AI 3603 Artificial Intelligence: Principles and Techniques

By: Benhao Huang (521030910073)

HW#: 2

November 6, 2023

Contents

1 Introduction

In this lab, students are asked to utilized learned knowledge about Reinforcement Learning to complete two game tasks: Cliff-walking Environment using Q-learning and Sarsa, Lunar-lander using DQN. More detailed requirements are as follow,

- Implement Sarsa and Q-learning algorithms, visualize the results.
- Analyze the learning process and describe the difference in the results of two algorithms in detail.
- Fill in every comments in DQN.py file to display your understandings of the code.
- Train and tune the DQN agent to achieve higher scores. Other ideas are encouraged.
- Choose one drawback of DQN, learn the corresponding improvements, and write your understandings.

2 Task1: Reinforcement Learning in Cliff-walking Environment

In this section, we will implement Sarsa and Q-learning algorithm to complete the Cliff-walking task, and give visualizations as well as results analysis.

2.1 Sarsa

Sarsa (State-Action-Reward-State-Action) is a reinforcement learning algorithm used to solve Markov decision processes (MDPs) and train agents to make decisions in an environment. It belongs to the category of on-policy reinforcement learning methods, meaning that it learns directly from the agent's interactions with the environment. A brief pseudocode of Sarsa algorithm representing its learning process is rendered below.

2.1.1 Formulation

Algorithm 1 Sarsa Algorithm

```
Initialize Q-value function, $Q(s, a)$, for all state-action pairs
for each episode do
  Initialize $s$
  repeat
    Choose an action $a$ using an epsilon-greedy strategy
    Execute the selected action $a$, observe the reward $r$ and next state $s'$
    Update Q-value using:
    $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [R + \gamma \cdot Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal
end for
```

```python
class SarsaAgent(object):
def __init__(self, all_actions):
    """initialize the agent. Maybe more function inputs are needed."""
    self.all_actions = all_actions
    self.epsilon = 0.1  # epsilon-greedy algorithm
    self.epsilon_lst = [self.epsilon]  # for plot
    self.lr = 0.5
    self.gamma = 0.9  # reward decay
    self.e_decay = 1  # epsilon-decay schema
    self.q_table = np.zeros((48, len(self.all_actions)))

def choose_action(self, observation):
    """choose action with epsilon-greedy algorithm.
    epsilon probs, choose the a that max q(s,a)
    1 - epsilon probs, choose a random action from self.all_actions
```

```python
        """
        if np.random.uniform() > self.epsilon:  # epsilon probs, np.random.uniform() return a random float in the
             interval [0.0, 1.0)
            action = np.argmax(self.q_table[observation, :])
        else:
            action = np.random.choice(self.all_actions)
        self.epsilon = self.epsilon * self.e_decay  # epsilon decay
        self.epsilon_lst.append(self.epsilon)
        return action

    def learn(self, state1, action1, reward, state2, action2):
        """learn from experience"""
        self.q_table[state1][action1] = self.q_table[state1][action1] + self.lr * (
                    reward + self.gamma * self.q_table[state2][action2] - self.q_table[state1][action1])
        return False
```

2.1.2 Parameter Settings

We have set hyper-parameters for SARSA as follows, where ‘Iteration‘ is the agent’s interaction times with environment in each episode, ϵ is used in epsilon-greedy algorithm for choosing actions. Learning Rate corresponds to α in its pseudo code 8, and γ is the reward decay. In order to provide convenience to reproduce the results, we set random seed fixed to 0.

Table 1: Hyper-parameters used for SARSA algorithm

Episode	Iteration	ϵ	Learning Rate	ϵ-decay	γ	Random Seed
1000	500	0.1	0.5	0.99995	0.9	0

Theoretically, ϵ should be set larger (like 0.8), in order to enable the agent to explore more. However, we found it hard for SARSA to converge when ϵ is set too large. Moreover, even ϵ is set to 0.1, SARSA can still performs well, which indicates the stability of SARSA algorithm.

