# Homework 4:

# Reinforcement Learning

# Report

110652019 林楷傑

## Part I. Implementation (-5 if not explain in detail):

- **Part1**

```python
# Begin your code
# TODO
"""
Use epsilon greedy select action
"""
if random.random() > self.epsilon:
    action = np.argmax(self.qtable[state])
else:
    action = random.randint(0, self.env.action_space.n-1)
return action
# End your code
```

```python
# Begin your code
# TODO
"""
update Q table by the formula: Q(s,a) = (1-alpha) * Q(s,a) + alpha * (r + gamma * max(Q(s')))
"""
self.qtable[state][action] = (1-self.learning_rate) * self.qtable[state][action] + self.learning_rate*(reward + self.gamma * max(self.qtable[next_state]))
# End your code
```

```python
# Begin your code
# TODO
"""
Return the max Q value of the giving state
"""
return max(self.qtable[state])
# End your code
```

- **Part2**

```python
# Begin your code
# TODO
"""
Use numpy function linspace to slice the interval into num_bins part
"""
bins = np.linspace(lower_bound, upper_bound, num_bins)
return bins[1:-1]
# End your code
```

```python
# Begin your code
# TODO
"""
Use numpy function digitize to discretize the value with given bins.
"""
bin_index = np.digitize(value, bins)
return bin_index-1
# End your code
```

```python
# Begin your code
# TODO
"""
discretize the giving state
"""
return [self.discretize_value(observation[i], self.bins[i]) for i in range(4)]
# End your code
```

```python
# Begin your code
# TODO
"""
Use epsilon greedy select action
"""
if random.random() > self.epsilon:
    return np.argmax(self.qtable[state[0]][state[1]][state[2]][state[3]])
return np.array(random.randrange(self.n_actions))
# End your code
```

```python
# Begin your code
# TODO
"""
update Q table by the formula: Q(s,a) = (1-alpha) * Q(s,a) + alpha * (r + gamma * max(Q(s')))
"""
if done:
    self.qtable[state[0]][state[1]][state[2]][state[3]][action] = self.qtable[state[0]][state[1]][state[2]][state[3]][action] + self.learning_rate * (
        reward - self.qtable[state[0]][state[1]][state[2]][state[3]][action])
else:
    self.qtable[state[0]][state[1]][state[2]][state[3]][action] = self.qtable[state[0]][state[1]][state[2]][state[3]][action] + self.learning_rate * (
        reward + self.gamma * np.max(self.qtable[next_state[0]][next_state[1]][next_state[2]][next_state[3]]) - self.qtable[state[0]][state[1]][state[2]][state[3]][action])
# End your code
```

```python
# Begin your code
# TODO
"""
Return max Q value of giving state
"""
state = self.discretize_observation(self.env.reset())
return np.max(self.qtable[state[0]][state[1]][state[2]][state[3]])
# End your code
```

- ## Part3

```python
# Begin your code
# TODO
"""
s: state; s':next state; a: action; r: reward
Sample a batch data from memory buffer.
Q_evaluate: use torch gather to get Q(s,a)
Q_next: use target net comput Q'(s', )
Q_target: r + gamma * Q'(s', argmax(Q'(s', .))) * (1-done)
Use mean square loss between Q and Q_target to update the evaluate net
"""
batch =  self.buffer.sample(self.batch_size)
#batch = (observations, actions, rewards, next_observations, done)
state_batch = Variable(Tensor(np.array(batch[0])))
action_batch = Variable(Tensor(batch[1]))
reward_batch = Variable(Tensor(batch[2]))
mask_batch = Variable(Tensor(batch[4]))
next_state_batch = Variable(Tensor(np.array(batch[3])))

Q_evaluate = torch.gather(self.evaluate_net(state_batch), 1, action_batch.view(-1, 1).long())
Q_next = self.target_net(next_state_batch).detach()
Q_target = reward_batch.view(-1, 1) + self.gamma * Q_next.max(1)[0].view(self.batch_size, 1) * (1-mask_batch.view(self.batch_size, 1))
loss = F.mse_loss(Q_evaluate, Q_target)

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

# End your code
```

