Homework 4:

Reinforcement Learning

110550108 施柏江

Part I. Implementation:

taxi.py:

choose_action

```
"""
If the random number is less than epsilon, the agent will choose a random action.
Otherwise, the agent will choose the action with the highest estimated value from
its Q-table for the current state. This is the exploitation step.
"""
# Begin your code
if np.random.uniform() < self.epsilon:
    action = env.action_space.sample()
else:
    action = np.argmax(self.qtable[state])
return action
# End your code
```

learn

```
"""
'next_max' calculates the maximum expected reward for any action that can be taken
in the next state.
'updated' is the estimate of the expected reward for taking action in state,
calculated using the Q-learning update rule.
"""
# Begin your code
original = self.qtable[state][action]
next_max = self.check_max_Q(next_state)
updated = (1 - self.learning_rate) * original + \
            self.learning_rate * (reward + self.gamma * next_max)
self.qtable[state][action] = updated
# End your code
```

check_max_Q

```
"""
Returns the maximum expected reward for any action that can be taken in
the current state, as estimated by the Q-table.
"""
# Begin your code
max_q = max(self.qtable[state])
return max_q
# End your code
```

cartpole.py:

init_bins

```
"""
Creates a set of evenly spaced bin edges for discretizing a continuous variable
into a discrete variable with 'num_bins', within a given range of values between
lower_bound and upper_bound.
"""
# Begin your code
bins = np.linspace(lower_bound, upper_bound, num_bins + 1)
return bins[1:-1]
# End your code
```

discritize_value

```
"""
Discretizes a continuous value into a discrete value based on a set of bin edges.
Then returns the index of the bin to which value belongs.
"""
# Begin your code
discretized = np.digitize(value, bins)
return discretized
# End your code
```

discritize_observation

```
"""
Discretizes a continuous observation into a discrete state based on a set of bin edges.
Then returns a list containing the discretized values for each dimension of the observation.
"""
# Begin your code
state = []
for i in range(4):
    state.append(self.discretize_value(observation[i], self.bins[i]))
return state
# End your code
```

choose_action

```
"""
If the random number is less than epsilon, the agent will choose a random action.
Otherwise, the agent will choose the action with the highest estimated value from
its Q-table for the current state. This is the exploitation step.
"""
# Begin your code
if np.random.uniform() < self.epsilon:
    action = env.action_space.sample()
else:
    action = np.argmax(self.qtable[tuple(state)])
return action
# End your code
```

learn

```
"""
'next_max' calculates the maximum expected reward for any action that can be taken
in the next state.
'updated' is the estimate of the expected reward for taking action in state,
calculated using the Q-learning update rule.
"""
# Begin your code
original = self.qtable[tuple(state)][action]
next_max = max(self.qtable[tuple(next_state)])
updated = (1 - self.learning_rate) * original + \
        self.learning_rate * (reward + self.gamma * next_max)
self.qtable[tuple(state)][action] = updated
# End your code
```

check_max_Q

```
"""
Discretizes the initial observation obtained from resetting the environment and
returns the maximum Q-value for the corresponding state.
"""
# Begin your code
state = self.discretize_observation(self.env.reset())
max_q = max(self.qtable[tuple(state)])
return max_q
# End your code
```

DQN.py:

learn

```
"""
We update the Q-network using a batch of transitions sampled from the replay buffer
and a target network that is periodically updated with the weights of the evaluation
network. The loss function used to update the Q-network is the mean squared error
between the current estimate of the Q-values and the target values. By minimizing
this loss, the Q-network learns to approximate the optimal Q-function, which is
used to select the actions that maximize the expected cumulative reward over time.
"""
# Begin your code

# Sample trajectories of batch size from the replay buffer.
observations, actions, rewards, next_observations, done = self.buffer.sample(self.batch_size)
observations = torch.tensor(np.array(observations), dtype=torch.float)
actions = torch.tensor(actions, dtype=torch.long)
rewards = torch.tensor(rewards, dtype=torch.float)
next_observations = torch.tensor(np.array(next_observations), dtype=torch.float)
done = torch.tensor(done, dtype=torch.bool)
```

```
# Forward the data to the evaluate net and the target net.
evaluation = self.evaluate_net(observations).gather(1, actions.unsqueeze(1))
next_max = self.target_net(next_observations).detach()
for i in range(len(done)):
    if done[i]:
        next_max[i] = 0
target = rewards.reshape(self.batch_size, 1) + self.gamma * next_max.max(1).values.unsqueeze(
```

```
# Compute the loss with MSE.
func = nn.MSELoss()
loss = func(evaluation, target)

# Zero-out the gradients.
self.optimizer.zero_grad()
```

```
# Backpropagation
loss.backward()

# Optimize the loss function.
self.optimizer.step()

# End your code
```

