# AI 3603 Artificial Intelligence: Principles and Techniques

By: Dexuan He (521030910063)

HW#: 2

November 16, 2023

## I. INTRODUCTION

In this homwwork, we are goning to take a look into Reinforcement Learning.

#### 1. *Task1*

In this assignment, we will implement Reinforcement Learning agents to find a safe path to the goal in a grid-shaped maze. The agent will learn by trail and error from interactions with the environment and finally acquire a policy to get as high as possible scores in the game.

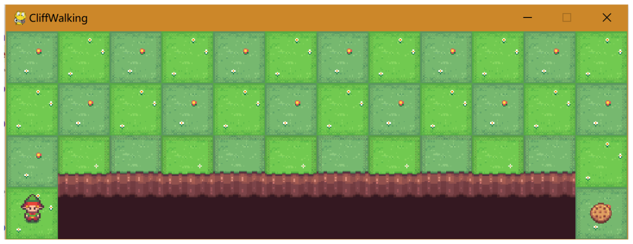


FIG. 1: Task1.field

#### 2. *Task2*

In this part, we will implement a DQN agent to control a lunar lander. We are asked to read and running the given code to train our intelligent lander agent. The performance of the agent may not be satisfactory and we have to tune it to get higher scores.

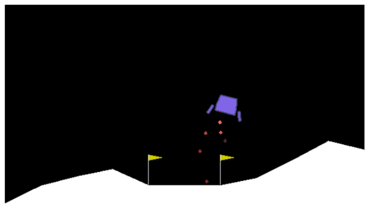


FIG. 2: Task2.field

### A. Environment

- Python 3.9.18
- Pytorch 2.1.0
- Pygame 2.1.0
- Gym 0.25.2
- Box2d-py 2.3.5

## II. THEOREM & METHOD

For a Reinforcement learning task, the agent interact with the environment to get information, called state $s$, from surroundings, and takes action $a$ to get reward $r$ and change its state $s$. The tricky problem is to find a strategy to get more reward in different unknown environment.

Here we introduce 3 common methods for RL: Q Learning, Sarsa, which we use in task1, and DQN, which we use in task2.

### A. Q Learning

A good policy (strategy) is to choose the action which lead to max expected reward at each state, we denote the expected longtime reward when the agent take action $a$ at state $s$ as $Q(s, a)$, store all the Qs in a list, and learn the list from interacting with the env.

```
Init Q(s,a)
Repeat (for each episode):
    Init s
    Repeat (for each step):
        Choose a using epsilon-greedy
        Take a and observe r and s'
        Q(s,a) += lr*[r+gamma*max(Q(s',a'))-Q(s,a)]
        s = s'
```

$\epsilon$-greedy is a strategy, who chooses the best rewarding action with a probability of $1-\epsilon$, and a random action with a probability of $\epsilon$., and that $\epsilon$ , which balances exploration and exploitation, will decay in training to converge to a optimal path.

The $a'$ used in the learning process may not be the actual action to take next in Q Learning.

### B. Sarsa

```
Init Q(s,a)
Repeat (for each episode):
    Init s
    Choose a using epsilon-greedy
    Repeat (for each step):
        Take a and observe r and s'
        Choose a' using epsilon-greedy
        Q(s,a) += lr*[r+gamma*Q(s',a')-Q(s,a)]
        s = s', a = a'
```

The Sarsa method is similar to Q Learning, and the biggest difference is the learning process. Sarsa uses the actual action to take to optimize $Q(s, a)$, which makes great difference in the result we're going to see.

### C. DQN

Both Q Learning and Sarsa use a list to store and learn $Q(s, a)$, which takes large time and storage cost, especially in continue time and multiple action problems. Deep Q-Network (DQN) use NNs to learn $Q(s, a)$

The unique point of DQN is the two networks: the evaluate network for $Q(s, a)$, and the target network for $r + gamma * max(Q(s', a'))$. In the learning process, parameters in the evaluate network are synchronously update, while those in the target network are not. This feature maintains the stability of the gradient when the network parameters are updated. Besides, DQN uses a replay buffer to randomly learn from the experience in the past, in order to eliminate the timing dependency of samples.

