikov et al., 2018) employed both Clipped double DQN and CEM to reduce the possible overestimation of the q-value. Furthermore, it provides a distributed asynchronous structure to achieve parallel training of one agent network.

Based on distributional DQN technique, the same team presented Q2-Opt(Bodnar et al., 2020), which changes the q-value to the q-distribution and represents the action state relation more comprehensively. Q2-Opt (Bodnar et al., 2020) is a distributional variant of QT-Opt. Two sub-versions of this enhancement are provided: Q2R-Opt based on Quantile-Regression DQN (QR-DQN) and Q2F-Opt based on Implicit Quantile Network (IQN). In the QR-DQN the p-Wasserstein metric (Vaserstein, 1969, 64–72.) replacing the KL-divergence is utilized to describe the difference between two distributions. A Wasserstein metric considers not merely the probability of the outcome events. The distance between them is also important. It is more appropriate to the q-value distribution since we concern more about the underlying similarity of the two distributions than matching their likelihoods exactly. To minimize the p-Wasserstein distance, the distributional Bellman update is projected onto a parameterized quantile distribution over the set of quantile midpoints. Then it utilizes the Quantile regression method (Koenker and Hallock, 2001) for an unbiased stochastic approximation of the distribution. Since the quantile regression loss is not derivable at zero, a quantile Huber-Loss(Dabney et al., 2018b) operates as the loss function in QR-DQN, given $\tau \in [0,1]$ as the selected quantile. QR-DQN could regulate its range of return value without projection support. The number of atoms in the quantile distribution is also manually adjustable.

IQN (Dabney et al., 2018a) is similar to QR-DQN in which IQN also return a parameterized quantile distribution. However, instead of the fixed τ in QR-DQN, the result in IQN is over a randomly generated $\tau \sim U([0,1])$ distorted by a distortion risk measure $\beta : [0,1] \to [0,1]$. The basic network structure of Q2-Opt is similar to Figure 1 in the DNN Model block. A significant improvement has been made by this extension as shown in (Bodnar et al., 2020, Results). In this paper we do not have a preference for risk management, thus we only use the basic risk matrix $\beta(\tau) = \tau$ in Q2F-Opt. The correlation between risk distortion and other improvements could be tested in future works. For the CEM in Q2-Opt, it maximizes the score mapped from distribution vector q which is the output of the network.

To help achieve higher performance various research groups focus on improving the structural composition perspective of RL. For instance, *Prioritized experience replay* (PER) presented by Schaul, Tom, et al. (Schaul et al., 2015), and *Noisy network* suggested by M. Fortunato et al. (Fortunato et al., 2017). PER prioritizes the experience in the replay buffer to increase agents' learning efficiency based on data efficiency enhancement and data correlations during training provided by the original replay buffer structure. On the other hand, the noisy network is aimed at exploration by attaching noise to network weights instead of action selection, which has the purpose to change consistent, potentially complex, state-

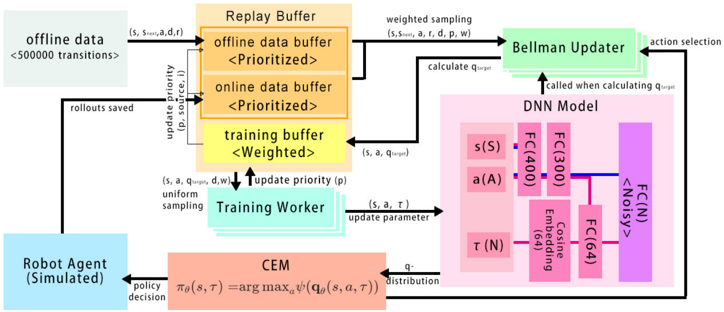


Figure 1: System architecture of our implementation of the extensions based on Q2-Opt. The rollouts simulated in the environment are saved in the replay buffer after each 32-gradient update. The transitions from both data buffers are sampled non-uniformly to the Bellman Updater, which appends the target q-distributions and is pushed to the training buffer which could be asynchronously consumed by Training workers to compute the gradients. The parameters in the model deep neural network are updated by gradient descent. Notice the lines in the DNN Model block, the pink line links the structure of Q2R-Opt and the blue line links the structure of Q2F-Opt.

dependent behavior patterns. Our focus is also these two techniques together with the distributed DQN approach, applied on Q2-Opt, which have been successfully applied in rainbow (Hessel et al., 2018) paper for basic DQN algorithms.

