Homework 4:

Reinforcement Learning

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

- Please screenshot your code snippets of Part 1 ~ Part 3, and explain your implementation
- Part 1, taxi

```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.")
        # Choose the best action according to the qtable and epsilon
        if np.random.uniform(0, 1) < self.epsilon or np.sum(self.qtable[state]) == 0:
            # Explore
            action = self.env.action_space.sample()
        else:
            # Exploit
            action = np.argmax(self.qtable[state])
        return action
        # 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 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
        # TODO
        # raise NotImplementedError("Not implemented yet.")
        # Calculate the new Q-value based on the Q-learning formula
        q_next = np.max(self.qtable[next_state])if not done else 0
        q_current = self.qtable[state, action]

        # Update the Q-table with the new Q-value
        q_newValue = (1 - self.learning_rate) * q_current + self.learning_rate \
                     * (reward + self.gamma * q_next)

        # Record the Q-value
        self.qtable[state, action] = q_newValue
        # 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
        # TODO
        # raise NotImplementedError("Not implemented yet.")
        # Find the max Q-value of the given state
        max_q = np.max(self.qtable[state])
        return max_q
        # End your code"
```

- Part 2, cartpole

```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.")
        # Slice the interval into #num_bins parts
        array = np.linspace(lower_bound, upper_bound, num_bins + 1)
        return array[1:-1]
        # End your code

    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
        # TODO
        # raise NotImplementedError("Not implemented yet.")
        # Discretize the value with given bins
        return np.digitize(value, bins)
        # End your code

    def discretize_observation(self, observation):
        """s

        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
```

```python
        # Begin your code
        # TODO
        # raise NotImplementedError("Not implemented yet.")
        state = []
        for index in range(4):
            # Discretize the observation
            state.append( self.discretize_value(observation[index], self.bins[index]) )
        return state
        # End your code

    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.")
        # Choose the best action according to the qtable and epsilon
        if np.random.uniform(0, 1) < self.epsilon or np.sum(self.qtable[state]) == 0:
            # Explore
            action = self.env.action_space.sample()
        else:
            # Exploit
            action = np.argmax(self.qtable[tuple(state)])
        return action
        # 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 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
        # TODO
        # raise NotImplementedError("Not implemented yet.")
        # Calculate the new Q-value based on the Q-learning formula
        q_next = np.max(self.qtable[tuple(next_state)])if not done else 0
        q_current = self.qtable[tuple(state)][action]

        # Update the Q-table with the new Q-value
        q_newValue = (1 - self.learning_rate) * q_current + self.learning_rate \
                    * (reward + self.gamma * q_next)

        # Record the Q-value
        self.qtable[tuple(state)][action] = q_newValue
        # End your code
        np.save("./Tables/cartpole_table.npy", self.qtable)

    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.")
        state = self.discretize_observation(self.env.reset())
        max_q = np.max(self.qtable[tuple(state)])
        return max_q
        # End your code
```