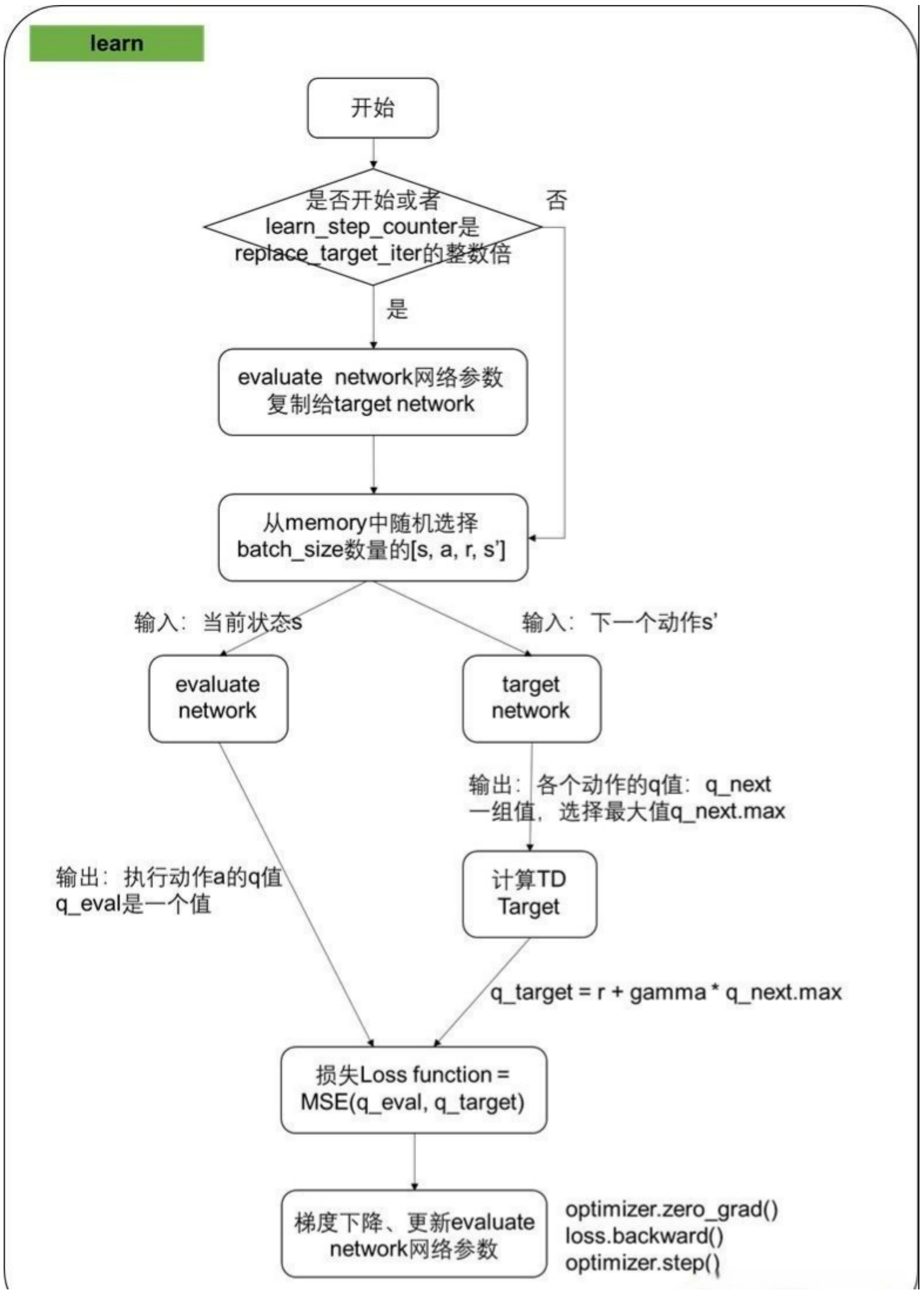

learn
开始
是否开始或者 learn_step_counter是 replace_target_iter的整数倍
否
是
evaluate network网络参数复制给target network
从memory中随机选择 batch_size数量的[s, a, r, s']
输入：当前状态s
输入：下一个动作s'
evaluate network
target network
输出：各个动作的q值：q_next 一组值，选择最大值q_next.max
输出：执行动作a的q值 q_eval是一个值
计算TD Target
q_target = r + gamma * q_next.max
损失Loss function = MSE(q_eval, q_target)
梯度下降、更新evaluate network网络参数
optimizer.zero_grad()
loss.backward()
optimizer.step()

