CS7642 Project 2 – Solving LunarLander problem with Double Deep Q-Learning

Qinghui Ge

Abstract—**In this report, I will demonstrate using Double Deep Q-Learning (DDQL) to solve a model reinforcement learning problem – LunarLander [1]. I will discuss the implementation DDQL algorithm, study the effect of a few hyperparameters. With proper implementation and hyperparameters, the trained DDQL agent is able to achieve +200 average rewards in 100 independent episodes – the criterion of solving the problem.**
git hash: 702fd3e72db88d8224e3085a64bcc37bbb0c5681

I. INTRODUCTION TO LUNARLANDER

In the LunarLander problem, the goal is to control a lander to land between two flags without crash. The state of the lander is specified by 8 numbers: $(x, y, v_x, v_y, \theta, v_\theta, leg_L, leg_R)$. x and y are coordinates of the lander position. v_x and v_y are velocity components on the x and y axes. θ and v_θ are the angle and angular velocity. leg_L and leg_R are binary values indicating whether the left or right leg of the lander is touching the ground. There are four possible actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

In Opengym's [2] implementation of LunarLander environment, moving the lander from top of the screen to landing pad receives 100-140 points; the same amount would be penalized if lander landed outside the designed area; crash or successful landing get -100 and +100 points, respectively; each leg ground contact is worth +10 points; firing the main engine incurs a -0.3 point penalty for each occurrence. A decent landing policy should receive +200 points on average. Designing a reward function like above is a useful technique called reward shaping [3], [4] in reinforcement learning, which is supposed to give more frequent feedback to the learner and make the learning process easier.

II. DEEP Q-LEARNING

In basis Q-Learning, the value of each state-action pair (expected reward if an agent is in state s and performs action a, and then continues playing until the end of the episode following some policy π) is stored in a matrix Q(s,a) – the Q-table. The Q-Learning procedure keeps updating the Q-table using the experiences so that the Q-values converge to it's optimal value.

Keep a record of the entire Q-table can be impractical if the number of state and actions are large, or if the state or action is continuous, as in the case of LunarLander. In case of continuous state and discrete action space, Deep Q-Learning (DQL) [5] uses a neural network to generate approximate Q-values for any state on the fly (see Fig. for a comparison between Q-Learning and Deep Q-Learning).

Instead of updating the Q-table, the learning process updates the neural network that predict Q values.

At time step t, the action a_t can be selected from ϵ-greedy algorithm using the prediction of neural network Q of state s_t (Algorithm.1):

Algorithm 1 ϵ-greedy
---
    function ϵ-GREEDY(s, Q, ϵ)
        if random_number() $< \epsilon$ **then**
            return random action
        else
            return $argmax_a Q(s, a)$
        end if
    end function

If we know the ground truth Q_{true}, we can simply update the neural network parameter ω by minimize some Loss function such as L2 loss/MSE:

$$Loss = \frac{1}{N_s N_a} \sum_{s,a} (Q_{true}(s,a) - Q_\omega(s_t, a_t))^2 \quad (1)$$

However, unlike supervised learning, we do not know the ground truth Q_{true}, instead we use Bellman equation to estimate Q_{true}, or the TD-target. The TD-error of neural network with parameter Q_w is estimated by:

$$Error = Q_{target} - Q_\omega(s_t, a_t) \quad (2)$$
$$= R(s_t, a_t) + \gamma * max_a Q_\omega(s_{t+1}, a) - Q_\omega(s_t, a_t) \quad (3)$$

The estimation of Q_{target} would in turn depends on Q_ω, which makes the training unstable. To combat this problem, Deep Q-Learning method introduces another target network Q^t. Q^t is used to compute the TD-target, while Q is used to determine the ϵ-greedy action a_t:

$$Q_{target} = R(s_t, a_t) + \gamma max_a Q^t_{\omega'}(s_{t+1}, a) \quad (4)$$

Note

The target network Q^t has the same architecture as the prediction network Q, but its parameter ω' is updated in a slower fashion than Q. For example, update Q^t for every K times after Q is updated, or soft-update Q^t with some parameter $\tau < 1$: $Q^t \leftarrow \tau Q + (1 - \tau)Q^t$. This way, the target is kept frozen for some period and the training is more stable.