2.2 Q-Learning

Q-Learning is a reinforcement learning algorithm used to train agents to make decisions in an environment with the goal of maximizing cumulative rewards. This algorithm is based on the Q-value function, which estimates the expected long-term rewards for taking specific actions in particular states. A pseudocode representing its learning process is shown below:

2.2.1 Formulation

Algorithm 2 Q-Learning Algorithm

```
Initialize Q-value function, $Q(s, a)$, for all state-action pairs
for each episode do
    Initialize $s$
    repeat
        Choose an action $a$ using an epsilon-greedy strategy
        Execute the selected action $a$, observe the reward $r$ and next state $s'$
        Update Q-value using:
        $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [R + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$
    until s is terminal
end for
```

```python
class QLearningAgent(object):
    def __init__(self, all_actions):
        """initialize the agent. Maybe more function inputs are needed."""
        self.all_actions = all_actions
        self.epsilon = 0.8  # epsilon-greedy algorithm
        self.epsilon_lst = [self.epsilon]  # for plot
        self.lr = 0.5
        self.gamma = 0.9  # reward decay
        self.e_decay = 0.99995  # epsilon-decay schema
        self.q_table = np.zeros((48, len(self.all_actions)))  # Q-table

    def choose_action(self, observation):
        """choose action with epsilon-greedy algorithm.
        epsilon probs, choose the a that max q(s,a)
        1 - epsilon probs, choose a random action from self.all_actions
        """
        if np.random.uniform() > self.epsilon:  # epsilon probs, np.random.uniform() return a random float in the
            interval [0.0, 1.0)
            action = np.argmax(self.q_table[observation, :])
        else:
            action = np.random.choice(self.all_actions)
        self.epsilon = self.epsilon * self.e_decay  # epsilon decay
        self.epsilon_lst.append(self.epsilon)
        return action

    def learn(self, state1, action1, reward, state2):
        """learn from experience"""
        max_q_s2 = np.max(self.q_table[state2, :])
        self.q_table[state1][action1] = self.q_table[state1][action1] + self.lr * (
                reward + self.gamma * max_q_s2 - self.q_table[state1][action1])
        return None
```

2.2.2 Parameter Settings

We have set hyper-parameters for Q-Learning as follows. Since detailed explanation has been rendered in SARSA section, we won't elaborate it again here.

Table 2: Hyper-parameters used for Q-Learning algorithm

Episode	Iteration	ϵ	Learning Rate	ϵ-decay	γ	Random Seed
1000	500	0.8	0.5	0.99995	0.9	0

2.3 Visualization and Analysis

In this section, we will render the visualization of two algorithms, which includes the plot of episode reward, ϵ plot as well as the final path chosen by the agent. Combined with those figures, We will further analyze the difference between two methods.

2.3.1 Different Trajectory

As shown in the figure 1, the trajectory derived from SARSA tends to be more conservative, which means it will stay as far away from the cliff as possible. By contrast, Q-Learning algorithm tends to be more radical, which chooses the shortest but the most dangerous way.

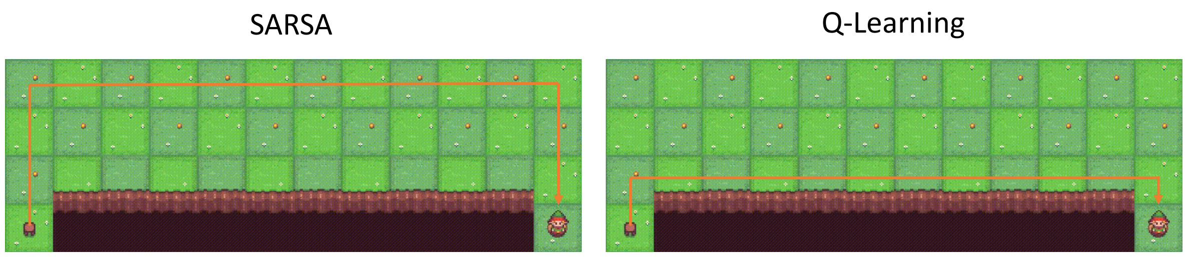

Figure 1: Different Trajectory of Two Algorithms

For more details, we also draw the heatmap of the visiting time of each grid by the agent during the training process, which clearly shows the different learned strategy of each agents with these two algorithms respectively. The brighter the color of the grid, the more times it has been accessed.

As we can see, for SARSA, the start point (0,3) is visited 1238 times and the end point is reached 997 times, while Q-learning visits start point for 3755 times and successfully reaches end point for 995 times, which means:

- SARSA algorithm is highly efficient. Among each episode, it can quickly find the endpoint, because 1238 is relatively close to episode = 1000, which means it just go through a few inner loops of 500 iterations to reach the endpoint.
- SARSA algorithm is quite "safe". As indicated in the figure, it's successful rate is $997/1238 \approx 80.5\%$, while Q-Learning is $995/3755 \approx 26.5\%$, which means unlike Q-learning algorithm that frequently lets the agent fall into the cliff, SARSA manages to avoid falling into cliff.

