Implementation and Analysis of Reinforcement Learning Algorithms

Darshan Jain jain.d@husky.neu.edu

Zimou Gao gao.zim@husky.neu.edu

Xupeng Zhu zhu.xup@husky.neu.edu

Abstract

In our project, we implemented different algorithms of Deep Q-Network (DQN) and Policy Gradient (PG) methods in Classic control, Atari and Box2D environment of Gym to analyse their performance.
After proper steps of training, the agent learned and improved its performance with most algorithms we implemented. Basically, algorithms of PG method can learn faster than DQN in our test environments and different algorithms will lead to different performance and learning curves. We will discuss the comparison result in depth in the following part.

Introduction

DQN and PG methods are frequently used in modern Reinforcement Learning. Based on these two methods, people derived a bunch of algorithms to train the agent. We are curious about how different algorithms with same hyperparameters will perform in different environments.

The algorithms we are using:
Deep Q-Network method: DQN with Target Network and Experience Replay, Dueling-DQN, Double-DQN.
Policy Gradient method: REINFORCE, REINFORCE with baseline, Actor-Critic, Deep Deterministic Policy Gradient.

The Gym environment we chose:
Atari games: BreakOut
States: RGB image of the screen, with position of paddle, ball, and blocks.
Action: Discrete, push cart to the left or right.
Reward: As the games score changes, positive reward for getting the blocks and negative reward for losing a ball.
Neural network: Use CNN to get function approximation value from image states.

Classic Control and Box2D Environments: CartPole, MountainCar, Acrobot, LunarLander
States: Vector states, including all the information of current status.
Action: Discrete, based on the game policy.
Rewards: Based on the game policy.
Neural network: Used Fully Connected neural network to get function approximation value from vector states.

Background

To solve the desired MDP problem we can learn estimates of the optimal value of each action, defined as the expected sum of future rewards when taking that action and following the optimal policy thereafter. Under a given policy π, the true value of an action in a state s is

$$Q_\pi(s,a) \equiv \mathbb{E}\left[R_1 + \gamma R_2 + \ldots \mid S_0 = s, A_0 = a, \pi\right],$$

where $\gamma \in [0, 1]$ is a discount factor that trades-off the importance of immediate and future rewards. The optimal value is then $Q_*(s,a) = \max_\pi Q_\pi(s,a)$.
An optimal policy is easily derived from the optimal values by selecting the highest valued action in each state. Estimates for the optimal action values can be learned using SARSA, Expected SARSA and Q-learning , all of them are a form of temporal difference learning method. Although in reality, most of the problems are too large to learn all action values in all states separately. This would result in increase in the amount memory and time required explore each state as required in a Q-table making in unrealistic. Instead, we explore the idea of approximating these Q-values with machine learning models like Neural Network, where we can learn a parameterized value function Q(s, a; θt).
The standard Q-learning update now for the parameters after taking action A_t in state S_t and observing the immediate reward R_{t+1} and resulting state S_{t+1} is then

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(Y_t^{\mathrm{Q}} - Q(S_t, A_t; \boldsymbol{\theta}_t))\nabla_{\boldsymbol{\theta}_t} Q(S_t, A_t; \boldsymbol{\theta}_t)$$

Related Work

Deep Q Network (Mnih et al., Playing Atari with Deep Reinforcement Learning) extended reinforcement learning play games in high dimensional sensor inputs, such as playing multiple Atari games. However, DQN suffers from several limitations. The first limitation is to learning a proper policy, DQN needs millions of episode to train in most of the mentioned Atari games. Schaul et al. (Prioritized experience replay) prioritized the experience in the replay buffer according to the TD error, then designed

weight for each TD update to correct the bias lead by the prioritization. This replay buffer made full use of the TD error thus speed up the training. However, the experience replay buffer is hard to implement, so we did not implemented it. Wang et al. proposed a dueling structure in DQN that used an advantage function. The advantage function is state-dependent and can generalize the learning process, which achieved better performance than DQN.

Another limit of DQN is the maximum bias introduced by Q learning. Hasselt et al. applied double Q-learning in DQN to avoid bias. Comparing to DQN, Hasselt's neuralnet avoid return-drop after long training, and is more stable.

Moreover, DQN is essentially table-base reinforcement learning algorithm, the action is discrete. Though continuous value can be discretized, the resolution is limited. On the other hand, policy gradient based algorithms are innate with the capability to estimate the distribution of the action, which can solve this problem. There are various variants of policy based algorithms, which can generate action continuously.