3 METHODOLOGY

Figure 1 shows the architecture of our implementation of the QT-Opt algorithm which has five main blocks: replay buffer, Bellman updater, training worker, DNN model, and CEM. Our modifications are mainly on replay buffer and DNN Model blocks, while other blocks merely attached several adaptions. With the same operation process as vanilla QT-Opt described in the preliminary section, our system run in single computing machine compare to distributed version. Moreover, we also implemented Q2-Opt based on the description in (Bodnar et al., 2020).

3.1 Prioritized Replay

For an experience replay priority method extension to QT-Opt, the data structure in the replay buffer is replaced by a sum-segment-tree (Berg et al., 1997). The priority value is set as the node value in the tree structure. Considering the replay buffer structure in QT-Opt, the PER method could be implemented in data- or training buffer or both.

The sampling process in a prioritized replay buffer started by uniformly sampling a batch of random values in the range [0,1). With the random batch multiplies by a sum over all priorities, we could use the retrieve function in the sum-segment tree to get the batch of the final sampled index. The transitions are picked from the buffer and sent together with the index and corresponding weights calculated by corresponding priority as in the ensuing Equation, where p_i is the priority of i-th transition in the buffer:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{\mathcal{P}(i)}\right)^{\beta} \tag{1}$$

$$\mathcal{P}(i) = \frac{p_i^{\alpha}}{\sum_k p_k^{\alpha}} \tag{2}$$

In our implementation, data buffer prioritization is a conversion from the original PER definition: using segment trees for importance sampling, picking out and updating the transition data by indices. For training buffer, it is more complex. The training buffer has to save the data source and corresponding indices whether prioritized or not since we need to update the priority value into the original on- or offline data buffer together with the Bellman update. Priority values are updated after each training batch and the IS weight should be recalculated in each mini-batch. Therefore, the training buffer also needs to calculate the current weight with the locally updated priority buffer if the gradient update per Bellman-update (gpb) is larger than 1. To simplify the process we fixed gpb as 1 in our experiments.

Under the circumstances, in which the training buffer is not prioritized, the sample weights are from the data buffer importance sampling while Bellman updating. We created a replay buffer subclass for this kind of training buffer, which saves the corresponding additional features and split the priority update batch for two data buffers. If prioritized, the sample weights are calculated from the training buffer itself. In other words, when both types of buffers are prioritized, our implementation is not saving the sample weights from the data buffer, only sampling non-uniformly. To update the priority which also adapts to distributional q-network results, considering the suggestion in Rainbow paper (Hessel et al., 2018, 'The Integrated Agent'), we are using the element-wise loss value as the priority. Two hyper-parameters are needed in the PER: α is denoted as the prioritization scale, which is fixed as 0.5 in our training. β determines the decay of probabilities, which is fixed as linearly growing from 0.4 to 1 as suggested.

3.2 PAL

Regarding the time penalty and RAM consumption with importance sampling, due to the segment-sum-tree, making PAL(Fujimoto et al., 2020) as an alternative to PER is worth considering. As a mirrored loss function of LAP(Fujimoto et al., 2020), PAL has an equivalent expected gradient to a non-uniformly sampled replay buffer using importance sampling to avoid its bias. In other words, PAL is a loss function alternative to a prioritized replay buffer implementation.