FIG. 3: DQN learning

## III. IMPLEMENT & RESULT

### A. Task1

To implement Q Learning and Sarsa, we first use a dict to store the Q list, which is fast to hash and convenience to enlarge (for unknown space of the environment). The process to choose action are same: (In additoion, we use SummaryWriter() for visualize and RecordVideo() for episode capture. See more detail in the code.

```
def e_decay(epsilon, e_decay):
    return epsilon / (1 + e_decay)

def epsilon_greedy(epsilon, all_actions, max_action):
    p = np.random.random()
    if p < epsilon:
        return np.random.choice(all_actions)
    else:
        return max_action

def choose_action(self, observation):
    """choose action with epsilon-greedy algorithm."""
    #action = np.random.choice(self.all_actions)
    Qs = self.Qlist.get(observation)
    if Qs is None:
        Qs = np.array([0.0, 0.0, 0.0, 0.0])
        self.Qlist[observation] = Qs
    max_action = self.all_actions[np.argmax(Qs)]
    action = epsilon_greedy(self.epsilon, self.all_actions, max_action)
    return action
```

#### 1. *Q Learning*

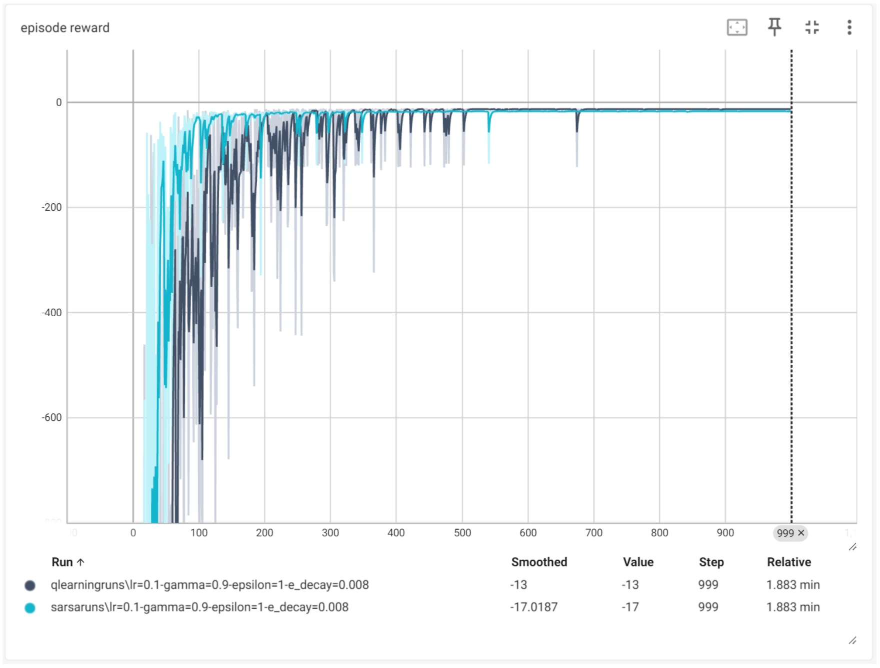
For Q Learning, the learning process don't use the actual action to take:

```
def learn(self, last_state, last_action, now_state, reward):
    """learn from experience"""
    Qs_now = self.Qlist.get(now_state)
    if Qs_now is None:
        Qs_now = np.array([0.0, 0.0, 0.0, 0.0])
        self.Qlist[now_state] = Qs_now
    self.Qlist[last_state][last_action] += self.lr*(reward+self.gamma*max(Qs_now)
                                            -self.Qlist[last_state][last_action])
```

