# Homework 4:

# Reinforcement Learning

# Report Template

Please keep the title of each section and delete examples. Note that please keep the questions listed in Part III.

## 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:

```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
    '''
    Since we are going to find a good balance between exploration and exploitation to produce higher reward, with epsilon
    equal to 0.05, I generated a number in 0~1. If it was bigger than epsilon, do exploitation; otherwise, do exploration.
    '''
    # TODO

    if random.uniform(0,1) < self.epsilon:
        return self.env.action_space.sample()

    return np.argmax(self.qtable[state])

    # raise NotImplementedError("Not implemented yet.")
    # 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
    '''
    Deviate the formula of optimal Q value for Q learning
    optimal Q = Qt
    optimal Q for next state = Qt'
    learning rate = Lt
    Qt -> (1-Lt)*Qt + Lt*(reward + Lt*(max of Qt'))
    '''
    # TODO

    if done:
```

```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
    '''
    In order to find the maximum Q value of actions under specific state in Q table.
    '''
    # TODO

    return np.max(self.qtable[state])
```

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
    '''
    Based on the num_bins, cut the interval into required pieces. And remove 2 heads at the both ends.
    '''
    # TODO

    return np.linspace(lower_bound, upper_bound, num_bins)[1:-1]    "linspace": Unknown word.
```

```python
def discretize_value(self, value, bins):    "discretize": Unknown word.
    """
    Discretize the value with given bins.    "Discretize": Unknown word.
    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.    "discretize": Unknown word.
        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
    '''
    Using function in numpy, and find which interval the target value is in.
    '''
    # TODO

    return np.digitize(value, bins)
```

```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
    '''
    Cut continuous original states into discrete states to produce a list.
    '''
    # TODO

    res = []
    for i in range(4):
        res.append(self.discretize_value(observation[i],self.bins[i]))
```

```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
    '''
    Since we are going to find a good balance between exploration and exploitation to produce higher reward, with epsilon
    equal to 0.05, I generated a number in 0~1. If it was bigger than epsilon, do exploitation; otherwise, do exploration.
    '''
    # TODO

    if random.uniform(0,1) < self.epsilon:
        return self.env.action_space.sample()

    return np.argmax(self.qtable[tuple(state)])
```

```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.    "enviornment": Unknown word.
        action: The exacuted action.    "exacuted": Unknown word.
        reward: Obtained from the enviornment after taking the action.    "enviornment": Unknown word.
        next_state: The state of the enviornment after taking the action.    "enviornment": Unknown word.
        done: A boolean indicates whether the episode is done.
    Returns:
        None (Don't need to return anything)
    """
    # Begin your code
    '''
    Deviate the formula of optimal Q value for Q learning
    optimal Q = Qt
    optimal Q for next state = Qt'
    learning rate = Lt
    Qt -> (1-Lt)*Qt + Lt*(reward + Lt*(max of Qt'))
    '''
    # TODO

    if done:
        nextBest = 0
    else:
        nextBest = np.max(self.qtable[tuple(next_state)])    "qtable": Unknown word.
```

```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
    '''
    In order to find the maximum Q value of actions under specific state in Q table.
    '''
    # TODO

    return np.max(self.qtable[tuple(self.discretize_observation(self.env.reset()))])
```

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.    "Backpropagation": Unknown word.
    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
    '''
    Deviate the formula of loss function
    optimal Q with weight = Qtw
    undone optimal Q of next state = V
    learning rate = Lt
    loss = average{[Qtw - (reward + Lt*V)]^2}
    '''
    # TODO
```

第 147 行 第 40 欄

```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.    "enviornment": Unknown word.
        (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
        '''
        Since we are going to find a good balance between exploration and exploitation to produce higher reward, with epsilon
        equal to 0.05, I generated a number in 0~1. If it was bigger than epsilon, do exploitation; otherwise, do exploration.
        '''
        # TODO

        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()    "argmax": Unknown word.

        # raise NotImplementedError("Not implemented yet.")
        # End your code
    return action
```