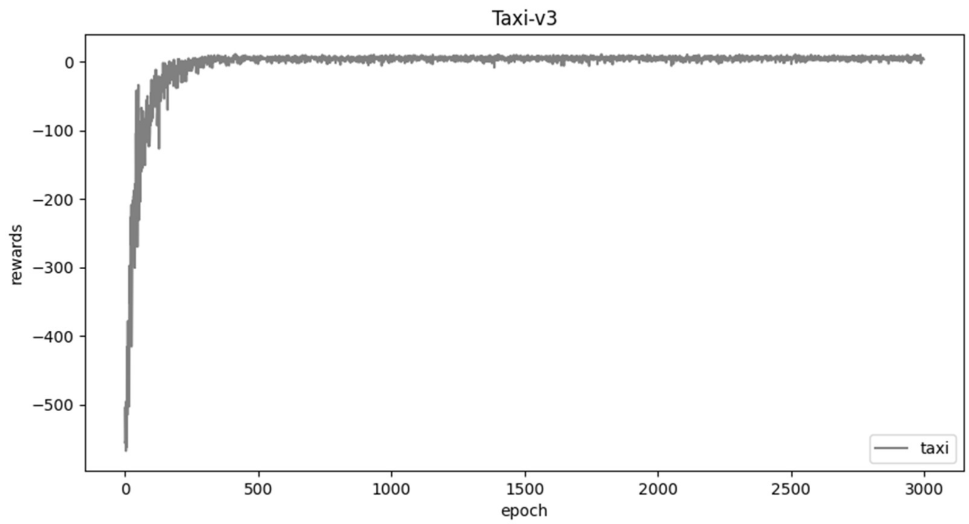
- Part 3, DQN

```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.")
        # Sample trajectories of batch size from the replay buffer.
        observations, actions, rewards, next_observations, done = self.buffer.sample(self.batch_size)

        # Forward the data to the evaluate net and the target net.
        observations = torch.FloatTensor(np.array(observations))
        actions = torch.LongTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_observations = torch.FloatTensor(np.array(next_observations))
        done = torch.BoolTensor(done)

        # Compute the loss with MSE.
        evaluate = self.evaluate_net(observations).gather(1, actions.reshape(self.batch_size, 1))
        nextMax = self.target_net(next_observations).detach()
        target = rewards.reshape(self.batch_size, 1) + self.gamma * nextMax.max(1)[0].view(self.batch_size, 1) * (~done).reshape(self.batch_size, 1)

        # Zero-out the gradients.
        MSE = nn.MSELoss()
        loss = MSE(evaluate, target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        # End your code
        torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")

    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.")
            if random.uniform(0,1) < self.epsilon:
                action = self.env.action_space.sample()
            else:
                action = torch.argmax(self.evaluate_net.forward(torch.FloatTensor(state))).item()
            # 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
        # TODO
        # raise NotImplementedError("Not implemented yet.")
        max_q = torch.max(self.evaluate_net.forward(torch.FloatTensor(self.env.reset()))).item()
        return max_q
        # End your code
```

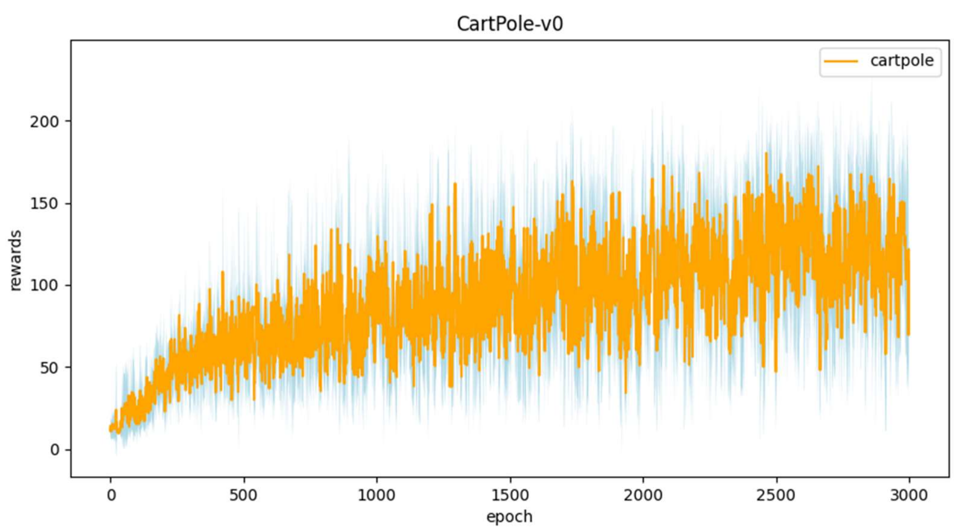
Part II. Experiment Results:

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

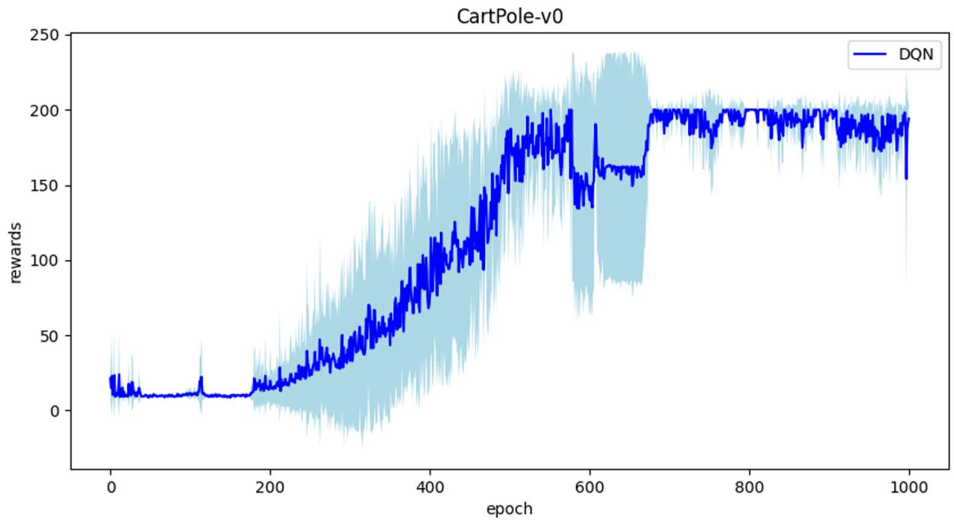
1. taxi.png:



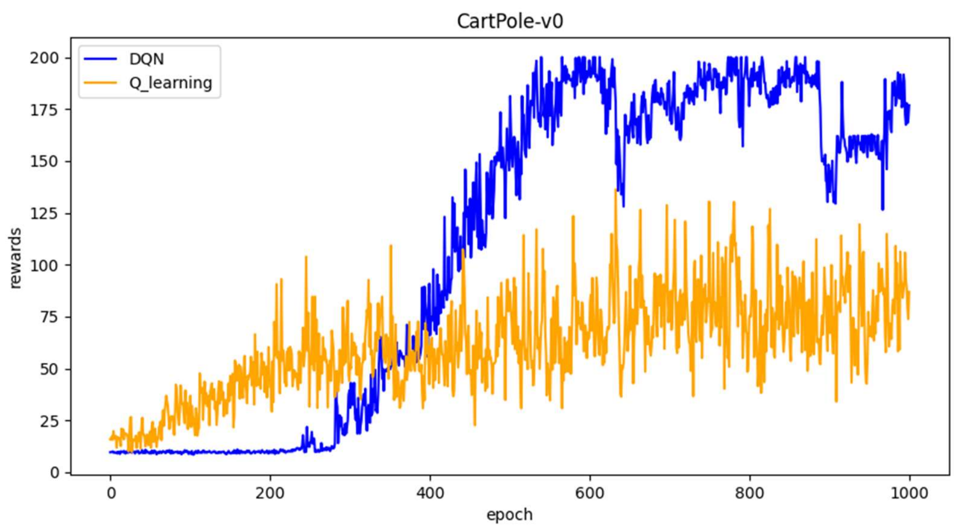
2. cartpole.png



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%)

$$Q_{\text{opt}}(s,a) = \sum_{s'} T(s,a,s')[\text{Reward}(s,a,s') + \gamma V_{\text{opt}}(s')]$$

$Q_{opt} \approx 1.6226$,approximately equals to the max Q we get.

```
(ai_hw4) alfonso@ubuntu:~/Git_workspace/Intro-to-AI-2023/HW4/AI_HW4$ python
3 taxi.py
#1 training progress
100%|                                        | 3000/3000 [00:52<00:00, 56.96it/s]
#2 training progress
100%|                                       | 3000/3000 [00:28<00:00, 105.46it/s]
#3 training progress
100%|                                       | 3000/3000 [00:09<00:00, 314.10it/s]
#4 training progress
100%|                                       | 3000/3000 [00:29<00:00, 101.72it/s]
#5 training progress
100%|                                       | 3000/3000 [00:20<00:00, 146.03it/s]
average reward: 7.54
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146699999995
```