We integrated PAL to QT-Opt by adding an alternative loss function instead of Huber-Loss or MSE. While adapting to the q-distribution network, we change the loss function in the original Q2-Opt by replacing the Huber-loss in ρ with PAL-loss. The final loss function for PAL attached Q2-Opt is as shown in Equation 3.

$$
\begin{aligned}
&\mathcal{L}_{PAL}(u) = \frac{1}{\lambda}\begin{cases} \frac{1}{2}|\delta_{TD-Error}|^2 & \text{if } |\delta_{TD-Error}| \leq \kappa \\ \frac{|\delta_{TD-Error}|^{\alpha+1}}{\alpha+1} & \text{otherwise} \end{cases}\\
&\lambda = \frac{\sum_j^N \max(|\delta_{TD-Error}(j)|, 1)}{N}\\
&\rho_\tau(u) = |\tau - I_{\delta_{TD-Error}<0}| \cdot \mathcal{L}_{PAL}(\delta_{TD-Error}) \qquad (3)\\
&I_{x<0} = \begin{cases} 1 & x < 0 \\ 0 & x \geq 0 \end{cases}
\end{aligned}
$$

(Dabney et al., 2018b; Fujimoto et al., 2020) The definition of the One hyper-parameter is needed in this improvement implementation: α as in PER. In our experiments, the value is fixed at 0.5.

3.3 Noisy Nets

A noisy layer is appended after the feature network and other extension implementations, as the final output layer. In our experiments, a noisy dense layer is defined as a subclass of standard linear dense layer, adding $\mu_w, \mu_b, \sigma_w, \sigma_b$ as trainable variables, and generating $\varepsilon_w, \varepsilon_b$ as random variables. As shown in the original paper (Fortunato et al., 2017), the function realized by the fully connected noisy linear layer is:

$$y = (\mu_w + \sigma_w \odot \varepsilon_w)x + (\mu_b + \sigma_b \odot \varepsilon_b) \qquad (4)$$

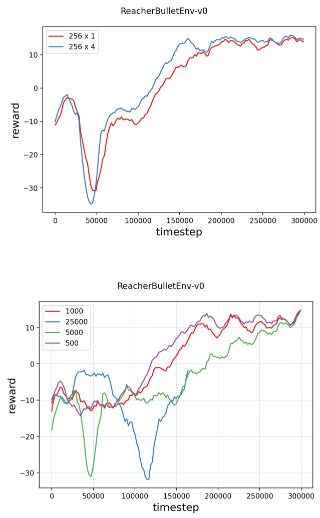


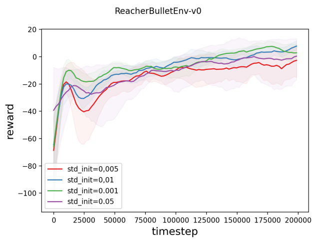
Figure 2: Evaluation reward as a function of the training time-step on *ReacherBulletEnv-v0* environment. The data is collected from on-policy training results. The plot on the top shows the training efficiency difference caused by different training batch sizes and the bottom plot shows the training efficiency difference caused by variant training buffer sizes. All curves in the plot are trained with the same random seed 0.

To generate random variables, we choose factorized Gaussian noise as suggested in (Fortunato et al., 2017) to reduce compute time. If we use independent Gaussian noise, since we are using a single-thread qt-opt-based agent, the computational overhead for our network structure is too large for our computing resource.

Besides the implementation in the DNN-model, we have to reset its noise every training step to activate a noisy network. Thus we need to label all noisy layers by network creation and call their reset functions at every gradient step. As shown in Figure 1, in our implementation, when a noisy network is on, the last layer in the current network is changed into a noisy layer. One hyper-parameter is needed in the noisy network, the initial standard deviation value used by initializing the σ variables.

4 EXPERIMENT

We tested several improvement techniques for QT-Opt, including PER, noisy network, and distributional DQN. Simulation environments with continuous control tasks generated by *PyBulletGym* engine are used in the training process. We used *ReacherBulletEnv-v0* environment from the default setting as well as few other robotic environments limited to time and com-



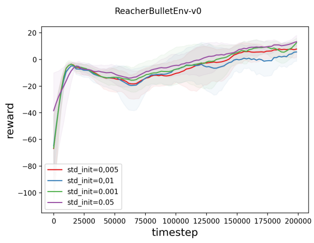


Figure 3: Evaluation reward as a function of the training time-step on *ReacherBulletEnv-v0* environment. The data of the top plot is collected from on-policy training results and the data in the bottom plot is collected from mixed-policy training results. All curves are from noisy attached QT-Opt agents with different initial standard deviations. The red, blue, green, and purple curves correspond to an agent with an initial standard deviation equal to 0.005, 0.01, 0.001, and 0.05. All curves in the plot are averaged by three separate runs with three different random seeds (133156, 254306, 369070). The half-transparent area shows the range between the max and min value at this time-step in three runs.

puting power costs.

