Homework 4: Reinforcement Learning

Part I. Implementation

Part 1: Q learning in Taxi-v3 (-5 if not explain in detail):

choose_action:

```
    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
        # choose a random action with probability epsilon
        if np.random.uniform(0, 1) < 1-self.epsilon:
            return self.env.action_space.sample()
        # choose the action with the highest Q-value for the current state
        return np.argmax(self.qtable[state]) # return the index of the highest Q-value
        # End your code
```

→ I randomly choose number between 0 and 1. If the number is less than (1-epsilon), then I just randomly choose an action by using the env.action_space.sample(). The reason I use (1-epsilon) but not epsilon is that the epsilon is equal to 0.95, which is too large. It may lead to exploration most of the time. Next, if the number is not less than (1-epsilon), I will use argmax function in numpy library to find the action which is equal to the index of the max number in qtable[state].

learn:

```python
    def learn(self, state, action, reward, next_state, done):
        """
        Calculate the new q-value base on the reward and state transformation observered after taking 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
        # Q-learning algorithm
        current_value = self.qtable[state, action]  # current Q-value for the state/action couple
        next_max = np.max(self.qtable[next_state])  # next best Q-value

        # Compute the new Q-value with the Bellman equation
        self.qtable[state, action] = current_value + self.learning_rate*(reward + self.gamma*next_max - current_value)

        # End your code
        np.save("./Tables/taxi_table.npy", self.qtable)
```

→ Update the value in q table. First of all, find the Q value of current state and action. Subsequently, find the maximal Q value in the next state of Q table by using max function in numpy library. Finally, utilize them to undate the Q value of current state and action. The formula of update is:

$$\overbrace{\boxed{Q(S_t, A_t)}}^{\text{Q-value (for a state (S) and action(A))}} \leftarrow \boxed{Q(S_t, A_t)} + \underbrace{\boxed{\alpha}}_{\text{Learning rate}} \Big[\overbrace{\boxed{R_{t+1}}}^{\text{Reward}} + \underbrace{\boxed{\gamma}}_{\text{Discount factor}} \overbrace{\boxed{\max_a Q(S_{t+1}, a)}}^{\text{Maximum expected future reward}} - \boxed{Q(S_t, A_t)}\Big]$$

check_max_Q:

```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
        q_values = self.qtable[state]
        max_q = np.max(q_values)
        return max_q
        # End your code
```

→ Utilize the max function in numpy library to find the maximal Q value in the given state of the Q table.

Part 2: Q learning in CartPole-v0

init_bins:

```python
    def init_bins(self, lower_bound, upper_bound, num_bins):
        """
        Slice the interval into #num_bins parts.
        Parameters:
            lower_bound: The lower bound of the interval.
            upper_bound: The upper bound of the interval.
            num_bins: Number of parts to be sliced.
        Returns:
            a numpy array of #num_bins - 1 quantiles.
        Example:
            Let's say that we want to slice [0, 10] into five parts,
            that means we need 4 quantiles that divide [0, 10].
            Thus the return of init_bins(0, 10, 5) should be [2. 4. 6. 8.].
        Hints:
            1. This can be done with a numpy function.
        """
        # Begin your code
        return np.linspace(lower_bound, upper_bound, num_bins, endpoint=False)[1:]
        # End your code
```

→ Utilize linspace function in numpy library to slice the interval into #num_bins parts. The endpoint = False means elcluding the end point and [1:] means excluding the start point.

discretize_value:

```python
    def discretize_value(self, value, bins):
        """
        Discretize the value with given bins.
        Parameters:
            value: The value to be discretized.
            bins: A numpy array of quantiles
        returns:
            The discretized value.
        Example:
            With given bins [2. 4. 6. 8.] and "5" being the value we're going to discretize.
            The return value of discretize_value(5, [2. 4. 6. 8.]) should be 2, since 4 <= 5 < 6 where [4, 6) is the 3rd bin.
        Hints:
            1. This can be done with a numpy function.
        """
        # Begin your code
        return np.digitize(value, bins)
        # End your code
```