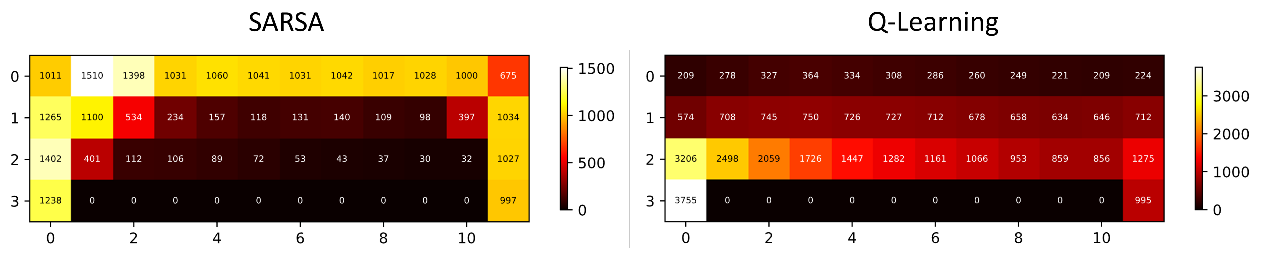

Figure 2: Heatmap which shows the visited times of each box in the Cliff-walking environment

The main reason for the difference in the behavior of SARSA and Q-Learning lies in the fact that: SARSA calculates its Q-Value based on the action that it decides to take, while Q-Learning calculates its Q-value based on the local optimal action, which it may not take in practice.

- SARSA: $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [R + \gamma \cdot Q(s',a') - Q(s,a)]$; $s \leftarrow s'$, $a \leftarrow a'$
- Q-Learning: $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [R + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$; $s \leftarrow s'$

2.3.2 Reward and Epsilon Plot

SARSA and Q-Learning algorithms are also very characteristic in the change of reward value.

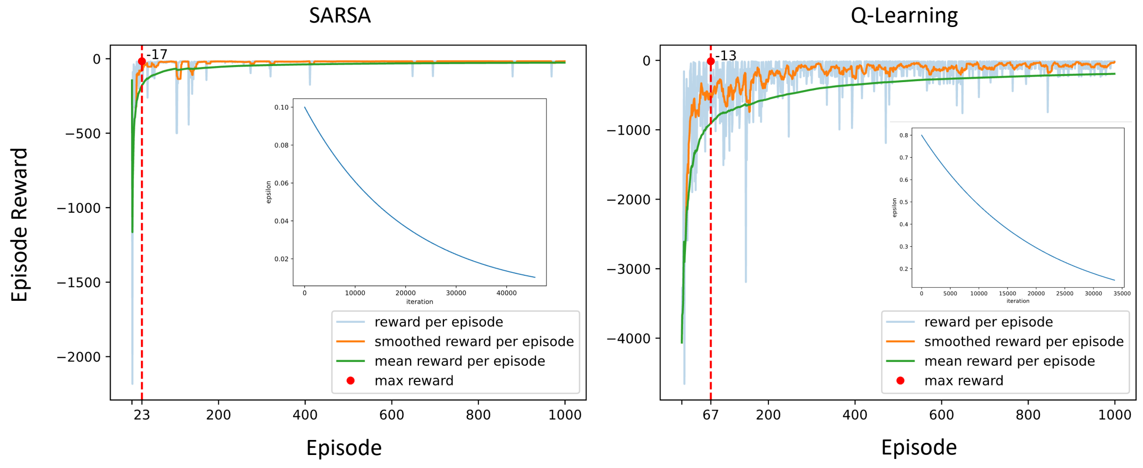

Figure 3: The episode reward earned by the player and epsilon value during the training process

In figure 3 above, the SARSA algorithm converges quickly under this parameter setting, which can be seen from the shape of the curve of the reward value and its mean value after smoothing. In contrast, the convergence speed of Q-Learning algorithm is much slower, and from the average curve, its performance is worse than that of SARSA algorithm.

However, we can also see that the number of steps required by Q-Learning algorithm is less than that of SARSA algorithm, which indicates that the upper bound of Q-Learning is higher than SARSA.

