Homework 4:

Reinforcement Learning

Report Template

110550012黃鵬軒

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

Part1:

```python
# Begin your code
"""
Generate a random number between [0, 1]. If number is less than epsilon, takes a random action to explore the environment.
Otherwise, it selects the action with the highest expected reward based on its current estimate of the Q-values for each state-action pair.
"""
if random.uniform(0, 1) < self.epsilon:
    return self.env.action_space.sample()
else:
    return np.argmax(self.qtable[state])
# End your code
```

```python
# Begin your code
"""
Use the formula of Q-learning to calculate new Q-value and update Q-table.
"""
self.qtable[state, action] = (1 - self.learning_rate) * self.qtable[state, action] + self.learning_rate * (reward + self.gamma * np.max(self.qtable[next_state]))
# End your code
```

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

Part2:

```python
# Begin your code
"""
Use numpy's linspace to generate equally spaced numbers between the lower and upper bounds,
then returns all values except the first as the upper bounds of the bins that the interval is divided into.
"""
return np.linspace(lower_bound, upper_bound, num_bins, endpoint=False)[1:]
# End your code
```

```python
# Begin your code
"""
Discretize the value with given bins. Return the index of the interval
that the value belongs to by using the numpy.digitize function.
"""
return np.digitize(value, bins)
# End your code
```

```python
# Begin your code
"""
discretize_observation function takes in an observation which is a list of four features and
returns a tuple of four discretized features that represent the state. It discretizes each
feature by iterating through the observation list, calling the discretize_value function
with the current feature and the corresponding bin array, and appending the discretized
value to the state list. Finally, it returns a tuple of the discretized state.
"""
state = []
for i in range (len(observation)):
    state.append(self.discretize_value(observation[i], self.bins[i]))
return tuple(state)
# End your code
```

```python
# Begin your code
"""
Generate a random number between [0, 1]. If number is less than epsilon, takes a random action to explore the environment.
Otherwise, it selects the action with the highest expected reward based on its current estimate of the Q-values for each state-action pair.
"""
if random.uniform(0, 1) < self.epsilon:
    return self.env.action_space.sample()
else:
    return np.argmax(self.qtable[state])
# End your code
```

```python
# Begin your code
"""
Use the formula of Q-learning to calculate new Q-value and update Q-table.
"""
max_next_q = np.max(self.qtable[tuple(next_state)])
q = self.qtable[tuple(state)][action]
new_q = q + self.learning_rate * (reward + self.gamma * max_next_q - q) if not done else q + self.learning_rate * (reward - q)
self.qtable[tuple(state)][action] = new_q
# End your code
```

```python
# Begin your code
"""
Discretize the initial state, and return the max Q value.
"""
return max(self.qtable[self.discretize_observation(self.env.reset())])
# End your code
```

Part3:

