AI51201 Final Report

20225231 Kangbeen Lee

June 2023

1 Description of Implemented DQN

1.1 Parameter Setting

Through all deep reinforcement learning models and all environments, mentioned parameters from now on are fixed during training. Learning rate is set to 0.0005, training episodes is set to 2000, maximum steps per episode is set to 1000, minimum epsilon is set to 0.01, epsilon decaying rate is set to 0.995, and (hard) target network update period is set to 300. The reason why training episode, minimum epsilon, and epsilon decaying rate is set such like that is explained below.

1.1.1 The Number of Training Episodes

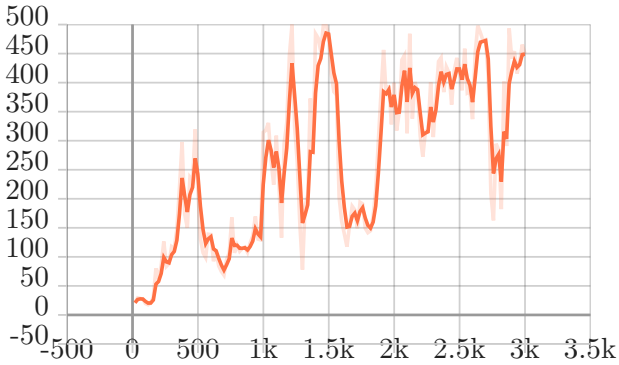


Figure 1: Score result of vanilla DQN with 3000 training episodes

From the above Figure 1, training with 3000 episodes is too much and may cause overfitting to deep learning model. So, 2000 is chosen as training episodes.

1.1.2 Minimum Epsilon and Epsilon Decaying Rate

To choose proper minimum epsilon and epsilon decaying rate, two tests are conducted with minimum epsilon 0.01 and 0.001. From Figure 2, deep blue curve is the training result of minimum epsilon 0.01 which shows more better performance than the one with 0.001. With regard to epsilon

decaying rate, I find that enough training must be proceeded after converging to minimum epsilon. Experimentally, 0.995 is chosen as proper epsilon decaying rate.

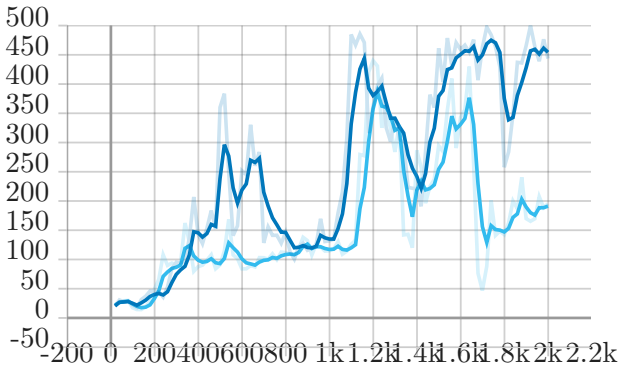


Figure 2: Score results of vanilla DQN with mininum epsilons 0.01 and 0.001

1.2 Description of Code Structure

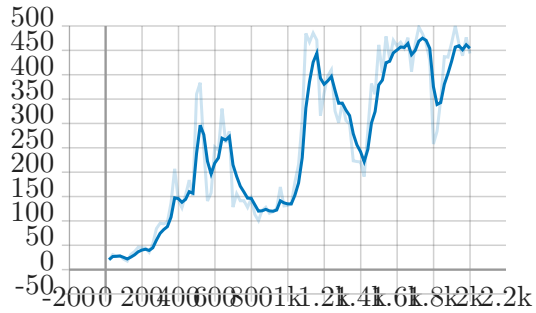
Code for this final project are written with Pytorch. My code roughly consists of four parts: Deep Q-network Class, Replay Buffer Class, Trainer Class, and main function.

1.2.1 Deep Q-network Class

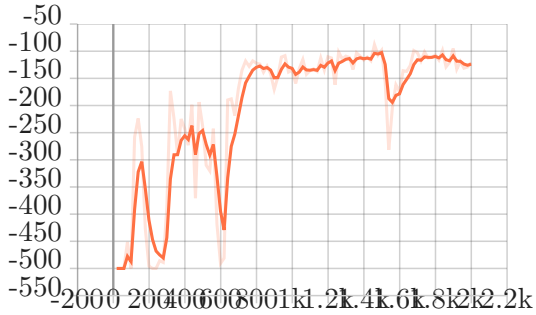
There are two types of deep Q-network: DQN [1] and Dueling-DQN [2]. First one is most basic deep Q-neural network which is composed of input layer, hidden layer, and output layer. All layers are linear and dimension of hidden layer is set to 128, but you can change with argument "–mlp_hidden_dim". The second one is Deuling-DQN. It seems similar to DQN, but there exist value stream and advantage stream rather than last two linear layers in DQN. You can select which type of neural network you will use with argument "–network_type".

