HW4 report

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

- 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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        action = np.argmax(self.qtable[state]) # Choose best action by Q-table
        if np.random.rand() >= self.epsilon: # Decide whether to explore
            action = self.env.action_space.sample()
        return action

        # 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 acti

        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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        self.qtable[state, action] = (1 - self.learning_rate) * self.qtable[state, action] + \
        self.learning_rate * (reward + self.gamma * self.check_max_Q(next_state)) # Update Q-table

        # End your code
```

```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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        return np.max(self.qtable[state]) # Calculate max Q-value of given state

        # End your code
```

- Part 2

```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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        bins = np.linspace(lower_bound, upper_bound, num_bins, endpoint = False)
        return bins[1:] # Use linspace to spilt into 7 parts and do not take first part

        # 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 go
            The return value of discretize_value(5, [2. 4. 6. 8.]) should
        Hints:
            1. This can be done with a numpy function.
        """
        # Begin your code
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        return np.digitize(value, bins) # Use digitize to discretize value

        # 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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        dis_value = () # Define a empty tuple
        for obs, bin in zip(observation, self.bins):
            dis_value += (self.discretize_value(obs, bin),) # Discretized the 4 features of observation
        return dis_value

        # 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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        action = np.argmax(self.qtable[state]) # Choose best action by Q-table
        if np.random.rand() >= self.epsilon: # Decide whether to explore
            action = self.env.action_space.sample()
        return action
```

```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 acti
        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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        state_action_pair = state + (action,) # Use tuple to be Q-table's index
        self.qtable[state_action_pair] = (1 - self.learning_rate) * self.qtable[state_action_pair] + \
        self.learning_rate * (reward + self.gamma * np.max(self.qtable[next_state])) # Update Q-table

        # End your code
```

```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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        initial_state = self.discretize_observation(self.env.reset()) # Discretize the initial state
        return np.max(self.qtable[initial_state]) # Calculate max Q-value of initial state

        # 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
        # TODO
        #raise NotImplementedError("Not implemented yet.")
```

```python

        # 2. Sample trajectories of batch size from the replay buffer and make some modifications.
        state_batch, action_batch, reward_batch, next_state_batch, mask_batch = self.buffer.sample(self.batch_size)
        state_batch = torch.Tensor(state_batch)
        action_batch = torch.LongTensor(action_batch).reshape(self.batch_size, 1)
        reward_batch = torch.Tensor(reward_batch).reshape(self.batch_size, 1)
        next_state_batch = torch.Tensor(next_state_batch)
        mask_batch = torch.Tensor(mask_batch).reshape(self.batch_size, 1)

        # 3. Forward the data to the evaluate net and the target net and make some modifications.
        q_eval = self.evaluate_net(state_batch).gather(1, action_batch)
        next_q = self.target_net(next_state_batch)
        next_q = torch.Tensor([max(q) for q in next_q]).reshape(self.batch_size, 1)
        q_target = reward_batch + self.gamma * next_q * (1 - mask_batch)
        # I use 1 - mask to make sure if episode is done, it don't take additional Q-value

        # 4. Compute the loss with MSE.
        loss = F.mse_loss(q_eval, q_target)
```

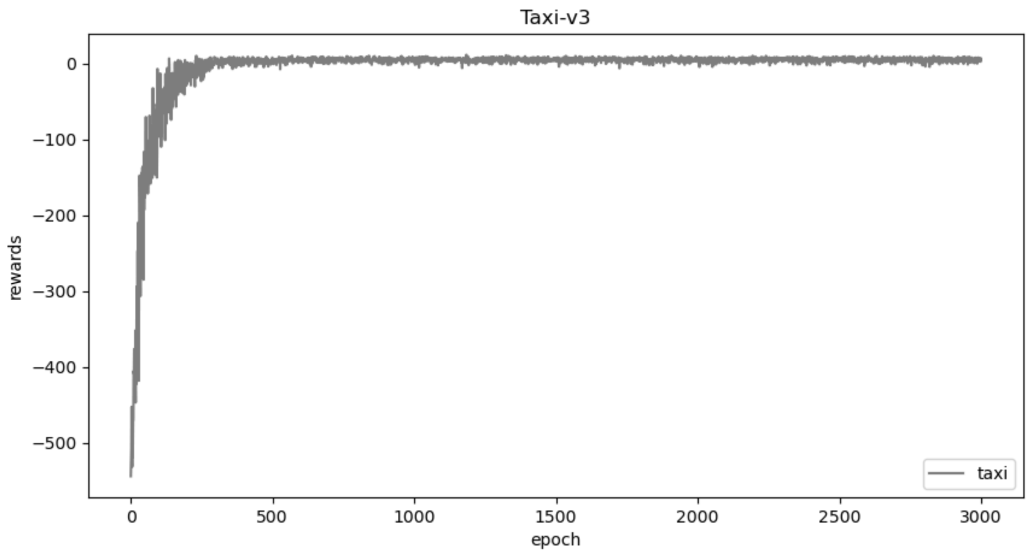
```python