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%)

$$\frac{1}{1-\gamma} = \frac{1}{1-0.97} = 33.3333$$

```
(ai_hw4) alfonso@ubuntu:~/Git_workspace/Intro-to-AI-2023/HW4/AI_HW4$ python
3 cartpole.py
#1 training progress
100%|                                        | 3000/3000 [02:06<00:00, 23.63it/s]
#2 training progress
100%|                                        | 3000/3000 [02:00<00:00, 24.84it/s]
#3 training progress
100%|                                        | 3000/3000 [02:03<00:00, 24.39it/s]
#4 training progress
100%|                                        | 3000/3000 [02:25<00:00, 20.67it/s]
#5 training progress
100%|                                        | 3000/3000 [01:57<00:00, 25.54it/s]
average reward: 172.96
max Q:31.094193147630833
```

```
(ai_hw4) alfonso@ubuntu:~/Git_workspace/Intro-to-AI-2023/HW4/AI_HW4$ python
3 DQN.py
#1 training progress
100%|                                        | 1000/1000 [02:01<00:00,  8.26it/s]
#2 training progress
100%|                                        | 1000/1000 [02:26<00:00,  6.82it/s]
#3 training progress
100%|                                        | 1000/1000 [02:34<00:00,  6.46it/s]
#4 training progress
100%|                                        | 1000/1000 [02:24<00:00,  6.92it/s]
#5 training progress
100%|                                        | 1000/1000 [08:44<00:00,  1.91it/s]
reward: 200.0
max Q:0.0336914137005806
```

3.

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

A: Discretize the observation would make the algorithm more easier to converge. In this case, the dimension of the state space is very high, and using continuous observations would make the algorithm difficult to train.

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

A: Better, because by increasing the number of bins, we can increase the granularity of the discretization and make the state space more fine-grained. This can make the policy more precise and enable better policies to be learned.

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

A: Increasing the number of bins will increase the dimension of the state space, which will make training time longer and require more memory to store the Q-table.

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

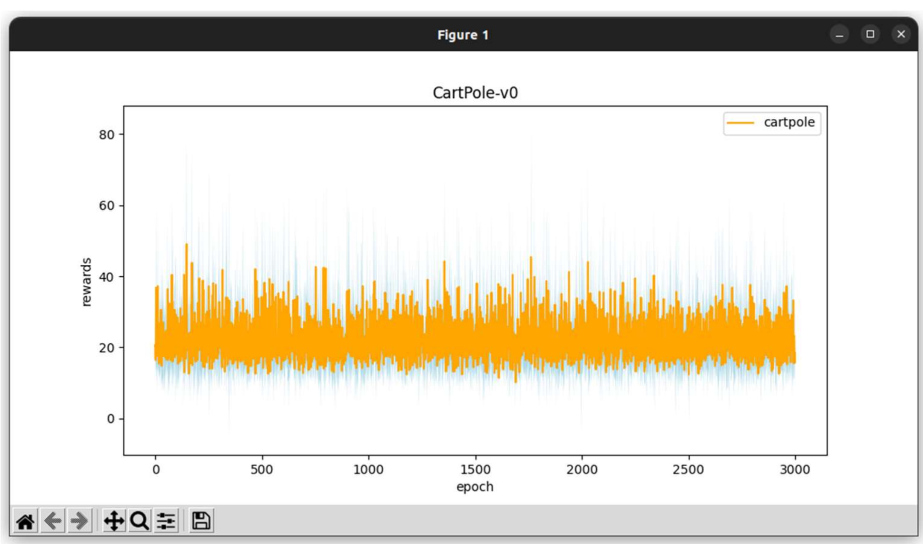
A: DQN, since CartPole-v0 has a continuous state space, which makes it challenging to create a discrete Q-table for Q-learning. Discretizing the state space can lead to a loss of information and reduced performance. On the other hand, DQN uses a neural network to approximate the Q-values, which can handle continuous state spaces more effectively.

5.

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

The main purpose of the epsilon greedy algorithm is to choose between exploration and exploitation when the agent has none or limited knowledge about the environment.

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



The result may looks like the plot above, the agent is not going to explore the unknown enviroment.

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

   Yes, softmax may reach same performance.

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

   Because the agent has already known the enviroment.

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

   A: It is used to disable gradient computation during the evaluation of the Q-values of the current state. During the testing phase, we only want to evaluate the learned policy and not update the weights of the network. Disabling gradient computation using "with torch.no_grad():" saves memory and computation time by preventing the calculation of gradients.