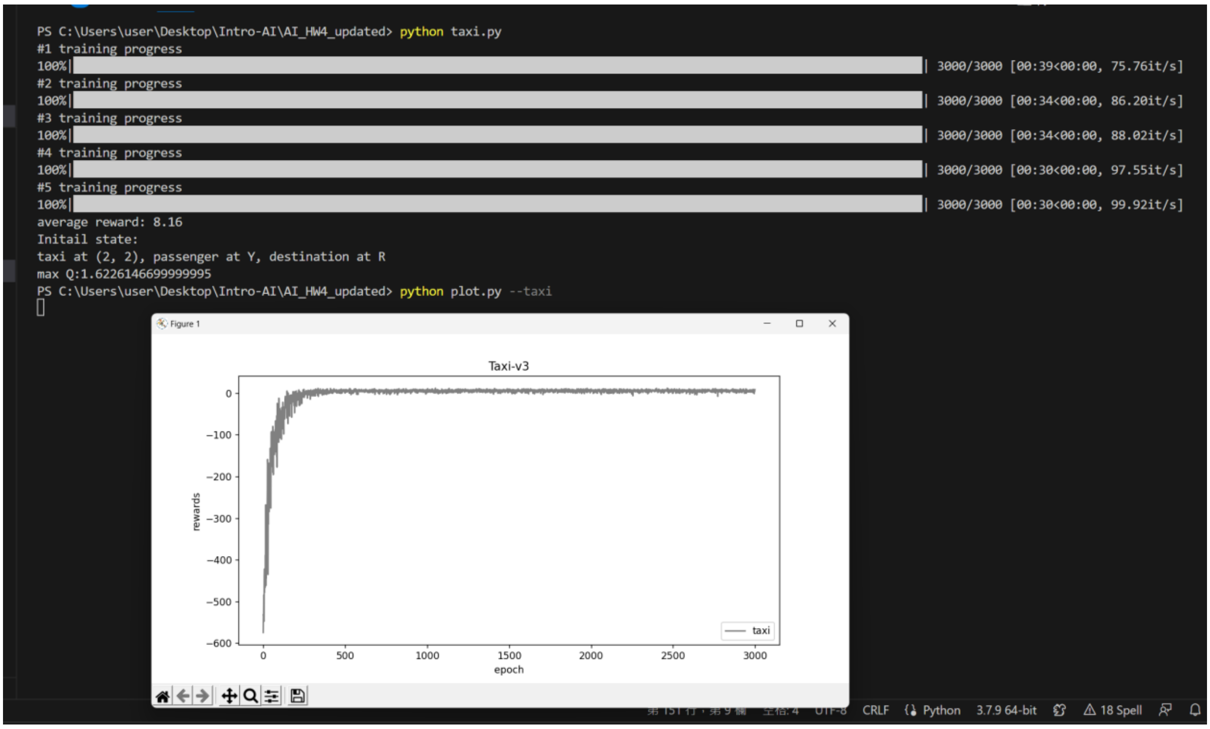
```python
def check_max_Q(self):
    """
    - Implement the function calculating the max Q value of initial state(self.env.reset()).
    - Check the max Q value of initial state
    Parameter:
        self: the agent itself.
        (Don't pass additional parameters to the function.)
        (All you need have been initialized in the constructor.)
    Return:
        max_q: the max Q value of initial state(self.env.reset())
    """
    # Begin your code
    '''
    In order to find the maximum Q value of actions under specific state in Q table.
    '''
    # TODO

    return torch.max(self.target_net(torch.FloatTensor(self.env.reset())))
```

## Part II. Experiment Results:

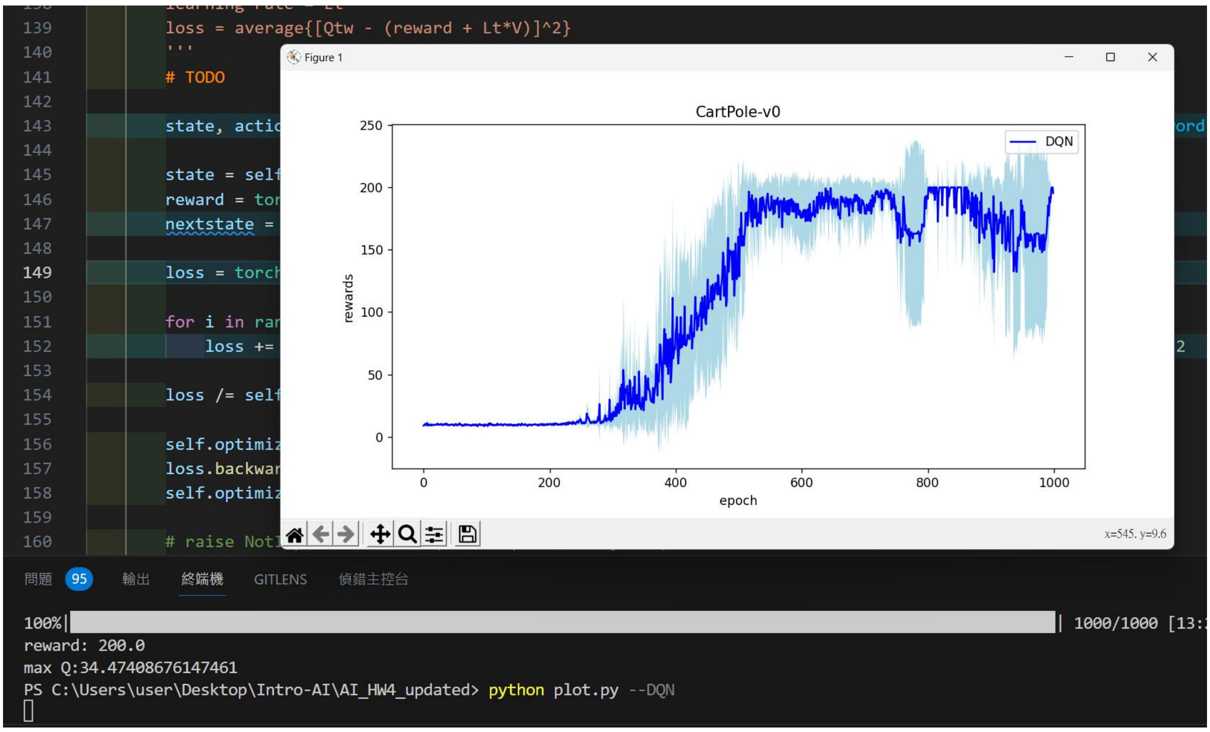
Please paste taxi.png, cartpole.png, DQN.png and compare.png here.