choose_action

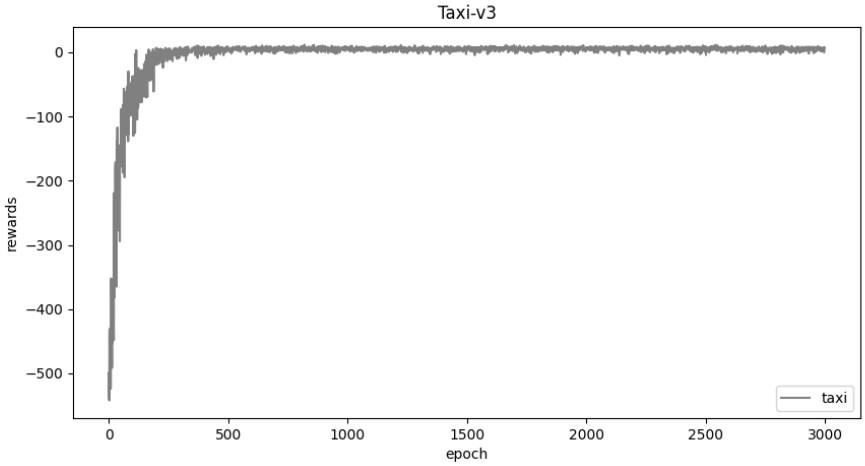
```
"""
We use 'with torch.no_grad()' to turn off gradient computation to speed up inference.
If the random number is greater than epsilon, the agent selects the action with the highest
Q-value in the current state.
"""
with torch.no_grad():
    # Begin your code
    if np.random.uniform() < self.epsilon:
        action = env.action_space.sample()
    else:
        x = torch.tensor(state, dtype=torch.float).unsqueeze(0)
        action_values = self.evaluate_net(x)
        action = torch.argmax(action_values, dim=1).item()
    # End your code
return action
```

check_max_Q

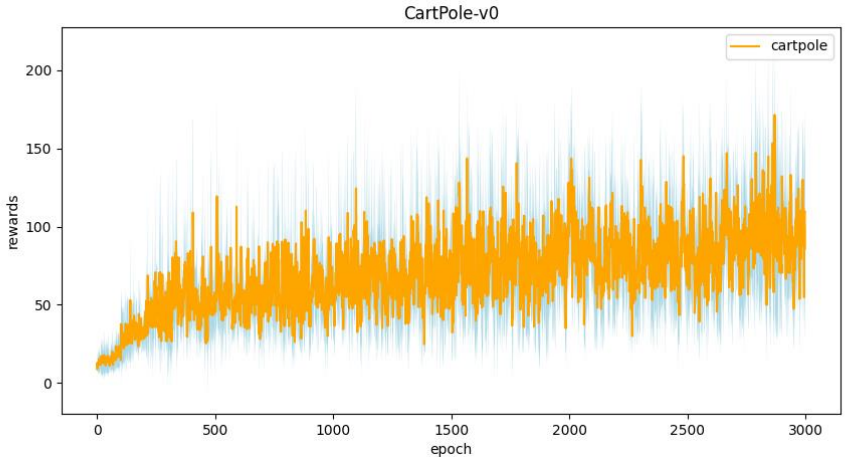
```
"""
We calculate the maximum Q-value of the current state using the target network.
"""
# Begin your code
state = self.env.reset()
state_tensor = torch.tensor(state, dtype=torch.float).unsqueeze(0)
with torch.no_grad():
    qvalues = self.target_net(state_tensor)
max_q = qvalues.max().item()
return max_q
# End your code
```

