Assignment 4

余北辰 519030910245

1 Experimental Purpose

- Implement the DQN algorithm and its improved algorithm DDQN
- Play with them in the classical RL control environment MountainCar
- Compare the performance of DQN and DDQN

2 Experimental Background

In this section, we present the methods of DQN and DDQN separately. In addition, we present the background of the environment MountainCar.

2.1 DQN

DQN, or Deep Q-Network, approximates a state-value function in a Q-Learning framework with a neural network.

The DQN algorithm use two same network: policy network Q and target network $\hat{Q}$. In each episode, the agent interact with the environment using the given state s_t and an action a_t. The algorithm stores the reward r_t and next state s_{t+1} together with s_t and a_t into the buffer as (s_t, a_t, r_t, s_{t+1}). Then, randomly sample a batch from the buffer, and estimate the target as $y = r + \gamma \max_a \hat{Q}(s, a)$. After that, optimize the mean square error between the estimation and Q value as $\min(y - Q(s, a))^2$. Finally, update $\hat{Q}$ by setting $\hat{Q} := Q$ every C steps.

The pseudo-code for DQN is shown below:

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

2.2 DDQN

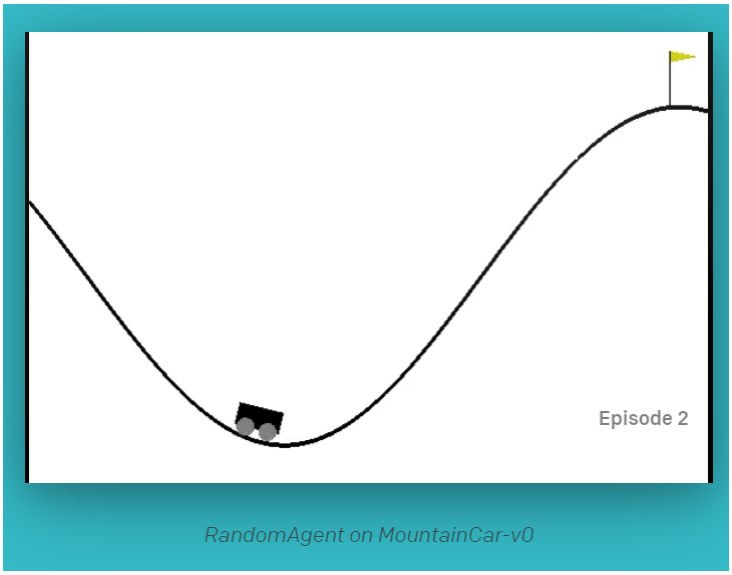
The DDQN(double DQN) is an improvement of DQN. To reduce overestimating of Q value, DDQN modified the method of estimating TD target as $y = r + \gamma\hat{Q}(s, \arg\max_a Q(s, a))$. If Q overestimates a, then $\hat{Q}$ will give it a proper value; If $\hat{Q}$ overestimates a, then a will not be selected. Thus, overestimation is avoided.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

2.3 MountainCar

MountainCar is a classical gym environment. In MountainCar, A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass.



There are three possible actions. 0 means push left, 1 means do nothing , and 2 means push right. The observation of the car is a vector of two dimension: the first dimension represents the position of the car and the second one represents the velocity.

3 Experimental Procedure

The source code for this experiment includes `DQN.py`, `DDQN.py` and `main.py`. The implement of DQN references the blog and is modified based on our experimental scenario.

3.1 DQN.py

`DQN.py` includes two classes: a class of network and a class of DQN agent.

Network

We will utilize `torch.nn` to construct our neural network for DQN.

```
class Net(nn.Module):
    def __init__(self, ):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(N_STATES, 50)
        self.fc1.weight.data.normal_(0, 0.1)
        self.out = nn.Linear(50, N_ACTIONS)
        self.out.weight.data.normal_(0, 0.1)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        actions_value = self.out(x)
        return actions_value
```

DQN agent

The initialization of DQN agent is seen below.

```
class DQN(object):
    def __init__(self):
        self.eval_net, self.target_net = Net(), Net()

        self.learn_step_counter = 0
        self.memory_counter = 0
        self.memory = np.zeros((MEMORY_CAPACITY, N_STATES * 2 + 2))
        self.optimizer = torch.optim.Adam(self.eval_net.parameters(), lr=LR)
        self.loss_func = nn.MSELoss()
```