Another important element in Deep Q-Learning is experience replay [**?**], [6]. In basic Q-Learning, Q values are updated for every observed state-action pair. This would be inefficient if we want to apply stochastic gradient decent (SGD) to update

the weights of network Q. Further, SGD works best when the samples in each batch is independent, which is unlikely for state-action pairs coming from a same episode. Thus DQL use a replay buffer to store accumulated experiences in memory. Then for each time step, a batch of samples can be randomly draw from the buffer to feed into SGD. Each experience contains the following information: [state (s_t), action (a_t), reward (R_t), next_state (s_{t+1}), done(end of episode)]. With experience replay, experiences can be reused multiple times, this is could be another significant advantage if interacting with environment is expensive.

III. DOUBLE Q-LEARNING

In basic Q-Learning and Deep Q-Learning, the max operator in computing TD-target(Eq.4) uses same value for evaluating and selecting an action. This causes a bias that the actions whose Q-values are overestimated are more likely to be selected. For basic Q-Learning, van Hasselt et.al proposed using two independent Q-tables to decouple evaluation and selection [7]. In each iteration, one Q-table is used to select the action, another is used to evaluate the Q-value, and the choice is made by random 50%-50% chance. This is the original Double Q-Learning algorithm.

IV. DOUBLE DEEP Q-LEARNING

For Deep Q-Learning, van Hasselt et.al suggest using the target network to partially decouple evaluation and selection. This introduce a minimal change on top of the existing Deep Q-Learning algorithm without introducing another network. In this Double Deep Q-Learning algorithm [8], instead of taking max of target network Q^t, we first determine the greedy action with Q, then evaluate Q^t at the greedy action:

$$Q_{target} = R(s_t, a_t) + \gamma Q^t_{\omega'}(s_{t+1}, a^*) \quad (5)$$
$$a^* = argmax_a Q_\omega(s_{t+1}, a) \quad (6)$$

Summarizing the above, we follow Algorithm 2 to implement Double Deep-Q Learning in this work. Note in actual implementation, the for loop in each batch is vectorized to improve computational efficiency.

There are a number of hyperparameters involved in DDQL:

1) ϵ / $decay$. In ϵ-decay algorithm, ϵ controls explore rate. With larger ϵ, random action is choose more frequently and thus more exploration, while small ϵ leads to greedy action. ϵ can be annealled with a decay factor, such that exploration is favored in earlier episodes, as training goes on, the algorithm shift towards more exploitation.
2) max_size of replay buffer. The replay buffer is set to have a max_size. When max_size is reached, most recent experience will overwrite oldest experience. The size should be neither too small nor too large. A small buffer will have a lot of correlated samples and the training will be unstable; large buffer may contain out-dated sample computed from less optimized Q that leads to slow convergence.
3) τ. The parameter controls soft-update. For larger τ, the target network Q^t is more update-to-date with Q, so the

---

Algorithm 2 Double Deep-Q Learning

function UPDATE(Q, Q^t, buffer)
    batch = random_choice(buffer, batch_size)
    for each sample=[s_t, a_t, R_t, s_{t+1}, done] in batch **do**
        if done **then**
            $Q_{target}(s_t, a_t) = R_t$
        else
            $a^* = argmax_a Q_{(s_{t+1}, a)}$
            $Q_{target}(s_t, a_t) = R_t + \gamma * Q^t_a(s_{t+1}, a^*)$
        end if
        for other $a \neq a_t$: $Q_{target}(s_t, a) = Q(s_t, a)$
    end for
    update Q: one step of gradient decent using $x = s_t(batch)$, $y = Q_{target}(batch)$.
    update Q^t: $Q^t \leftarrow \tau Q + (1 - \tau)Q^t$
end function