→ Utilize the digitize function in numpy library to find the indices of the bins to which each value in input array belongs, which can achieve the goal of discretizing the value with given bins.

discretize_observation:

```python
    def discretize_observation(self, observation):
        """
        Discretize the observation which we observed from a continuous state space.
        Parameters:
            observation: The observation to be discretized, which is a list of 4 features:
                1. cart position.
                2. cart velocity.
                3. pole angle.
                4. tip velocity.
        Returns:
            state: A list of 4 discretized features which represents the state.
        Hints:
            1. All 4 features are in continuous space.
            2. You need to implement discretize_value() and init_bins() first
            3. You might find something useful in Agent.__init__()
        """
        # Begin your code
        state = []
        for i in range(len(observation)):
            discrete_value = self.discretize_value(observation[i], self.bins[i])
            state.append(discrete_value)
        return tuple(state)
        # End your code
```

→ First of all, create an empty list called state. Supsequently, use discretize_value function to get the discrete value and put it into the state list. Finally, convert the list to tuple and return it.

choose_action:

```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
        # choose a random action with probability epsilon
        if np.random.uniform(0, 1) < 1-self.epsilon:
            return self.env.action_space.sample()
        # choose the action with the highest Q-value for the current state
        return np.argmax(self.qtable[tuple(state)]) # return the index of the highest Q-value
        # End your code
```

→ I randomly choose number between 0 and 1. If the number is less than (1-epsilon), then I just randomly choose an action by using the env.action_space.sample(). The reason I use (1-epsilon) but not epsilon is that the epsilon is equal to 0.95, which is too large. It may lead to exploration most of the time. Next, if the number is not less than (1-epsilon), I will use argmax function in numpy library to find the action which is equal to the index of the max number in qtable[state].

learn:

```
    def learn(self, state, action, reward, next_state, done):
        """
        Calculate the new q-value base on the reward and state transformation observered after taking 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
        # Get the Q value of the current state and next state
        current_value = self.qtable[state + (action,)]
        next_max = max(self.qtable[next_state])
        if done:
            next_max = 0

        # Update Q value with Q-learning
        self.qtable[state + (action,)] = (1 - self.learning_rate) * current_value + self.learning_rate * (reward + self.gamma * next_max)
        if done:
        # End your code
            np.save("./Tables/cartpole_table.npy", self.qtable)
```

→ Update the value in q table. First of all, find the Q value of current state and action. Subsequently, find the maximal Q value in the next state of Q table by using max function in numpy library. If done is not equal to 0, set next_max to 0. Finally, utilize them to undate the Q value of current state and action.

check_max_Q:

```
    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
        # Reset the environment to obtain the initial state
        initial_state = self.discretize_observation(self.env.reset())

        # Find the maximum Q-value for the initial state
        max_q = np.max(self.qtable[tuple(initial_state)])
        return max_q
        # End your code
```

→ First of all, find the initial state by utilizing the reset function and discretize_observation function. Next, use max function in numpy to find the maximal Q value in the initial state of the Q table and return it.

Part 3: DQN in CartPole-v0

learn:

```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
        # 2. Sample trajectories of batch size
        batch_state, batch_action, batch_reward, batch_next_state, done = self.buffer.sample(self.batch_size)

        # 3. Forward the data to the evaluate net and the target net.
        actions = torch.tensor(np.array(batch_action).reshape(len(batch_action), 1), dtype=torch.long)
        rewards = torch.tensor(np.array(batch_reward).reshape(len(batch_reward), 1), dtype=torch.float)

        # Compute Q-values for current states and selected actions
        current_q_value = self.evaluate_net(torch.tensor(np.array(batch_state), dtype=torch.float)).gather(1, actions)

        # Compute Q-values for next states using the target net
        next_q_values = self.target_net(torch.tensor(np.array(batch_next_state), dtype=torch.float)).detach()
        target_q_values = rewards + self.gamma * next_q_values.max(1).values.unsqueeze(-1) # target Q value
        for i in range(len(done)):
            if done[i]:
                target_q_values[i][0] = 0

        # 4. Compute the loss with MSE.
        loss = F.mse_loss(current_q_value, target_q_values)

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

        # 6. Backpropagation.
        loss.backward()

        # 7. Optimize the loss function.
        self.optimizer.step()
        # End your code
        torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
```