Project Description

Deep Q-Networks

In Deep Reinforcement Learning algorithms we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. Two important components of the Deep Q-Networks and its extensions algorithm is the use of target network and experience replay. The target network, with parameters θ^-, is the same as the online network except that its parameters are copied every τ steps from the online network, so that then $\theta^-_t = \theta_t$, and kept the same on all other steps.

The target used by DQN is then

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-) .$$

For experience replay, the transitions are stored in a memory bank and then randomly sampled to update the network. Both the target network and the experience replay play a key role in improving the performance of the algorithm.

Dueling DQN

Dueling DQN uses a specialized Dueling Q Head in order to separate Q to an A (advantage) stream and a V stream as shown in Figure 1. Adding this type of structure to the network head allows the network to better differentiate actions from one another, and significantly improves the learning.

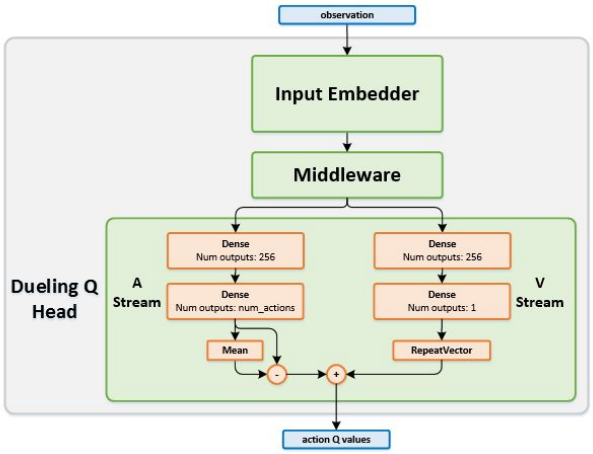

Figure 1: Network Structure of Dueling DQN

In many states, the values of the different actions are very similar, and it is less important which action to take. This is especially important in environments where there are many actions to choose from. In DQN, on each training iteration, for each of the states in the batch, we update the `Q` values only for the specific actions taken in those states. This results in slower learning as we do not learn the Q values for actions that were not taken yet. On dueling architecture, on the other hand, learning is faster -- as we start learning the state-value even if only a single action has been taken at this state.

Double DQN

The idea of Double Q-learning is to reduce overestimations by breaking down the max operation in the target into action selection and action evaluation. With the use of target network in the DQN architecture, there would be no need to introduce additional networks. We therefore propose to evaluate the greedy policy according to the online network, but using the target network to estimate its value. In reference to both Double Q-learning and DQN, we refer to the resulting algorithm as Double DQN. Its update is the same as for DQN, but replacing the target Y_t^{DQN} with:

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-) .$$

The goal is to get most of the benefit of Double Q-learning, while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead.

REINFORCE

Reinforce algorithm is based on policy gradient method. The general idea of REINFORCE is to maximize the value

function over the initial state distribution. This can write down as objective function:

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0),$$

From the Policy Gradient Theorem (Sutton et al., 1999), the performance gradient with respect to the policy can be simplified as:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}),$$

Sampling a trajectory by the learned policy and policy gradient can be applied to the policy $\pi(a|s,\boldsymbol{\theta})$. By taking the policy ascending, the policy neuralnet will be adjusted in a way that increasing the object function $J(\boldsymbol{\theta})$. The pseudocode is shown below:

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

REINFORCE with Baseline

In barebone REINFORCE, the return G was estimated by Monte Carlo methods, with is an unbiased but high-variance estimation of V(s). Here, a baseline was introduced to reduce the variance thus increasing convergence. The pseudocode is below:

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Actor Critic

The REINFORCE with Baseline used reward G as target and the gradient descent for parameter w is essentially reducing mean-square-error between V(s) and reward G. Due to the reward G is high-variance, the learning process is high-variance. In Actor-Critic, the target G is replaced by one-step TD estimation, then use TD error training the value function to approach the true value. The pseudocode is shown below:

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Deep Deterministic Policy Gradient (DDPG)