function DOUBLE DEEP-Q LEARNING( )
    initialize Q, Q^t to have the same random weights.
    initialize replay buffer with max_size
    for each episode **do**
        reset environment
        for step t in episode **do**
            $a_t = \epsilon$-greedy(s, Q, ϵ)
            take action a_t, observe s_{t+1}, R_t, done
            add [s_t, a_t, R_t, s_{t+1}, done] to replay buffer
            update(Q, Q^t, buffer)
        end for
        $\epsilon *= decay$
    end for
end function

---

training may become unstable. For small τ, updates of Q^t is lagged behind Q, this could make training stable but converge slower.
4) γ the discount factor as in regular Q-learning.
5) Neural-net related hyperparamters, such as network structure, loss function, optimizer, learning rate lr, batch-size, etc.

V. EXPERIMENT RESULTS

A. Training

Using the following setting, we obtain a trained agent that is able to achieve +200 reward for an average of 100 episodes:

1) inital ϵ=1, $decay$=0.995
2) max_buffer_size = 9600
3) τ = 0.001
4) γ = 0.99
5) Neural Network with two hidden layers of 64 and 32 units, both using LeakyRelu activation. Output layer has N_A (4 in the case of LunarLander) output numbers, using Linear activation. Adam optimizer with learning rate lr = 0.0005 is used to minimize MSE loss. Batch_size = 32 for sampling replay buffer.
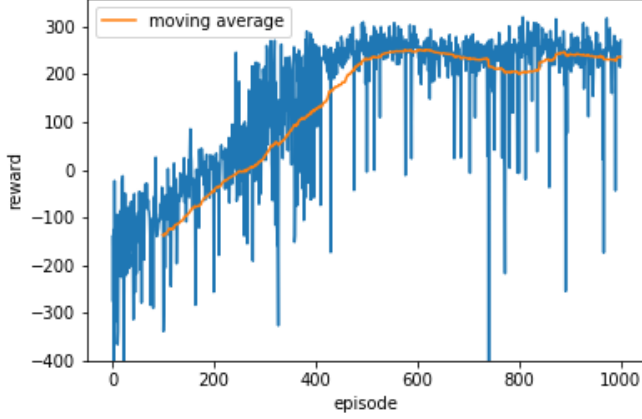
Fig. 1. Episode rewards of 1000 episodes during training process. $lr = 0.0005$, $decay = 0.995$, $\tau = 0.001$, $\gamma = 0.99$. The moving average of 100 episodes is shown in the orange line.
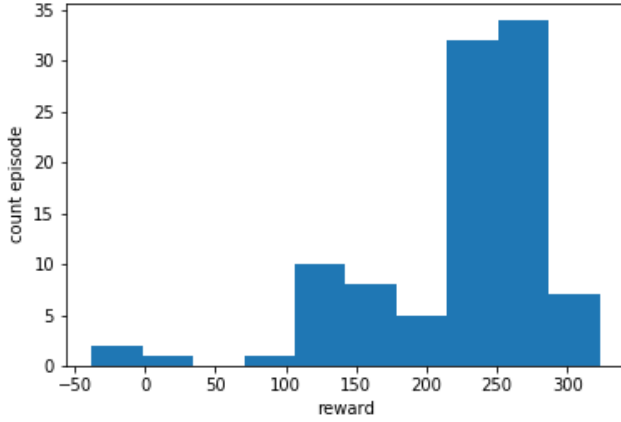

Fig. 2. Distribution of rewards for 100 independent episodes using the trained agent.

We train the agent for 1000 episodes. Fig. 1 shows the reward of each episode during training process. The moving average reward of 100 episodes reaches +200 at around 500 episodes. Afterwards the rewards can still be negative for certain episodes, although the moving average remains above +200 until the end. Using the weights obtained at the end of 1000 episodes, we can evaluate total reward of 100 independent episodes. The distribution of 100 rewards in show in Fig. 2. We are able to achieve average total reward of 227, with standard deviation = 55.