Part II. Experiment Results:

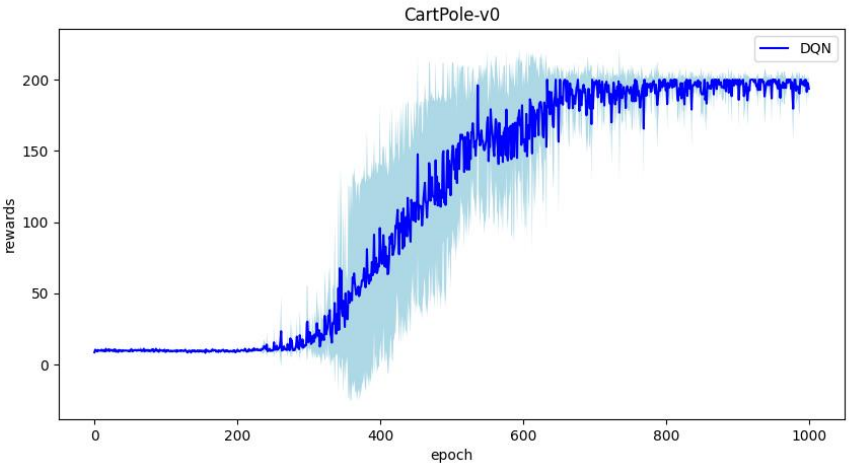
1. taxi.png:



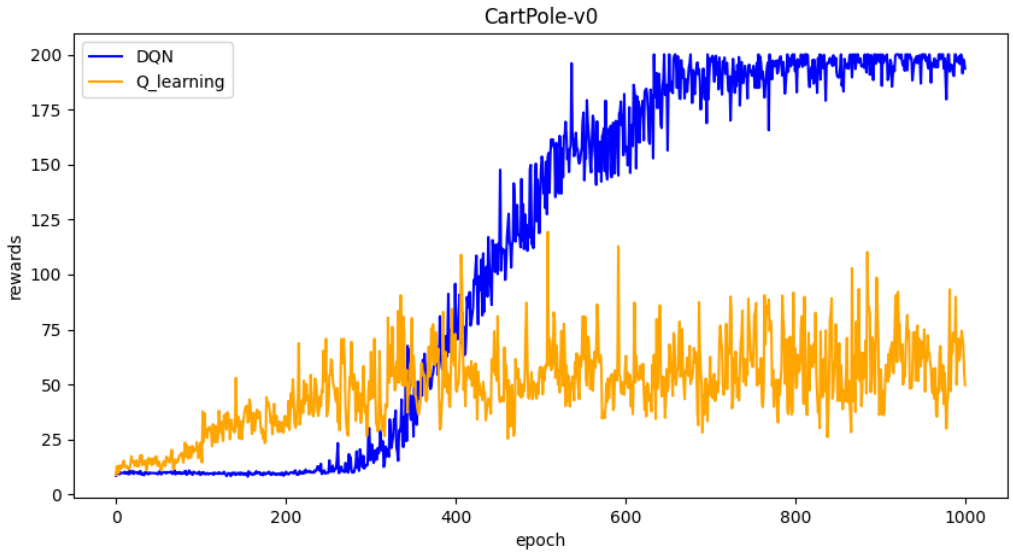
2. cartpole.png



3. DQN.png



4. compare.png



Part III. Question Answering:

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%)

Ans:

$$\hat{Q}_{\text{opt}}(s,a) = \sum_{s'} \hat{T}(s,a,s')[\widehat{\text{Reward}}(s,a,s') + \gamma \hat{V}_{\text{opt}}(s')]$$

The taxi is at (2, 2) in the beginning, and we want to sent the passenger from Y to R, so we need 9 steps to travel to the destination.

