Homework 4:

Reinforcement Learning Report

Part I. Implementation

Part 1: Q-learning in Taxi-v3

choose_action (self, state):

```
# Begin your code
"""
Generate a random number which interval is between [0, 1]. If the number is bigger than
epsilon, determine to exploit and return the index of the maximum Q of the given state
in Q-table; otherwise, determine to explore and return random action.
"""
if np.random.uniform(0, 1) > self.epsilon:
    return np.argmax(self.qtable[state])
else:
    return env.action_space.sample()
# End your code
```

learn (self, state, action, reward, next_state, done):

```
# Begin your code
"""
Based on the formula of Q-learning, calculate the new Q-value and update in the 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
np.save("./Tables/taxi_table.npy", self.qtable)
```

check_max_Q (self, state):

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

Part 2: Q-learning in Cartpole-v0

init_bins(self, lower_bound, upper_bound, num_bins):

```
# Begin your code
"""
Slice the interval based on the given lower_bound, upper_bound, and num_bins.
Because np.linspace() will contain the lower_bound, return the list started from index 1.
"""
return np.linspace(lower_bound, upper_bound, num_bins, endpoint=False)[1:]
# End your code
```

discretize_value(self, value, bins):

```
# Begin your code
"""
Discretize the value with given bins. Using np.digitize() to determine the value in
which interval of bins.
"""
return np.digitize(value, bins)
# End your code
```

discretize_observation(self, observation):

```
# Begin your code
"""
Discretize the continuous observation. Using the function "discretize_value()" to get
the discretized data of the 4 features in observation.
"""
return tuple([self.discretize_value(obs, bin) for obs, bin in zip(observation, self.
bins)])

# End your code
```

choose_action(self, state):

```
# Begin your code
"""
Generate a random number which interval is between [0, 1]. If the number is bigger than
epsilon, determine to exploit and return the index of the maximum Q of the given state
in Q-table; otherwise, determine to explore and return random action.
"""
if np.random.uniform(0, 1) > self.epsilon:
    return np.argmax(self.qtable[tuple(state)])
else:
    return env.action_space.sample()
# End your code
```

learn (self, state, action, reward, next_state, done):

```
# Begin your code
"""
Based on the formula of Q-learning, calculate the new Q-value and update in the Q-table.
"""
self.qtable[tuple(state)][action] += self.learning_rate * (
    reward
    + self.gamma * np.max(self.qtable[tuple(next_state)])
    - self.qtable[tuple(state)][action]
)
if done:
# End your code
    np.save("./Tables/cartpole_table.npy", self.qtable)
```

check_max_Q (self, state):

```
# Begin your code
"""
Return the max Q-value of given state, and before returning the max Q-value, the initial
state should be discretized first.
"""
return np.max(self.qtable[tuple(self.discretize_observation(self.env.reset()))])
# End your code
```

Part 3: DQN in Cartpole-v0

learn (self):

```
# Begin your code
"""
Sample trajectories of batch size from the replay buffer. Use function "sample" defined
in class "replay_buffer" to get the sampled data. Then, convert these sampled data into
tensor.
"""
sample = self.buffer.sample(self.batch_size)
states = torch.tensor(np.array(sample[0]), dtype=torch.float)
actions = torch.tensor(sample[1], dtype=torch.long).unsqueeze(1)
rewards = torch.tensor(sample[2], dtype=torch.float)
next_states = torch.tensor(np.array(sample[3]), dtype=torch.float)
done = torch.tensor(sample[4], dtype=torch.bool)
"""
Forward the data to the evaluate net and the target net. Variable "q_value" is predicted
values from evaluate network which is extracted based on "action". Variable
"q_next_value" is actual values from target network. Variable "target_q_value" is the
expected Q-values obtained from the formula "reward + gamma * max(q_next_value)".
"""
q_value = self.evaluate_net(states).gather(1, actions)
q_next_value = self.target_net(next_states).detach() * (~done).unsqueeze(-1)
target_q_value = rewards.unsqueeze(-1) + self.gamma * q_next_value.max(1)[0].view(self.
batch_size, 1)
"""
Compute the loss of "q_value" and "target_q_value" with nn.MSELoss().
"""
loss_func = nn.MSELoss()
loss = loss_func(q_value, target_q_value)

"""
Zero-out the gradients before doing back propagation, or the gradient would be a
combination of
the old gradient. Next, do back propagation, and update the parameters.
"""
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
# End your code
```

choose_action (self, state):