To evaluate the algorithm performance, we are using the reward value from the default tasks. One evaluation step is taken after each 2048 training time step. For each evaluation, we run 10 tests with the temporal agent and collect the averaged reward value per step. One test means a complete run until the task is done or a maximum step (150 for "Reacher" and 1000 for other environments) is reached. For reproducibility of the results, random seeds are set for TensorFlow, NumPy, and gym environments. To ensure the validity of the test, Three full pieces of the training run with different random seeds are taken for all setups. The final reward values are taken from the average among three tests and the range (max and min value among the three tests) are also saved. The curves in the following plots have been smoothed by a Savitsky-Golay filter with a window size equal to 15 and an order of 4. Ranges around the curve show the range between max and min values over three tests, both values are smoothed with the same filter.

Compared to pure on-policy training, we are providing mixed-policy training as explained in the original QT-Opt paper (Kalashnikov et al., 2018). A mixed policy starts by training with offline data sets and keeps collecting online data for one training rollout for each 32 training steps. The online data percentage grows linearly from 0% to 50% in the first 50% training iterations. The offline data used in mix-policy experiments are generated by pure online training. All 500k transitions in the online data buffer after a full training run (300k steps for Reacher, 1M for other environments) are saved, which are loaded afterwards for a mixed training as offline data. No priority values are saved or passed on by creating a new offline data buffer.

Firstly, we tuned our model with different training buffer sizes and training batch sizes, as shown in Figure 2. For the training buffer, the closer to the Bellman-update batch size the better result of the test gets. The gradient update batch size is more straightforward: the larger it is, the better it gets, and it reaches its limit depending on the training buffer size. Therefore in the following experiments, we make the training buffer size equal to the gradient update batch size. Besides, in our case, we consider more of a single QT-Opt rather than a distributed asynchronous QT-Opt as our experiment target. So we set the training buffer as the same size as the Bellman-update batch to provide a single model. Hence these three sizes are selected equally in the following experiments.

With noisy network standard variation value initialized differently, the result does not catch up to the vanilla training with a explore probability of 0.02. But with a specific standard deviation initial value, the noisy network could reach a similar result. If the initial standard-deviation value is more than 0.05 or less than 0.0001, the agent does not end up with a positive reward. To see how this hyperparameter affects the result and to choose the best for the rest of the training, we run a small test with 2/3 steps of full training for four different initial standard deviation values within the range of 0.05 and 0.0001. Each value for three times with inconsistent random seeds. The results are shown in Figure 3. The optimal initial values for on and mixed policy are different even though they are trained with the same parameters and random seeds. We use 0.01 for on-policy training and 0.05 in mixed-policy training as the default initial standard deviation value for a noisy network in the following experiments.

From Figure 4, we can see that the PER techniques have no progressive results compared to the original QT-Opt algorithm with both policies on all tested environments excluding *ReacherBulletEnv-v0*. However, PER technique does not hurt the learning ability of the QT-Opt agents in any of the tasks.