In summary, if you want to have a stable and excellent performance, SARSA algorithm is better, but if you want to perform to the extreme and are not afraid of risk, then Q-Learning algorithm is also good.

3 Task2: Deep Reinforcement Learning

In this section, we will describe the process we train the DQN agent and render the corresponding results. Besides, we will choose one drawback and write our understanding of the corresponding improvements.

3.1 Introduction

Deep Q-Network (DQN) is a reinforcement learning algorithm that combines the stability of Q-Learning with the approximation power of deep neural networks.

In reinforcement learning, an agent learns to interact with an environment in a way that maximizes some notion of cumulative reward. The DQN algorithm does this by learning a representation of the Q-function, a function that estimates the expected return (i.e., the total reward) of taking a certain action in a particular state.

The Q-function is learned by iteratively updating the Q-values based on the Bellman equation. However, traditional Q-Learning can be unstable and even diverge when combined with function approximators such as neural networks. Therefore, DQN introduces two key techniques to address these issues:

Experience Replay: To break the correlation between consecutive experiences, DQN stores the agent's experiences in a **buffer**. During training, minibatches of experiences are sampled from this buffer, breaking the sequential nature of the experiences and thus stabilizing the learning process.

Target Network: DQN uses a separate network to generate the target Q-values for each update, reducing the correlation between the target and the parameters being updated.