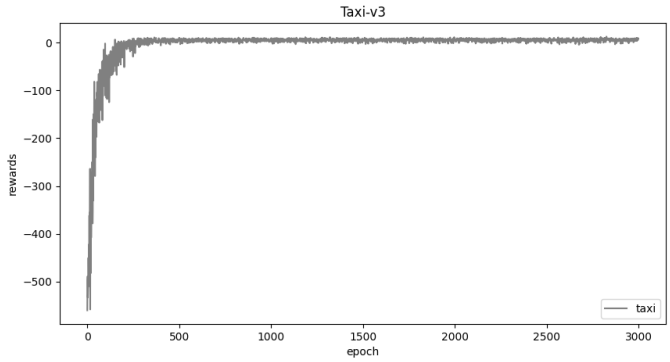
```
# Begin your code
"""
Generate a random number which interval is between [0, 1]. If the number is bigger
than epsilon, determine to exploit and return the index of the maximum Q of the
given state forwarded by the evaluate neural network; otherwise, determine to
explore and return random action.
"""
if np.random.uniform(0, 1) > self.epsilon:
    action = torch.argmax(
        self.evaluate_net(torch.tensor(state, dtype=torch.float))).item()
else:
    action = env.action_space.sample()
# End your code
```

check_max_Q (self)

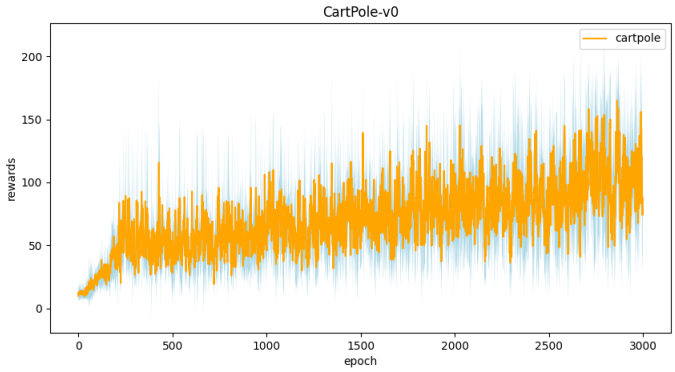
```
# Begin your code
"""
First, convert the initial state to tensor. Next, forward the tensor in the target
neural network.
Then, find the max Q-value and return it.
"""
return torch.max(self.target_net(torch.FloatTensor(self.env.reset()).unsqueeze(0))).item
()

# End your code
```

Part II. Experiment Results:

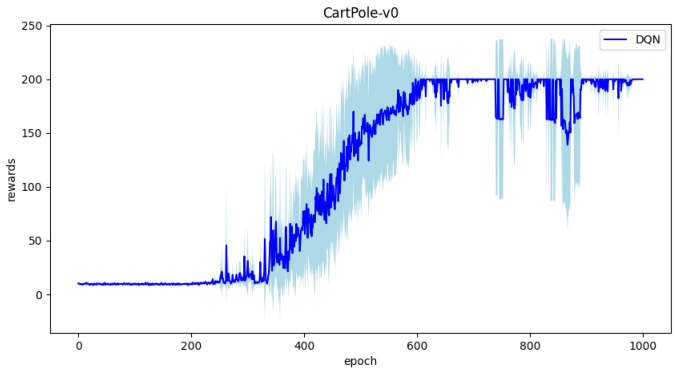
1. taxi.png:



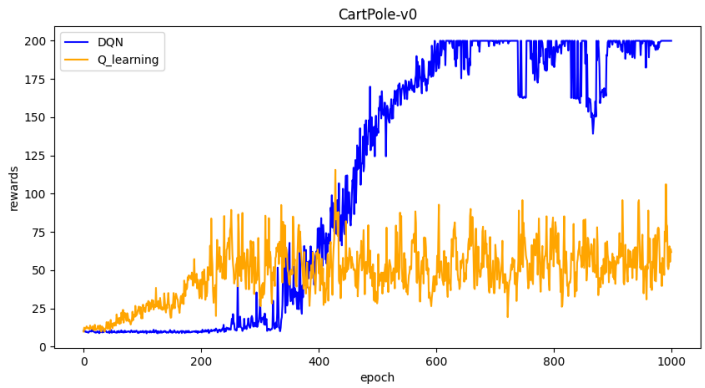
2. cartpole.png:



3. DQN.png:



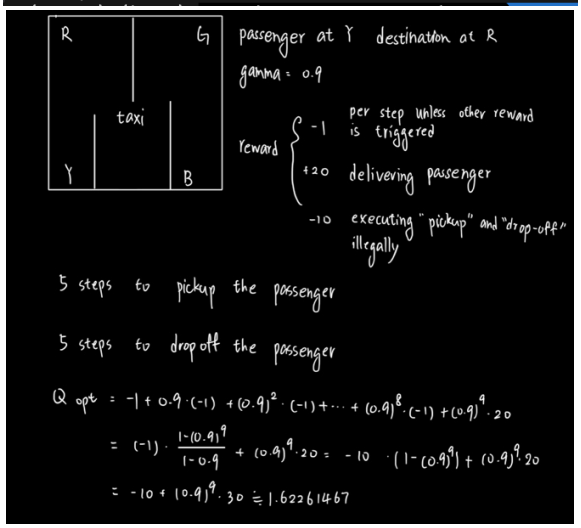
4. compare.png:



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%)**

```
average reward: 7.78
Initial state:

Taxi at (2,2),passenger at Y, destination at R
max Q:1.6226146699999995
```



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

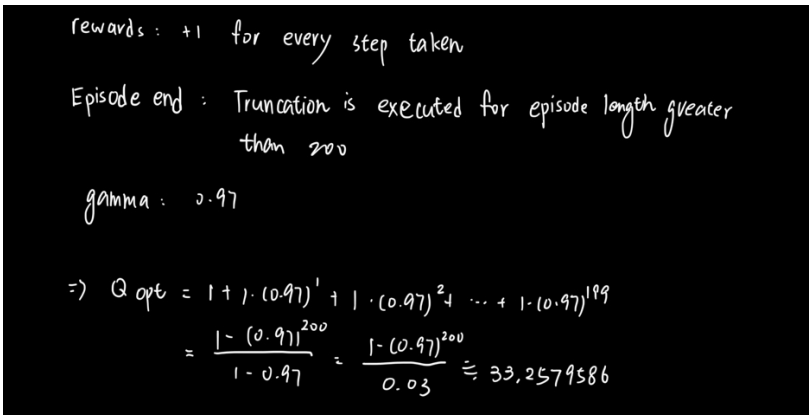
cartpole

```
average reward: 59.05
max Q:31.619468979618897
```

DQN

```
reward: 200.0
max Q:32.524513244628906
```

Optimal Q-value is close to the max Q of the Q learning, yet DQN is even closer.



3.

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

Because the interval is continuous, which is not easy to determine the state of the cartpole. As a result, we need to discretize it to get the state.

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

In my opinion, the performance will become better if we increase the num_bins, which represents the number of the states in the bounded interval. Because when we increase the number of the states, it implies that we have more states to approximate the continuous interval, which leads to the better performance.

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

Increasing "num_bins" can result in concerns such as longer time required to update and save the @ table due to the increased number of states, as well as the increased memory required to save the larger Q table.

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

DQN outperforms discretized Q learning in the Cartpole-v0 environment. The reason for this is that Q learning discretizes the continuous data into states, which can result in data lo ss. In contrast, DQN can use the continuous data and preserve more details, leading to better performance.

5.

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

The epsilon-greedy algorithm serves the purpose of balancing exploration and exploitation in action selection. It allows for the selection of the best-known action while also exploring new options, thus taking advantage of prior knowledge, and discovering new possibilities.

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

If the epsilon-greedy algorithm is not used in the CartPole-v0 environment, there are two potential scenarios. If only exploration is used, there will be no way to choo se the best-known action, as all actions will be random. If only exploitation is used, the algorithm can only rely on the known information and may miss unknown high-performance conditions without any randomness or 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%)**

   It may be possible to achieve the same performance without the epsilon-greedy algorithm in the CartPole-v0 environment if we can find another method to replace the learning rate in the algorithm while maintaining the same proportion of exploration/exploitation. With the same distribution, there is a possibility to achieve the same performance.

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

   The epsilon-greedy algorithm is not needed during the testing section because it is used for exploration and exploitation during training. In the testing section, the goa is simply to find the best path and obtain the best reward, and therefore there is no need to use the algorithm to train the data.

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

   The use of "with torch.no_grad():" inside the "choose_action" function in DQN disables gradient calculation for every tensor, meaning that requires_grad is set to False. This is because the function is used to choose an action and update the Q-values, but the gradients are not needed for these operations and can be disabled to improve computational efficiency.