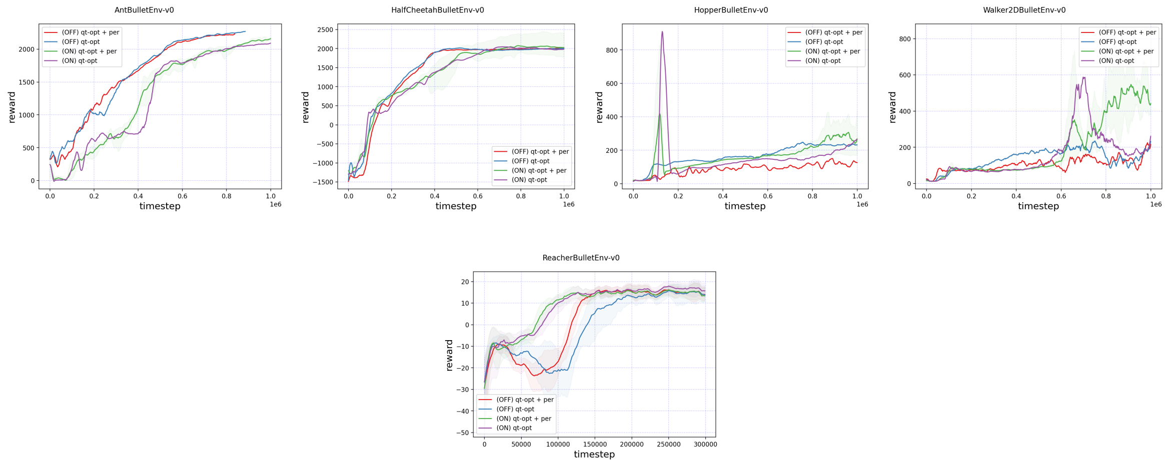


Figure 4: Evaluation reward as a function of the training time-step on *AntBulletEnv-v0*, *HalfCheetahBulletEnv-v0*, *HopperBulletEnv-v0*, *Walker2DBulletEnv-v0*, and *ReacherBulletEnv-v0* environment. The data is collected from mixed-policy training results. The red curve is with PER attached QT-Opt agent trained in mixed-policy. The blue curve is with vanilla QT-Opt agent trained in mixed-policy. The green curve is with PER attached QT-Opt agent trained in on-policy. The purple curve is with the QT-Opt agent trained in on-policy. All curves in the other plots are averaged by three separate runs, while all curves in the plot are trained with the same random seed.

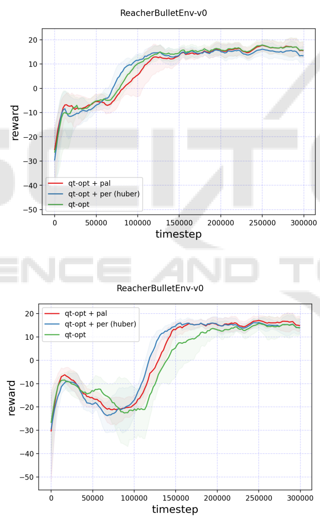


Figure 5: Evaluation reward as a function of the training time-step on *ReacherBulletEnv-v0* environment. The data in the top plot is collected from on-policy training results and the data in the bottom plot is collected from mixed-policy training results. The red curve is with PAL attached QT-Opt agent. The blue curve is with PER attached QT-Opt agent. The green curve is the vanilla QT-Opt agent. All curves in the plot are averaged by three separate runs with three different random seeds (133156, 254306, 369070). The half-transparent area shows the range between the max and min value at this time-step in three runs.

Notice that multi-pivot robot tasks have an obvious advantage in mixed-policy training in evaluation rewards over on-policy training. However, in the *ReacherBulletEnv-v0* condition, it is the opposite. Additionally, in mixed-policy training, the PER-attached QT-Opt agent has a more pronounced effect than the vanilla one. Therefore the following experiments mostly proceed in mixed-policy. For *HopperBulletEnv-v0* and *Walker2DBulletEnv-v0* tasks using humanoid robot structures, none of the agents tend to a stabilized state.

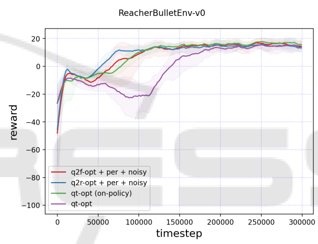


Figure 6: Evaluation reward as a function of the training time-step on *ReacherBulletEnv-v0* environment. The data is mainly collected from mixed-policy training results: The blue curve is with PER and noisy attached Q2R-Opt agent, the red curve is with PER and noisy attached Q2F-Opt agent, and the purple curve is vanilla QT-Opt agent. Except for the green curve, which is collected from on-policy with vanilla QT-Opt agent. All curves in the plot are averaged by three separate runs with three different random seeds (133156, 254306, 369070). The half-transparent area shows the range between the max and min value at this time-step in three runs.

As shown in Figure 5, PAL has almost as good performance as PER in mixed-policy and could reach a slightly higher final mean reward. Compared to the vanilla QT-Opt, both PAL and PER could provide more learning efficiency. Meanwhile, with on-policy tests, it performs even slower than vanilla QT-Opt.

