Homework 4: Reinforcement Learning

學號: 111550129　姓名: 林彥亨

Part I. Implementation

Q-learning in taxi

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

        # use random number from 0 to 1
        # if the number is bigger than the epsilon, then return the maximum Q in the Q table.
        # if the number is smaller than the epsilon, then return the random action.
        if np.random.uniform(0, 1) > self.epsilon:
            return np.argmax(self.qtable[state])
        else:
            return env.action_space.sample()

        # End your code
```

- 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
        # TODO
        # raise NotImplementedError("Not implemented yet.")
                                          (variable) learning_rate: float
        # calculate the new q-value with th
        self.qtable[state, action] += self.learning_rate * (
            reward + self.gamma * np.max(self.qtable[next_state]) - self.qtable[state, action]
        )

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

- choose_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
    # TODO
    # raise NotImplementedError("Not implemented yet.")
    # to return the maximum Q
    return np.max(self.qtable[state])

    # End your code
```

Q-learning in Cartpole

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

    # use the linspace to slice the interval with given lower bound, upper bound and number of bins.
    # return the list from index 1.
    return np.linspace(lower_bound, upper_bound, num_bins, endpoint=False)[1:]

    # End your code
```

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

        # Discretize the value with given bins.
        # Using np.digitize() to determine the value in which interval of bins.
        return np.digitize(value, bins, right=False)


        # End your code
```

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

        return [self.discretize_value(observation[i], self.bins[i]) for i in range(4)]
        # End your code
```

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

        if np.random.uniform(0, 1) > self.epsilon:
            return np.argmax(self.qtable[tuple(state)])
        else:
            return env.action_space.sample()

        # End your code
```

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

    self.qtable[tuple(state)][action] += self.learning_rate * (
        reward
        + self.gamma * np.max(self.qtable[tuple(next_state)])
        - self.qtable[tuple(state)][action]
    )

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

- check_max_Q( )

```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.")

    # check the maximum value of Q
    return np.max(self.qtable[tuple(self.discretize_observation(self.env.reset()))])

    # End your code
```

DQN in Cartpole

- learn( )

```python
        # Begin your code
        # TODO
        # raise NotImplementedError("Not implemented yet.")

        # Use function "sample" defined in class "replay_buffer" to get the sampled data.
        sample = self.buffer.sample(self.batch_size)
        # convert these sampled data into tensor.
        states = torch.tensor(np.array(sample[0]), dtype=torch.float)
        actions = torch.tensor(sample[1], dtype=torch.long).unsqueeze(1)
        rewards = torch.tensor(sample[2], dtype=torch.float)
        next_states = torch.tensor(np.array(sample[3]), dtype=torch.float)
        done = torch.tensor(sample[4], dtype=torch.bool)

        # "q_eval" is predicted values from evaluate network which is extracted based on "action".
        q_eval = self.evaluate_net(states).gather(1, actions)
        # "q_next" is actual values from target network.
        q_next = self.target_net(next_states).detach() * (~done).unsqueeze(-1)
        # "q_target" is the expected Q-values obtained from the formula "reward + gamma * max(q_next)".
        q_target = rewards.unsqueeze(-1) + self.gamma * q_next.max(1)[0].view(self.batch_size, 1)

        # use nn.MSELoss() func. to evaluate the loss of q_eval and q_target
        loss_func = nn.MSELoss()
        loss = loss_func(q_eval, q_target)

        # zerp-out the gradients before doing backpropagation.
        self.optimizer.zero_grad()
        loss.backward()
        # update the parameters.
        self.optimizer.step()

        # End your code
        torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
```

- choose_action( )

```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.")

            if np.random.uniform(0, 1) > self.epsilon:
                action = torch.argmax(
                    self.evaluate_net(torch.tensor(state, dtype=torch.float))
                ).item()
            else:
                action = env.action_space.sample()

            # End your code
        return action
```