#### 2. *Sarsa*

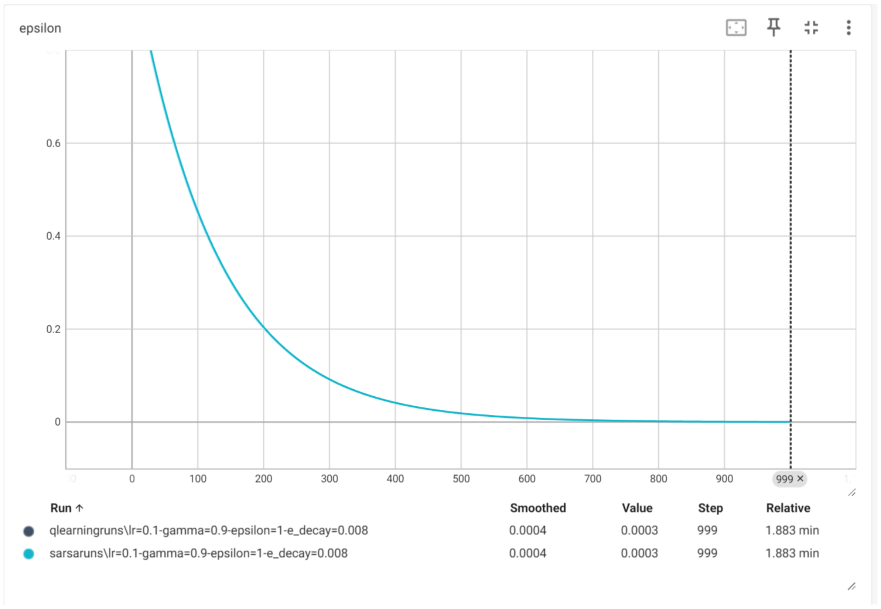
For Sarsa, the learning use the actual action to take:

```
def learn(self, last_state, last_action, now_state, now_action, reward):
    """learn from experience"""
    self.Qlist[last_state][last_action] += self.lr*
        (reward+self.gamma*self.Qlist[now_state][now_action]
        -self.Qlist[last_state][last_action])
```



| | Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|---|
| ● | qlearningruns\lr=0.1-gamma=0.9-epsilon=1-e_decay=0.008 | -13 | -13 | 999 | 1.883 min |
| ● | sarsaruns\lr=0.1-gamma=0.9-epsilon=1-e_decay=0.008 | -17.0187 | -17 | 999 | 1.883 min |

(a) Reward

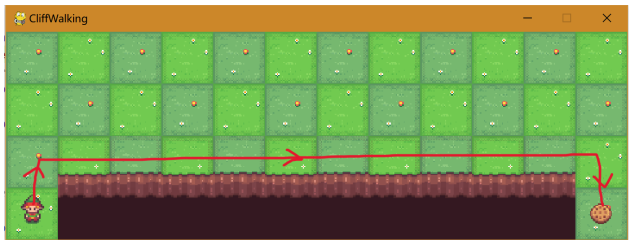


| | Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|---|
| ● | qlearningruns\lr=0.1-gamma=0.9-epsilon=1-e_decay=0.008 | 0.0004 | 0.0003 | 999 | 1.883 min |
| ● | sarsaruns\lr=0.1-gamma=0.9-epsilon=1-e_decay=0.008 | 0.0004 | 0.0003 | 999 | 1.883 min |

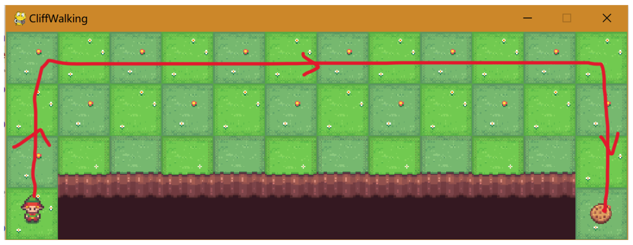
(b) epsilon

FIG. 4: Comparison between Q Learning and Sarsa

From the reward curve we see that Sarsa has a better convergence while Q Learning gets a better reward. Since they share a $\epsilon$-decay function, they have the same $\epsilon$



(a) Q Learning



(b) Sarsa

FIG. 5: Comparison between Q Learning and Sarsa

From the path we see that Q Learning finds the global optimal, and Sarsa seems to be fear with the cliff (more detail in the videos).