In contrast to the stochastic policy that the policy is a mapping $\pi_\theta : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$, the deterministic policy has a deterministic action output: $\mu_\theta(s)$. Silver et al., derived the Deterministic Policy Gradient Theorem. Furthermore, to ensure the compatibility of using function approximation to approximate the Q function, Silver et al., constrained the Q function parameterization. Based on his work, Lillicrap et al., designed deep version of the Deterministic Policy Gradient (DPG) as Deep Deterministic Policy Gradient (DDPG). With the capability of deep neural net, DDPG can learn a policy from raw sensor data. To accelerate the training, Lillicrap et al. used trick experience replay. Moreover, to ensure the semi-gradient work, target networks was applied. Specifically, they used soft update for the target networks. To ensure enough exploration, Ornstein-Uhlenbeck process was used as behaviour policy. The pseudocode for DDPG is the following:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i,a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Experiments

In this section we analyse the performance of all algorithms discussed above for its returns at the end of each episode, learning agility and robustness. We chose Box2D environments as they are easy to implement and are not computationally expensive which would be an effective way to test our algorithm and visualize its behaviour. After the validation we used our algorithm to learn how to solve Breakout (Atari game), which would be computationally expensive.

Table 1: List of hyperparameters and their values

Hyperparameter	Value - Box2d	Value - Breakout	Description
batch size	128	32	Number of training cases over which each stochastic gradient descent (SGD) update
replay size	10,000	100,000	SGD updates are sampled from this number of most recent transitions.
replay prepopulate size	5,000	10,000	Number of transitions pre populated in the memory before training.
exploration	(1, 0.05, 200,000)	(1, 0.05, 1,000,000)	(Initial ε, Final ε, number of steps in which ε is exponentially annealed in)
learning rate	5e-3	1e-4	The learning rate used by RMSProp.
target network update frequency	1,000	1,000	The frequency with which target network is updated.
num_steps	1,000,000	2,000,000	Number of training steps used.
discount factor	0.99	0.99	Discount factor gamma used in the Q-learning update.
num_layers	2	3	Number of layers in the Neural Network.
layer_type	linear	convolution	Type of layer in the Neural Network.