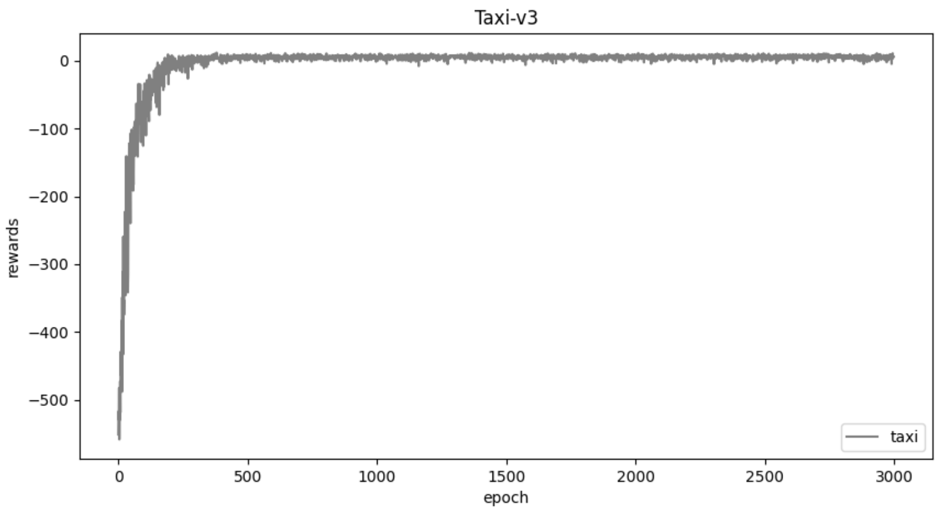
- check_max_Q( ).

```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.")

        # change the initial state to tensor, forward to the NN.
        x = torch.unsqueeze(torch.tensor(self.env.reset(), dtype=torch.float), 0)
        # find out the max Q and return it.
        return torch.max(self.target_net(x)).item()

        # End your code
```