As seen, we use two same network to fit policy and target separately.

```
    def choose_action(self, x):
        x = torch.unsqueeze(torch.FloatTensor(x), 0)
        if np.random.uniform() < EPSILON:
            actions_value = self.eval_net.forward(x)
            action = torch.max(actions_value, 1)[1].data.numpy()
            action = action[0] if ENV_A_SHAPE == 0 else action.reshape(
                ENV_A_SHAPE)
        else:
            action = np.random.randint(0, N_ACTIONS)
            action = action if ENV_A_SHAPE == 0 else action.reshape(
                ENV_A_SHAPE)
        return action
```

We use a ϵ-greedy policy to choose action. If the random number is less then ϵ, the greedy action is chosen. Otherwise, a random action is chosen.

```
    def store_transition(self, s, a, r, s_):
        transition = np.hstack((s, [a, r], s_))
        index = self.memory_counter % MEMORY_CAPACITY
        self.memory[index, :] = transition
        self.memory_counter += 1
```

We use a buffer to store the transitions. If the memory buffer is full, then the old transitions are filled.

```
    def learn(self):
        if self.learn_step_counter % TARGET_REPLACE_ITER == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())
        self.learn_step_counter += 1

        sample_index = np.random.choice(MEMORY_CAPACITY, BATCH_SIZE)
        b_memory = self.memory[sample_index, :]
        b_s = torch.FloatTensor(b_memory[:, :N_STATES])
        b_a = torch.LongTensor(b_memory[:, N_STATES:N_STATES+1].astype(int))
        b_r = torch.FloatTensor(b_memory[:, N_STATES+1:N_STATES+2])
        b_s_ = torch.FloatTensor(b_memory[:, -N_STATES:])

        q_eval = self.eval_net(b_s).gather(1, b_a)
        q_next = self.target_net(b_s_).detach()
        q_target = b_r + GAMMA * q_next.max(1)[0].view(BATCH_SIZE, 1)
        loss = self.loss_func(q_eval, q_target)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

`learn()` is the key function. A transition (s_t, a_t, r_t, s_{t+1}) is randomly sampled from the buffer. The loss is calculated as introduced in previous, and the parameters of the networks are updated.

3.2 DDQN.py

The most parts of `DDQN.py` is the same as `DQN.py`. The only difference is the `learn()` function.

```
    def learn(self):
        if self.learn_step_counter % TARGET_REPLACE_ITER == 0:
            self.target_net.load_state_dict(self.eval_net.state_dict())
        self.learn_step_counter += 1

        sample_index = np.random.choice(MEMORY_CAPACITY, BATCH_SIZE)
        b_memory = self.memory[sample_index, :]
        b_s = torch.FloatTensor(b_memory[:, :N_STATES])
        b_a = torch.LongTensor(b_memory[:, N_STATES:N_STATES+1].astype(int))
        b_r = torch.FloatTensor(b_memory[:, N_STATES+1:N_STATES+2])
        b_s_ = torch.FloatTensor(b_memory[:, -N_STATES:])

        q_eval = self.eval_net(b_s).gather(1, b_a)
        q_next = self.target_net(b_s_)
        q_next_cur = self.eval_net(b_s_)
        q_target = b_r + GAMMA * \
            q_next.gather(1, torch.max(q_next_cur, 1)
                          [1].unsqueeze(1)).squeeze(1)
```

```
        loss = self.loss_func(q_eval, q_target.detach())

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

The code of the `learn()` function is modified based on the equation in the section 2.2.