B. Effect of Hyperparameters

Next we will study the effect of a few hyperparameters: learning rate lr, decay factor, γ, and soft-update parameter τ. Other parameters are kept the same though all experiments: initial $\epsilon = 1$, batch_size = 32 , max_buffer_size = 9600.
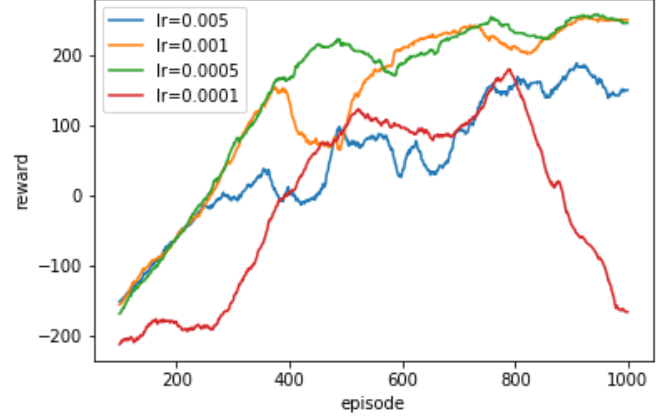

Fig. 3. Effect of learning rate. Moving average reward of 1000 episodes for different $decay$ values during training process. $decay = 0.995$, $\tau = 0.001$, $\gamma = 0.99$ for all runs. The moving average is evaluated with a window of 100 episodes.

First, we retrain the agent at different learning rate: $lr = 0.005, 0.001, 0.0005, 0.0001$, with other parameters fixed. The learning curves of different learning rate are shown in Fig. 3. $lr = 0.001$ and 0.0005 is able to achieve moving average reward > 200 during training. For larger learning rate 0.005, although it performs similarly at the beginning, the rewards does not improve to +200 later on. Small learning rate 0.0001 learns slowly at the beginning, rewards decrease again towards negative in later episodes, as if the agent "forget" what it has learned.

We then turn to the decay rate for ϵ-greedy algorithm. We experimented with $decay = 0.999, 0.995, 0.99, 0.95$. Smaller decay would make the training enter exploitation phase earlier, and the rewards can increase faster in the beginning. As show in Fig. 4, in the first 100 episodes, the reward increase quickest for $decay = 0.95$, and slowest for $decay = 0.999$. However, without enough explore, the agent do not perform great in the long term: $\epsilon = 0.995/0.99$ out-performs $decay = 0.95$ in the end. Too large $decay$ is not idea either, for $decay = 0.999$, the learning becomes very slow, because the agent is unable to exploit the learned greedy actions.

The soft-update parameter τ acts as a learning rate for the target network. In Fig. 5, we can see larger τ values like $\tau = 0.005$ leads to more instability during the training process, as the target network is updated more aggressively. On the other hand small τ values like $\tau = 0.0002$ and $\tau = 0.0001$ will slow down the learning. $\tau = 0.0005$ or 0.001 achieves best performance by balancing the stability and learning efficiency, and are able to achieve moving average reward > 200 without too much fluctuation.

