Reinforcement Learning Assignment 4

Chunlin Chen, 519021910463

April 13, 2023

1 Problem Description

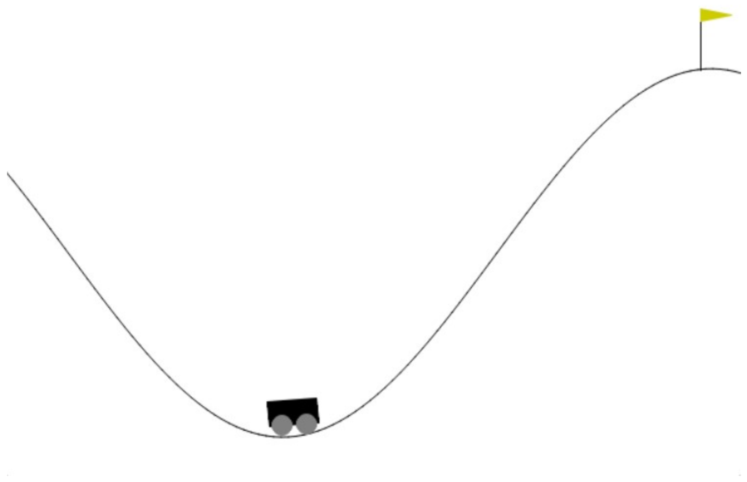


Figure 1: Mountain Car

Consider a classic RL control problem Mountain Car shown in the Figure 1, which consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the problem is to strategically accelerate the car to reach the goal state on top of the right hill.

The observation is a 2-dimensional array, where dim-0 represents the position of the car along the x-axis and dim-1 represents the velocity of the car.

There are 3 discrete deterministic actions: 0 represents "Accelerate to the left", 1 represents "Don't accelerate", and 2 represents "Accelerate to the right".

Given an action, the mountain car follows the following transition dynamics:

$$velocity_{t+1} = velocity_t + (action - 1) * force - cos(3 * position_t) * gravity$$

$$position_{t+1} = position_t + velocity_{t+1}$$

where $force = 0.001$ and $gravity = 0.0025$. The collisions at either end are inelastic with the velocity set to 0 upon collision with the wall. The position is clipped to the range $[-1.2, 0.6]$ and velocity is clipped to the range $[-0.07, 0.07]$.

The goal is to reach the flag placed on top of the right hill as quickly as possible, as such the agent is penalised with a reward of -1 for each timestep.

The position of the car is assigned a uniform random value in $[-0.6, -0.4]$. The starting velocity of the car is always assigned to 0.

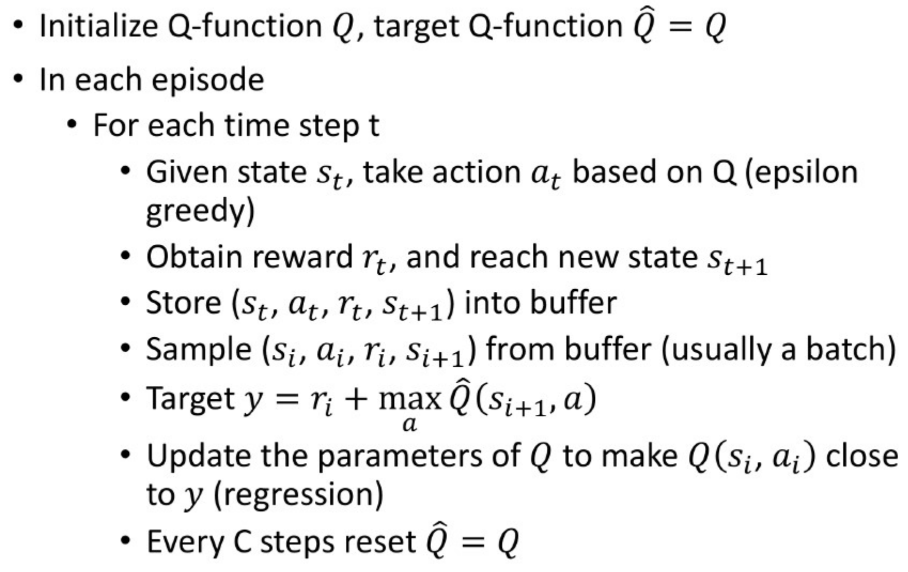
We solve the problem by implementing the following 2 algorithms and compare their performances respectively:

