Homework 4:
Reinforcement Learning

Part I. Implementation

- part 1

```python
def choose_action(self, state):
    """
    Choose the best action with given state and epsilon.

    Parameters:
        state: A representation of the current state of the enviornment.
        epsilon: Determines the explore/expliot rate of the agent.

    Returns:
        action: The action to be evaluated.
    """
    # Begin your code

    # Exploration vs. Exploitation:

    if np.random.rand() > self.epsilon:  # Exploration
        action = np.random.randint(self.env.action_space.n)  # Choose a random action
    else:  # Exploitation
        max_q_value = np.max(self.qtable[state])
        best_actions = np.where(self.qtable[state] == max_q_value)[0]
        action = np.random.choice(best_actions)
# Choose a random action among those with the maximum Q-value


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

def learn(self, state, action, reward, next_state, done):
    """
    Calculate the new q-value base on the reward and state transformation observered after taking th
e action.

    Parameters:
        state: The state of the enviornment before taking the action.
        action: The exacuted action.
        reward: Obtained from the enviornment after taking the action.
        next_state: The state of the enviornment after taking the action.
        done: A boolean indicates whether the episode is done.

    Returns:
        None (Don't need to return anything)
    """
    # Begin your codd


    # Calculate the new Q-value using the Q-learning update rule

    old_q = self.qtable[state, action]
    if done:
        next_max_q = 0  # If the episode is done, there's no future reward
    else:
        next_max_q = np.max(self.qtable[next_state])
# Otherwise, use the maximum Q-value for the next state

    # Calculate the new Q-value
    new_q = (1 - self.learning_rate) * old_q + self.learning_rate * (reward + self.gamma *
next_max_q)

    # Update the Q-value for the state-action pair
    self.qtable[state, action] = new_q

    # Save the Q-value for tracking
    self.qvalue_rec.append(new_q)


    #raise NotImplementedError("Not implemented yet.")
    # End your code
    np.save("./Tables/taxi_table.npy", self.qtable)
```

```python
def check_max_Q(self, state):
        """
        - Implement the function calculating the max Q value of given state.
        - Check the max Q value of initial state

        Parameter:
            state: the state to be check.
        Return:
            max_q: the max Q value of given state
        """
        # Begin your code

        # Implement the function calculating the max Q value of given state.
        # Check the max Q value of initial state
        max_q = np.max(self.qtable[state])
        return max_q
        #raise NotImplementedError("Not implemented yet.")
        # End your code
```

- part 2:

```
def choose_action(self, state):
        """
        Choose the best action with given state and epsilon.
        Parameters:
            state: A representation of the current state of the enviornment.
            epsilon: Determines the explore/expliot rate of the agent.
        Returns:
            action: The action to be evaluated.
        """
        # Begin your code

        if np.random.uniform(0, 1) > self.epsilon:
            return self.env.action_space.sample()  # Exploration
        else:
            return np.argmax(self.qtable[state])  # Exploitation
        #raise NotImplementedError("Not implemented yet.")
        # End your code

    def learn(self, state, action, reward, next_state, done):
        """
        Calculate the new q-value base on the reward and state transformation observered after takin
g the action.
        Parameters:
            state: The state of the enviornment before taking the action.
            action: The exacuted action.
            reward: Obtained from the enviornment after taking the action.
            next_state: The state of the enviornment after taking the action.
            done: A boolean indicates whether the episode is done.
        Returns:
            None (Don't need to return anything)
        """
        # Begin your code


        # Calculate the old Q-value
        old_q = self.qtable[state][action]

        # Calculate the maximum Q-value for the next state
        next_max = np.max(self.qtable[next_state]) if not done else 0

        # Update the Q-value using the Q-learning algorithm
        new_q = (1 - self.learning_rate) * old_q + self.learning_rate * (reward + self.gamma *
next_max)

        # Update the Q-value for the state-action pair
        self.qtable[state][action] = new_q
        #raise NotImplementedError("Not implemented yet.")
        # End your code
        np.save("./Tables/cartpole_table.npy", self.qtable)
```

```python
def check_max_Q(self):
        """
        - Implement the function calculating the max Q value of initial state(self.env.reset()).
        - Check the max Q value of initial state
        Parameter:
            self: the agent itself.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Return:
            max_q: the max Q value of initial state(self.env.reset())
        """
        # Begin your code

        # Implement the function calculating the max Q value of initial state(self.env.reset()).
        # Check the max Q value of initial state
        state = self.discretize_observation(self.env.reset())
        return np.max(self.qtable[state])
        #raise NotImplementedError("Not implemented yet.")
        # End your code
```