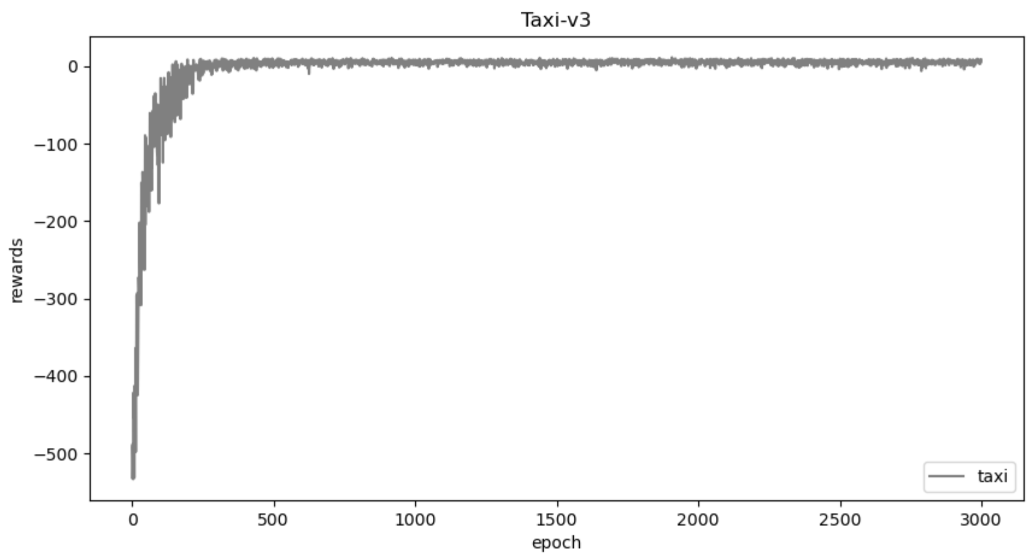
```python
# Begin your code
# TODO
"""
Use epsilon greedy select action
"""
state = torch.Tensor(state)
if random.uniform(0,1) > self.epsilon:
    Q_val = self.evaluate_net(state)
    action = torch.argmax(Q_val).item()
else:
    action = np.array(random.randrange(self.n_actions))
# End your code
```

```python
# Begin your code
# TODO
"""
Return the max Q that initial state pass through the target net.
"""
return max(self.target_net(Tensor(self.env.reset())))
# End your code
```

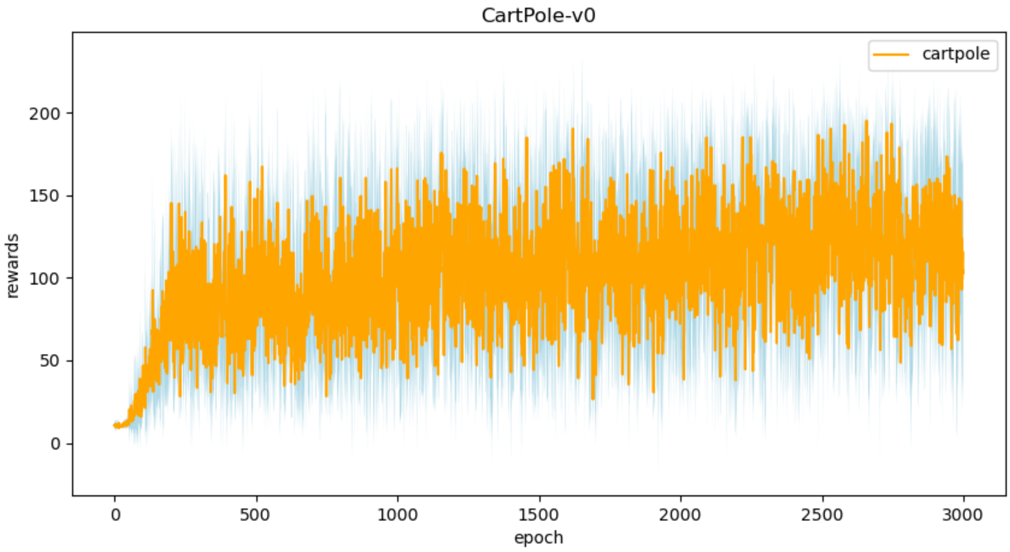
## Part II. Experiment Results:

**Please paste taxi.png, cartpole.png, DQN.png and compare.png here.**

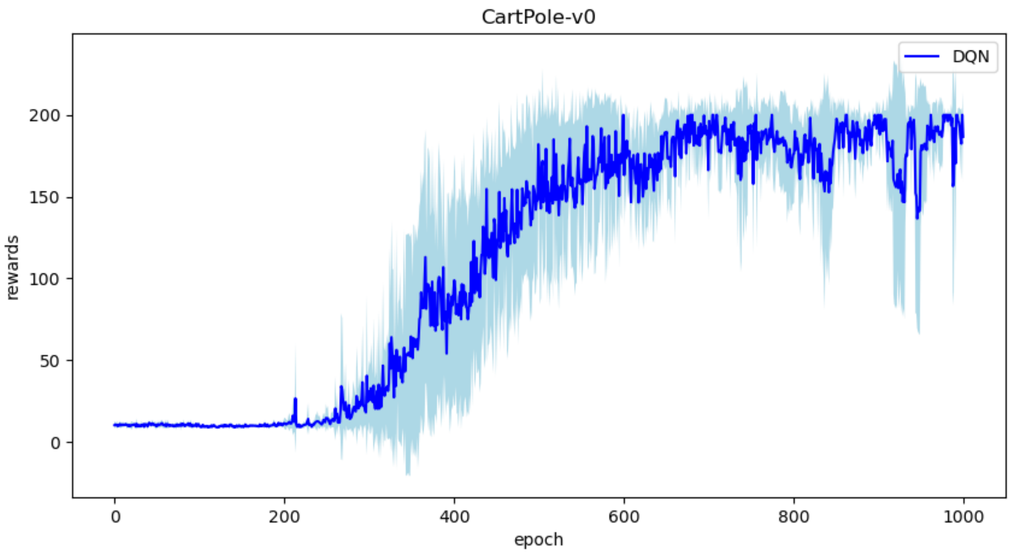
### 1. taxi.png:



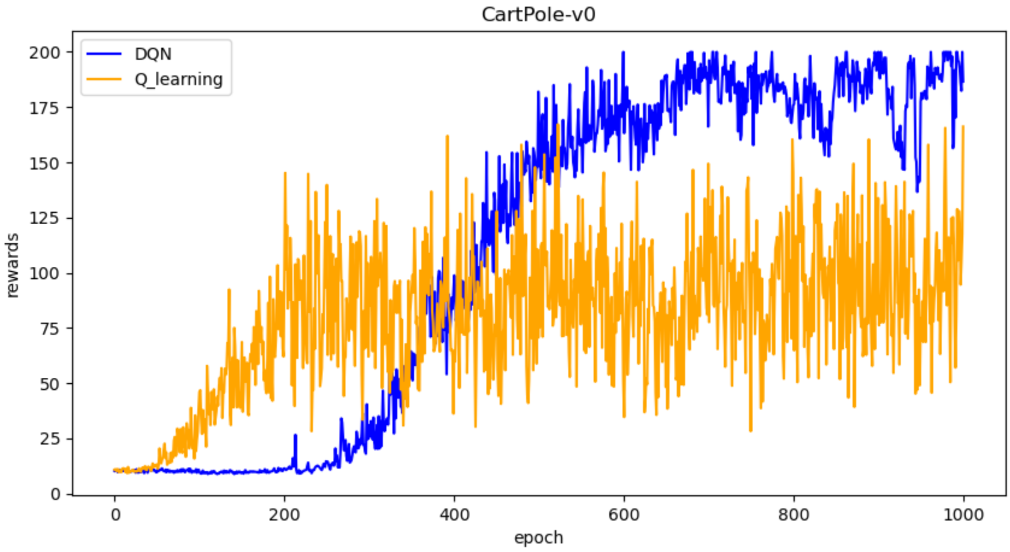
## 2. cartpole.png




## 3. DQN.png



### 4. compare.png




## Part III. Question Answering (50%):

1. Calculate the optimal Q-value of a given state in Taxi-v3, and compare with the Q-value you learned (Please screenshot the result of the "check_max_Q" function to show the Q-value you learned). **(10%)**

```
# Q opt: for state 243, at least need 9 step to reach the goal.
print("Q opt:", -(1-self.gamma**9)/(1-self.gamma) + 20 * self.gamma**9)
return max(self.qtable[state])
# End your code
```

```
average reward: 7.56
Initail state:
taxi at (2, 2), passenger at Y, destination at R
Q opt: 1.6226146700000017
max Q:1.6226146699999995
```

2. Calculate the max Q-value of the initial state in CartPole-v0, and compare with the Q-value you learned. (Please screenshot the result of the "check_max_Q" function to show the Q-value you learned) **(10%)**

```
# Q opt: Need at least 200 step to reach the final goal. The optimal reward is 200.
print("Q opt: ", (1-self.gamma**200)/(1-self.gamma))
state = self.discretize_observation(self.env.reset())
return np.max(self.qtable[state[0]][state[1]][state[2]][state[3]])
# End your code
```

```
average reward: 196.82
Q opt:  33.25795863300011
max Q:30.639871238426274
```

3.
   a. Why do we need to discretize the observation in Part 2? **(3%)**

      Since the observation of cartpole environment is continuous. We need to discretize the observation to present on the Q table.

   b. How do you expect the performance will be if we increase "num_bins"? **(3%)**

The run time of the program will be longer, but the performance will be better. Since the agent can gain more knowledge from the environment.

c. Is there any concern if we increase "num_bins"? **(3%)**
The runtime of the program would be longer.

4. Which model (DQN, discretized Q learning) performs better in Cartpole-v0, and what are the reasons? **(5%)**

   DQN. The neural network can deal the continuous space. It solves the problem that discretized Q learning encountered, that is continuous space is hard to presented on the tabular method. Thus, DQN would perform better.

5.
   a. What is the purpose of using the epsilon greedy algorithm while choosing an action? **(3%)**

      For exploration, use epsilon greedy algorithm to make the agent fully explore the  evironment.

   b. What will happen, if we don't use the epsilon greedy algorithm in the CartPole-v0 environment?  **(3%)**

      The policy will stuck in a local optimal action since the agent will always choose the max Q which first update previously and is not alctually optimal. The agent is lack of exploration.

   c. Is it possible to achieve the same performance without the epsilon greedy algorithm in the CartPole-v0 environment? Why or Why not? **(3%)**

      No, without exploration, the agent can not fully explore actions that is likely to be better.

   d. Why don't we need the epsilon greedy algorithm during the testing section? **(3%)**

      Since we have successfully trained out agent, the agent can perform well policy. We do not need randomness to disturb its decision.

6. Why does "`with torch.no_grad():`" do inside the "choose_action" function in DQN? **(4%)**

   `with torch.no_grad():` means the back propagation will not propagate through the computation. Since the decision making is depend on choosing the maximum Q value, we don'  t need to tack the gradient of this process.