- DQN
- Double DQN(DDQN)

2 Implementation of Deep Q-Learning Algorithms

2.1 DQN

The workflow of DQN is shown in Figure 2:



```
• Initialize Q-function $Q$, target Q-function $\hat{Q} = Q$
• In each episode
    • For each time step t
        • Given state $s_t$, take action $a_t$ based on Q (epsilon greedy)
        • Obtain reward $r_t$, and reach new state $s_{t+1}$
        • Store $(s_t, a_t, r_t, s_{t+1})$ into buffer
        • Sample $(s_i, a_i, r_i, s_{i+1})$ from buffer (usually a batch)
        • Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
        • Update the parameters of $Q$ to make $Q(s_i, a_i)$ close to $y$ (regression)
        • Every C steps reset $\hat{Q} = Q$
```

Figure 2: The DQN Algorithm

According to the above algorithm, we can define the DQN agent in a class DQN, which includes the following variables and methods:

```python
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()
```

The variable “learn_step_counter” is to help the target network copy the parameters of the updated network at a certain frequency, and “memory_counter” is to help store the new transition into the memory buffer and maintain it within a fixed capacity. Here we use the Adam algorithm as the optimizer and Mean Square Error as the loss function for the training process.

The action at each timestep is chosen based on an ϵ-greedy policy as follows:

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

The memory buffer is maintained as follows:

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

And the Q-learning process is implemented in a function as shown below:

```
def learn(self):
    # target net parameter update
    if self.learn_step_counter % TARGET_UPDATE_FREQ == 0:
        self.target_net.load_state_dict(self.eval_net.state_dict())

    # sample batch transitions
    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 w.r.t the action in experience
    q_eval = self.eval_net(b_s).gather(1, b_a) # shape (batch, 1)
    q_next = self.target_net(b_s_).detach() # detach from graph, don't
        backpropagate
    q_target = b_r + GAMMA * q_next.max(1)[0].view(BATCH_SIZE, 1) #
        shape (batch, 1)
    loss = self.loss_func(q_eval, q_target)

    self.learn_step_counter += 1

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

As for the network part, we build a Q network consisting of 2 Fully-Connected Layers and 1 activation function, which is shown below:

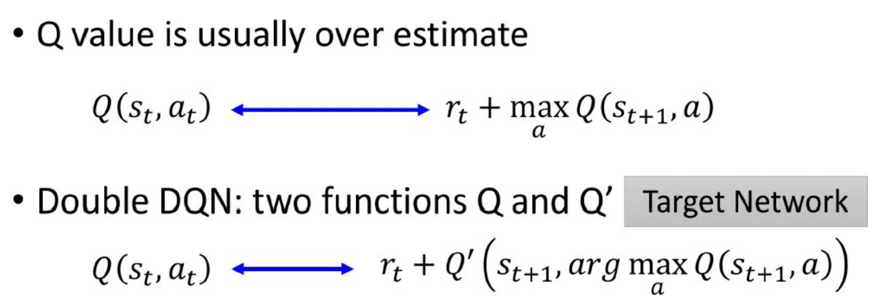
```
class Net(nn.Module):
    def __init__(self, ):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(N_STATES, 128)
        self.out = nn.Linear(128, N_ACTIONS)

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

Finally, we can run the DQN algorithm like this:

```
agent = DQN()

for i_episode in range(N_EPISODES):
    s = env.reset()[0]
    ep_r = 0
    steps = 0

    while True:
        env.render()
        a = agent.choose_action(s)

        # take action
        next_s, r, done, info, _ = env.step(a)
        # update the memory buffer
        agent.store_transition(s, a, r, next_s)

        ep_r += r
        steps += 1
        # update the Q network
        agent.learn()

        if steps == MAX_STEPS:
            done = True

        if done:
            break

        s = next_s
```