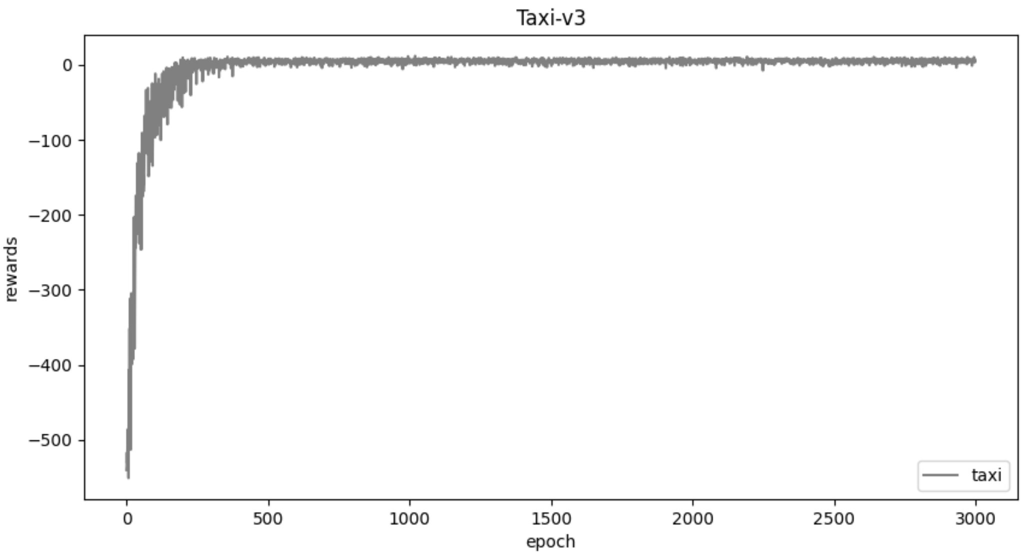
```python
# Begin your code
"""
The function samples a batch of trajectories from the replay buffer and converts them into PyTorch tensors. The observations tensor
contains the state values for each trajectory in the batch, the actions tensor contains the actions taken in each state, the rewards
tensor contains the rewards received for each action, the next_observations tensor contains the resulting states after each action,
and the done tensor indicates whether each trajectory has ended.
Use the evaluation network to predict the Q-values for the current state-action pairs, and selects the Q-values
that correspond to the actions taken in each trajectory using the gather function. The function also uses the target network to predict
the Q-values for the resulting states after each action, and selects the maximum Q-value for each trajectory using the max function.
Calculates the target Q-values by multiplying the maximum Q-values by the discount factor gamma, adding the rewards for each action,
and setting the Q-values for the final states to 0 if the corresponding trajectory has ended.
And calculates the mean squared error (MSE) loss between the predicted Q-values and the target Q-values using the PyTorch MSELoss function.
Set the gradients to zero, performs backpropagation to calculate the gradients, and updates the parameters of the evaluation network using the optimizer.
"""
observations, actions, rewards, next_observations, done = self.buffer.sample(self.batch_size)

observations = torch.tensor(observations, dtype=torch.float32)
actions = torch.tensor(actions, dtype=torch.int64)
rewards = torch.tensor(rewards, dtype=torch.float32)
next_observations = torch.tensor(next_observations, dtype=torch.float32)
done = torch.tensor(done, dtype=torch.bool)

evaluate = self.evaluate_net(observations).gather(1, actions.view(self.batch_size, 1))
nextMax = self.target_net(next_observations).detach()
target = rewards.view(self.batch_size, 1) + self.gamma * nextMax.max(1)[0].view(self.batch_size, 1) * (~done).view(self.batch_size, 1)

MSE = nn.MSELoss()
loss = MSE(evaluate, target)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
# End your code
```

```python
# Begin your code
"""
Generate a random number between [0, 1]. If the number is less than epsilon, the agent selects a random action.
Otherwise, it selects the action with the highest predicted Q-value from the evaluation network for the current state.
"""
if random.uniform(0,1) < self.epsilon:
    action = self.env.action_space.sample()
else:
    action = torch.argmax(self.evaluate_net.forward(torch.FloatTensor(state))).item()
# End your code
```

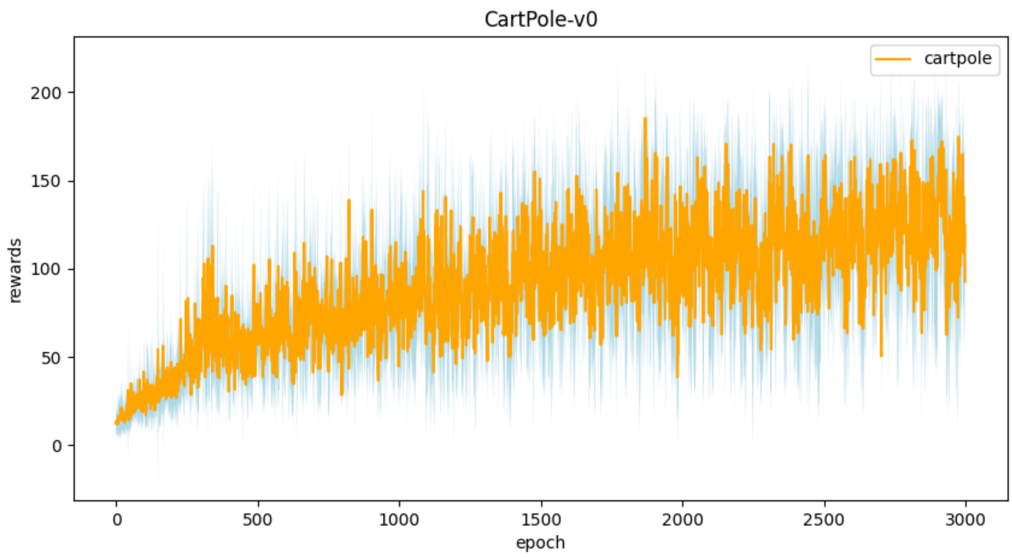
```python
# Begin your code
"""
Returns the maximum Q value predicted by the target network for the initial state of the environment.
It resets the environment and passes the resulting state to the target network, then returns the
maximum Q value from the resulting tensor.
"""
return torch.max(self.target_net(torch.FloatTensor(self.env.reset())))
# End your code
```

