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.**

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, θ, v_θ, leg_L, leg_R). x and y are coordinate 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 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 in reinforcement learning, which is supposed to give more frequent feedback to the learner and make the learning process easier.

II. Double 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 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 ϵ-greedy

```
function ϵ-GREEDY(s, Q, ϵ)
    if random_number() < ϵ 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 MSE:

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

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:

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

The estimation of Q_{target} in turn depends on Q_ω, which makes the training unstable. The Double Deep Q-Learning method tries to combat this problem by introducing another target network Q^t. Q^t is used to compute the TD-target, while Q is used to determine the greedy action at which Q^t is evaluated:

$$\begin{aligned} Q_{target} &= R(s_t,a_t) + \gamma * Q^t_{\omega'}(s_{t+1},a^*) && (4)\\ a^* &= argmax_a Q_\omega(s_{t+1},a) && (5) \end{aligned}$$

Note a_t is still obtained from ϵ-greedy algorithm using Q.

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 C 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 technique used in DDQL is replay buffer. In basic Q-Learning, Q values are updated for every new 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 DDQL use a replay buffer to store accumulated in a memory buffer. Then for each time step, a 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].

Summarizing the above, we follow Alg to implement Double Deep-Q Learning in this work:

Algorithm 2 Double Deep-Q Learning

```
function UPDATE(Q, Q^t, buffer)
    batch = random_choice(buffer, batch_size)
    for each sample 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 + γ * Q^t_a(s_{t+1}, a*)
        end if
        for other a ≠ 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 ← τQ + (1 − τ)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 = ε-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
        ε *= decay
    end for
end function
```

In actual implementation, the for loop in each batch can be 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 explore, while small ϵ leads to greedy action. ϵ can be annealled with a decay factor, such that explore is favored in earlier episodes, as training goes on, the algorithm shift towards more exploit.
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 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.

III. EXPERIMENT RESULTS

A. Training Agent

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.

We train the agent for 1000 episodes, Fig. X 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. X. We are able to achieve average total reward of 227, with standard deviation = 55.

B. Effect of Hyperparameters

Next we explore the effect of a few hyperparameters.

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. X. $lr = 0.001$ 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 is 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 exploit phase earlier. The rewards can increase faster in the beginning. As show in Fig. X, the reward increase quickest for $decay = 0.95$ in the first 100 episodes, and slowest for $decay = 0.999$.

IV. CONCLUSION