Homework 4:

Reinforcement Learning

Report Template

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

Part 1:

Choose Action

```python
# Begin your code

r = np.random.uniform(0,1) # choose a random r from 0~1
if(r > self.epsilon): # choose best action from the current state
    action = np.argmax(self.qtable[state])
else: # choose a random action
    action = env.action_space.sample()

return action

# End your code
```

Learn

```python
# Begin your code

if done: max_next_Qopt = 0
else: max_next_Qopt = np.max(self.qtable[next_state])

Qopt_old = self.qtable[state, action]
# Qopt(s, a) = (1-eta) * Qopt(s, a) + eta * (reward + gamma * max_Qopt(s', a'))
Qopt_new = (1 - self.learning_rate) * Qopt_old + self.learning_rate * (reward + self.gamma * max_next_Qopt)
self.qtable[state, action] = Qopt_new # update qtable to new Qopt

# End your code
```

Check Max Q

```python
# Begin your code

return np.max(self.qtable[state]) # return the max Q value of given state

# End your code
```

Part 2:

Init Bins

```python
        # Begin your code

        # Returns num evenly spaced samples, calculated over the interval [lower, upper] without the upper point.
        bins = np.linspace(lower_bound, upper_bound, num_bins, endpoint=False)
        bins = np.delete(bins, 0) # Delete the lower point --> num-1 evenly spaced samples.

        return bins

        # End your code
```

Discretized Values

```python
        # Begin your code

        # Return the indices of the bins to which each value in input array belongs.
        return np.digitize(value, bins, right=False) # bins[i-1] ≤ value < bins[i], return i

        # End your code
```

Discretize Observation

```python
        # Begin your code

        # state = [0~6, 0~6, 0~6, 0~6] because there are 7 bins for each observation value
        state = [0, 0, 0, 0]

        for i in range(len(observation)):
            # discretize each observation value accordingly to put each value in one of the 7 bins
            state[i] = self.discretize_value(observation[i], self.bins[i])

        return state

        # End your code
```

Choose Action

```python
        # Begin your code

        r = np.random.uniform(0,1) # choose a random r from 0~1
        if(r > self.epsilon): # choose best action from the current state
            action = np.argmax(self.qtable[state[0], state[1], state[2], state[3]])
        else: # choose a random action
            action = env.action_space.sample()

        return action

        # End your code
```

Learn

```python
# Begin your code

# max_Qopt(s', a')
if done: max_next_Qopt = 0
else: max_next_Qopt = np.max(self.qtable[next_state[0], next_state[1], next_state[2], next_state[3]])

Qopt_old = self.qtable[state[0], state[1], state[2], state[3], action]
# Qopt(s, a) = (1-eta) * Qopt(s, a) + eta * (reward + gamma * max_Qopt(s', a'))
Qopt_new = (1 - self.learning_rate) * Qopt_old + self.learning_rate * (reward + self.gamma * max_next_Qopt)
self.qtable[state[0], state[1], state[2], state[3], action] = Qopt_new # update qtable to new Qopt

# End your code
```

Check Max Q

```python
# Begin your code

state = self.env.reset() # reinitialize state
state = self.discretize_observation(state) # discretize the initial state values

return np.max(self.qtable[state[0], state[1], state[2], state[3]]) # return the max Q value of the inital state

# End your code
```

Part 3:

Learn:

```python
# Begin your code

# sample trajectories of batch size from the replay buffer
sample = self.buffer.sample(self.batch_size)
state = torch.tensor(np.array(sample[0]), dtype=torch.float)
action = torch.tensor(sample[1],dtype=torch.long)
reward = torch.tensor(sample[2],dtype=torch.float)
next_state = torch.tensor(np.array(sample[3]),dtype=torch.float)
done = sample[4]

# forward the data to the evaluate net and the target net
m = torch.LongTensor( [0]*32)
eval = self.evaluate_net(state).gather(1, action.unsqueeze(1)) + m
# m = torch.LongTensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
# 0, 0, 0, 0, 0, 0, 0, 0])
# eval = eval + m
next = self.target_net(next_state).detach()
for i in range (self.batch_size) :
    if done[i]: next[i] = 0
target = reward + self.gamma * next.max(1).values.unsqueeze(-1)

# compute loss with MSE
loss_func = nn.MSELoss()
loss = loss_func(eval.float(), target.float())

# zero-out the gradients
self.optimizer.zero_grad()

# backpropagation
loss.backward()
self.optimizer.step()

# End your code
```