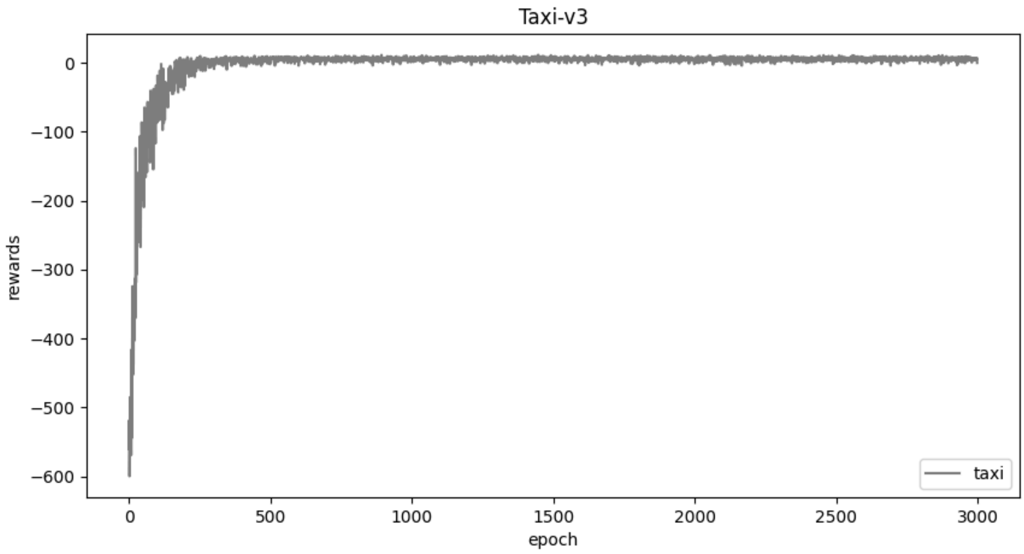
- part 3:

```python
def learn(self):
        '''
        - Implement the learning function.
        - Here are the hints to implement.
        Steps:
        -----
        1. Update target net by current net every 100 times. (we have done this for you)
        2. Sample trajectories of batch size from the replay buffer.
        3. Forward the data to the evaluate net and the target net.
        4. Compute the loss with MSE.
        5. Zero-out the gradients.
        6. Backpropagation.
        7. Optimize the loss function.
        -----
        Parameters:
            self: the agent itself.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Returns:
            None (Don't need to return anything)
        '''
        if self.count % 100 == 0:
            self.target_net.load_state_dict(self.evaluate_net.state_dict())

        # Begin your code

        # Step 2: Sample trajectories of batch size from the replay buffer
        state, action, reward, next_state, done = self.buffer.sample(self.batch_size)

        # Convert data to PyTorch tensors
        state = torch.FloatTensor(np.array(state))
        action = torch.LongTensor(action).unsqueeze(1)
        reward = torch.FloatTensor(reward).unsqueeze(1)
        next_state = torch.FloatTensor(np.array(next_state))
        done = torch.FloatTensor(done).unsqueeze(1)

        # Step 3: Compute Q values
        q_val = self.evaluate_net(state).gather(1, action)

        # Step 4: Compute target Q values using target network (with no gradient calculation)
        with torch.no_grad():
            next_q = self.target_net(next_state).max(1, keepdim=True)[0]
            target_q = reward + self.gamma * next_q * (1 - done)

        # Step 5: Compute loss with MSE
        loss = F.mse_loss(q_val, target_q)

        # Step 6: Zero-out the gradients
        self.optimizer.zero_grad()

        # Step 7: Backpropagation and optimization
        loss.backward()
        self.optimizer.step()

        # Increment step count
        self.count += 1
        #raise NotImplementedError("Not implemented yet.")
        # End your code
        torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
```