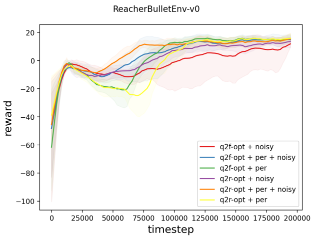


Figure 7: Evaluation reward as a function of the training time-step on *ReacherBulletEnv-v0* environment. The data is collected from mixed-policy training results. The red curve is with noisy attached Q2F-Opt agent. The green curve is with PER attached Q2F-Opt agent. The blue curve is with PER and noisy attached Q2F-Opt agent. The purple curve is with noisy attached Q2R-Opt agent. The yellow curve is with PER attached Q2R-Opt agent. The orange curve is with PER and noisy attached Q2R-Opt agent. All curves in the plot are averaged by three separate runs with three different random seeds (133156, 254306, 369070). The half-transparent area shows the range between the max and min value at this time-step in three runs.

Considering its operational efficiency, PAL could be a proper alternative to PER when no further changes in loss function are required for an integrated agent. Otherwise, it needs more theoretical proof for the unbiasedness of the formula.

We begin with the final results from the fully integrated agent, comparing with the performance of vanilla QT-Opt. As shown in Figure 6, with both PER and noisy network switched on as the base algorithm, we add Q2-Opt techniques separately. We found that Both Q2-Opt algorithms have better results in mixed-policy tests, while comparing with the on-policy vanilla QT-Opt they even maintains a proper advantage. Comparing the results based on two Q2-Opt variants, Q2R-Opt has better collaboration with PER and noisy network methods regarding iteration efficiency.

Then we evaluated the two techniques separated and integrated on Q2-Opt as in Figure 7. Compared to pure Q2R-Opt and pure Q2F-Opt, these two algorithms attached with PER make a more obvious efficiency increase than it has made in vanilla QT-Opt. For Q2R-Opt, PER increases more with the ascent speed of learning, since the curve is steeper by increasing. Meanwhile, for Q2F-Opt, PER only increases the efficiency of the training steps by starting the rewarding climb early. Its learning curve has almost the same, even more, gradual slope. We could say that PER and Q2-Opt have good compatibility. Meanwhile, noisy exploration on Q2R- and Q2F-Opt speeds up the training by smoothing the learning at the start, easily avoiding the slump in the beginning but hardly being stable at around 50% of the time steps. Noisy has a better correlation with Q2F-Opt, but it also adds a large scale of instability to the evaluation results. So both PER and Noisy Network adds to the base and could increase the speed of learning. When both of the techniques are switched on, the learning curve could be increased furthermore.

Lastly, we extend the experiments with other *Pybullet* simulated mujoco-like environments like *AntBulletEnv-v0*, all the agents have demonstrated their ability to learn and solve the tasks (Figure 8). In these more complex environments, without hyperparameter tuning, PER and Q2-Opt variations do not harm the learning abilities of the QT-Opt agent. However, these extensions could not overcome QT-Opt's weakness in humanoid-like robot fields.

5 DISCUSSION

In this paper, we have re-implemented the QT-Opt and Q2-Opt (Kalashnikov et al., 2018; Bodnar et al., 2020) algorithm with additional extension: PER (Schaul et al., 2015) and Noisy network (Fortunato et al., 2017) techniques. We conducted several experiments separately on each extension to validate their optimization effect on the algorithm, and proceed on integrated tests combination of all techniques. Moreover, we tested use of PAL (Fujimoto et al., 2020) instead of original PER. In the Reacher environment by Pybullet simulation, we observe an obvious improvement in the learning process for a QT-Opt agent with integrated methods. Meanwhile, in other related robotic environments we confirmed no harm in time-step-efficiency when attaching these techniques on qt-opt. Despite relatively considerable results, we still observe few drawbacks with these extensions. First, PER is not very cost-effective as an improvement to QT-Opt, while PAL as its simplify equivalent have lost most of the efficiency advantage. Second, the Noisy Network extension is not suitable for the current version of QT-Opt, and proof for the compatibility of this extension with QT-Opt is needed further research. Finally, for the Q2-Opt, the performance of its agent in a simple task is not as evident as it has in a vision-based complex task like grasping. Moreover, due to its expansion of the computing network, its time-consuming, and memory-occupying problems are evident. Nevertheless, its combined effect with PER is proven worth consideration.