optimal ↓

```
step = 9
optimal_q = -(1 - pow(self.gamma, step)) / (1 - self.gamma) + 20 * pow(self.gamma, step)
print(f'optimal Q:{optimal_q}')
```

```
optimal Q: 1.6226146700000017
```

learned ↓

```
#1 training progress
100%|| 3000/3000 [00:02<00:00, 1443.84it/s]
#2 training progress
100%|| 3000/3000 [00:02<00:00, 1326.21it/s]
#3 training progress
100%|| 3000/3000 [00:02<00:00, 1439.35it/s]
#4 training progress
100%|| 3000/3000 [00:02<00:00, 1474.37it/s]
#5 training progress
100%|| 3000/3000 [00:02<00:00, 1474.32it/s]
average reward: 8.24
Initail state:
taxi at (2, 2), passenger at Y, destination at R
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%)

Ans:

formula ↓

```
optimal_max_q = (1 - pow(self.gamma, np.mean(total_reward))) / (1 - self.gamma)
print(f'optimal_max_q:{optimal_max_q}')
```

Q-learning updates Q-values based on estimates of future rewards. During early stages of learning, the agent has limited experience. Estimation errors can cause the learned Q-values to deviate from the true optimal values.

Q learning ↓

```
optimal_max_q:33.21773529895501
```

```
#1 training progress
100%|| 3000/3000 [00:08<00:00, 362.45it/s]
#2 training progress
100%|| 3000/3000 [00:08<00:00, 341.66it/s]
#3 training progress
100%|| 3000/3000 [00:09<00:00, 305.11it/s]
#4 training progress
100%|| 3000/3000 [00:08<00:00, 349.29it/s]
#5 training progress
100%|| 3000/3000 [00:09<00:00, 320.20it/s]
average reward: 185.96
max Q:31.521006009494407
```

DQN employs experience replay and target networks to stabilize and improve the learning process. Thus, the result is very closed to the optimal value.

DQN ↓

```
optimal_max_q:33.257659580339144
```

```
#1 training progress
100%|| 1000/1000 [02:48<00:00,  5.95it/s]
#2 training progress
100%|| 1000/1000 [04:08<00:00,  4.02it/s]
#3 training progress
100%|| 1000/1000 [03:59<00:00,  4.18it/s]
#4 training progress
100%|| 1000/1000 [03:49<00:00,  4.36it/s]
#5 training progress
100%|| 1000/1000 [04:27<00:00,  3.74it/s]
reward: 199.87
max Q:33.76601028442383
```

3.

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

Ans:

The observations in part 2 are continuous values representing the state of the system. However, the reinforcement learning algorithms we use require discrete states and actions to effectively learn and make decisions.

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

Ans:

It can lead to a higher resolution in the discretized observation space. The agent can distinguish more subtle differences in the state of the system. This increased level of detail can help the algorithm converge faster and find optimal more efficiently.

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

Ans:

The size of the tabular representation to store the Q-values grows. This can result in increased computational and memory requirements, potentially slowing down the learning process.

4.

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

Ans:

DQN performs better. DQN can generalize its knowledge across similar states due to neural networks. The agent can transfer its learned policy to unseen states that are similar to the training states. But in Discretized Q-learning, it cannot easily generalize because each state is treated independently, without considering the relationships between similar states.

5.

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

Ans:

The purpose is to balance the exploration of new actions and the exploitation of the currently known best action.

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

Ans:

If we always select the action that is currently believed to be the best, it will lead to a lack of exploration. As a result, it might miss out on discovering better strategies that are not initially apparent. It could get stuck in suboptimal policies that are discovered early on.

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

Ans:

Yes. We can achieve with other algorithms, such as Upper Confidence Bound algorithm. It encourages exploration by selecting actions that have the potential for high uncertainty or high potential for improvement. It uses a confidence bound to balance exploration and exploitation, favoring actions that are less explored or have higher estimated value uncertainty.

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

Ans:

Because during testing, the goal is to evaluate the performance of the learned policy and exploit the knowledge the agent has acquired during training. The agent is expected to select actions based on its learned policy, which is optimized for exploitation and maximizing the expected rewards.

6. What does “with torch.no_grad():” do inside the “choose_action” function in DQN? (4%)

Ans:

It can speed up the training process. It allows us to select an action without performing any weight updates or back propagation because our objective is simply to choose an action based on the current network weights. Disabling gradient calculations within the block helps save time by avoiding unnecessary computations during action selection.