2.2 Double DQN(DDQN)

The Double DQN algorithm differs from the DQN only in its way of updating the Q values, shown in Figure 3:



- Q value is usually over estimate

$$Q(s_t, a_t) \longleftrightarrow r_t + \max_a Q(s_{t+1}, a)$$

- Double DQN: two functions Q and Q' Target Network

$$Q(s_t, a_t) \longleftrightarrow r_t + Q'\left(s_{t+1}, arg \max_a Q(s_{t+1}, a)\right)$$

Figure 3: The Double DQN Algorithm

which needs a little modification as follows:

```
# q_eval w.r.t the action in experience
q_eval = self.eval_net(b_s).gather(1, b_a) # shape (batch, 1)
q_next = self.target_net(b_next_s).detach() # detach from graph, don't
    backpropagate
# double DQN
b_opt_a = self.eval_net(b_s).max(1)[1].view(BATCH_SIZE, 1) # derive the
    optimal action from eval net
q_target = b_r + GAMMA * q_next.gather(1, b_opt_a).view(BATCH_SIZE, 1) #
    compute the target Q value using target net
```

The rest is totally the same as DQN.

3 Results Analysis

3.1 DQN

To avoid tedious episodes during training, we set the max steps per episode be 500. With regard to the hyperparameters, we let the sample batch size be 32, the learning rate be 0.01, ϵ=0.9, and the discount factor γ=0.9. And we test the performance of DQN under different memory buffer capacities(5000, 10000, 50000) and target network update frequencies(5, 50, 100). And the results are as follows:

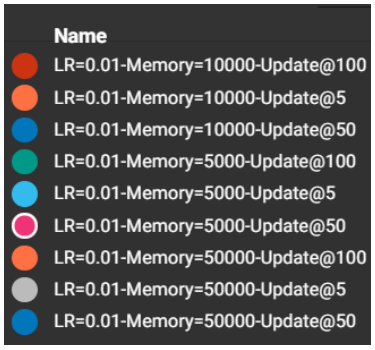

Name
● LR=0.01-Memory=10000-Update@100
● LR=0.01-Memory=10000-Update@5
● LR=0.01-Memory=10000-Update@50
● LR=0.01-Memory=5000-Update@100
● LR=0.01-Memory=5000-Update@5
● LR=0.01-Memory=5000-Update@50
● LR=0.01-Memory=50000-Update@100
● LR=0.01-Memory=50000-Update@5
● LR=0.01-Memory=50000-Update@50

Figure 4: Legend of Following Figures

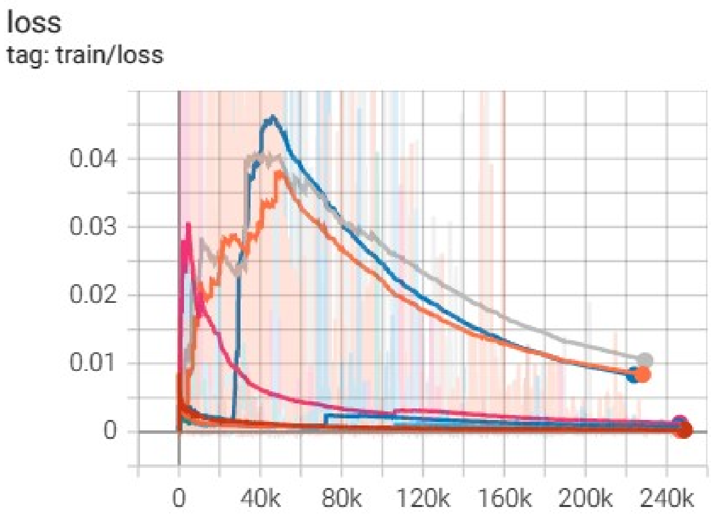


Figure 5: Losses at Different Settings

According to the Figure 5-7, we can find that the training curves of agents with 50000 memory buffer capacity significantly differs from other agents with