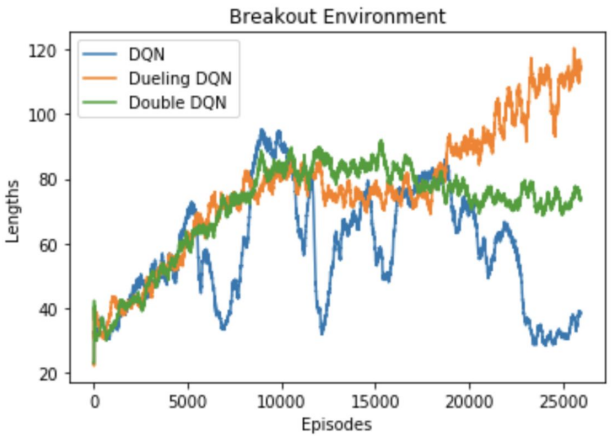

Figure 2: Learning curve of Breakout Environment

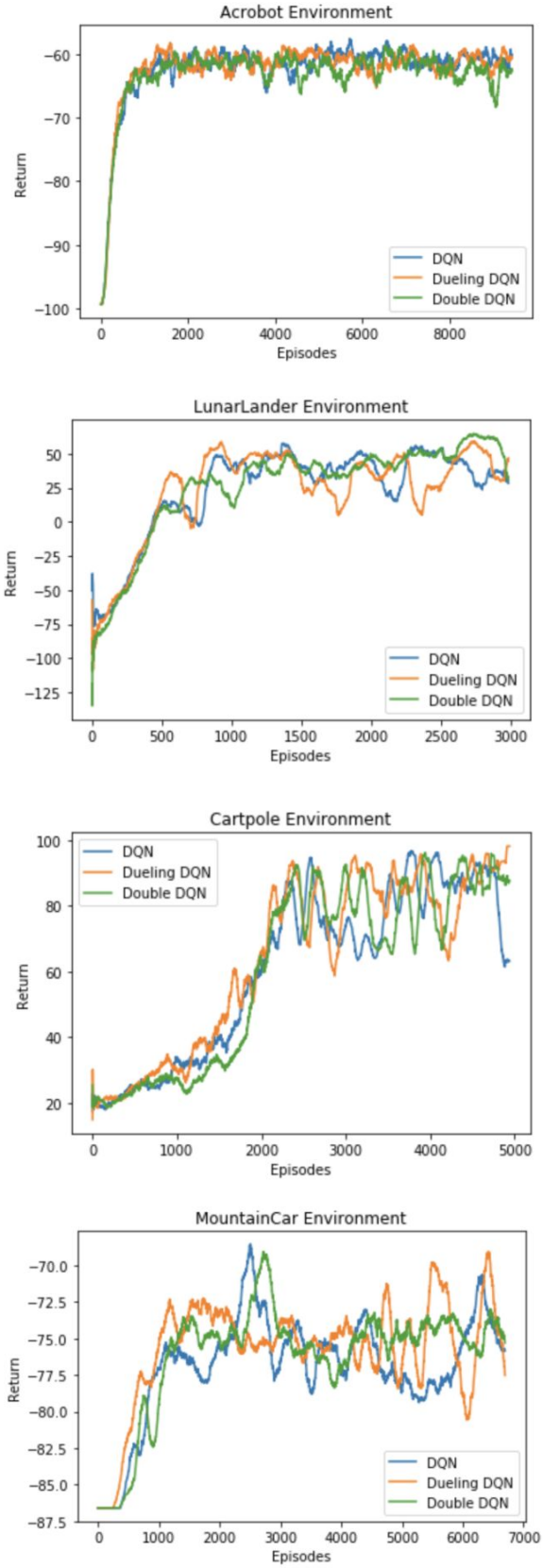

Figure 3: Learning curves of Box2D Environment

Analysis I: DQN v/s Dueling DQN v/s Double DQN

Our intuition based on the their network architecture suggested that Dueling DQN would learn with agility and Double DQN would learn with robustness, while DQN performing averagely. In the figure above, we can see that DQN, Dueling DQN, and Double DQN perform quite similarly with respect to each other in terms of return for all Box2d environments.

In CartPole, MountainCar and Breakout Environments, we see that Dueling DQN learned more efficient than DQN and Double DQN, which satisfied our expectations. Its neural architecture allows it to learn the state-value function efficiently, as with every update of the Q values in the Dueling DQN, the value stream V is updated whereas in DQN only value for one action is updated. However, we can see the learning curves of Dueling DQN have drops similar to DQN because it cannot deal with the maximum bias.

Double DQN appears more robust in these environments, suggesting that appropriate generalizations occur and that the found solutions do not exploit the determinism of the environments. This is appealing, as it indicates progress towards finding general solutions rather than a deterministic sequence of steps that would be less robust.

Compared to other Box2D environments, the Acrobot environment has relatively stable learning curves after 1000 episodes. The reason we guessed is that we updated the target network of Acrobot task in the right frequency so all algorithms learned fast and stable very soon.

In Figure 2 of the learning curve of Breakout, Dueling DQN has a tendency of rising instead of dropping, which might due to the deficit of training steps.

Hyperparameters Tuning

We think proper hyperparameters can help the agent to learn better and quicker. We tuned the hyperparameters of our assignment to the value shown in Table 1, and the performance improved significantly as shown in Figure 4.

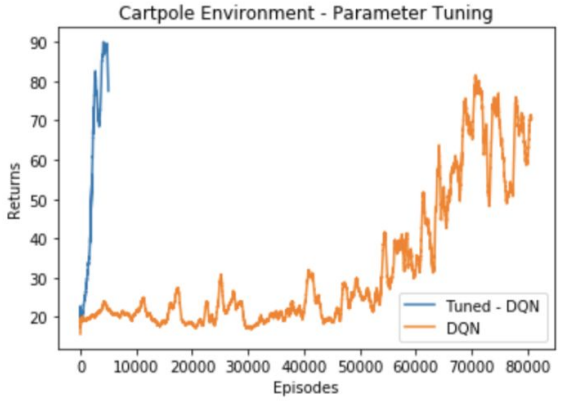

Figure 4: Return achieved after parameter tuning significantly increase result in a short amount of time.

Analysis II: DQN v/s REINFORCE

The DQN structure is the same as defined in previous. REINFORCE used two hidden layers with dimension 128. Learning rate was set as 3e-3.