1.2.2 Replay Buffer Class

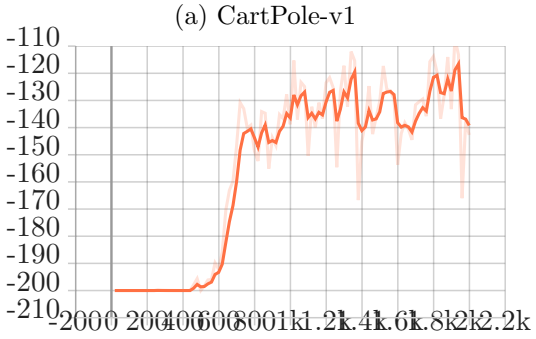
DQN and its variants are off-policy algorithms. So, they can utilize experience replay. This experience replay is necessary to avoid sample correlation and make training more robust with respect to training samples. Replay buffer in a lot of open-source codes is usually implemented with deque data-structure in python collection module. However, I make a replay buffer class in a different way and this is described in Section 2. There is also advanced version of replay buffer: Prioritized Replay Buffer (PER) [3]. With this advanced buffer, deep Q-neural network can be trained fast because PER gives priority to transition samples which has large magnitude of TD-error. I refer to https://github.com/keep9oing/DQN-Family and https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py to make PER class.



(a) CartPole-v1



(b) Acrobot-v1



(c) MountainCar-v0

Figure 3: Score results of vanilla DQN in each environment

1.2.3 Trainer class

In Trainer class, reinforcement learning procedure with chosen environment and parameters is described. There are several member functions and "learn()" function is the most importance one among them. During the training, average score per 20 episodes is stored to visualize the training results with tensorboard. This average period was set to 100 episodes in skeleton code, but it was too large to show the flow of the training.

1.2.4 main function

The all parameters used in reinforcement learning and training are written with argparse module. For simple implementation of the code, there are just few lines in main function. After setting parameters said above, Trainer object will train the agent.

2 Errors and Solutions

2.1 UserWarning: Creating a tensor from a list of numpy.ndarrays

If replay buffer is implemented with deque from collection module, then it may cause above warning. So, I make a replay buffer with numpy rather than using deque. For more details, please see my code.

2.2 torch no grad to TD target

To train deep Q-network, Q value, $Q(s, a)$, for sampled batch transitions must be close to TD-target. This TD-target must be untracked from computational graph in Pytorch with torch.no_grad() or detach() functions.

3 Performance of Vanilla DQN

Figure 3 is about average score curves in each environments with vanilla DQN. For this vanilla DQN, hard target network update and reward clipping are applied. In all Figures 3a, 3b, and 3c, the training results are very unstable and their convergence speeds are also slow. Additionally, vanilla DQN is not able to get high enough score both in Acrobot-v1 and MountainCar-v0 environments.

4 Performance Comparison in CartPole-v1 Environment

4.1 Reward clipping

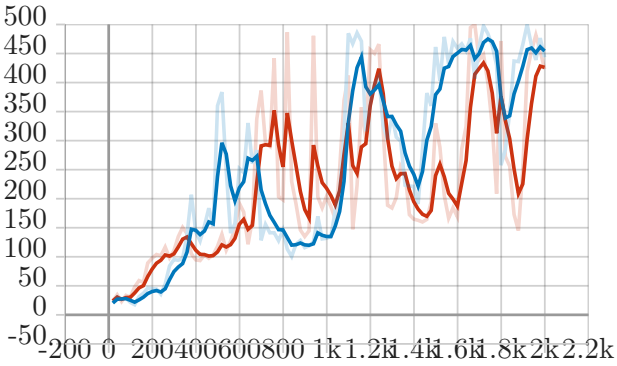


Figure 4: Score results of DQN with reward clipping and unclipping

In all environments, CartPole-v1, Acrobot-v1 and MountainCar-v0, reward is given as -1 or +1 at every time step. The magnitude of reward is small enough, but I train agent with one hundredth of the original reward. In Figure 4, blue curve is the result with reward clipping, whereas brown curve is not. Result shows that clipping reward gives more stability in training. So, clipping reward applied in every algorithms including vanilla DQN.

4.2 Soft Target Update

Rather than hard target network update in [1], DQN with soft target network update [4] shows more stable training results. In Figure 5, pink curve is the result of soft target update and blue is for hard target update. Even if the result with hard update shows more higher average score at the

end of the episode, the overall training stability with soft update cannot be negligible. So, soft target update is applied to all DQN variants except for vanilla version.

