**Assignment 4: DQN and improvement of DQN**

1. Implementation of DQN

#Define a DQN network

class DQN(nn.Module):

    def \_\_init\_\_(self, input\_dim, output\_dim):

        super(DQN, self).\_\_init\_\_()

        self.fc1 = nn.Linear(input\_dim, 128)

        self.relu = nn.ReLU()

        self.fc2 = nn.Linear(128, output\_dim)

    def forward(self, x):

        x = self.relu(self.fc1(x))

        x = self.fc2(x)

        return x

# Define a Buffer class

class ReplayBuffer:

    def \_\_init\_\_(self, capacity):

        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next\_state, done):

        self.buffer.append((state, action, reward, next\_state, done))

    def sample(self, batch\_size):

        sample = random.sample(self.buffer, batch\_size)

        states, actions, rewards, next\_states, dones = zip(\*sample)

        return np.array(states), np.array(actions), np.array(rewards), np.array(next\_states), np.array(dones)

    def \_\_len\_\_(self):

        return len(self.buffer)

These are definition of classes DQN and replay buffer. Then we imply the function of DQN train:

# Train function of DQN

def train\_dqn(env\_name):

    env = gym.make(env\_name)

    input\_dim = env.observation\_space.shape[0]

    output\_dim = env.action\_space.n

    policy\_net = DQN(input\_dim, output\_dim)

    target\_net = DQN(input\_dim, output\_dim)

    target\_net.load\_state\_dict(policy\_net.state\_dict())

    target\_net.eval()

    optimizer = optim.Adam(policy\_net.parameters())

    buffer = ReplayBuffer(10000)

    episodes = 300

    batch\_size = 64

    gamma = 0.99

    epsilon = 0.1

    for episode in range(episodes):

        state = env.reset()

        total\_reward = 0

        while True:

            if random.random() < epsilon:

                action = env.action\_space.sample()

            else:

                state\_t = torch.FloatTensor(state).unsqueeze(0)

                action = policy\_net(state\_t).max(1)[1].item()

            next\_state, reward, done, \_ = env.step(action)

            buffer.push(state, action, reward, next\_state, done)

            state = next\_state

            total\_reward += reward

            if len(buffer) > batch\_size:

                states, actions, rewards, next\_states, dones = buffer.sample(batch\_size)

                states = torch.FloatTensor(states)

                actions = torch.LongTensor(actions)

                rewards = torch.FloatTensor(rewards)

                next\_states = torch.FloatTensor(next\_states)

                dones = torch.FloatTensor(dones)

                q\_values = policy\_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

                next\_q\_values = target\_net(next\_states).max(1)[0]

                expected\_q\_values = rewards + gamma \* next\_q\_values \* (1 - dones)

                loss = (q\_values - expected\_q\_values.detach()).pow(2).mean()

                optimizer.zero\_grad()

                loss.backward()

                optimizer.step()

            if done:

                break

        if episode % 10 == 0:

            target\_net.load\_state\_dict(policy\_net.state\_dict())

        print(f'Episode: {episode}, Total reward: {total\_reward}')

    env.close()

Here I set some parameters as the following:

    episodes = 300

    batch\_size = 64

    gamma = 0.99

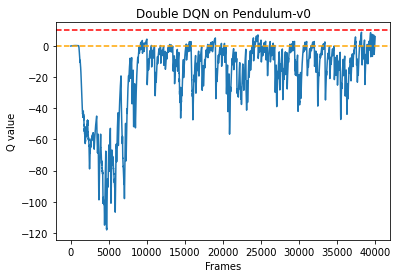
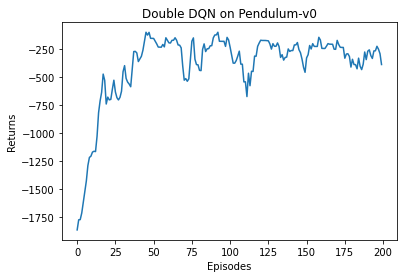
epsilon = 0.1

1. Implementation of double DQN

Double DQN is order to solve the over estimating problem of DQN, in this part we need to change the next\_actions and next\_q\_values.

1. Comparison between DQN and Double-DQN

Problems occurs when I update the state of the DQN and Double-DQN, it seems that the environment MountainCar-v0 cannot provide a correct 4 parameters state\_info, so I tried another environment “Pendulum” and got the following results:



We notice that comparing to normal DQN, the Q value of Double DQN is less frequently up to 0, which means the overestimation of Q value is solved.