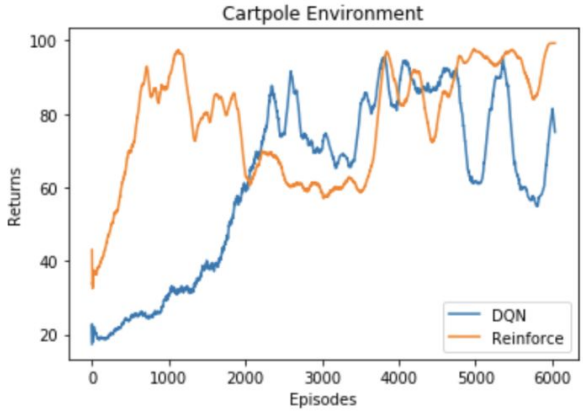

Figure 5: Learning curves of Cartpole Environment

Figure 5 shows the comparison on DQN and REINFORCE. For the cartpole environment, REINFORCE surpase the performance of DQN in the first one-third. These may due to the cart-pole problem, it is easier to learn the policy directly than learn a Q function. The drop of REINFORCE in the middle may due the Mente Calo update that lead to instability.

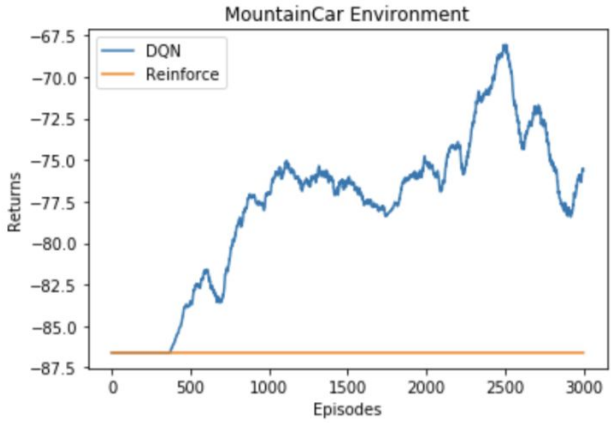

Figure 6: Learning curves of MountainCar Environment

For the Mountain car environment, REINFORCE failed to learn a policy, even failed to obtain a reward. This failure may lead by the stochastic policy cannot move the car consistently, so that the car can obtain enough velocity to go to the goal. In DDPG, this exploration problem was solved by using a behaviour policy, which incorporated Ornstein-Uhlenbeck process to explore efficiently.

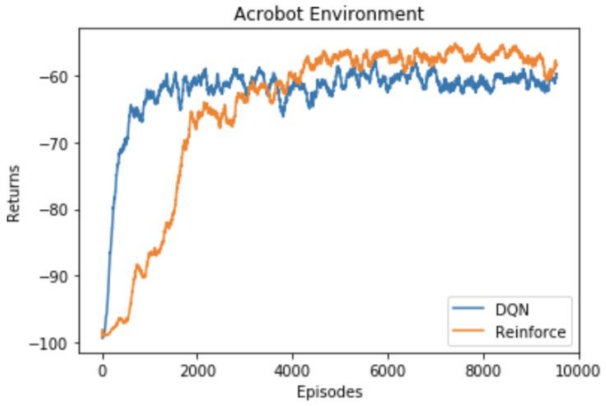

Figure 7: Learning curves of Acrobot Environment

For the AcroBot environment, REINFORCE surpassed DQN in the last one-third. This showed by only learning the policy, it can perform better than learning the Q function, where Q include more environmental information. Notice that in the first one-third, REINFORCE learns slower than DQN. It may lead by REINFORCE cannot explore the action-state space efficiently thus it takes longer time to find a good policy.

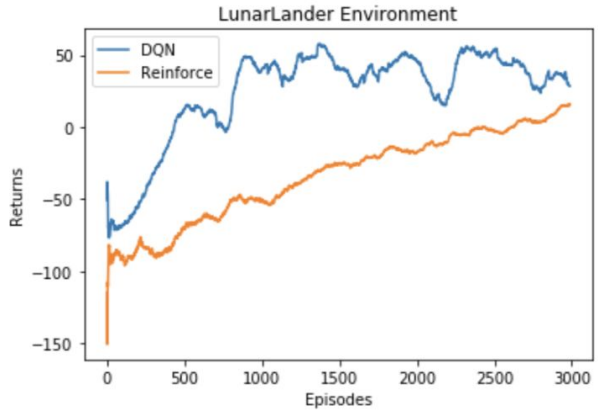

Figure 8: Learning curves of LunarLander Environment

In the LunarLander environment, DQN learns faster than REINFORCE at start. However, after the half of the time interval, REINFORCE raises fast and tends to surpass DQN. The fact that we used a mild parameter setting in REINFORCE (i.e., small learning rate, small batch size, etc) cased REINFORCE learns slow at first.