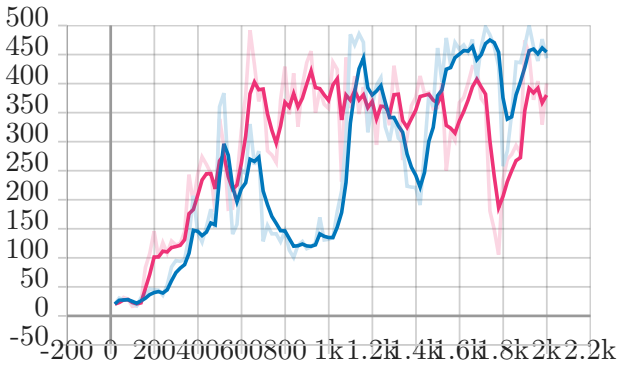


Figure 5: Score results of DQN with hard update and soft update

4.3 DQN-variants: Double-DQN, Dueling-DQN, Dueling Double-DQN

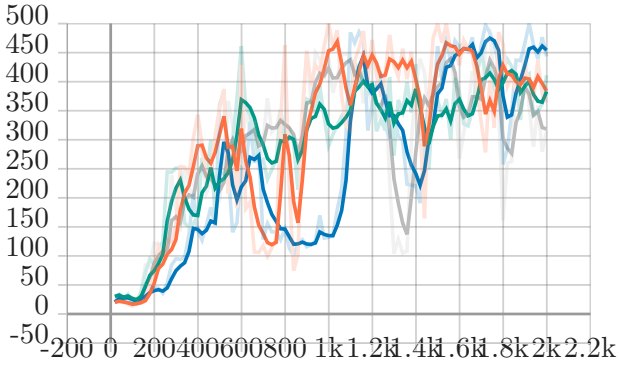


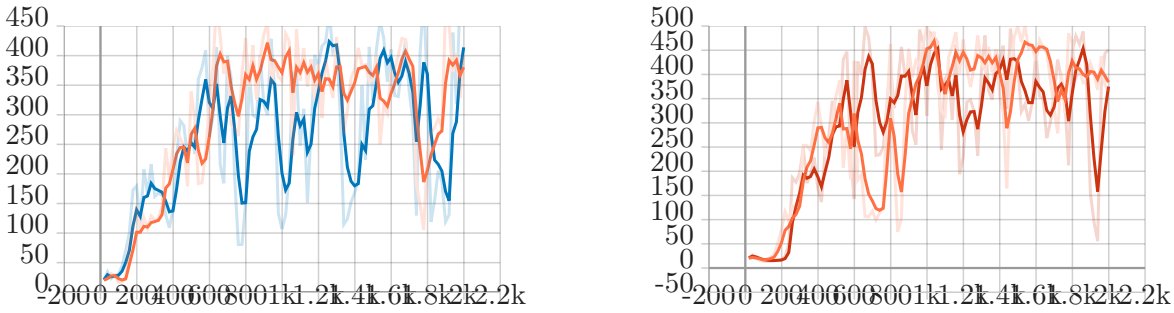
Figure 6: Score results of DQN and its variants

After DQN, a lots of its variants emerged such as Double-DQN (DDQN) [5], Dueling-DQN [2], Dueling Double-DQN (D3QN), and so on. DDQN, Dueling-DQN, and D3QN are chosen to compared to vanilla DQN in this project. In Figure 6, blue curve is the result of DQN, green curve is for DDQN, gray curve is for Dueling-DQN, and orange curve is for D3QN. The curves of DQN and D3QN show most zig-zag shape compared to the others. Dueling-DQN's curve is dramatically decreasing around 1.2k episode, but immediately increasing. DDQN shows the most stable average score curve during training.

4.4 Prioritized Replay Buffer (PER)

As explained already in section 1.2.2, PER gives priority to transition sample with high magnitude of TD-error to train neural network with more precious transition sample rather than opposite one.

However, contrary to my expectation, training with PER seems to make learning unstable. In Figure 7a, blue curve is the result of DQN with only soft update and orange curve is for DQN with PER and soft update. Additionally, in Figure 7b, orange curve is the result of D3QN with only soft update and brown curve is for D3QN with PER and soft update. In both experiments, PER makes average score curve more zig-zag shape compared to without PER, even if make D3QN get high enough score fast (not meaning that converge to high score).