Finally, we take a brief look at the discount factor γ. As the goal is to optimize the undiscounted reward during the entire episode, our intuition is $\gamma = 1$ may yield the best results. However, interestingly the training seems to be less stable with $\gamma = 1$. In Fig. 6, we compare the four independent runs of
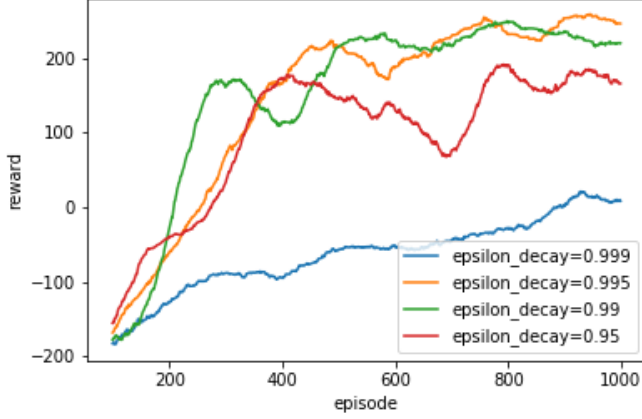
Fig. 4. Effect of epsilon_decay. Moving average reward of 1000 episodes for different *decay* values during training process. $\tau = 0.001$, $lr = 0.0005$, $\gamma = 0.99$ for all runs. The moving average is evaluated with a window of 100 episodes.
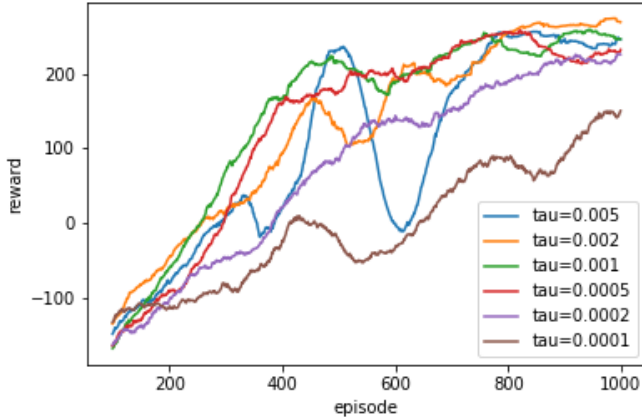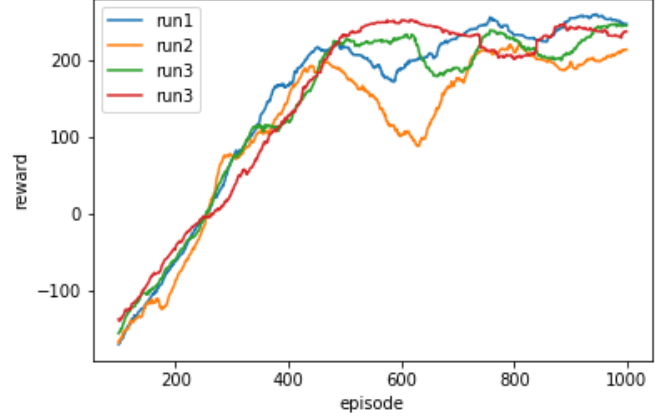

(a) $\gamma = 0.99$

Fig. 5. Effect of τ. moving average reward of 1000 episodes for different τ values during training process. $decay = 0.995$, $lr = 0.0005$, $\gamma = 0.99$ for all runs. The moving average is evaluated with a window of 100 episodes.

(b) $\gamma = 1$

Fig. 6. Comparison between $\gamma = 0.99$ (a) and $\gamma = 1$ (b). Each graph shows the moving average reward of four independent runs of given γ value, with $decay = 0.995$, $lr = 0.0005$, $\tau = 0.001$. The difference between 4 runs of same γ value comes from random seed settings. It is shown that $\gamma = 0.99$ gives more stable learning curves.
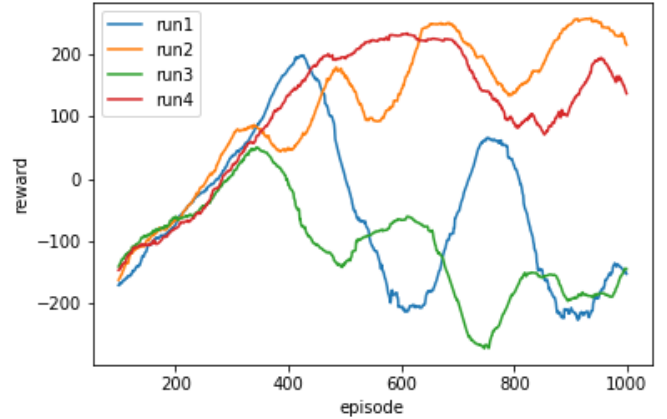
$\gamma = 1$ and $\gamma = 0.99$. The learning curve for $\gamma = 1$ has larger fluctuation, and the rewards can drop quickly to negative even though the initial training seems to be on the right track. Choosing γ to be smaller than 1 partly alleviate the problem, as shown in the 4 runs of $\gamma = 0.99$.