Now we explain those differences. Q Learning take max in updating $Q(s, a)$, that is, do not consider the value of the final walk to a large negative reward, only consider to get the maximum reward eventually, so Q Learning more courageous, not afraid of error, which enable it to get the optimal path, while slowing down the converge process for more mistakes to take.

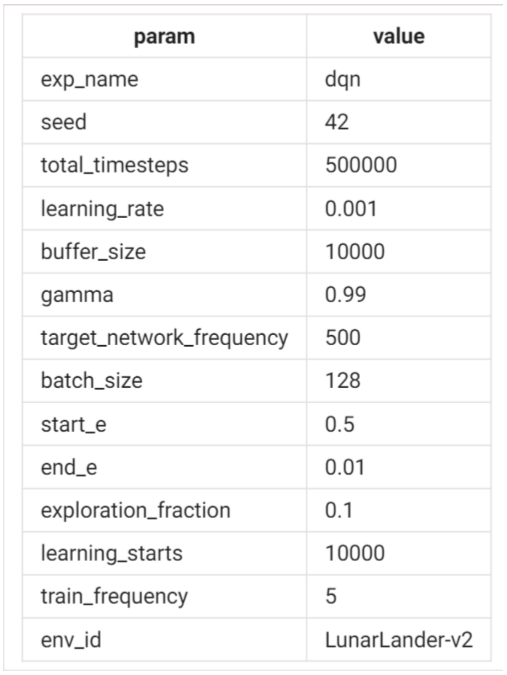
On the other hand, Sarsa takes the next step to learn, and as long as there are mistakes around it (big negative rewards) then there is a chance to get this bad reward, and the whole path will be rated poorly. After that it will be avoided as much as possible. Then it eventually leads to the fact that Sarsa will be more sensitive to making mistakes and will stay away from the points where mistakes are made and be more conservative, resulting in a higher cost but a better convergence.

## B. Task2

To land the lunarcar we use a DQN. In fact, the DQN is given and the only thing we need to do is to adjust parameters and leave comments on the code. Comments can be find in the code file.

### 1. *Hyper-Parameters*

Compare to default parameters, we turn up *gamma* to learn faster, enlarge begin_$\epsilon$ and turn down end_$\epsilon$ for better exploration in early stage and more exploitation in ending. A bigger train_frequency to accelerate converge process. And a much larger lr, which will change over time because we use a scheduler.



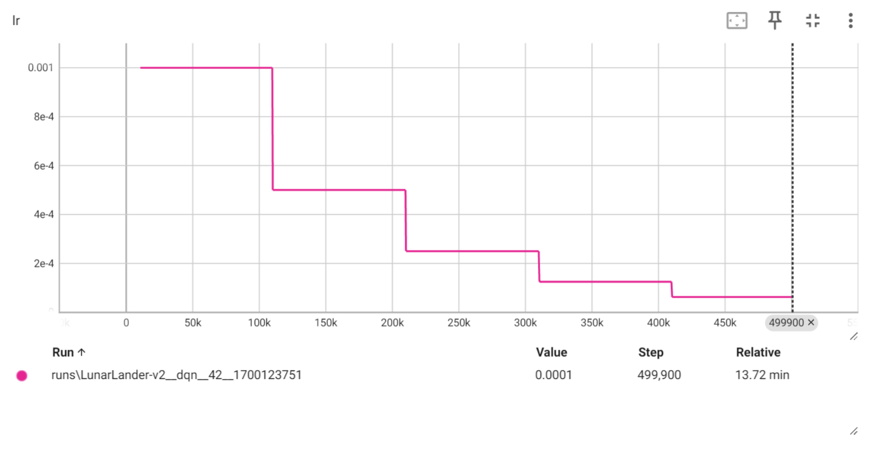
| param | value |
|---|---|
| exp_name | dqn |
| seed | 42 |
| total_timesteps | 500000 |
| learning_rate | 0.001 |
| buffer_size | 10000 |
| gamma | 0.99 |
| target_network_frequency | 500 |
| batch_size | 128 |
| start_e | 0.5 |
| end_e | 0.01 |
| exploration_fraction | 0.1 |
| learning_starts | 10000 |
| train_frequency | 5 |
| env_id | LunarLander-v2 |