3.2 Understanding DQN Code

In this section, we will render the code used for training the DQN Agent, and explain the training process in comments as requested.

```python
# -*- coding:utf-8 -*-
import argparse
import os
import random
import time

import gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from stable_baselines3.common.buffers import ReplayBuffer
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm
from gym import wrappers
def parse_args():
    """parse arguments. You can add other arguments if needed."""
    parser = argparse.ArgumentParser()
    parser.add_argument("--exp-name", type=str, default=os.path.basename(__file__).rstrip(".py"),
        help="the name of this experiment")
    parser.add_argument("--seed", type=int, default=42,
        help="seed of the experiment")
    parser.add_argument("--total-timesteps", type=int, default=500000,
        help="total timesteps of the experiments")
    parser.add_argument("--learning-rate", type=float, default=5e-4,
        help="the learning rate of the optimizer")
    parser.add_argument("--buffer-size", type=int, default=10000,
        help="the replay memory buffer size")
    parser.add_argument("--gamma", type=float, default=0.99, #
        help="the discount factor gamma")
    parser.add_argument("--target-network-frequency", type=int, default=500,
        help="the timesteps it takes to update the target network")
    parser.add_argument("--batch-size", type=int, default=128,
        help="the batch size of sample from the reply memory")
    parser.add_argument("--start-e", type=float, default=0.5,
        help="the starting epsilon for exploration")
    parser.add_argument("--end-e", type=float, default=0.05,
        help="the ending epsilon for exploration")
    parser.add_argument("--exploration-fraction", type=float, default=0.1,
        help="the fraction of `total-timesteps` it takes from start-e to go end-e")
    parser.add_argument("--learning-starts", type=int, default=10000,
        help="timestep to start learning")
    parser.add_argument("--train-frequency", type=int, default=2,
        help="the frequency of training")
    parser.add_argument("--test", type=bool, default=True,
        help="test the results")
    args = parser.parse_args()
    args.env_id = "LunarLander-v2"
    return args

def make_env(env_id, seed):
    """construct the gym environment"""
    env = gym.make(env_id)
    env = gym.wrappers.RecordEpisodeStatistics(env)
    # env = wrappers.RecordVideo(env, './videos/')
    env.seed(seed)
    env.action_space.seed(seed)
    env.observation_space.seed(seed)
    return env

class QNetwork(nn.Module):
    """comments: Q network used for DQN, which could provide the q value of current obs"""
    def __init__(self, env):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(np.array(env.observation_space.shape).prod(), 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, env.action_space.n),
        )

    def forward(self, x):
        return self.network(x)

def linear_schedule(start_e: float, end_e: float, duration: int, t: int):
    """comments:  linearly decay the epsilon from start_e to end_e"""
    slope = (end_e - start_e) / duration
    return max(slope * t + start_e, end_e)

if __name__ == "__main__":

    """parse the arguments"""
    print("start training")
    args = parse_args()
    isTest = args.test
    model_path = "/Users/husky/Three-Year-Aut/AI-theory/AI3603_HW2/models/LunarLander-v2__dqn__42__1699249744.pth"
    run_name = f"{args.env_id}__{args.exp_name}__{args.seed}__{int(time.time())}"

    """we utilize tensorboard yo log the training process"""
    writer = SummaryWriter(f"runs/{run_name}")
    writer.add_text(
        "hyperparameters",
        "|param|value|\n|-|-|\n%s" % ("\n".join([f"|{key}|{value}|" for key, value in vars(args).items()])),
    )

    """comments: set the random seed so that the results could be reproduced"""
    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.backends.cudnn.deterministic = True
    device = torch.device("cuda" if torch.cuda.is_available() else "mps")

    """comments: get the environment we use for training"""
    envs = make_env(args.env_id, args.seed)
    """comments: init two networks, one for training, one for target; init the optimizer"""
    if not isTest:
        q_network = QNetwork(envs).to(device)
        optimizer = optim.Adam(q_network.parameters(), lr=args.learning_rate)
        target_network = QNetwork(envs).to(device)
        target_network.load_state_dict(q_network.state_dict())
    else:
        q_network = QNetwork(envs).to(device)
        q_network.load_state_dict(torch.load(model_path))
        target_network = QNetwork(envs).to(device)
        target_network.load_state_dict(q_network.state_dict())

    """comments: rb is used to store the experience"""
    rb = ReplayBuffer(
        args.buffer_size,
        envs.observation_space,
        envs.action_space,
        device,
        handle_timeout_termination=False,
    )

    if isTest:
        obs = envs.reset()
        for global_step in range(10, args.total_timesteps):

            q_values = q_network(torch.Tensor(obs).to(device))
            actions = torch.argmax(q_values, dim=0).cpu().numpy()
            """comments: step the env and get the next_obs, rewards, dones, infos,
            here infos include the episodic_return and episodic_length, """
            next_obs, rewards, dones, infos = envs.step(actions)
            envs.render() # close render during training

            """comments: learn from experience, store the experience in the replay buffer"""
            rb.add(obs, next_obs, actions, rewards, dones, infos)

            """comments: step to the next observation"""
            obs = next_obs if not dones else envs.reset()

    else:
        """comments: init the observation"""
        obs = envs.reset()
        for global_step in range(args.total_timesteps):

            """comments: linearly decay the epsilon from start_e to end_e"""
            epsilon = linear_schedule(args.start_e, args.end_e, args.exploration_fraction * args.total_timesteps,
                 global_step)

            """comments: epsilon-greedy algorithm"""
            if random.random() < epsilon:
                actions = envs.action_space.sample()
            else:
                q_values = q_network(torch.Tensor(obs).to(device))
                actions = torch.argmax(q_values, dim=0).cpu().numpy()

            """comments: step the env and get the next_obs, rewards, dones, infos,
            here infos include the episodic_return and episodic_length, """
            next_obs, rewards, dones, infos = envs.step(actions)
            # envs.render() # close render during training

            if dones:
                print(f"global_step={global_step}, episodic_return={infos['episode']['r']}")
                writer.add_scalar("charts/episodic_return", infos["episode"]["r"], global_step)
                writer.add_scalar("charts/episodic_length", infos["episode"]["l"], global_step)

            """comments: learn from experience, store the experience in the replay buffer"""
            rb.add(obs, next_obs, actions, rewards, dones, infos)

            """comments: step to the next observation"""
            obs = next_obs if not dones else envs.reset()

            if global_step > args.learning_starts and global_step % args.train_frequency == 0:

                """comments: choose a batch of data from the replay buffer"""
                data = rb.sample(args.batch_size)

                """comments: evaluate the q value of the current network, and contrast it with the target network
                    """
                with torch.no_grad():
                    target_max, _ = target_network(data.next_observations).max(dim=1)
                    td_target = data.rewards.flatten() + args.gamma * target_max * (1 - data.dones.flatten())
                old_val = q_network(data.observations).gather(1, data.actions).squeeze()
                loss = F.mse_loss(td_target, old_val)

                """comments: for visualizationm, we log the loss and q values"""
                if global_step % 100 == 0:
                    writer.add_scalar("losses/td_loss", loss, global_step)
                    writer.add_scalar("losses/q_values", old_val.mean().item(), global_step)

                """comments: optimize the network"""
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                """comments:update target network, every args.target_network_frequency steps"""
                if global_step % args.target_network_frequency == 0:
                    target_network.load_state_dict(q_network.state_dict())
        if not isTest:
            torch.save(q_network.state_dict(), f"models/{run_name}.pth")
        """close the env and tensorboard logger"""
    envs.close()
    writer.close()
```

3.3 Train DQN Agent

As requested, we will tune the hyper-parameters of the agent, especially gamma value, epsilon value, and epsilon decay schema in this section.