Analysis III: DQN v/s REINFORCE v/s Baseline v/s Actor-Critic

In the figure, four algorithms DQN, REINFORCE, Baseline, and Actor-Critic are compared in the Cart Pole environment. All the parameters are set as mentioned before. Besides, the critic network V(s) is the same as Q(s, a) network in DQN, but taken the maximum: V(s) = max[Q(s, a)].

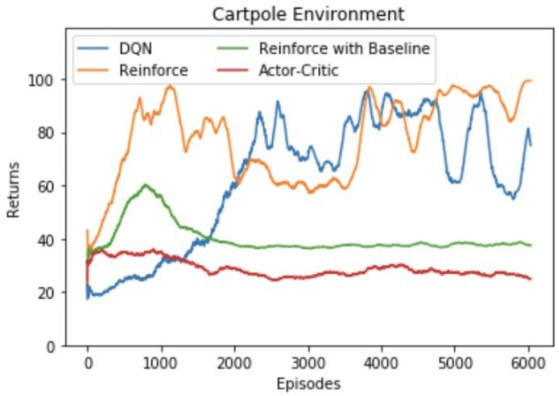

Figure 9: Comparison on DQN, Policy Gradient Methods

The green line is REINFORCE with Baseline. Compares to REINFORCE, the Baseline curve is more stable, it has less spikes than REINFORCE and DQN. The Baseline increased at start but dropped in the middle of the figure. The reason could be the code was implemented the same as the pseudocode in the project description, but it is problematic. In pseudocode, the update for the value network follows the episode trajectory, which breaked the i.i.d. assumption of the training dataset. So at start the cart pole fell down quickly and created sort and i.i.d. trajectories. However, when the agent learned some policy, the trajectory of cart pole becomes longer and related. The algorithm failed to lean a good value function, thus failed to learn a good policy. This problem can be solved by creating an experience replay buffer and using random sampling to do mini-batch gradient descent on the value function.

The Actor-Critic algorithm, as shown in red curve, is even worse than Baseline. Here the same problem, non-i.i.d. training data as discussed in Baseline comparison, deteriorates the learning of the value function. On the other hand, for the TD learning, people usually use semi-gradient. The problem of semi-gradient may also the same case as here: the target network is correlated with the

objective network. So use a target network that is uncorrelated with the objective network may also improve the learning. For instance, use low frequency to update the target network or use a soft update methods, as in DDPG.

Analysis IV: Deep Deterministic Policy Gradient

Here we applied DDPG in two continuous-action-space environment: pendulum and bipedal walker. DDPG calculate deterministic continuous action from high dimension image input.

The actor in DDPG is a three-layer fully connected neural net. The hidden layer's dimension is 128. Learning rate was set as 1e-4. The critic in DDPG is a three-layer fully connected neural net, the dimensions are state_dimension, 256, 128, action_dimension. Same learning rate as actor.

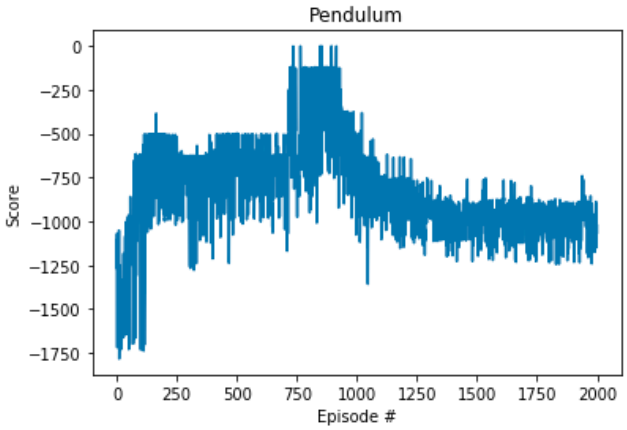

Figure 10: Learning curves of Pendulum Environment

For the pendulum, DDPG learned a policy to control it by outputting continuous actions. Notice on the figure 12 the continuous actions are shown as curved arrows.