FIG. 6: Hyper-parameters

### 2. *Scheduler*

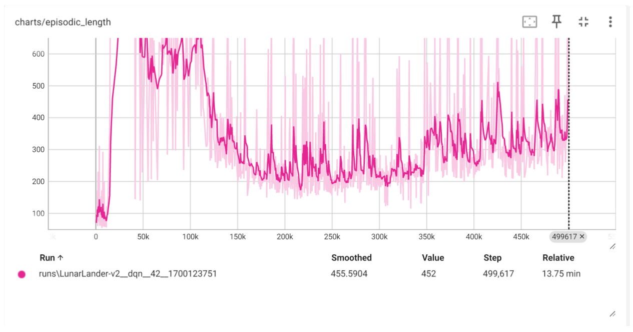
In the early trainings, we find that the training process is too long to run while the model can't converge to a positive reward, jumping between several values perically. We think that the reason might be learning rate, too small step in the initial period and too large step in the final period. Therefore we use a scheduler to dynamically adjusts learning rate with epoches.

```
optimizer = optim.Adam(q_network.parameters(), lr=args.learning_rate)
'''use a scheduler for a quick learning and fine convergence'''
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=2e4, gamma=0.5)
```



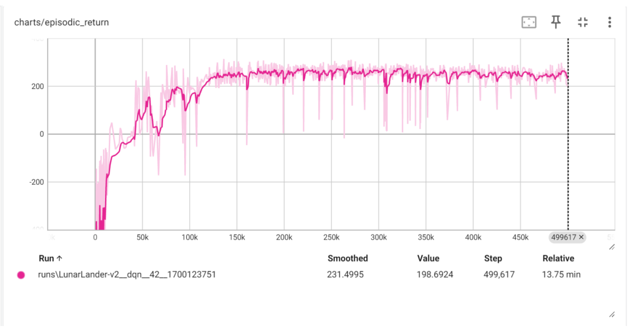
| Run ↑ | Value | Step | Relative |
|---|---|---|---|
| ● runs\LunarLander-v2__dqn__42__1700123751 | 0.0001 | 499,900 | 13.72 min |

FIG. 7: Change of learning rate



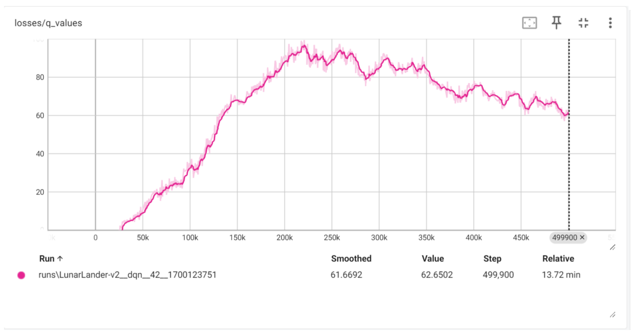
| Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| ● runs\LunarLander-v2__dqn__42__1700123751 | 455.5904 | 452 | 499,617 | 13.75 min |

(a) episode length



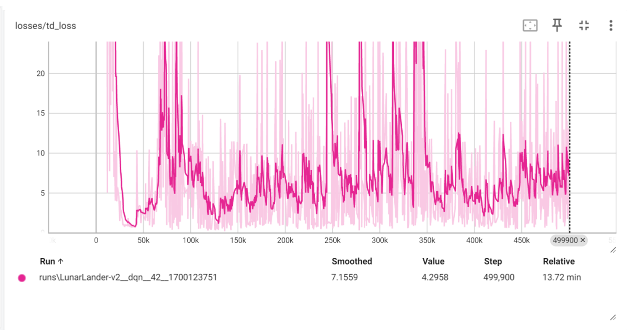
| Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| ● runs\LunarLander-v2__dqn__42__1700123751 | 231.4995 | 198.6924 | 499,617 | 13.75 min |

(b) episode reward



| Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| ● runs\LunarLander-v2__dqn__42__1700123751 | 61.6692 | 62.6502 | 499,900 | 13.72 min |

(c) loss evaluate net



| Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| ● runs\LunarLander-v2__dqn__42__1700123751 | 7.1559 | 4.2958 | 499,900 | 13.72 min |

