Homework 4:

Reinforcement Learning

Report Template

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

- taxi.py

```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
    if (np.random.uniform(0, 1) > self.epsilon): # exploitation
        return np.argmax(self.qtable[state])
    else: # exploration
        return self.env.action_space.sample()
    # End your code
```

```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
    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
    if done:
        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
    # return max Q
    return np.max(self.qtable[state])
    # End your code
```

- cartpole.py

```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
```

```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
```

```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
    return tuple([self.discretize_value(obs, bin) for obs, bin in zip(observation, self.bins)])
    # End your code
```

```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
    if (np.random.uniform(0, 1) > self.epsilon): # exploitation
        return np.argmax(self.qtable[state])
    else: # exploration
        return self.env.action_space.sample()
    # End your code
```

```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
    # Get the max Q value of the next state
    max_next = np.max(self.qtable[next_state])
    if done: max_next = 0

    # Get the Q value of the current state
    original = self.qtable[state + (action,)]

    # Update Q value with Q-learning
    self.qtable[state + (action,)] = (1 - self.learning_rate) * original + self.learning_rate * (reward + self.gamma * max_next)

    if done:
    # End your code
        np.save("./Tables/cartpole_table.npy", self.qtable)
```

- DQN.py

```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

    # Get the Q value of initial state
    initial_state = torch.tensor(self.env.reset(), dtype=torch.float)
    q_value = self.target_net(initial_state).detach()

    # Get max Q value
    max_q = q_value.max(0).values.unsqueeze(-1)
    max_q = float(max_q[0])
    return max_q
    # End your code
```

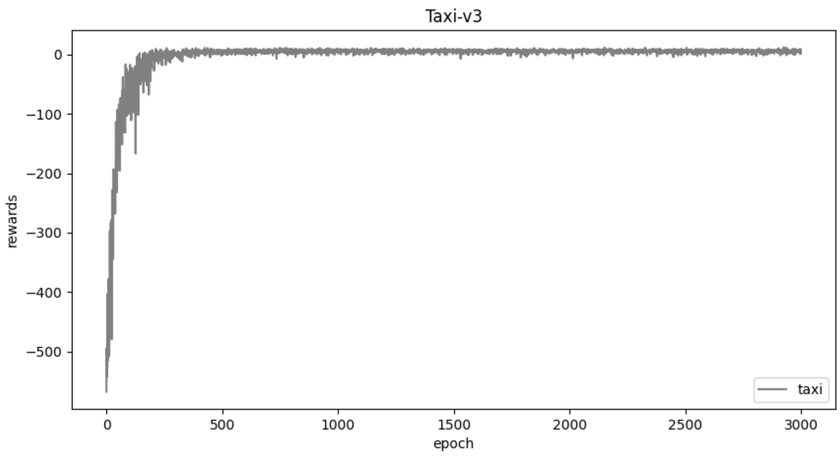
```python
def learn(self):
        '''
        - Implement the learning function.
        - Here are the hints to implement.
        Steps:
        -----
        1. Update target net by current net every 100 times. (we have done this for you)
        2. Sample trajectories of batch size from the replay 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
        # Add attribute step to record how many steps have done
        if not hasattr(self, "step"): self.step = 0

        # 2. Sample trajectories of batch size from the replay buffer.
        batch_state, batch_action, batch_reward, batch_next_state, batch_done = self.buffer.sample(self.batch_size)

        action=torch.tensor(np.asarray(batch_action).reshape(len(batch_action), 1), dtype=torch.long)
        reward=torch.tensor(np.asarray(batch_reward).reshape(len(batch_reward), 1), dtype=torch.float)

        # 3. Forward the data to the evaluate net and the target net.
        q_evaluate = self.evaluate_net(torch.tensor(np.asarray(batch_state), dtype=torch.float)).gather(1, action)
        q_next = self.target_net(torch.tensor(np.asarray(batch_next_state), dtype=torch.float)).detach()
        q_target = reward + self.gamma * q_next.max(1).values.unsqueeze(-1) # target Q value
        for i in range(len(batch_done)):
            if batch_done[i]:
                q_target[i][0] = 0

        # 4. Compute the loss with MSE.
        loss_function = nn.MSELoss()
        loss = loss_function(q_evaluate, q_target)

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

        # 6. Backpropagation.
        loss.backward()

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

        # Update count and step
        self.count += 1
        self.step += 1

        # Save the result every 100 steps
        if self.step % 100 == 0:
            torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
        # End your code
```

```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
        # TODO
        x = torch.tensor(state, dtype=torch.float)
        if np.random.uniform(0, 1) > self.epsilon: # exploration
            # Predict Q value from evaluate net
            action_values = self.evaluate_net(x)
            # Pick action that contains largest Q value
            action = torch.argmax(action_values).item()
        else: # exploitation
            action = env.action_space.sample()
        # End your code
    return action
```