→ First of all, every 100 steps, the parameters of the target network are updated to match the parameters of the evaluation network. Secondly, set the sample trajectories by using the sample function. Thirdly, I convert a batch of action and reward to numpy array and use reshape function to make it an appropriate dimension. Next, utilize tensor function to convert it to a given data type tensor. Subsequently, I use the evaluate net and state to calculate the current Q value and use the target net and next state to calculate the next Q value. We can use the next Q value to calculate the target Q value with the formula we've known. Note that I use a for loop to handle terminal states in the environment. Fourthly, use mse_loss function to compute the mean squared error loss between the current Q-values and the target Q-values. Fifthly, use the zero_grad function to set the gradients of all parameters in the neural network to zero in order to make gradients accumulate from zero. Sixth and seventh, update the parameters of the neural network in the direction that minimizes the loss function during optimization. Finally, save the parameters of the target network.

choose_action:

```
    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
            state_torch = torch.tensor(state, dtype=torch.float32)  # Convert state to a tensor
            q_values = self.evaluate_net(state_torch)  # Get Q-values for the state from the evaluate network
            if np.random.uniform(0, 1) < 1-self.epsilon:  # Choose a random action with epsilon probability
                action = env.action_space.sample()
            else:
                action = q_values.argmax().item()  # Choose the action with the highest Q-value
            # End your code
        return action
```

→ Convert the state to a tensor and use it to get the Q values. Subsequently, implement epsilon greedy algorithm. If the random number is smaller than (1 - epsilon), just randomly choose an action by using the env.action_space.sample(). Otherwise, choose the action with the highest Q value.

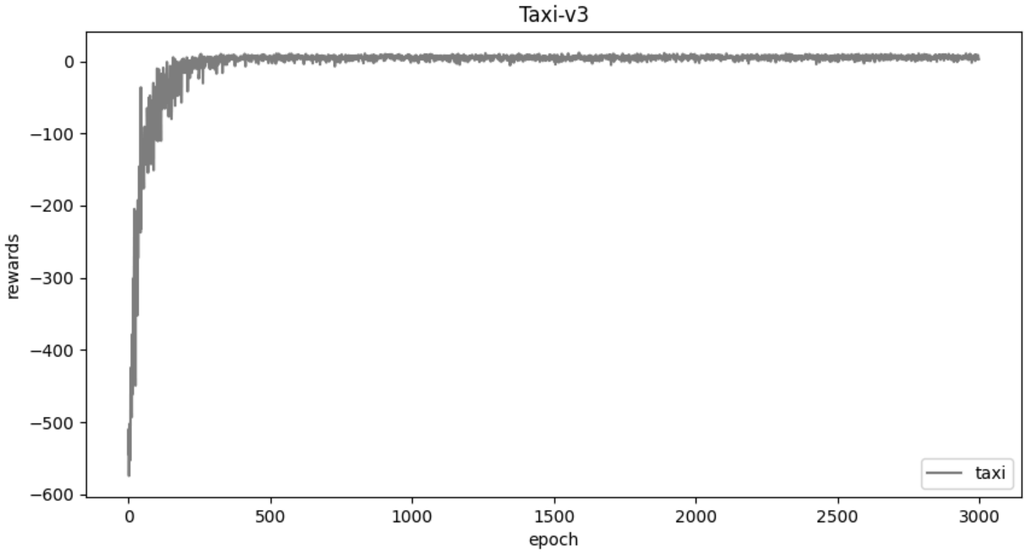
check_max_Q:

```
    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
        # Reset the environment to obtain the initial state
        initial_state = self.env.reset()
        # Convert the initial state to a tensor
        initial_state_tensor = torch.tensor(initial_state, dtype=torch.float32)
        # Get Q-values for the initial state from the target network
        q_values = self.target_net(initial_state_tensor).detach()
        # Calculate the max Q value
        max_q = q_values.max(0).values.unsqueeze(-1)
        max_q = float(max_q[0])
        return max_q
        # End your code
```