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

        # 6. Backpropagation.
        loss.backward()

        # 7. Optimize the loss function and update count.
        self.optimizer.step()
        self.count += 1

        # 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
            #raise NotImplementedError("Not implemented yet.")

            state = torch.Tensor(state)
            action = torch.argmax(self.evaluate_net(state)).item() # Choose best action by Q-table
            if np.random.rand() >= self.epsilon: # Decide whether to explore
                action = self.env.action_space.sample()

            # End your code
```

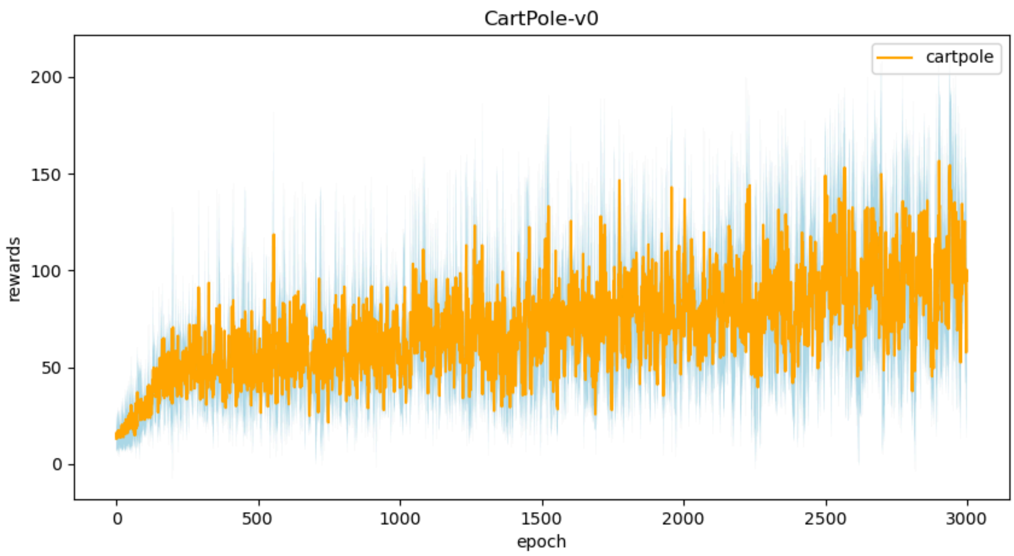
```python
    def check_max_Q(self):
        """
        - Implement the function calculating the max Q value of initial state(self.env.reset()).
        - Check the max Q value of initial state
        Parameter:
            self: the agent itself.
            (Don't pass additional parameters to the 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
        # TODO
        #raise NotImplementedError("Not implemented yet.")

        initial_state = torch.Tensor(self.env.reset())
        q_value = self.target_net(initial_state)
        return float(max(q_value)) # Calculate max Q-value of initial state