(d) loss target net

FIG. 8: result of DQN

During training, we get a fast convergence, and after that, a smoother landing (which increase the episode time), a better evaluating, and a better match between evaluate network and target network. (more detail in video)

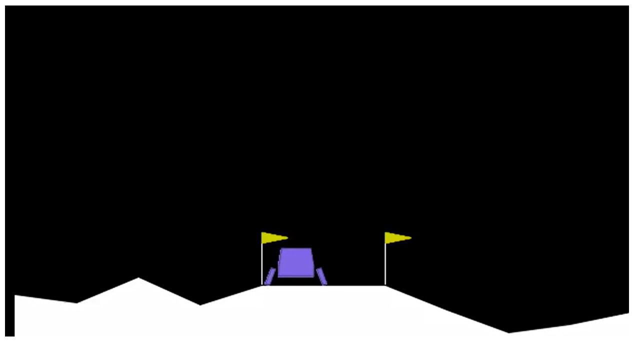


FIG. 9: Success landing

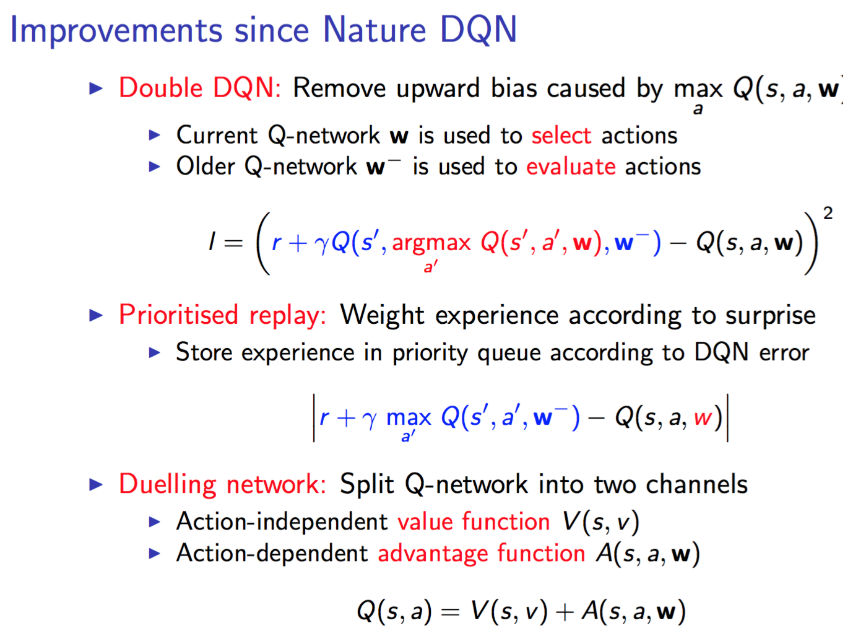
## IV. DISCUSSION & CONCLUSION

### A. Improved DQN

There're several improvements of DQN:

- improve the $Q(s, a)$
- improve the random selection of experience to learn
- improve the network

and the following works are the implement of these improvements.



Improvements since Nature DQN

- Double DQN: Remove upward bias caused by $\max_a Q(s, a, \mathbf{w})$
  - Current Q-network $\mathbf{w}$ is used to select actions
  - Older Q-network $\mathbf{w}^-$ is used to evaluate actions

$$l = \left( r + \gamma Q(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a', \mathbf{w}), \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right)^2$$

- Prioritised replay: Weight experience according to surprise
  - Store experience in priority queue according to DQN error

$$\left| r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, w) \right|$$

- Duelling network: Split Q-network into two channels
  - Action-independent value function $V(s, v)$
  - Action-dependent advantage function $A(s, a, \mathbf{w})$

$$Q(s, a) = V(s, v) + A(s, a, \mathbf{w})$$

FIG. 10: Imroved DQN

### B. Conclusion

In this homework, we complete two tasks about Reinforcement Learning, including Q Learning, Sarsa and DQN. We solve the previous two problem mainly by implementing the learning process, and adjust hyper-parameters and propose the scheduler method to solve the last one. During the process, we deepen understanding of RL methods, and improve our coding capacity.