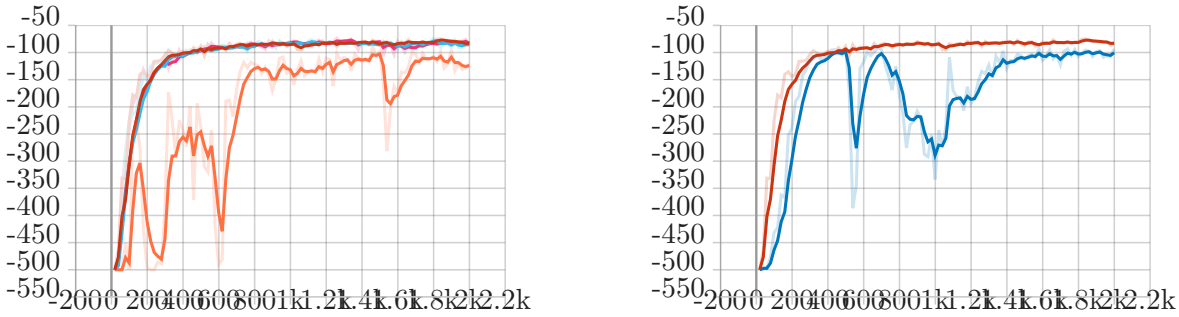
(a) DQN with soft update

(b) D3QN

Figure 7: Score results of DQN and D3QN with PER and without PER

5 Performance Comparison in Other Environments

5.1 Acrobot-v1



(a) Score results of vanilla DQN and its variants

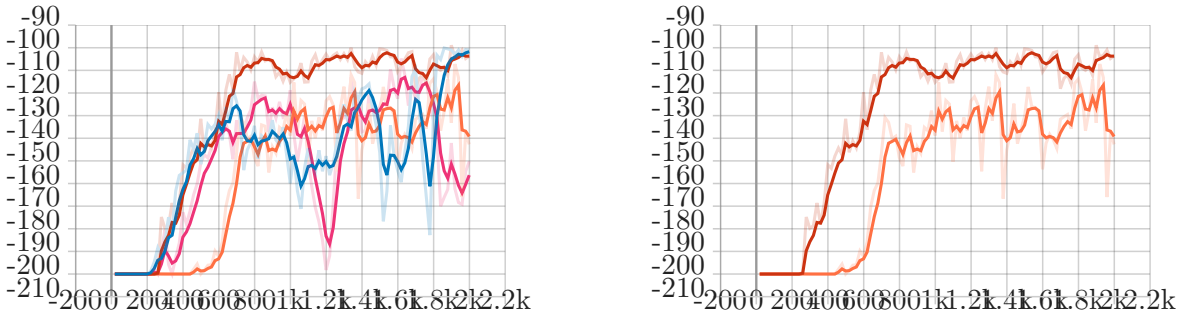
(b) Score results of D3QN with PER and not

Figure 8: Score results in Acrobot-v1

In Acrobot-v1 environment, vanilla DQN and its variants are trained with same number of training episodes, 2000, as in CartPole-v1. All models show fast convergence to high enough score compared to CartPole-v1. In Figure 8a (orange curve is result of vanilla DQN), all DQN variants (DDQN, Dueling-DQN, and D3QN) show very good training results with respect to convergence speed, average score, and training stability. Also there training results are very similar each other. However, contrary to

DQN variants, vanilla DQN is unable to converge high enough score and shows poor training stability. To analyze the effect of PER, additional experiment with PER also conducted. In Figure 8b, brown curve is the result of D3QN without PER and blue curve is the result without PER. This experiment implies that training with PER causes more instability.

5.2 MountainCar-v0



(a) Score results of vanilla DQN and its variants

(b) Score results of vanilla DQN and DDQN

Figure 9: Score results in MountainCar-v0

In Figure 9, orange curve is the result of vanilla DQN, brown is for DDQN, pink is for Dueling-DQN, blue is for D3QN. The training results of vanilla DQN, Duelingn-DQN, and D3QN are very unstable and only D3QN gets high enough score at the end of the training episode among them. However, contrary to others, DDQN shows very fast convergence speed and stable learning curve. In Figure 9b, performance of DDQN overwhelms the performance of vanilla DQN.

6 Conclusion

In this project, performance of vanilla DQN and its variants (DDQN, Dueling-DQN, and D3QN) are tested in discrete state and action environments (CartPole-v1, Acrobot-v1, and MountainCar-v0). For better performance, additional techniques are applied such as reward clipping, soft target update, and prioritized experience replay (PER). Reward clipping and soft target update increase stability to training, whereas PER makes training more unstable. With respect to DQN variants, DDQN shows best performance compared to others. As a result, DDQN with reward clipping and soft target update is best model in all environments and overwhelms vanilla DQN. For more details about my code, please see https://github.com/kangbeenlee/dqn-family-pytorch.git

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level

control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[2] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[3] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

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

[5] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.