Homework 4:

Reinforcement Learning

110550080何曉嫻

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

Part 1:

```python
        # Begin your code
        if np.random.uniform(0,1) > self.epsilon:
            action= np.argmax(self.qtable[state]) #exploitation
        else:
            action= env.action_space.sample()     # exploration
        return action
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

```python
        # Begin your code
        # calculate q value and update the qtable
        self.qtable[state, action]= self.qtable[state, action]+ self.learning_rate*\
            (reward+ (self.gamma* np.max(self.qtable[next_state]))- self.qtable[state, action])

        # End your code
```

```python
        # Begin your code
        max_q= max(self.qtable[state])
        return max_q
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

Part 2:

```python
        # Begin your code
        # create an list of arithmetic sequence(include lower_bound and upper_bound)
        bins= np.linspace(lower_bound, upper_bound, num_bins+1)
        # delete the head and tail
        bins= np.delete(bins, num_bins)
        bins= np.delete(bins, 0)
        return bins
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

```python
        # Begin your code
        # discretize the value, find the value is in which interval
        x= np.searchsorted(bins, value, side='right')
        return x
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

```python
# Begin your code
# Discretize the observation which we observed from a continuous state space.
# Observation is a list of 4 features.
state=[]
for i in range(len(observation)):
    state.append(self.discretize_value(observation[i], self.bins[i]))
return tuple(state)
raise NotImplementedError("Not implemented yet.")
# End your code
```

```python
# Begin your code
if np.random.uniform(0,1) > self.epsilon:
    action= np.argmax(self.qtable[state]) # exploitation
else:
    action= env.action_space.sample()      # exploration
return action
raise NotImplementedError("Not implemented yet.")
# End your code
```

```python
# Begin your code
# calculate q value and update the qtable (using tuple)
self.qtable[state+ (action,)]= self.qtable[state+ (action,)]+ self.learning_rate*\
    (reward+ (self.gamma* np.max(self.qtable[next_state]))- self.qtable[state+ (action,)])

# raise NotImplementedError("Not implemented yet.")
# End your code
```

```python
# Begin your code
# Discretize and find the max q
max_q= max(self.qtable[self.discretize_observation(self.env.reset())])
return max_q
raise NotImplementedError("Not implemented yet.")
# End your code
```

Part 3:

```python
# Begin your code
if random.uniform(0,1) > self.epsilon:
    # exploitation
    action= torch.argmax(self.evaluate_net.forward(torch.FloatTensor(state))).item()
else:
    # exploration
    action = self.env.action_space.sample()
return action
raise NotImplementedError("Not implemented yet.")
# End your code
```