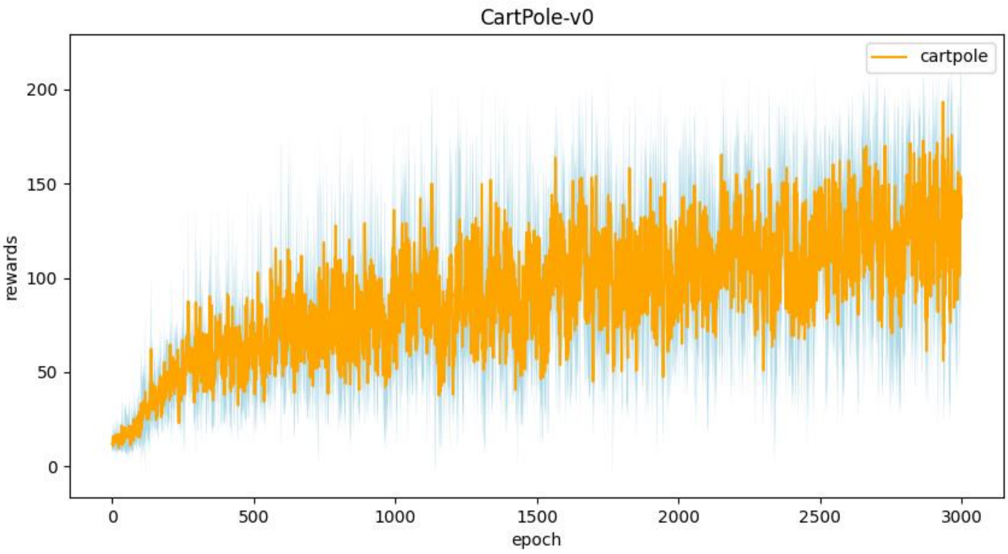
Part II. Experiment Results:

Please paste taxi.png, cartpole.png, DQN.png and compare.png here.

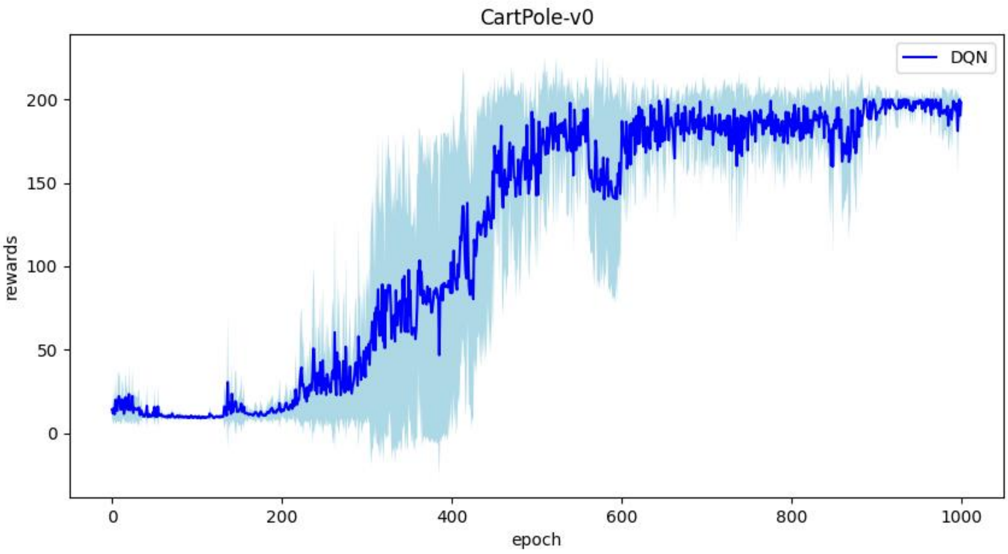
1. taxi.png:



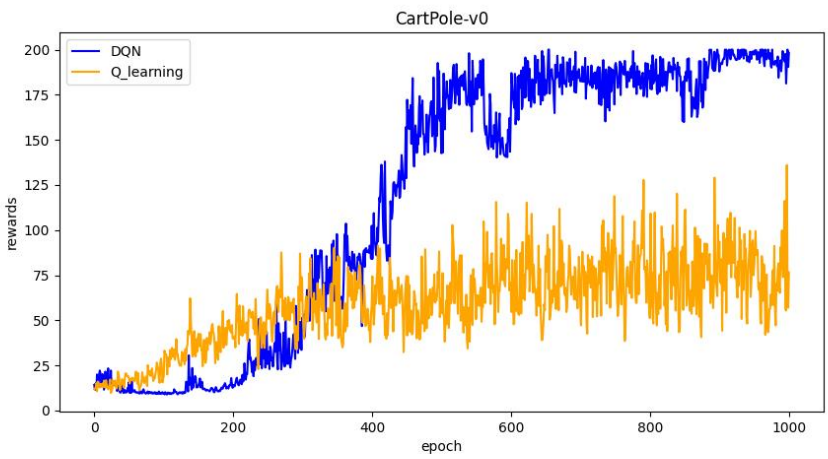
2. cartpole.png



3. DQN.png



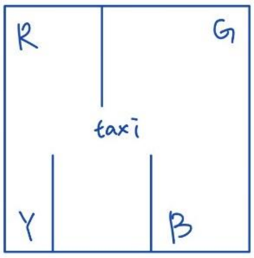
4. compare.png



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: 8.05
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146699999995
```



passenger at Y, destination at R

gamma = 0.9

reward { -1 per step unless other reward is triggered; +20 delivering passenger; -10 executing "pickup" and "drop-off" illegally.

5 steps to pickup the passenger

5 steps to drop-off the passenger

$$Q_{opt} = -1 + 0.9\cdot(-1) + (0.9)^2\cdot(-1) + \cdots + (0.9)^8\cdot(-1) + (0.9)^9\cdot 20$$

$$= (-1)\cdot\frac{1-(0.9)^9}{1-0.9} + (0.9)^9\cdot 20$$

$$= -10\cdot\{1-(0.9)^9\} + (0.9)^9\cdot 20$$

$$= -10 + (0.9)^9\cdot 30 \fallingdotseq 1.62261467$$

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%)**

```
average reward: 92.48
max Q:29.309158145736305
```

Rewards: +1 for every step taken

Episode end: Truncation is executed for episode length greater than 500

gamma = 0.97

$$\Rightarrow Q_{opt} = 1 + 1\cdot(0.97)^1 + 1\cdot(0.97)^2 + \cdots + 1\cdot(0.97)^{199}$$

$$= \frac{1-(0.97)^{200}}{1-0.97} = \frac{1-(0.97)^{200}}{0.03} \fallingdotseq 33.2579586$$

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 Q 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 loss. 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 choose 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 goal 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 in DQN? **(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.