Part II. Experiment Results:

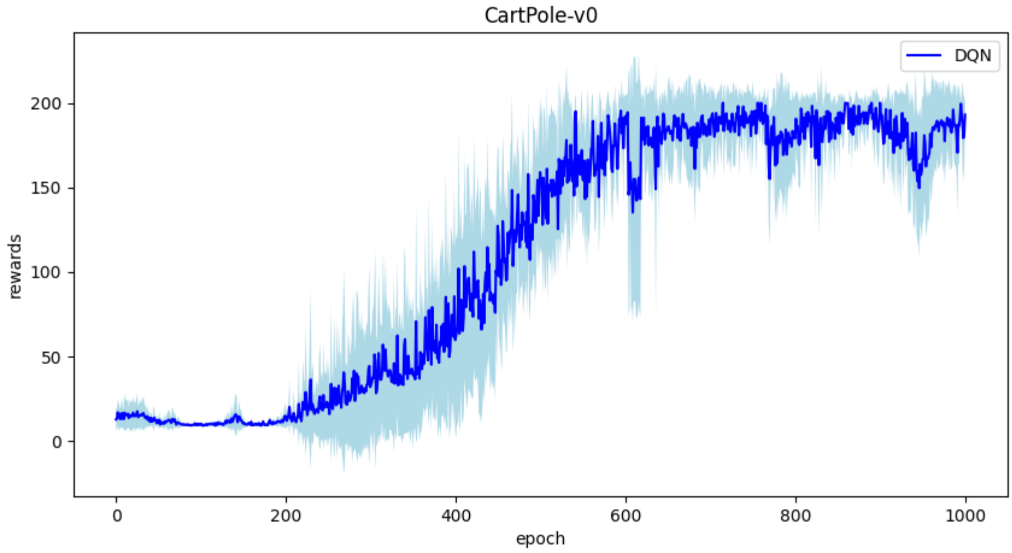
1. taxi.png:



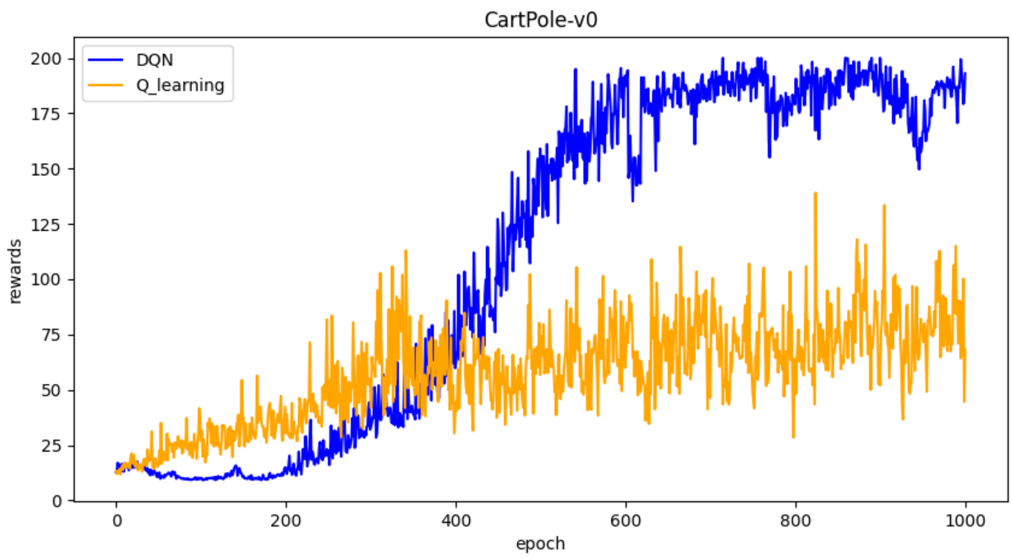
2. cartpole.png



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%)**

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')].$$

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

- -1 per step unless other reward is triggered.
- +20 delivering passenger.
- -10 executing “pickup” and “drop-off” actions illegally.

optimal Q-value = 1.6221614… (close to max Q)

```
#1 training progress
100%|██████████| 3000/3000 [00:05<00:00, 564.87it/s]
#2 training progress
100%|██████████| 3000/3000 [00:05<00:00, 570.15it/s]
#3 training progress
100%|██████████| 3000/3000 [00:05<00:00, 568.74it/s]
#4 training progress
100%|██████████| 3000/3000 [00:05<00:00, 526.31it/s]
#5 training progress
100%|██████████| 3000/3000 [00:05<00:00, 542.54it/s]
average reward: 7.99
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%)**
   1/(1-0.97) = 33.333… (close to max Q of Q-learning and more closer to max Q of DQN)
   Q-learning:

```
#1 training progress
100%|██████████| 3000/3000 [00:31<00:00, 96.40it/s]
#2 training progress
100%|██████████| 3000/3000 [00:38<00:00, 78.36it/s]
#3 training progress
100%|██████████| 3000/3000 [00:33<00:00, 90.62it/s]
#4 training progress
100%|██████████| 3000/3000 [00:31<00:00, 94.50it/s]
#5 training progress
100%|██████████| 3000/3000 [00:35<00:00, 84.85it/s]
average reward: 48.53
max Q:30.856744809846692
```

   DQN:

```
100%|██████████| 1000/1000 [01:43<00:00,  9.64it/s]
#2 training progress
100%|██████████| 1000/1000 [01:38<00:00, 10.15it/s]
#3 training progress
100%|██████████| 1000/1000 [01:54<00:00,  8.77it/s]
#4 training progress
100%|██████████| 1000/1000 [01:28<00:00, 11.31it/s]
#5 training progress
100%|██████████| 1000/1000 [01:57<00:00,  8.51it/s]
reward: 200.0
max Q:33.46462631225586
```

3.
   a. Why do we need to discretize the observation in Part 2? **(3%)**
      In the CartPole environment, the state values are continuous. It is difficult to represent them in a discrete table.

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

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

Increasing the number of bins will increase the number of states that the agent must consider, leading to increased computational complexity and longer training times.and we may need to add more training data.

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

   DQN, because it can handle continuous state spaces without the need for discretization. It uses a deep neural network to approximate the Q-function and employs experience replay and a target network to improve stability and convergence. This makes DQN a more flexible and robust approach to reinforcement learning in continuous state spaces.

5.

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

      The epsilon greedy algorithm is used to balance exploration and exploitation in reinforcement learning by occasionally choosing a random action while mostly choosing the best action based on current knowledge. It helps the agent to learn from experience and converge to a better policy.

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

      Without the epsilon greedy algorithm, it would be difficult to balance exploration and exploitation, leading to inefficient decision making and potentially a suboptimal policy.

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

      There is another possible way to achieve the same performance in the CartPole-v0 environment without using the epsilon-greedy algorithm, but it would be difficult to implement.

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

      Agent has enough information about the environment. During the testing section, the agent has already learned the optimal policy and does not need to explore new actions. Therefore, there is no need to use the epsilon greedy algorithm, and the agent can choose actions based on its learned Q-values.

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

   The "with torch.no_grad()" is used inside the "choose_action" function in DQN to temporarily disable the gradient computation in order to save memory and increase performance.