        # End your code
```

Part II. Experiment Results:

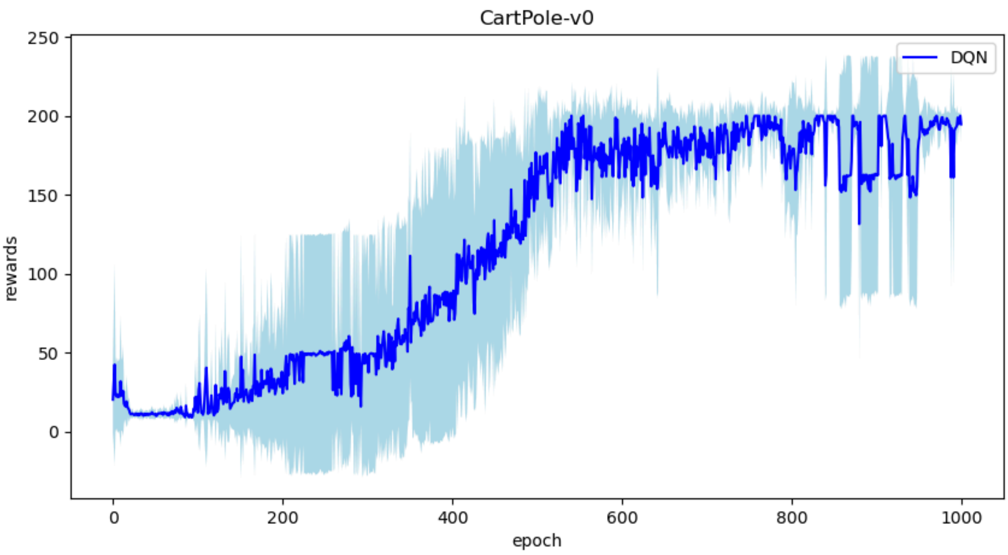
1. taxi.png



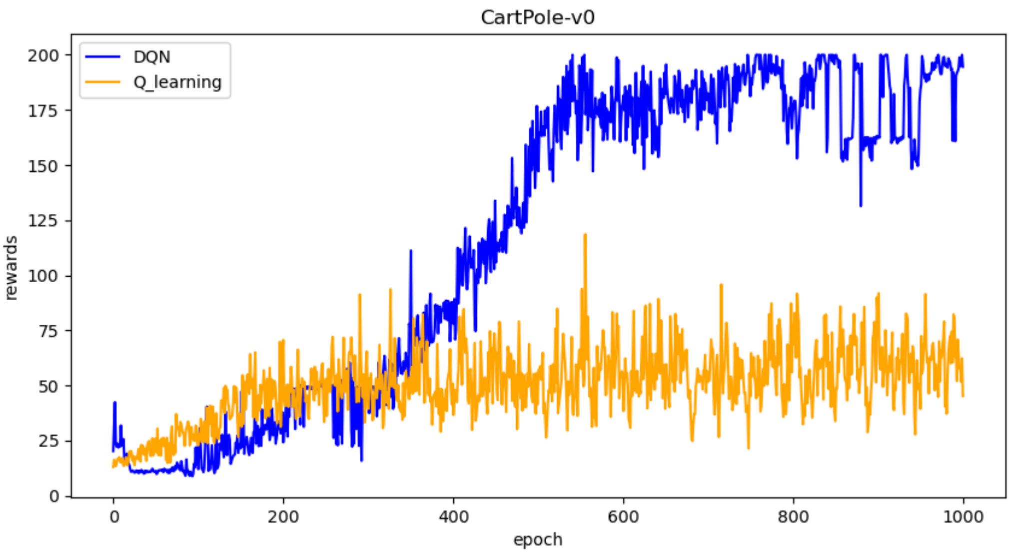
2. cartpole.png



3. DQN.png



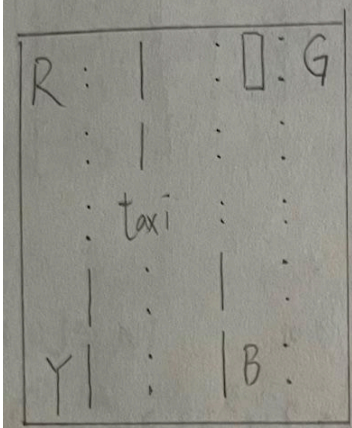
4. compare.png



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%)

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



reward $\begin{cases} -1, & \text{per step unless other reward is triggered} \\ +20, & \text{delivering passenger} \\ -10, & \text{executing "pick up" and "drop off" illegally} \end{cases}$

Need 5 steps to pick up passenger and 5 steps to drop off passenger

$$Q_{opt} = -(1+0.9+0.9^2+\cdots+0.9^8)+20\times 0.9^9$$
$$= -\left(\frac{1-(0.9)^9}{0.1}\right)+20\times 0.9^9 \fallingdotseq 1.6226$$

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%)

```
max Q:28.95079927460375
```

reward = +1, per step

Episode end when episode length reaches 200

$$\Rightarrow Q_{opt} = 1+0.97+0.97^2+\cdots+0.97^{199} = \frac{1-(0.97)^{200}}{0.03} \fallingdotseq 33.58$$

3.

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

      Because we use Q-table to store Q-value, if observation is continuous, it will be difficult to store.

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

      I think the performance will be better because it has a more accurate index for Q-table. But it will cost more memory for Q-table to store more accurate information.

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

      As I said in (b), it will cost more memory. Moreover, it will spend more time on updating the Q-table due to more states.

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

   DQN performs better than discretized Q learning in Cartpole-v0. Because discretized Q learning discretized the state space, which will lose accuracy contrast to original continuous state space.

5.

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

      To let the agent explore, not just follow the greedy choice.

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

      If we don't use epsilon greedy, but just use a common greedy algorithm, the performance will depend on whether you are lucky or not. If you are lucky, you may get all important information without exploration, otherwise you may not get good performance due to the loss of 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%)

      For sure you can achieve the same performance without epsilon greedy. As I said in (b), if you are lucky enough, or you can fine-tune the hyperparameter to achieve the similar performance.

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

Because we just want to test our model whether it is good or not, we don't need the epsilon greedy to make exploration.

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

   It means that it will not compute the gradients and do backpropagation for this part because there is no need for this. And it can speed up the computation.