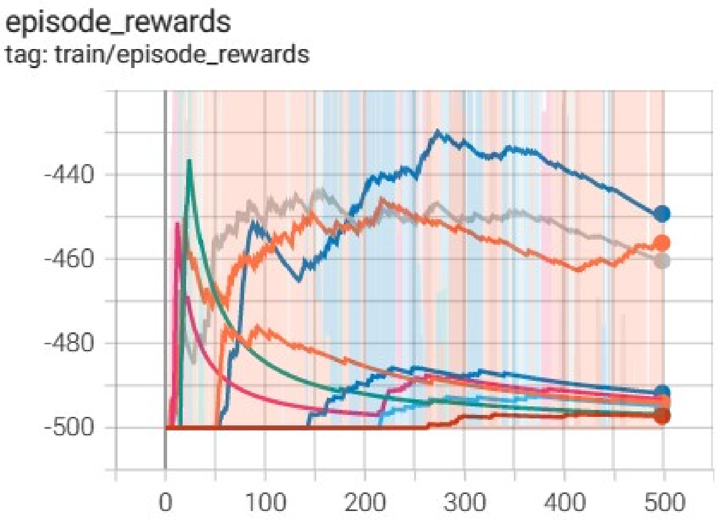


Figure 6: Episode Rewards at Different Settings

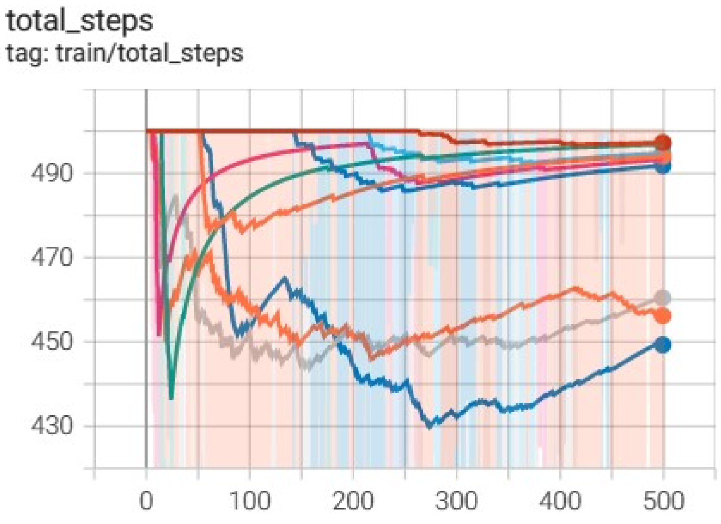


Figure 7: Steps taken per Episode at Different Settings

smaller memory capacity. Specifically, for agents with larger memory capacity, the rewards of each episode increase faster and tend to converge to a larger value(i.e., the steps taken at each episode decreases faster and tend to converge to a smaller value) during training, which means they are more likely to learn to accomplish the goal within the maximum steps and have a higher success rate.

3.2 Double DQN

Similar to the tests of DQN, we set the max steps per episode be 500, the sample batch size be 32, the learning rate be 0.01, ϵ=0.9, and the discount factor γ=0.9. Here, we test the performance of Double DQN under the memory buffer capacities(50000, 100000) and target network update frequencies(5, 100). And the results are as follows:

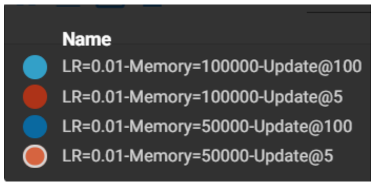

Name
● LR=0.01-Memory=100000-Update@100
● LR=0.01-Memory=100000-Update@5
● LR=0.01-Memory=50000-Update@100
● LR=0.01-Memory=50000-Update@5