VI. DISCUSSION

Training the agent with Double Deep Q-Learning is challenge in a few ways. The hyperparameters have to be carefully tuned to balance exploration and exploitation, stability and efficiency. With current hyperparameters set, I still experience in stability frequently, the training reward sometimes decrease a lot after reaching +200. We may consider stop the training once the reward reaches +200, or save the model weights though the way and pick the best weights afterwards. It is worth-noting the reward observed in the training curve is

not exactly the test reward, because the training reward is computed with different networks in each episode, although it may be roughly correlated test reward. Ideally, we can obtain test-reward during the training process. However, this could be expensive because test-reward is evaluated by averaging 100 independent runs in order to get some statistical significance.

Currently we are keeping other parameters fixed while studying the effect of a given hyperparameter, it would also be interesting to explore the correlations between parameters. For example, the smallest learning rate $lr = 0.0001$ performs poorly with $decay = 0.995$. It is possible that we may want to increase the $decay$ value (slower decay, more exploration) to adapt to the slower learning rate.

Each training currently takes about 30 minutes on CPU for 1000 episodes, this makes massive hyperparameter searching difficult. Surprisingly the bottleneck is not updating the network weights, but using the network to predict next action,

which is a step hard to be parallelized.

We also note that when using the Opengym implementation of the environment, unless using the "unwrapped" setting, the episode is going to be terminated at some max time step (1000 for LunarLander). In those unnatural termination, the Bellman equation for TD-Target should include the value of next state, rather than just reward. The results above were obtained without this tweak. Some initial trials with the modification did not give significant performance change. However, due to the randomness in the algorithm, we need more experiment (like the multiple runs for studying the effect of γ) to reach a solid conclusion. And we could monitor the frequency of reaching max time step to access the potential impact for this problem.

Various techniques have been proposed to improve the performance of (double) deep Q-Learning method. For example, priority replay [9] suggest giving samples with larger TD-error higher weight to be withdrawn from the replay buffer in order to learn more efficiently. Dueling Deep Q-Network [10] decompose the Q values to an action independent part $V(s)$ and an action dependent part $A(s, a)$. From $V(s)$, the network is able to learn which states are good/bad without learning the effect of actions. These are things worth trying out if I had more time.

VII. ACKNOWLEDGEMENT

Apart from the academic papers cited, a number of blog posts have helped me understand the intuition behind the algorithms, and provide hands-on guidance on the implementation [11]–[15].

REFERENCES

[1] "Lunarlander," https://gym.openai.com/envs/LunarLander-v2/.
[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.
[3] M. J. Mataric, "Reward functions for accelerated learning," in *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 181–189.
[4] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *ICML*, vol. 99, 1999, pp. 278–287.
[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
[6] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.
[7] H. V. Hasselt, "Double q-learning," in *Advances in neural information processing systems*, 2010, pp. 2613–2621.
[8] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461
[9] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.
[10] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.
[11] "Double deep q networks–tackling maximization bias in deep q-learning," https://towardsdatascience.com/double-deep-q-networks-905dd8325412.
[12] "Self learning ai-agents iii:deep (double) q-learning," https://towardsdatascience.com/deep-double-q-learning-7fca410b193a.
[13] "Deep q-learning, part2: Double deep q network, (double dqn)," https://medium.com/@qempsil0914/deep-q-learning-part2-double-deep-q-network-double-dqn-b8fc9212bbb2.
[14] "Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed...)," https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/.
[15] "Dueling deep q networks," https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751.