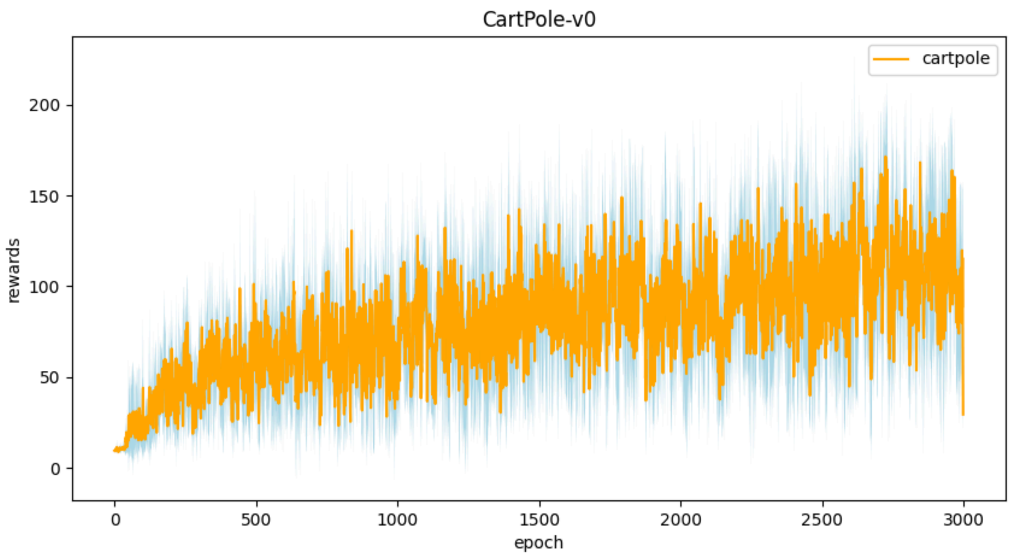
```python
def choose_action(self, state):
        """
        - Implement the action-choosing function.
        - Choose the best action with given state and epsilon
        Parameters:
            self: the agent itself.
            state: the current state of the enviornment.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Returns:
            action: the chosen action.
        """
        with torch.no_grad():
            # Begin your code
            action = None
            # Use epsilon-greedy strategy to choose action
            if random.uniform(0, 1) > self.epsilon:
                # Explore: choose a random action
                action = self.env.action_space.sample()
            else:
                # Exploit: choose the action with maximum Q-value
                state_tensor = torch.FloatTensor(state)
                q_values = self.evaluate_net(state_tensor)
                action = torch.argmax(q_values).item()

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

    def check_max_Q(self):
        """
        - Implement the function calculating the max Q value of initial state(self.env.reset()).
        - Check the max Q value of initial state
        Parameter:
            self: the agent itself.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Return:
            max_q: the max Q value of initial state(self.env.reset())
        """
        # Begin your code

        # Implement the function calculating the max Q value of initial state(self.env.reset()).
        # Check the max Q value of initial state

        state = self.env.reset()
        q_val = self.evaluate_net(torch.FloatTensor(state))
        max_q = q_val.max().item()
        return max_q

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

Part II. Experiment Results:

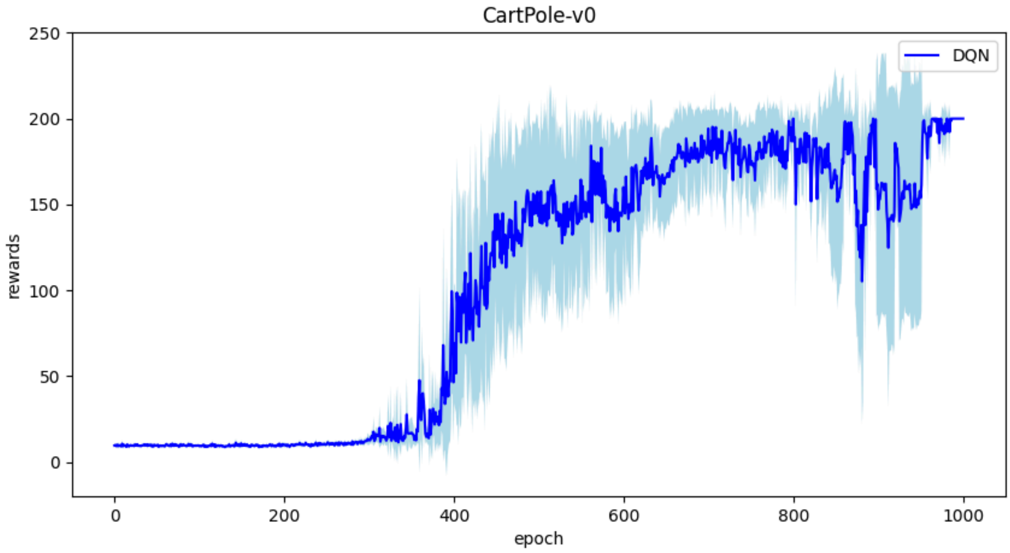
1. **taxi.png:**



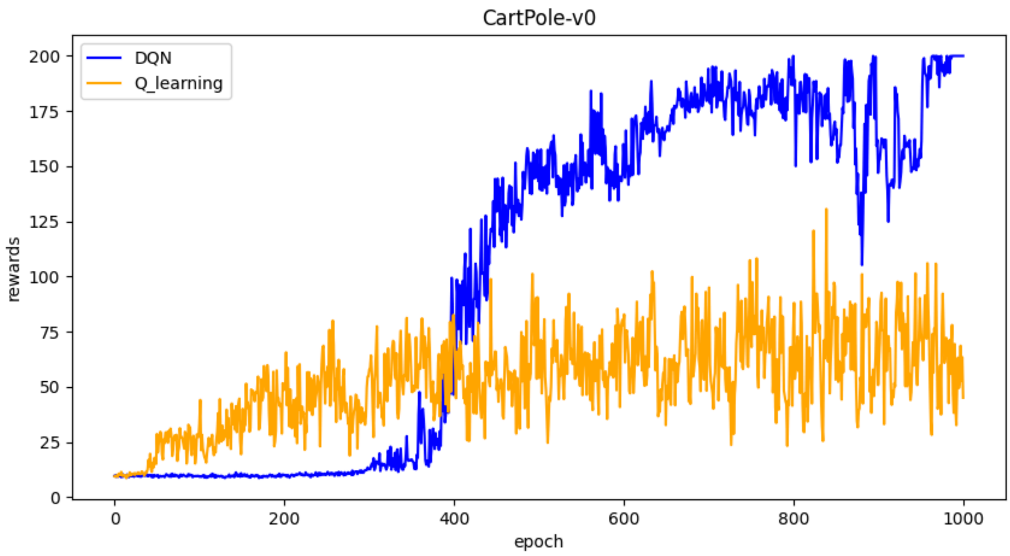
2. **cartpole.png**



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%)

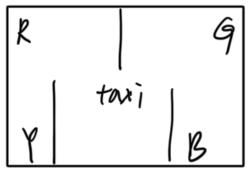
```
average reward: 7.61
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146699999995
```

Rewards:

$$\begin{cases} -1 & : \text{per step unless other reward is triggered} \\ +20 & : \text{delivering passenger} \\ -10 & : \text{executing "picking" \& "drop-off" actions illegally.} \end{cases}$$

$$Q_{opt} = -1 + 0.9 * (-1) * (0.9)^2 + \dots + 20 * (0.9)^9$$

$$\fallingdotseq 1.6226$$



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: 25.99
max Q:30.200060048620237
```