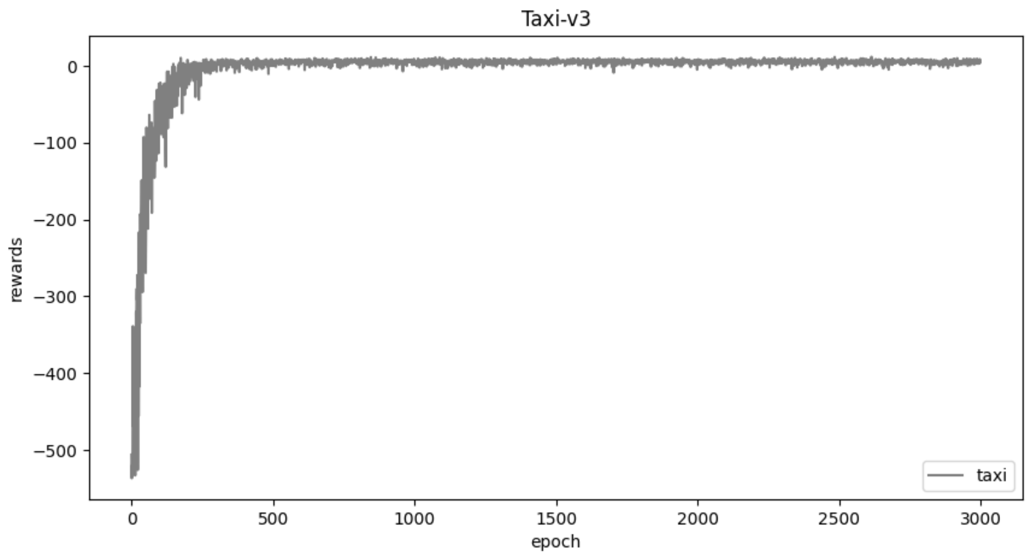
```python
# Begin your code
# Sample trajectories of batch size from the replay buffer.
states, actions, rewards, next_states, done= self.buffer.sample(self.batch_size)
states= torch.FloatTensor(np.array(states))
actions= torch.LongTensor(actions)
rewards= torch.FloatTensor(rewards)
next_states= torch.FloatTensor(np.array(next_states))
# Forward the data to the evaluate net and the target net.
q_eval= self.evaluate_net(states).gather(1, actions.reshape(self.batch_size, 1))
q_next= self.target_net(next_states).detach()
for i in range(len(done)):
    if (done[i]): q_next[i]= 0
q_target= rewards.reshape(self.batch_size, 1)+ self.gamma* q_next.max(1)[0].view(self.batch_size, 1)
# Compute the loss with MSE
func= nn.MSELoss()
loss= func(q_eval, q_target)
# Zero-out the gradients
self.optimizer.zero_grad()
# Backpropagation
loss.backward()
# Optimize the loss function
self.optimizer.step()
# End your code
```

```python
        # Begin your code
        # Use target network to estimate the maximum Q-value of the actions in the next state.
        return torch.max(self.target_net(torch.FloatTensor(self.env.reset())))
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

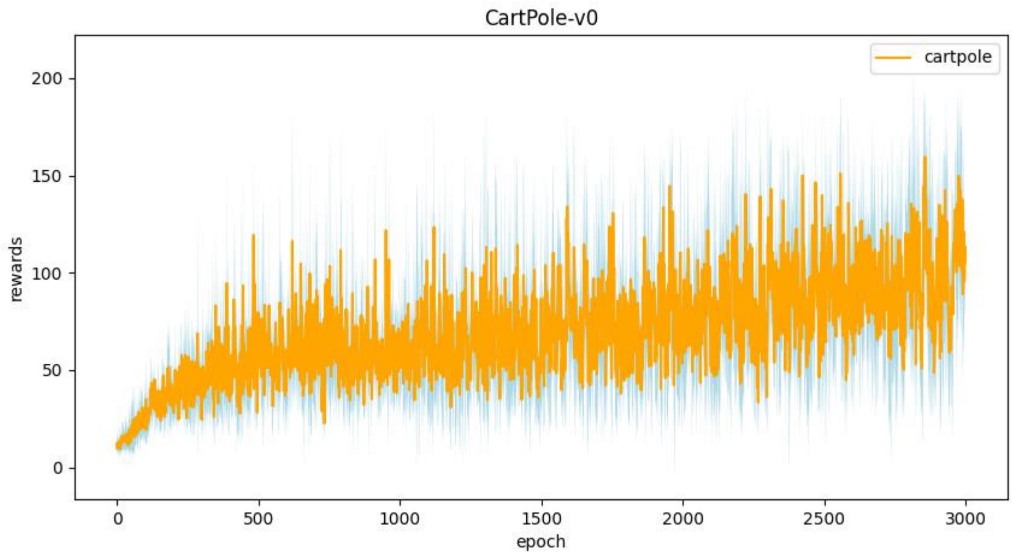
Part II. Experiment Results:

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

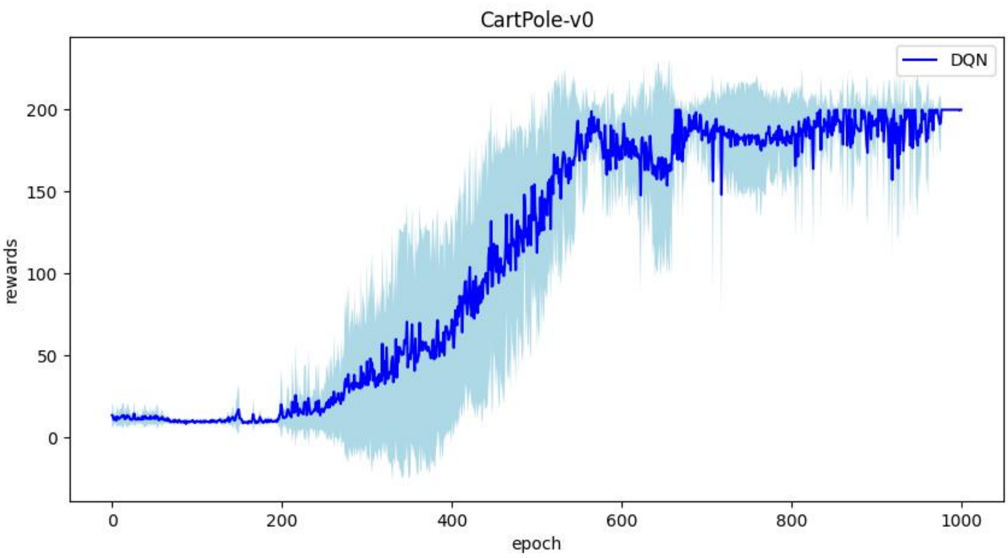
1. taxi.png



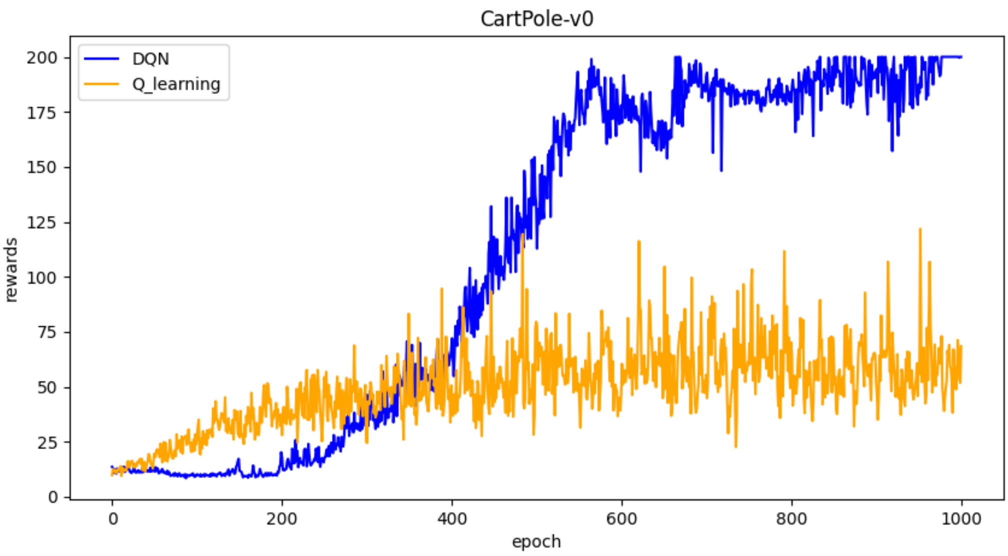
2. cartpole.png



3. DQN.png

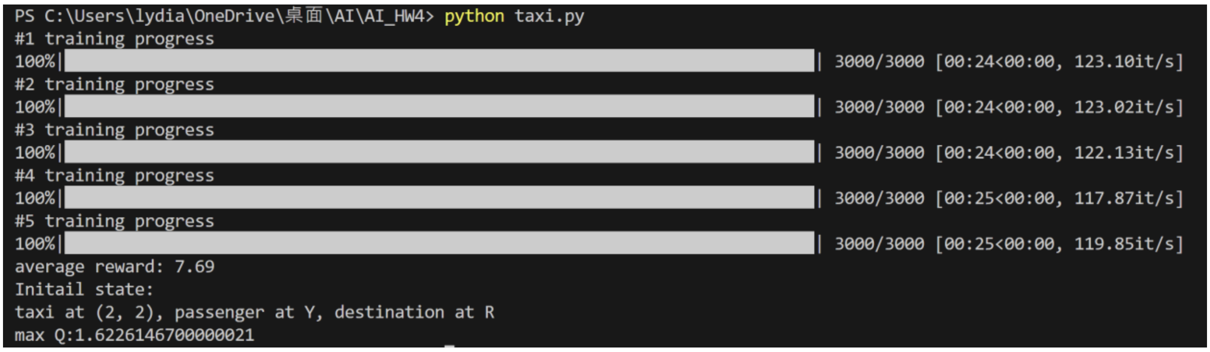


4. compare.png



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%)



action : ← ← ↓ ↓ pick ↑ ↑ ↑ ↑ drop $\gamma=0.9$

optimal Q-value = $(-1)*(1-\mathbf{0.9}^{9}) / (1-0.9) + 20*\mathbf{0.9}^{9} = 1.62$

The value is close !

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%)

$\gamma=0.97$ reward=181.62

optimal Q-value = $(1-\mathbf{0.97}^{181.62}) / (1-0.97) = 33.2013$

The value is close !

3.

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

The original obeservations are continous. However, QL work in a finite environm ent , so we need to discretize them.

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