3.3 main.py

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-n', '--episode_num', type=int, default=500)
    parser.add_argument('-a', '--action', type=str, default='plot')

    args = parser.parse_args()
    episode_num = args.episode_num
    action = args.action

    if action == 'plot':
        plot(episode_num)
    elif action == 'train':
        train(episode_num)


if __name__ == '__main__':
    main()
```

`main()` use parser to get the parameters entered on the command line, and depending on these parameters, training the models or drawing curves is chosen.

training

```
def train_net(net, episode_num):
    return_list = []
    step_list = []
    score_list = []
    for i in range(episode_num):
        s = env.reset()
        ep_r = 0
        step = 0
        score = 0
        discount = 1.0
        while True:
            step += 1
            # env.render()
            a = net.choose_action(s)
            s_, r, done, info = env.step(a)

            if s_[0] <= -0.5:
                r = 100 * abs(s_[1])
            elif s_[0] > -0.5 and s_[0] <= 0.5:
                r = math.pow(2, 5*(s_[0] + 1)) + (100 * abs(s_[1])) ** 2
            else:
                r = 1000
            net.store_transition(s, a, r, s_)
```

```
            score += discount * r
            discount *= GAMMA

            ep_r += r
            if net.memory_counter > MEMORY_CAPACITY:
                net.learn()
                if done:
                    return_list.append(ep_r)
                    step_list.append(step)
                    score_list.append(score)
            if done:
                break
            s = s_
    return return_list, step_list, score_list

def train(episode_num):
    dqn = DQN()
    ddqn = DDQN()
    dqn_return_list, dqn_step_list, dqn_score_list = train_net(
        dqn, episode_num)
    ddqn_return_list, ddqn_step_list, ddqn_score_list = train_net(
        ddqn, episode_num)
    np.save('dqn_return_list', dqn_return_list)
    np.save('dqn_step_list', dqn_step_list)
    np.save('dqn_score_list', dqn_score_list)
    np.save('ddqn_return_list', ddqn_return_list)
    np.save('ddqn_step_list', ddqn_step_list)
    np.save('ddqn_score_list', ddqn_score_list)
```

In the training part, The return, step number and score of each episode are stored in three lists. The lists are saved as the `.npy` files and will be used to plot the curves.

The origin reward is not good enough to train the model, so we modified the reward function.The chosen of the reward function references this blog.

$$r_t = \begin{cases} 1000, & \text{if } x_t \geqslant 0.5 \\ 2^{5(x_t+1)} + (100|v_t|)^2, & \text{if } -0.5 < x_t < 0.5 \\ 0 + 100|v_t|, & \text{if } x_t \leqslant -0.5 \end{cases}$$

When the car coordinates $x < -0.5$, although the position is not good, in order to accelerate to the right, we still need to make sure that the absolute value of velocity is large.
When the car coordinate $x > -0.5$, at this time, the more rightward the position, the bigger the reward; and the bigger the absolute value of speed, the reward should be steeper.

plotting

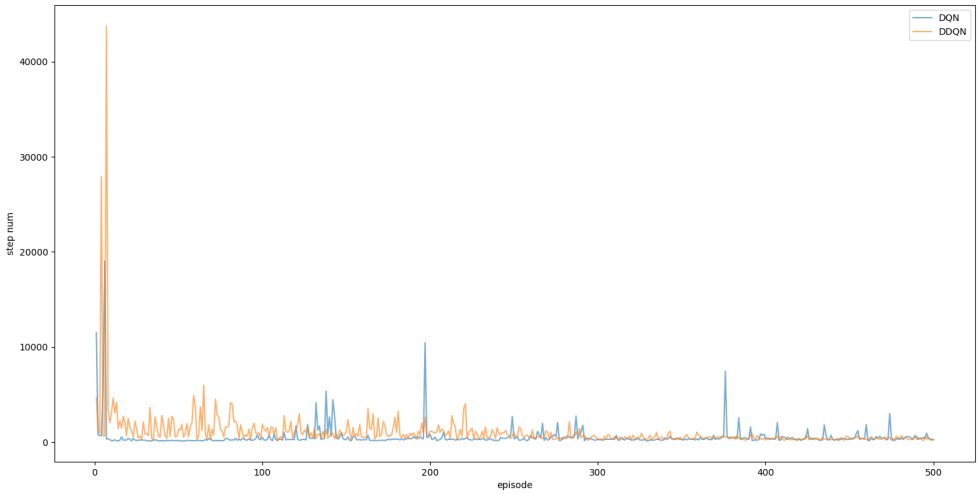
```
def plot(episode_num):
    dqn_return_list = np.load('dqn_return_list.npy', allow_pickle=True)
    dqn_step_list = np.load('dqn_step_list.npy', allow_pickle=True)
    dqn_score_list = np.load('dqn_score_list.npy', allow_pickle=True)
    ddqn_return_list = np.load('ddqn_return_list.npy', allow_pickle=True)
    ddqn_step_list = np.load('ddqn_step_list.npy', allow_pickle=True)
    ddqn_score_list = np.load('ddqn_score_list.npy', allow_pickle=True)

```