As we known, right parameters are of great significance to the model performance. Based on the given model structure, we mainly modified several parameters below:

- We changed **train_frequency** from 10 to 2, which aims to maximize the learning time and try to make agent sample more from buffer, which is an intuitive method to improve model's performance.
- we changed **gamma** from 0.6 to 0.99. Gamma is called reward-decay, a larger gamma value could enable the agent to think for long-term effects.

- we changed **start_e** from 0.3 to 0.5, which encourages the agent to explore more in the beginning.

seed	**total_timesteps**	**learning_rate**	**buffer_size**	**gamma**	**target_network_frequency**
42	500000	0.0005	10000	0.99	500
batch_size	**start_e**	**end_e**	**exploration_fraction**	**learning_starts**	**train_frequency**
128	0.5	0.05	0.1	10000	2

Table 3: Parameters Chosen for DQN

Actually, we have tried many possible choice of parameters and derived corresponding visualized results as below:

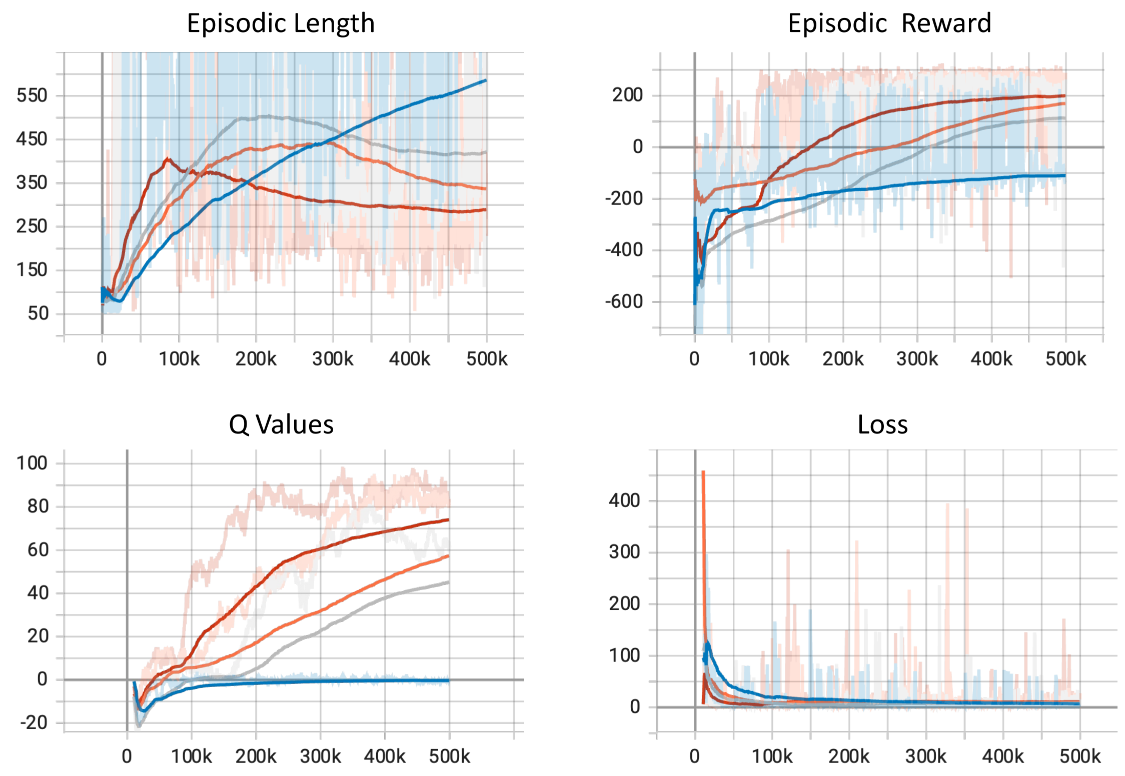

Figure 4: Training process of DQN agent with different parameters. For clarity, the plot in the figure is smoothed by 0.999. The red plot is the agent with parameters described in table 3.