Part II. Experiment Results

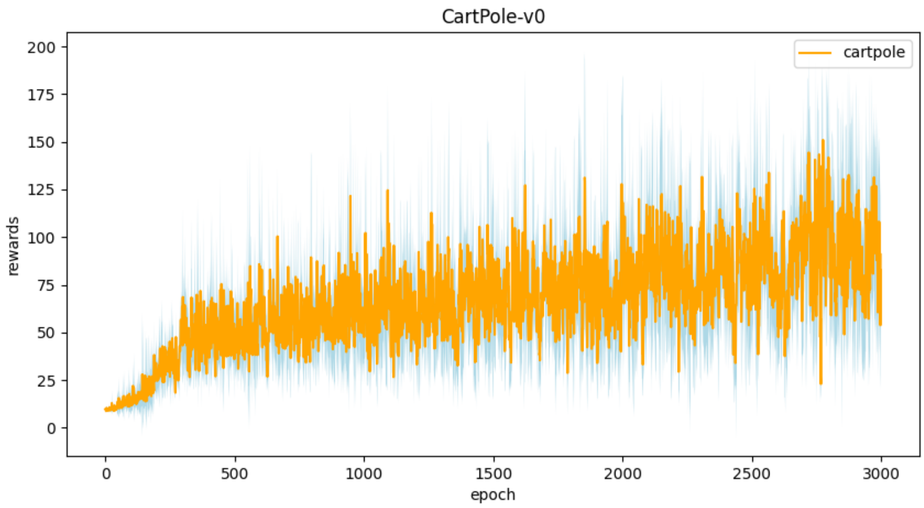
1. Q-learning in Taxi-v3

   - taxi.png



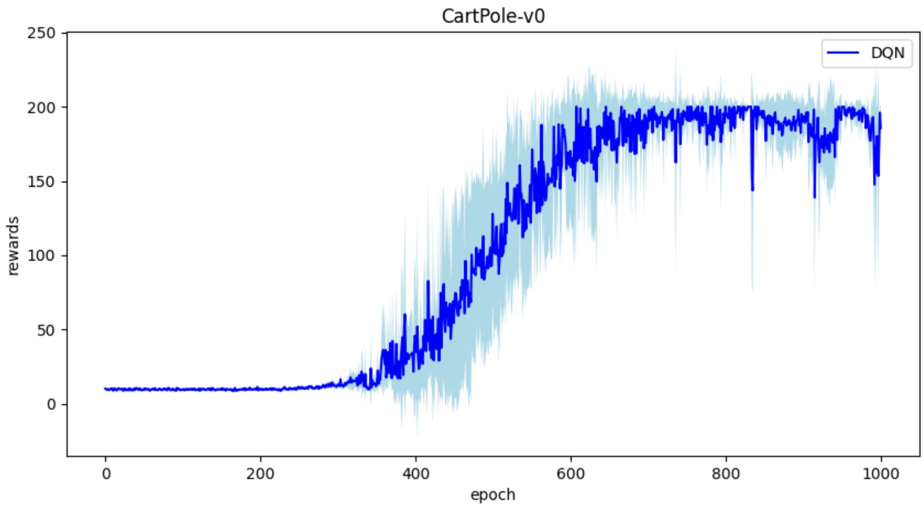
2. Q-learning in Cartpole-v0

   - cartpole.png



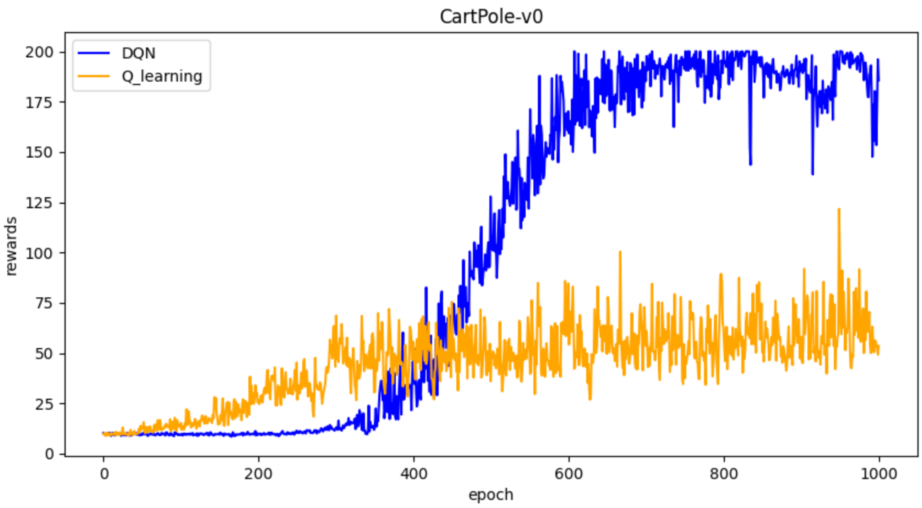
3. DQN in Cartpole-v0

   - DQN.png



4. Compare Q-learning with DQN

   - compare.png

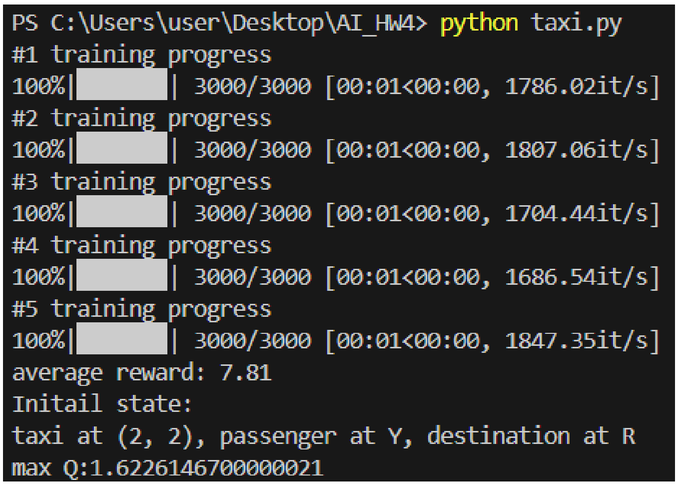


Part III. Question Answering

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%)