Figure 8: Legend of Following Figures

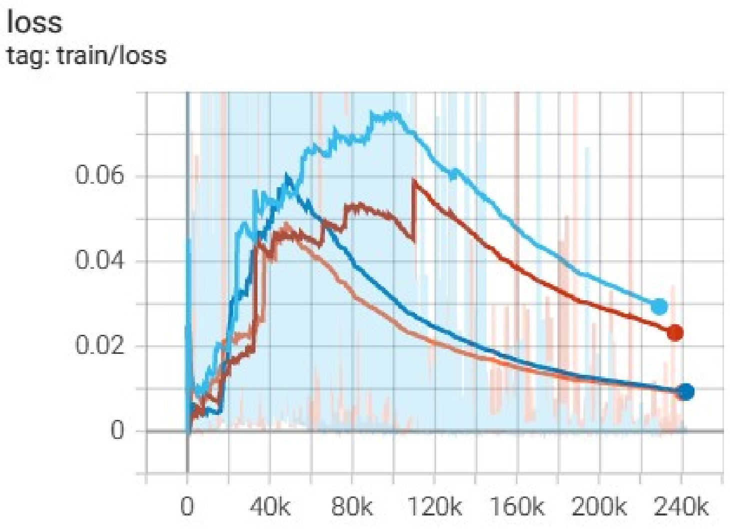


Figure 9: Losses at Different Settings

From Figure 8-10, we find that the agent with 100000 memory capacity and 100 target network update frequency shows different performances from agents with other settings, the reward curve has a strange twist during the training

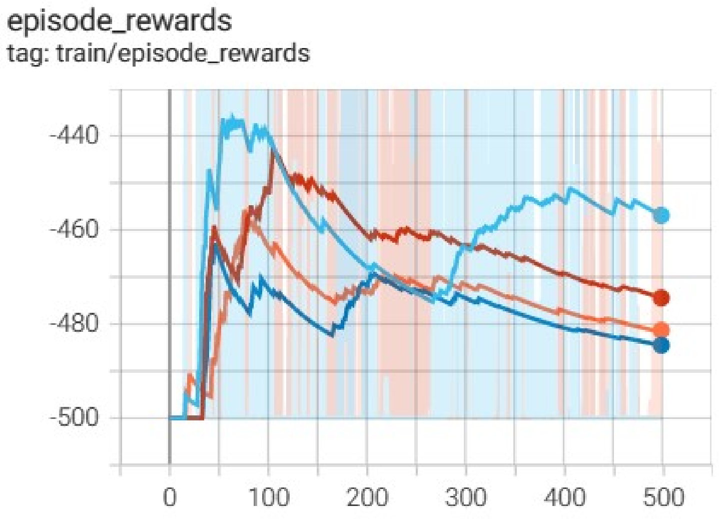


Figure 10: Episode Rewards at Different Settings

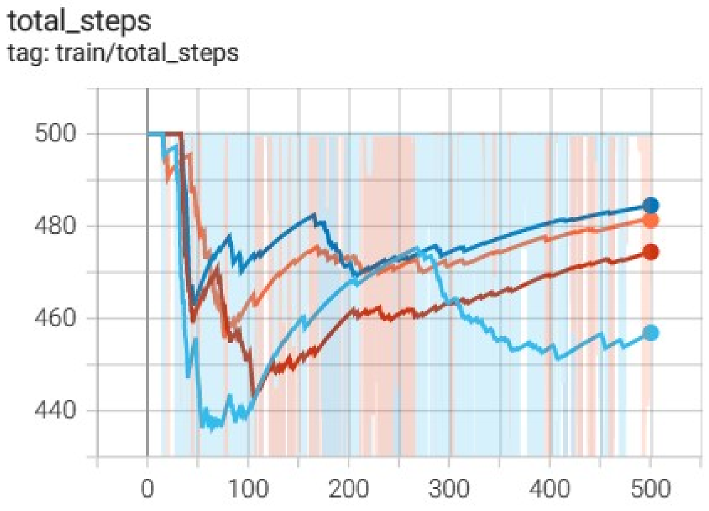


Figure 11: Steps taken per Episode at Different Settings

process, which may be caused by the high update frequency of the target network parameters.

3.3 Conclusion

From the above experiments, we find that agents with larger memory capacity significantly outperform agents with smaller memory capacity, while the target network update frequency seems to play a insignificant role in the agent's performance. However, large memory capacity undoubtedly will require large hardware memory consumption, which may pose a challenge in real-world applications. The differences of DQN and Double DQN are not very obvious in our experiments, since we did not evaluate the specific Q values they eventually converged to.