As we could see, while all the agent's loss converge to a small value, the agent with parameters described in table 3 has shortest episodic length and largest episodic reward(nearly to 200, which is upper bound reward in LunarLander-v2 environment) after 500k steps. For clarity, all the agent's parameters will be included in Appendix for detailed examination.

3.4 Improve DQN Agent

One common drawback of standard DQN is its instability and slow convergence during training. This instability can lead to slow learning and difficulties in achieving good results in many reinforcement learning tasks. Fortunately, "Double Deep Q-Networks (Double DQN)"[1] has been put forward to improve this situation.

The primary issue that Double DQN aims to mitigate is the overestimation bias in Q-value estimates. Standard DQN tends to overestimate the Q-values, which can result in sub optimal policies and longer training times. **The overestimation occurs because Q-learning targets are based on the maximum Q-value**, and the same network is used for both the action selection and value estimation.

Given this case, DDQN manages to decouple the action selection (estimate Q-network) from the action evaluation (target Q-network).

In DQN, we update the Q-value estimate by:

$$y_t \leftarrow r_{t+1} + \gamma \max_a q(s_{t+1}, a; w_t) \tag{1}$$

While in Double DQN, Q-value is updated by:

$$y_t \leftarrow r_{t+1} + q(s_{t+1}, argmax_a q(s_{t+1}, a, w_e); w_t) \tag{2}$$

Here y_t is target value, γ is reward decay, w_e is estimate network's weight, and w_t is target network's weight.

As we can see, In DDQN, the Q-values used for target calculations are determined by the estimate network (action selection), but the target network is used to estimate the value of that selected action. This change prevents the overestimation of Q-values that can occur when using the same network for both purposes, and thus improve the performance of DQN.

References

[1] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

4 Appendix

4.1 Parameters for Other Agents in Figure 4

param	**value**
exp_name	dqn
seed	42
total_timesteps	500000
learning_rate	0.0002
buffer_size	10000
gamma	0.9
target_network_frequency	500
batch_size	128
start_e	0.3
end_e	0.05
exploration_fraction	0.1
learning_starts	10000
train_frequency	10
env_id	LunarLander-v2

param	**value**
exp_name	dqn
seed	42
total_timesteps	500000
learning_rate	0.0005
buffer_size	10000
gamma	0.99
target_network_frequency	500
batch_size	128
start_e	0.1
end_e	0.05
exploration_fraction	0.1
learning_starts	10000
train_frequency	4
env_id	LunarLander-v2

param	value
exp_name	dqn
seed	42
total_timesteps	500000
learning_rate	0.0005
buffer_size	10000
gamma	0.99
target_network_frequency	500
batch_size	64
start_e	1.0
end_e	0.05
exploration_fraction	0.1
learning_starts	10000
train_frequency	4
env_id	LunarLander-v2

param	**value**
exp_name	dqn
seed	42
total_timesteps	500000
learning_rate	0.0005
buffer_size	10000
gamma	0.99
target_network_frequency	500
batch_size	128
start_e	0.5
end_e	0.05
exploration_fraction	0.1
learning_starts	10000
train_frequency	2
env_id	LunarLander-v2

Figure 5: Parameters Table of four agents shown in figure 4.

4.2 Results Demo of DQN in LunarLander-v2 Environment

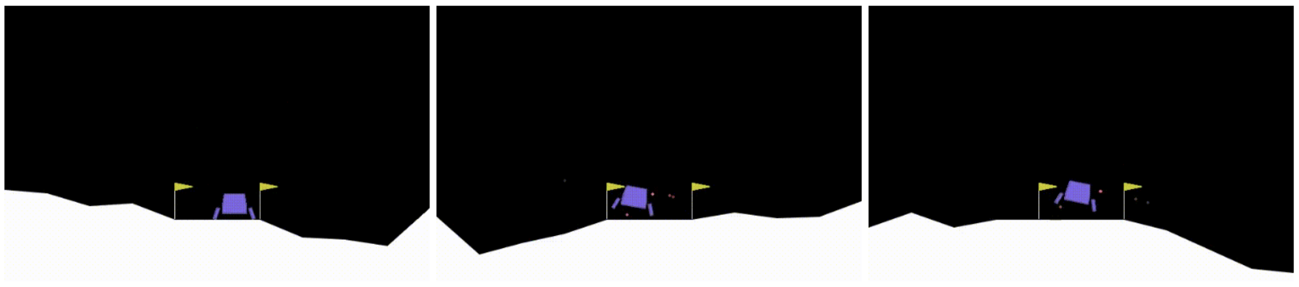

Figure 6: Task demo of this DQN agent with parameters above. GIF Demo could be found here.