The more bins would help the model account for more specific state space, and the performance will be better.

c. Is there any concern if we increase "num_bins" ? (3%)

More bins would require training with more time, and the computation need more memory.

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

DQN.

(1) DQN is capable of learning continuous state-action spaces without discretizing them.

(2) DQN uses experience replay and a target network to improve the stability of the learni ng process and prevent overfitting. Experience replay allows the agent to learn from a diverse set of experiences, while the target network is periodically updated to reduce th e correlation between the target and estimated Q-values.

5.

a. What is the purpose of using the epsilon greedy algorithm while choosing an actio n? (3%)

The epsilon greedy algorithm selects the action with the highest estimated reward most of the time. The aim is to have a balance between exploration and exploitatio n.

With a small probability of epsilon, we choose to explore (not to exploit what we h ave learned so far). In this case, the action is selected randomly, independent of the action-value estimates.

b. What will happen, if we dont use the epsilon greedy algorithm in the CartPole-v0 environment? (3%)

The exploration and exploitation will not be balanced. It may cause extreme senari os, like not exploit what it learn ornot explore the new things.

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

Yes, because we just need a method to balance exploitation and exploration, and there have many action selection methods. For example, the softmax action selection strategy controls the relative levels of exploration and exploitation by mapping values into action probabilities.

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

The agent have already got the information of the enviornment.

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

Operations performed within "with torch.no_grad(): " will not be tracked for backpropagation, which can save memory and computational resources. This disables gradient calculation during the forward pass through the neural network.