- DQN

```
reward: 200.0
max Q:34.69419479370117
```

$$1/(1-\gamma) \fallingdotseq 33.3$$

$\Rightarrow$ cartpole is not close 33.3 so much.

$\Rightarrow$ DQN is closer to 33.3 than cartpole become of discretizing.

3.

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

   The reason that we need to discretize to the observation is the CartPole-v0 environment provides observations that are continuous, but the Q-learning algorithm, which we're using to train the agent, requires discrete state-action pairs.

b. How do you expect the performance will be if we increase “num_bins”?

   Maybe will make the Q-table lager and the become more accurate.

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

   The overfitting problem also could happen, because If the number of bins is too high, and then the agent might overfit to the training data.

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

   DQN.
   There are two reason.First, DQN tends to be more sample-efficient compared to discretized Q-learning because it learn directly from original inputs and do not need the discretization. Secondly, DQN can also handle continuous action spaces efficiently. Hence, although discretized Q learning has some strengths , DQN is better.

5.

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

   It balances exploration and exploitation by allowing the agent to explore new actions with epsilon, like a probability, and old knowledge to choose a best-choice.

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

   Maybe can not find a best action to get a higher reward.

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

Maybe could, if we find another algorithm to replace the greedy algorithm to complete the all step with same logic.

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

   Because the agent was already learned from exploration during training, it can choose the action with the highest estimated value without the need for exploration.

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

   Because we do not need to update the Q-value.