### 1. taxi.png:

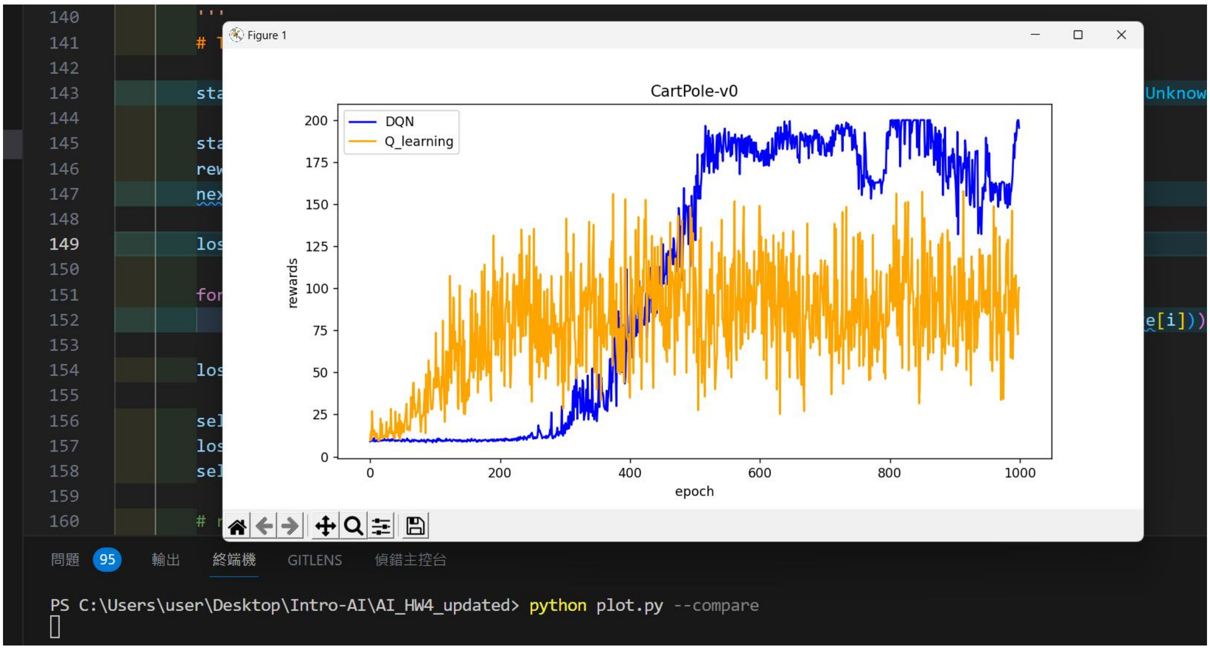


**2.** cartpole.png


**3.** DQN.png

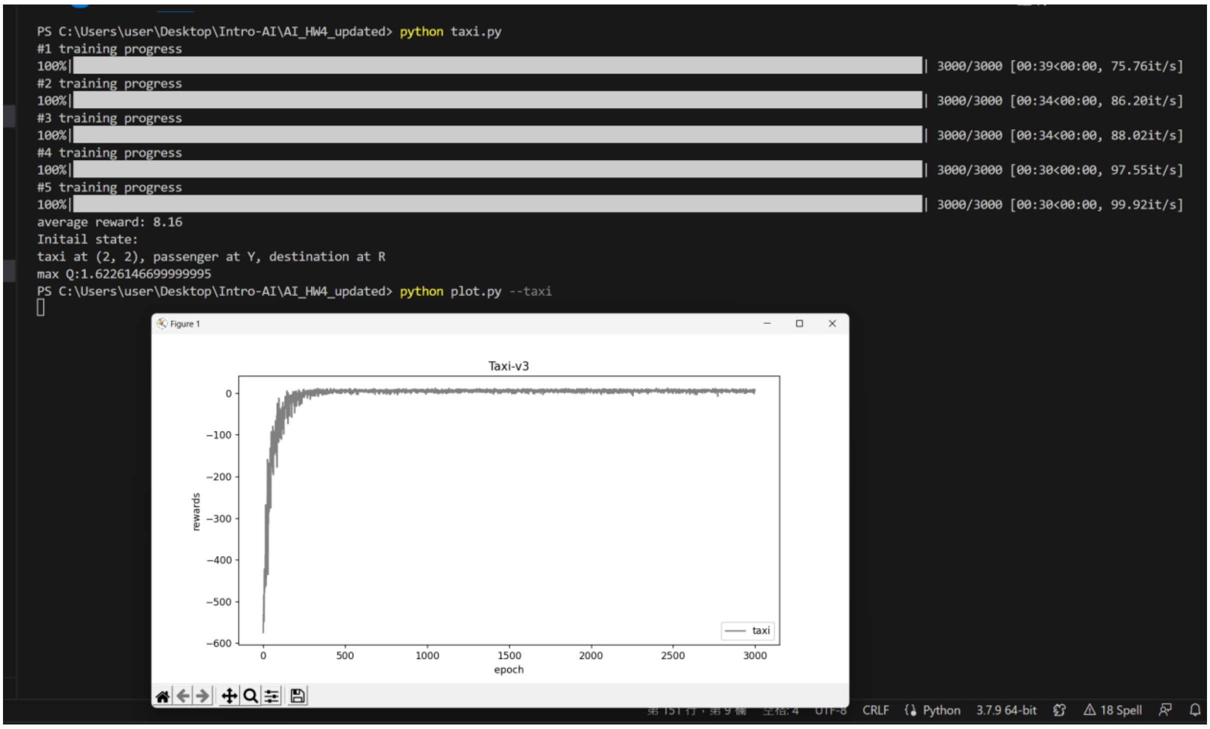


4. compare.png




## 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%)**



$$Q_{opt} = \sum_{i=0}^{8} (-1) \cdot 0.9^{i} + 20 \cdot 0.9^{9} = 1.62261\ldots$$

As we can see, the optimal Q value I calculate is very close to that in Taxi-v3.

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%)**


$$Q_{opt} = \sum_{i=0}^{199} 0.97^{i} = 33.25795\ldots$$

Since in Cartpole-v0 the states are discrete, therefore, the error between code and what I calculate is a little larger than Taxi-v3.

3.

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

The original states should be continuous which produce infinite actions. Thus we cut them into discrete groups to build a table containing limited number of members.

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

Increasing "num_bins" causes more groups in table mentioned above. If an interval we cut is thinner, the estimation is more closer to the real value. Thus, we have better result if getting higher "num_bins".

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

Increasing "num_bins" indicates we get more groups to calculate, which accumulates tasks for computer that increases space complexity, and also takes much more time to execute.

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

I think DQN will perform better. First, discretized Q learning utilizes discrete data that causes error between estimations and real values. Second, discretized Q learning executes based on Q table that is confined with the number of groups we provide. However, DQN constructs a deep network that can calculate estimations with much more states than Q learning. Thus, DQN handles more cases than Q learning, performing better.

5.

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

We need to find a balanced ratio of exploitation and exploration and produce higher value of reward.

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

It will exploit all the time, and won't find a new better action as exploration. In this case, we may obtain a low reward.

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, epsilon greedy algorithm is aimed to enhance the possibility of randomly choosing other states. It’ s still possible to form a totally same combination as a combination from all exploration or all exploitation without algorithm.

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

During the testing section, our agent has been alreadyfamiliar with the environment, which we finish optimizing Q table. Thus, agent can always choose the best actions.

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

There is a parameter “requires_grad” in tensor causing automatical update of the net. But we only need the action, which we set the parameter to be false, that is “no_grad,” to avoid calculation of gradient.