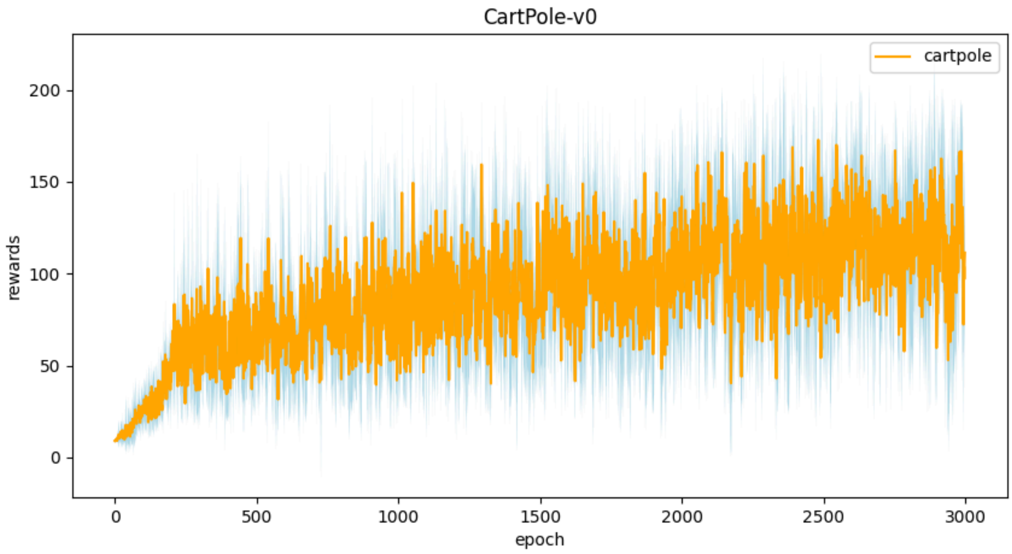
→ Utilize the reset function in env to get the initial state. Next, convert the initial state to the tensor and use it and target net to get a Q values. Finally, find the max Q value and return it.

Part II. Experiment Results:

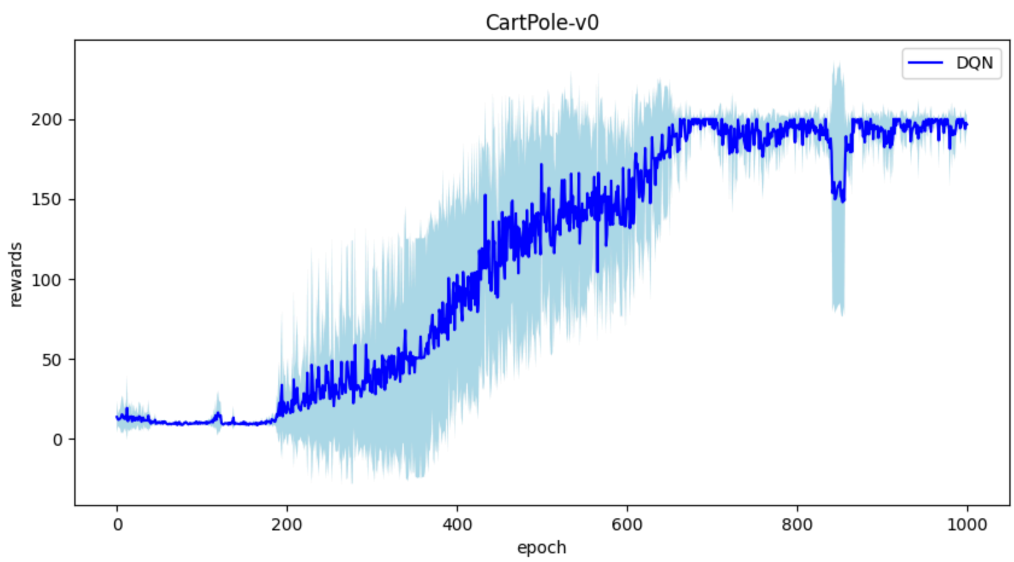
1. taxi.png



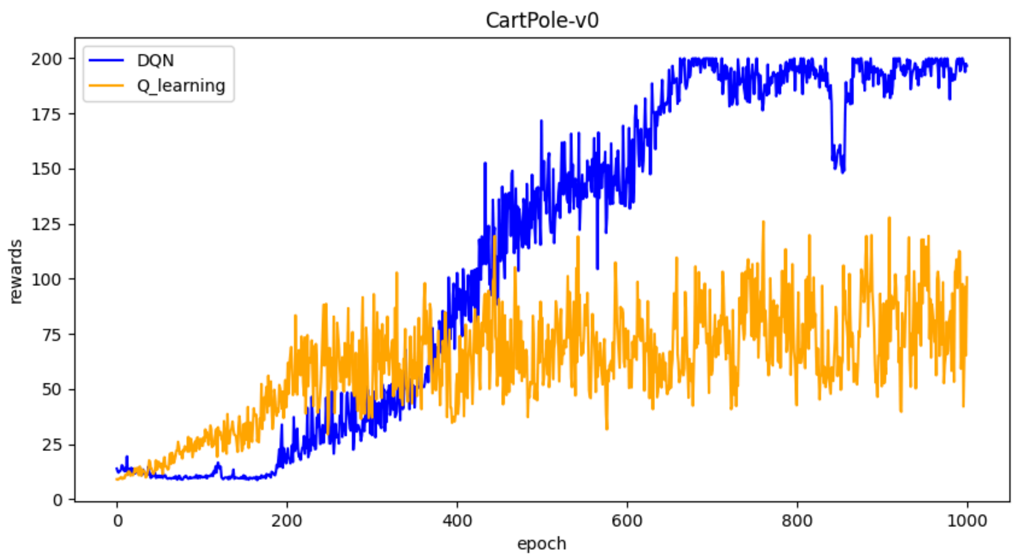
2. cartpole.png



3. DQN.png



4. compare.png



Part III. Question Answering:

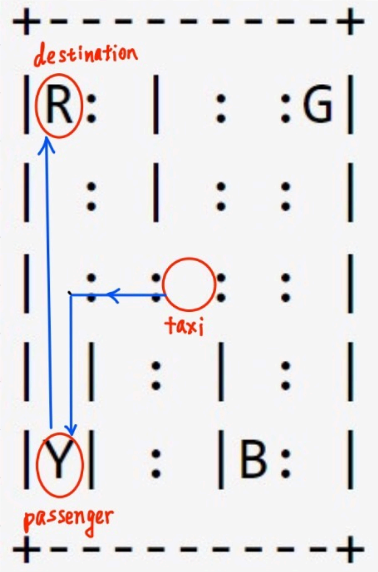
1. Calculate the optimal Q-value of a given state in Taxi-v3, and compare with the Q-value you learned (Please screenshot the result of the “check_max_Q” function to show the Q-value you learned). (10%)

learned Q value:

```
average reward: 8.07
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146700000021
```

optimal Q value:

1.



