Assignment 4

1. Introduction

Although Q-learning is already a good idea, but the problem is that the Q table of two dimensions(state and action) need to be completely stored. It's okay for simple problems like cliff walking, but for more complex problems where there are infinite number of states or actions, it's impossible to store the whole Q-table.

Therefore, we should use a value function to fit the values instead of storing all of them. One good way is using deep neural network, which is suitable for this. But as reinforcement learning tends to be unstable, directly applying deep neural network to it may not leads to convergence. On the basis of this, deep Q-network (DQN) is invented. And for further improvement, Double DQN is applied.

In this report, I will apply DQN and Double DQN to solve the problem of mountain car.

2. Task: Mountain Car

The game, mountain car, is shown in figure 1. The car is on a one-dimensional track, between two "mountains". The goal is to head to the mountain on the right. However, the car's engine isn't strong enough to get over the mountain by directly driving toward right. So the only way to succeed is to drive back and forth to build momentum. And our task is to get this car to the end on the right with as few steps as possible. And the episode would terminate whenever the car reaches the small yellow flag on the right or exceeds 200 iterations.

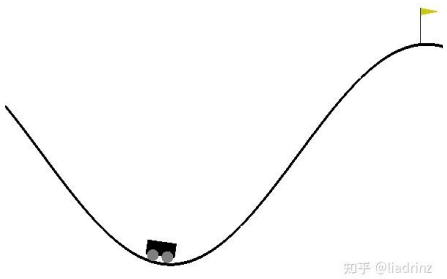

Figure 1. Mountain Car

The car has two states, velocity and position. And for each time step, the car has three discrete action space Action, 0 for push left, 1 for no push, and 2 for push right. For evert time step of these variables are updated by each other, velocity by action and position, position by velocity. The termination condition is position >= 0.6. In addition, Reward of each step is -1, while steps toward terminal state is 0.5.

In this experiment we use gym to provides this environment, just input current action, gym would output next state, reward and whether it's done.

3. Deep Q Network (DQN)

3.1 Introduction

As mentioned above, in comparison with Q-learning, DQN makes two main improvements.

First, DQN uses experience replay to solve the correlation and non-static distribution problem. Experience replay uses a random sample of prior actions instead of the most recent action. Each step the agent generates a sample of (state, action, next_action, reward), and it will be stored and randomly chosen to train the network. In this way, it removes correlations and improves the utilization of data.

Second, DQN uses a Q-network to fit the value function and a target network to decide the action. The Q-network will be updated after every step while the target network updates every N steps. Such a target network fixed the policy when Q-network updates which leads to stability.

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

Figure 2. DQN Algorithm

3.2 Implementation

The structure of network is:

1. Input: position and velocity;
2. Output: reward of three actions;
3. Hidden layer: two fully-connected layer with 256 nodes;

```
env = gym.make('MountainCar-v0')
episodes = 1000
reward_list = []
agent = DQN()
for episode in range(episodes):
    s = env.reset()
    total_reward = 0
    if episode >= 5:
        agent.update_greedy()
    while True:
        a = agent.get_best_action(s)
        next_s, reward, done, _ = env.step(a)
        agent.remember(s, a, next_s, reward)
        total_reward += reward
        if episode >= 5:
            agent.train()
        s = next_s
        if done:
            reward_list.append(total_reward)
            print('episode:', episode, 'reward:', total_reward, 'max_reward:', max(reward_list))
            break
    if np.mean(reward_list[-10:]) > -130:
        agent.save_model()
        break
env.close()
```

Figure 3. Main Function

For the first few episodes, the agent doesn't start training, they are used to fill the replay queue. And the greedy value decreases with time, means at first the agent choose rather random actions, and gradually choose best actions. And when the result is good enough, it stops training.

```
def create_model(self):
    STATE_DIM, ACTION_DIM = 2, 3
    model = models.Sequential([
        layers.Dense(256, input_dim=STATE_DIM, activation='tanh'),
        layers.Dense(256, input_dim=256, activation='tanh'),
        layers.Dense(ACTION_DIM, activation="linear")
    ])
    model.compile(loss='mean_squared_error',
                  optimizer=optimizers.Adam(0.001))
    return model
```

Figure 3. Definition of network

```
def train(self, batch_size=32, factor=1):
    self.step += 1
    if self.step % self.update_freq == 0:
        self.target_model.set_weights(self.model.get_weights())

    replay_batch = random.sample(self.replay_queue, batch_size)
    s_batch = np.array([replay[0] for replay in replay_batch])
    next_s_batch = np.array([replay[2] for replay in replay_batch])

    Q = self.model.predict(s_batch)
    Q_next = self.target_model.predict(next_s_batch)

    for i, replay in enumerate(replay_batch):
        _, a, _, reward = replay
        Q[i][a] = reward + factor * np.amax(Q_next[i])
    self.model.fit(s_batch, Q, verbose=0)
```