4.3 Trial on Modifying Q-Network Structure

Actually, we have also tried to modify the hidden layer of the Q-Network, but it doesn't lead to significant optimization, so we quit this method.

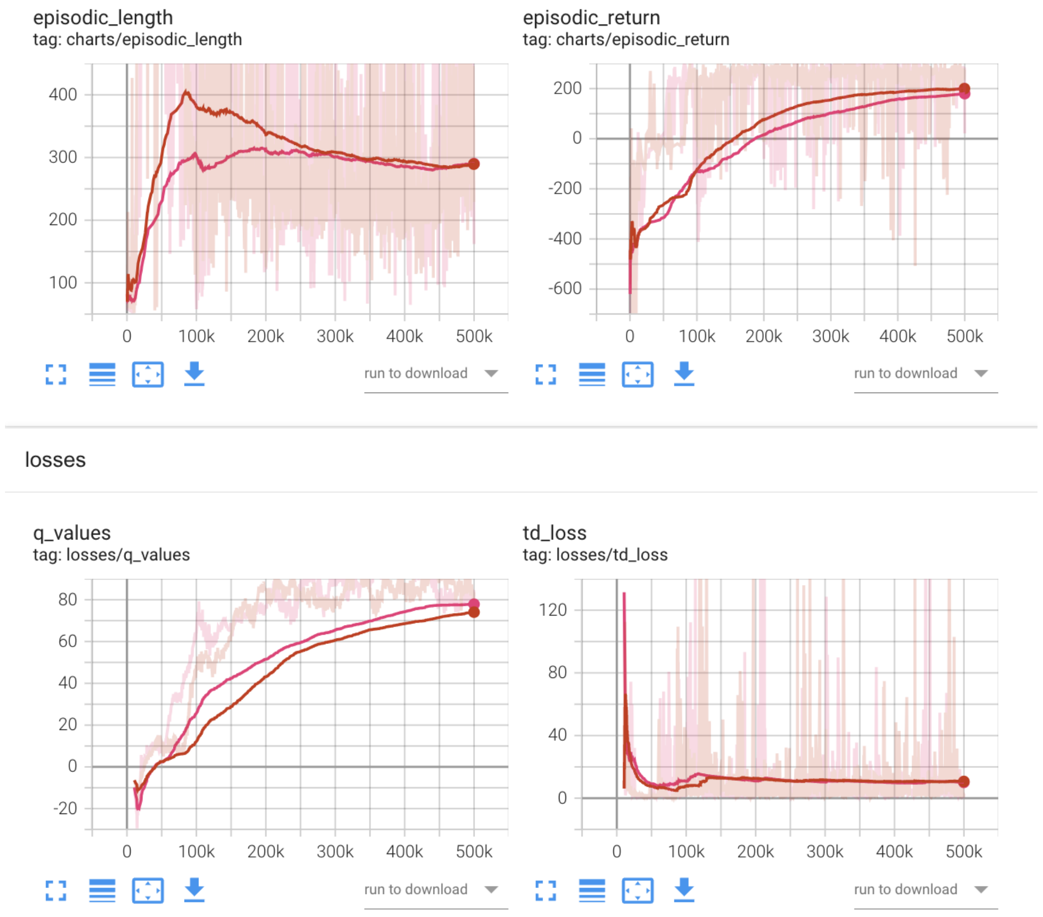

Figure 7: Results after changing the structure of the Q-Network. The red plot is from the same agent in figure 4 with same color, while the pink one is from the agent with modified Q-Network structure.

```python
class QNetwork(nn.Module):
"""comments: Q network used for DQN, which could provide the q value of current obs"""
def __init__(self, env):
    super().__init__()
    self.network = nn.Sequential(
        nn.Linear(np.array(env.observation_space.shape).prod(), 256),  # Increase hidden layer size
        nn.ReLU(),
        nn.Linear(256, 128),  # Increase hidden layer size
        nn.ReLU(),
        nn.Linear(128, env.action_space.n),
    )

def forward(self, x):
    return self.network(x)
```