   A: Firstly, the taxi is at (2, 2), the passenger at Y, and the destination is at R, the reward are 9 steps of -1 and 1 step of 20.

   optimal Q-value: $-1 * (1 - r^9)/1 - r + 20 * (r^9) = -1 * (1 - 0.9^9)/1 - 0.9 + 20(0.9^9)$ = 1.6226…



```
PS C:\Users\user\Desktop\AI_HW4> python taxi.py
#1 training progress
100%|          | 3000/3000 [00:01<00:00, 1786.02it/s]
#2 training progress
100%|          | 3000/3000 [00:01<00:00, 1807.06it/s]
#3 training progress
100%|          | 3000/3000 [00:01<00:00, 1704.44it/s]
#4 training progress
100%|          | 3000/3000 [00:01<00:00, 1686.54it/s]
#5 training progress
100%|          | 3000/3000 [00:01<00:00, 1847.35it/s]
average reward: 7.81
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146700000021
```

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%)

optimal q-value: 1-r^(avg - reward)/1-r

⇒ the result is around 33.3333…

The result of the maxQ of the DQN is closer to the optimal value than the Qmax of Q-learning.

```
PS C:\Users\user\Desktop\AI_HW4> python cartpole.py

#1 training progress
100%|██████████| 3000/3000 [00:07<00:00, 395.25it/s]
#2 training progress
100%|██████████| 3000/3000 [00:07<00:00, 381.98it/s]
#3 training progress
100%|██████████| 3000/3000 [00:07<00:00, 392.46it/s]
#4 training progress
100%|██████████| 3000/3000 [00:08<00:00, 369.58it/s]
#5 training progress
100%|██████████| 3000/3000 [00:08<00:00, 336.03it/s]
average reward: 120.48
max Q:30.630427737043938
```

```
PS C:\Users\user\Desktop\AI_HW4> python DQN.py
#1 training progress
100%|██████████| 1000/1000 [02:55<00:00,  5.69it/s]
#2 training progress
100%|██████████| 1000/1000 [03:09<00:00,  5.28it/s]
#3 training progress
100%|██████████| 1000/1000 [03:00<00:00,  5.54it/s]
#4 training progress
100%|██████████| 1000/1000 [03:54<00:00,  4.27it/s]
#5 training progress
100%|██████████| 1000/1000 [03:07<00:00,  5.34it/s]
reward: 200.0
max Q:34.19133377075195
```

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

   A: Because the states are continuous, it makes Q-learning complex. We have to discretize the observation first.

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

   A: In my opinion, the performance may become better, since the discretized data is more similar for observation.

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

   A: "Oversampling" may cause the complexity of the computation.

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

   A:

The DQN perform better.

Here is the reasons:

a. DQN can deal with the continuous data; Q-learning deal with discretized data.
b. Since the neural network, DQN can relieve the problem .
c. DQNs, being more complex, might be prone to overfitting the training data in such a small problem setting, while discretized Q-learning remains straightforward and resilient to overfitting.

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

   A: the epsilon-greedy approach allows the agent to explore new actions while still primarily relying on the best-known actions, thus helping it adaptively improve its policy over time.
   b. What will happen, if we don't use the epsilon greedy algorithm in the CartPole-v0 environment? (3%)

   A: Without exploration, the agent will take much longer to discover the optimal policy because it only tries a limited set of actions. It may cause poor 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%)

   A: It is possible. Since it is some alternative algorithm can be used.
   d. Why don't we need the epsilon greedy algorithm during the testing section? (3%)

   A: During testing, the focus is on exploiting the learned policy to its fullest extent, which means relying on the highest-value actions identified during training, rather than exploring potentially less optimal ones.

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

   In this step, we have to choose the next action which is estimated by neural network, and we don't need to calculate the gradient.

   The torch.no_grad( ) can optimize the memory usage, speed up computation and ensure the correctness.