From our experiments and assumptions, PER with distributional QT-Opt needs more trimming. We suggest two ways to achieve this. One is to find the best way to apply data sampling weight to training sampling, while the other is to provide theoretical support to PAL on Q2-Opt. Also, future studies could in-

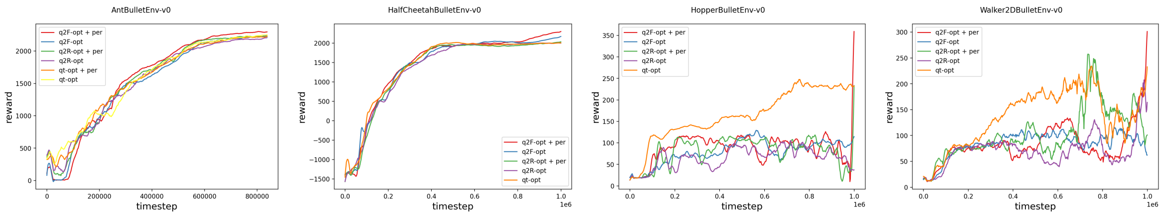


Figure 8: Evaluation reward as a function of the training time-step on *AntBulletEnv-v0*, *HalfCheetahBulletEnv-v0*, *HopperBulletEnv-v0*, and *Walker2DBulletEnv-v0* environment. The data is collected from mixed-policy training results. The red curve is with PER attached Q2F-Opt agent. The blue curve is with the Q2F-Opt agent. The green curve is with PER attached Q2R-Opt agent. The purple curve is with the Q2R-Opt agent. The orange curve is with PER attached QT-Opt agent. The yellow curve is with vanilla QT-Opt agent. All curves in the plot are trained with the same random seed 254306. The half-transparent area shows the range between the max and min value at this time-step in three runs. Notice total training time steps in Ant is 839,680, which is slightly lower than in other environments.

vestigate the conflict between Noisy net and QT-Opt. Though Noisy net is not stable in the current version of QT-Opt, there still exists research significance for this phenomenon theoretically.

REFERENCES

Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR.

Berg, M. d., Kreveld, M. v., Overmars, M., and Schwarzkopf, O. (1997). Computational geometry. In *Computational geometry*, pages 1–17. Springer.

Bodnar, C., Li, A., Hausman, K., Pastor, P., and Kalakrishnan, M. (2020). Quantile qt-opt for risk-aware vision-based robotic grasping. In *Proceedings of Robotics: Science and Systems*, Corvalis, Oregon, USA.

Coumans, E. and Bai, Y. (2016–2021). Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

Dabney, W., Ostrovski, G., Silver, D., and Munos, R. (2018a). Implicit quantile networks for distributional reinforcement learning. In *International conference on machine learning*, pages 1096–1105. PMLR.

Dabney, W., Rowland, M., Bellemare, M., and Munos, R. (2018b). Distributional reinforcement learning with quantile regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

De Boer, P.-T., Kroese, D. P., Mannor, S., and Rubinstein, R. Y. (2005). A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., et al. (2017). Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*.

Fujimoto, S., Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR.

Fujimoto, S., Meger, D., and Precup, D. (2020). An equivalence between loss functions and non-uniform sampling in experience replay. *Advances in Neural Information Processing Systems*, 33.

Hasselt, H. (2010). Double q-learning. *Advances in neural information processing systems*, 23.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.

Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., Quillen, D., Holly, E., Kalakrishnan, M., Vanhoucke, V., et al. (2018). Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on Robot Learning*, pages 651–673. PMLR.

Koenker, R. and Hallock, K. F. (2001). Quantile regression. *Journal of Economic Perspectives*, 15(4):143–156.

Li, Y. (2017). Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Vaserstein, L. N. (1969). Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii*, 5(3):64–72.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.