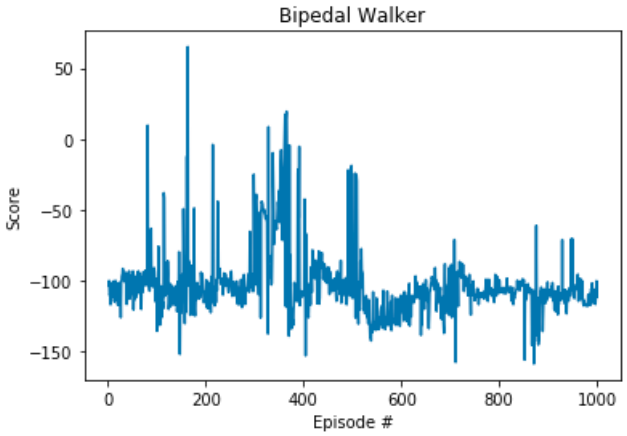

Figure 11: Learning curves of Bipedal Walker Environment

For the bipedal walker, DDPG does not learning a policy in these episodes (1000). If the agent was trained longer, it may achieve a better performance. There could have potential bugs.

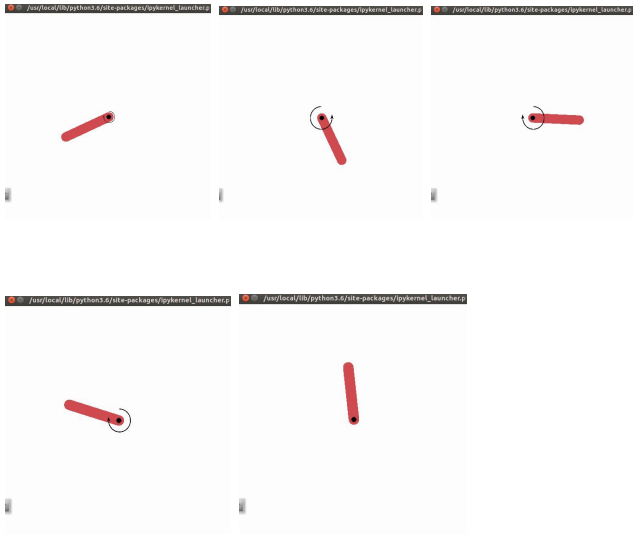

Figure 12: Screenshots for Pendulum Environment

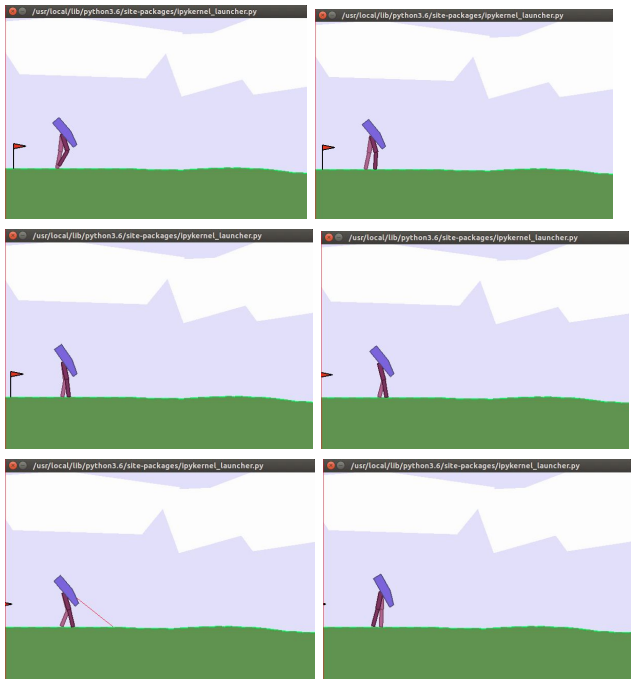

Figure 13: Screenshots for BipedalWalker Environment

Conclusion

In this paper we have implemented and compared various deep reinforcement learning algorithms. The main conclusion is that algorithms performance varies with respect to the environment, but those differences are depends on network architecture. For example, Dueling DQN can be seen to learn efficiently, Double DQN managed to learn robustly. Policy gradient can use simple network architecture to learn a policy directly.

Future work will focus on further focusing on the failure of achieving desired returns in the above algorithms by extensive hyperparameter optimization and applying all of the algorithms to more complex environments for better understanding.

References

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018, April). Rainbow: Combining improvements in deep reinforcement learning. In Thirty-Second AAAI Conference on Artificial Intelligence.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529.

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

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014, June). Deterministic policy gradient algorithms.

Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In Advances in neural information processing systems (pp. 1057-1063).

Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Thirtieth AAAI conference on artificial intelligence.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581.