Choose Action

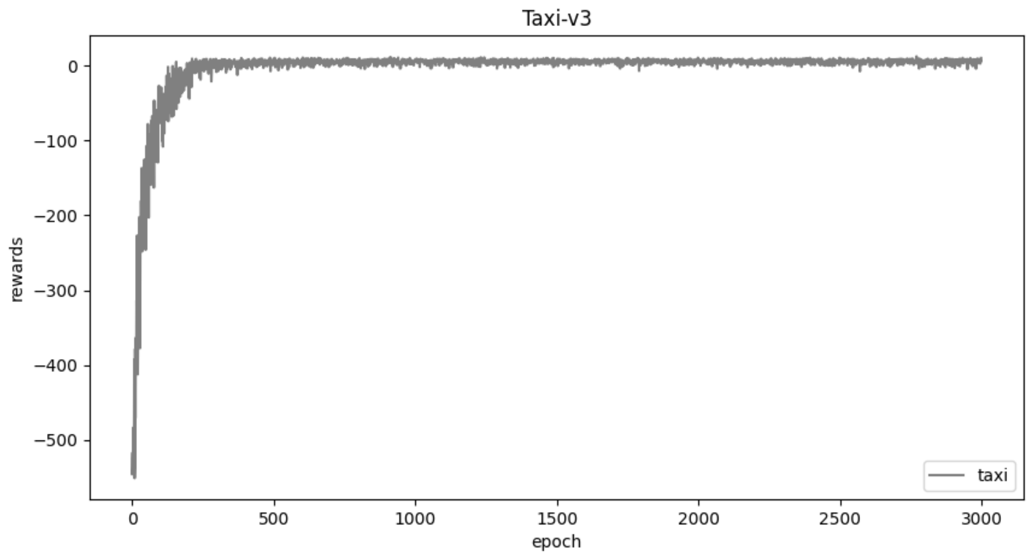
```
        # Begin your code

        x = torch.unsqueeze(torch.tensor(state, dtype=torch.float), 0)
        r = np.random.uniform(0,1) # choose a random r from 0~1
        if(r > self.epsilon): # choose best action from the current state
            action_values = self.evaluate_net(x)
            action = torch.argmax(action_values).item()
        else: # choose a random action
            action = env.action_space.sample()

        # End your code
```

Check Max Q

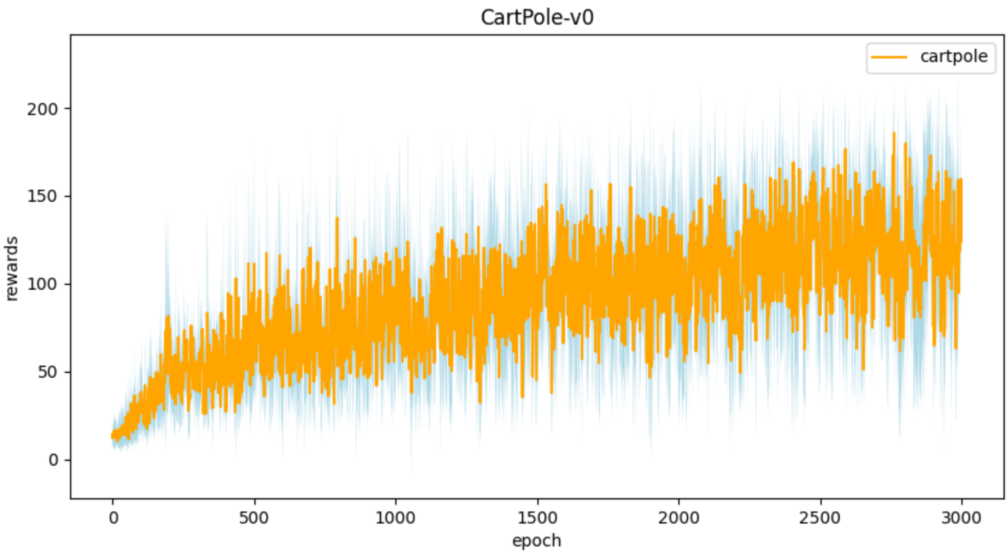
```
    # Begin your code

    state = self.env.reset() # reinitialize state
    x = torch.unsqueeze(torch.tensor(state, dtype=torch.float), 0)
    action_values = self.evaluate_net(x) # evaluate state to get Q values

    return torch.max(action_values).item() # return max Q value

    # End your code
```

Part II. Experiment Results:

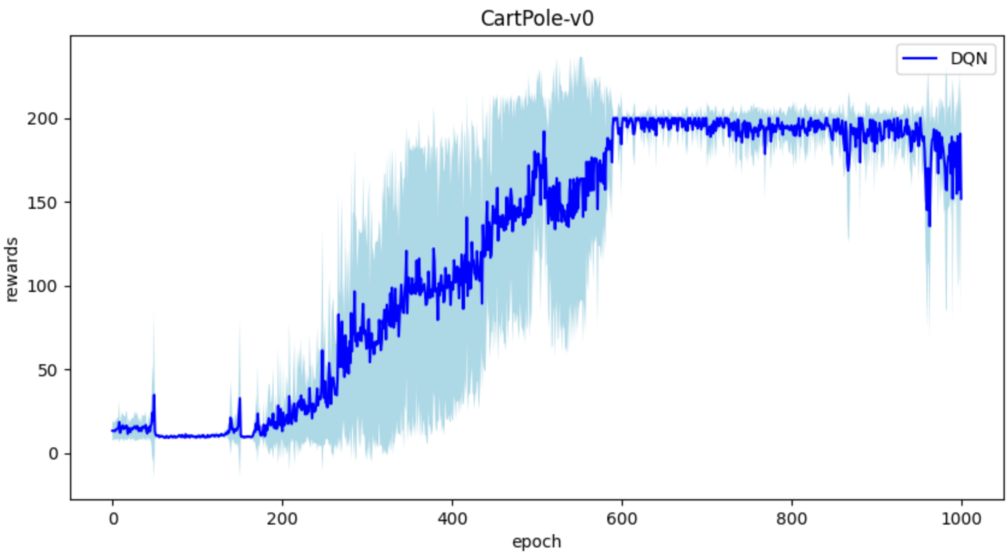
1. taxi.png:



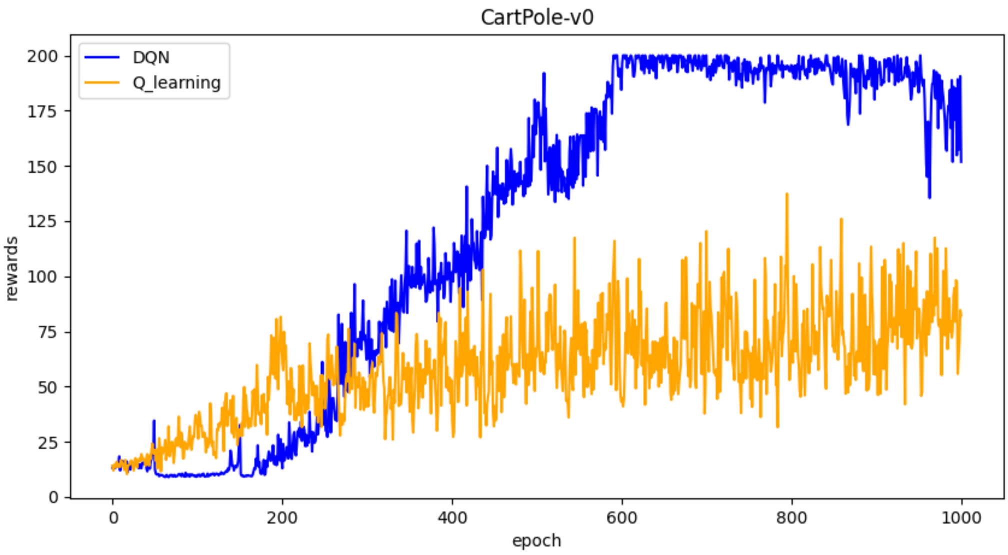
2. cartpole.png



3. DQN.png:



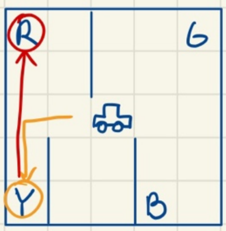
4. compare.png:



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%)

```
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146699999995
```



Taxi at (2,2)
Passenger at Y
Destination at R

5 steps to pick up (move ×4 + pickup ×1)
5 steps to drop off (move ×4 + dropoff ×1)
9 rewards of (-1) and 1 reward of (20)

Gamma = 0.9

successful dropoff

$$Q_{opt} = 0.9^0(-1) + 0.9^1(-1) + 0.9^2(-1) + \cdots + 0.9^8(-1) + 0.9^9(20)$$

$$= -1\left(\frac{1-0.9^9}{1-0.9}\right) + 0.9^9 \times 20 = 1.6226$$

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%)**

```
max Q:30.86389868745296
```

Gamma = 0.97

max steps taken

$$Q_{opt} = 0.97^0 + 0.97^1 + 0.97^2 + \cdots + 0.97^{199}$$

$$= \left(\frac{1-0.97^{200}}{1-0.97}\right) = 33.2579$$

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

    Because the observation is continuous, we have discretize them to put them into separate bins to be able to create a proper qtable.

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

    The performance would become better since there will be more qtable states that fit the data more properly.

    c. Is there any concern if we increase “num_bins” ? **(3%)**

    Increasing the num_bins would also increase the space and memory to save the increasing states in the qtable. Calculation time would also become longer as it takes more effort to go thorough all the states.

4. Which model (DQN, discretized Q learning) performs better in Cartpole-v0, and what are the reasons? **(5%)**
DQN performs better than discretized Q learning. The reason is stated in its name, discretized. Q learning discretizes its data to fit them onto a qtable. This increases efficiency but lowers performance since the data is not so accurate anymore after discretizing. DQN, on the other hand, uses raw, continuous data. It uses more training time to obtain higher performance.

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

    To maintain a balance in exploitation and exploration. When exploiting, choose the max(Q) action of current state from the constructed qtable. When exploring, choose a random action from the action space. This way, we can achieve both exploitation and exploration with balance.

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

All explore: the result would be random and there will be no learning curve.
All exploit: can only work on known information and may miss unknown high-rewarded results.

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

   If we can find another method to achieve balance in exploration and exploitation, then yes. Else, no, as reasons stated in (b).

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

   The purpose of the epsilon greedy algorithm is to balance exploration and exploitation when constructing the qtable. When testing, we will only be using the qtable to find the best result without overwriting it. So wee don't need the algorithm during testing.

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

   There's no need to calculate the gradient and back propagation in choose_action, so we disable gradiant calculation to increase performance with the function torch.no_grad().