```
    ep_range = [i + 1 for i in range(episode_num)]

    plt.figure(figsize=(18, 9))
    plt.plot(ep_range, dqn_step_list, label='DQN', alpha=0.6)
    plt.plot(ep_range, ddqn_step_list, label='DDQN', alpha=0.6)
    plt.xlabel('episode')
    plt.ylabel('step num')
    plt.legend()
    plt.savefig('step.png')

    plt.figure(figsize=(18, 9))
    plt.plot(ep_range, dqn_return_list, label='DQN', alpha=0.6)
    plt.plot(ep_range, ddqn_return_list, label='DDQN', alpha=0.6)
    plt.xlabel('episode')
    plt.ylabel('return')
    plt.legend()
    plt.savefig('return.png')

    plt.figure(figsize=(18, 9))
    plt.plot(ep_range, dqn_score_list, label='DQN', alpha=0.6)
    plt.plot(ep_range, ddqn_score_list, label='DDQN', alpha=0.6)
    plt.xlabel('episode')
    plt.ylabel('score')
    plt.legend()
    plt.savefig('score.png')
```

4 Experimental Result

We can enable `env.rander()` to observe the training of the car. We can find that both in DQN and DDQN, the car can easily get into the position of flag in just several episodes.

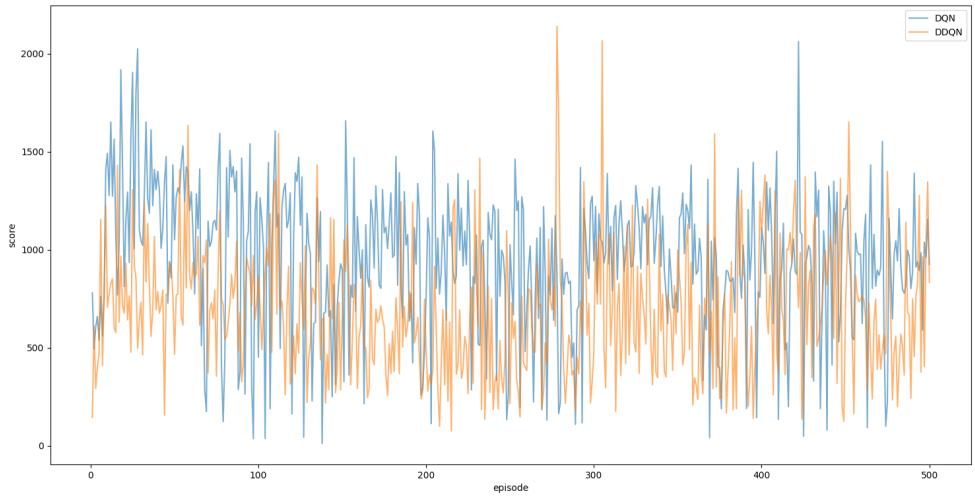
The curve of the step number:



DQN converges quickly while DDQN converges in around 250 episodes.

The curve of the total return:

The curve of the score:



From the above figures, we can conclude that DDQN is indeed more stable than DQN.

In fact, the differences between the result of DQN and the results of DDQN is not so large. I think the biggest difference between the two is that DDQN takes longer to converge, but is more stable than DQN after convergence. In some ways, such as score, DQN is even higher than DDQN in average. I think the reason for this phenomenon is that the hyper parameters chosen during training are not ideal, resulting in poor training of DDQN.

5 Experimental Summary

- Systematically learn and understand DQN and DDQN
- Learn to use pytorch to build neural networks
- Implement and successfully run the algorithm of DQN and DDQN based on pseudo-code
- Use matplotlib to plot curves