The shortest route will be the blue line in the graph.

score :

- lose 1 point for every timestep it takes
- receive 20 points for a successful drop-off
- 10 points penalty for illegal pick-up and drop-off actions

⇒ We will take 10 steps, 5 for picking up the passenger, and the others for dropping off the passenger

$$Q = (-1) + 0.9\times(-1) + (0.9)^2\times(-1) + \cdots + (0.9)^8\times(-1) + (0.9)^9 \cdot 20$$
$$= (-1)\times\frac{1-(0.9)^9}{1-0.9} + (0.9)^9\times 20$$
$$= (-10)\left(1-(0.9)^9\right) + (0.9)^9\times 20$$
$$= (-10) + 10\times(0.9)^9 + 20\times(0.9)^9$$
$$\fallingdotseq (-10) + 11.62261467 = 1.62261467 ※$$

→ The Q-value I learned is nearly identical to the optimal Q-value I calculated. It means that it trains well to evalute the Q value.

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%)

by cartpole .py:

```
average reward: 131.45
max Q:29.441717187149692
```

by DQN .py:

```
reward: 193.63
max Q:33.74856948852539
```

the optimal Q-value:

2.

Rewards : +1 for every step taken, including the termination step, is allotted.

Episode ends : When episode is greater than 200 (for v0), it will truncate.

⇒ We will get the optimal Q-value when we keep the pole upright until 200 th episode

$$Q = 1 + 1\times(0.97)^1 + 1\times(0.97)^2 + 1\times(0.97)^3 + \cdots + 1\times(0.97)^{199}$$
$$= \frac{1-(0.97)^{200}}{1-0.97} \fallingdotseq 33.25758633$$

→ Both of the learned Q-value are close to the optimal Q-value. However, it can be observed that the Q-value obtained from DQN.py (http://DQN.py) is even closer to the optimal Q-value, which means that DQN method can perform a better evaluation.

3.

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

The observation is continuous. In other words, it exist infinite state-action pairs. Due to it, it is hard to create a Q table. With an eye to solving it, we discretize the observation.

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

If we increase “num_bins”, the observation space will be divided into smaller intervals, which enable the agent to distinguish between more subtle differences in the environment. Therefore, it can contribute to more precise control actions and potentially better performance. In conclusion, I expect that the performance will be improved if we increase the the “num_bins” in modernration. However, it exists some problem. I explain it in c.

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

Increasing “num_bins” also means the increase of complexity. The time it takes may skyrocket, leading the slower training. Besides, it also needs more memory to store the larger Q table owing to the increased number of states.

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

DQN model performs better in Cartpole-v0. The reason is that DQN uses the continuous states, which can enable the agent to control actions precisely. Compared to DQN, discretized Q learning use discretized state, which leads to some data loss. Therefore, discretized Q learning model cannot perform as well as DQN model. It can also be observed by the result we got before. In the compare.png, the average reward of DQN is higher than the one of the discretized Q learning. Besides, the max Q value of the initial state of DQN is much closer to the optimal Q value.

5.

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

The purpose of using the epsilon-greedy algorithm is to strike a balance between exploration and exploitation. Exploration makes the agent explore the environment by randomly selecting actions with a probability of epsilon, which helps the agent learn more about the environment and improve its policy. As for exploitation, it allows the agent to select the action by using the value we have gained before. Via explointation, it can maximize the agent's short-term reward. In conclusion, epsilon greedy algorithm provides a way to make agent train data better.

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

If we don't use the epsilon greedy algorithm, it means that we only perform exploitation. In other words, the agent will tend to exploit the current policy it has learned so far without trying out new actions that could potentially lead to better long-term rewards, which contributes to the worse training performance.

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

In my opinion, it is possible to achieve the same performance without the epsilon greedy algorithm. We can use another way to do the similar thing. For instance, I browsed the Internet and find a method of exploration, called Boltzmann exploration. In this exploration, the agent draws actions from a boltzmann distribution (softmax) over the learned Q values. Despite the fact that it is more complecated, it can achieve the same or even better performance.

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

The purpose of testing section is to evaluate the performance of the learned policy in training section. Therefore, it does not need to explore. The only thing it should do is to follow the policy we get by training, which can allow us to determine whether the training is good or not.

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

In “choose_action” function, we should choose the best action with given state and epsilon, which is no need for performing gradient calculation. Therefore, we utilize “with torch.no_grad():” function to disable gradient calculation to improve the efficient.