Figure 4. Definition of training

3.3 Result

In the experiment, maximum number of episodes is set to 1000, but around 500 episodes are enough to produce a decent outcome. I record the total reward of each episode to evaluate DQN, which is shown in Figure 5. At the beginning, the total reward is -200, which means it could not reach terminal state within 200 steps so the environment will close. And as the training goes on, it could reach terminal state sometimes. Finally, for consecutive 10 episodes, the total reward is higher than -130, which means the policy at this point is quite good(the best possible outcome is around -100), so training is stopped and model is stored.

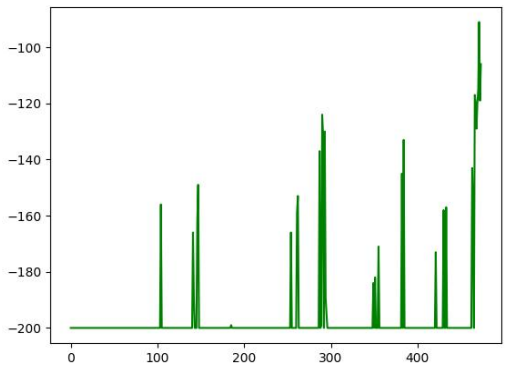

Figure 5. Record of total reward in DQN

And then to see what the strategy really looks like, load the model and play the game according to it. It takes 105 steps to the top, and I have record a video(DQN.mp4) of it.

4. Double DQN

4.1 Introduction

Although DQN achieve quite a good outcome, there is a problem. Because it always choose the highest Q-value to update neutral network, it tends to overestimate Q-Values. To solve this, Double DQN was invented. Double DQN introduces another network, and it reduces Q-Value overestimation by splinting max operator of DQN into action selection using original network and action evaluation using another network.

$$\text{DQN: } Q(s_t, a_t) \leftrightarrow r_t + \max_a Q(s_{t+1}, a)$$

$$\text{Double DQN: } Q(s_t, a_t) \leftrightarrow r_t + Q'(s_{t+1}, \operatorname{argmax}_a Q(s_{t+1}, a))$$

4.2 Implementation

As the natural DQN algorithm also uses an additional network for fixing Q-value for a while, so no additional networks are actually added compared with DQN. Weights of the second network are replaced with the weights of the target network for the evaluation of the policy. Target Network is still updated every N episodes, by copying parameters from Q-Network.

And to better converge, when calculating target Q value, if the next state is the terminal state(done = True), then the Q value would be considerably higher, leading the car this way.

```
def train(self, batch_size=32, factor=1):
    self.step += 1
    if self.step % self.update_freq == 0:
        self.target_model.set_weights(self.model.get_weights())

    replay_batch = random.sample(self.replay_queue, batch_size)
    s_batch = np.array([replay[0] for replay in replay_batch])
    next_s_batch = np.array([replay[2] for replay in replay_batch])

    Q = self.model.predict(s_batch)
    Q1 = self.target_model.predict(next_s_batch)
    Q2 = self.model.predict(next_s_batch)
    next_action = np.argmax(Q2, axis=1)

    for i, replay in enumerate(replay_batch):
        _, a, next_s, reward, done = replay
        Q[i][a] = reward + factor * Q1[i][next_action[i]] * (1 - done)
    self.model.fit(s_batch, Q, verbose=0)
```

Figure 6. Training of DDQN

4.3 Result

Here is the record of total rewards during training. Compared to natural DQN, it could reach terminal state more frequently, and after about 500 episodes, it reaches a rather steady state.

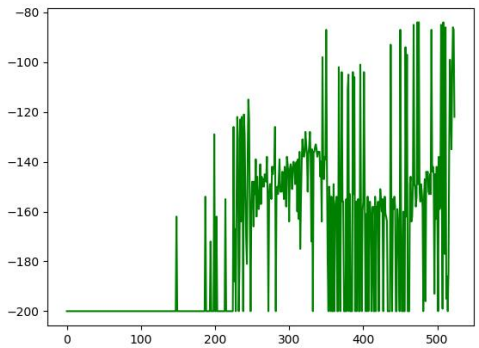

Figure 7. Record of total reward in Double DQN

In my testing, it takes 117 steps to the top, and I have record a video(Double DQN.mp4) of it.