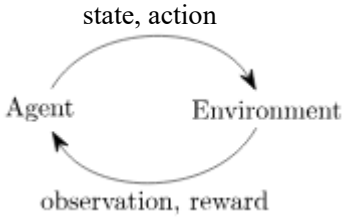
Deep Q Learning for cart-pole

1. Introduction：

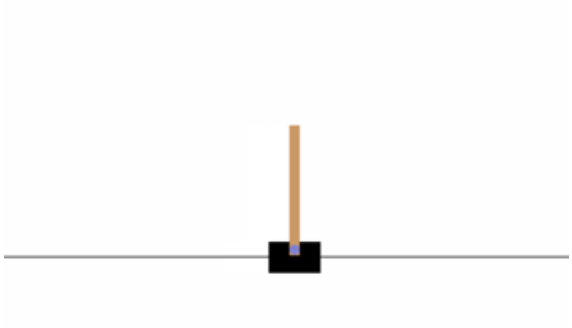


Agent-Environment loop

This is a basic process of deep learning, and computer calculates it again and again to learning policy which has optimal result. At each point in time, the agent (which can be thought of as the algorithm you wrote) chooses an action, and the environment returns the observation (Observation) and reward (Reward) of the last action.

2. Environment：

We made use of OpenAI Gym to provide interfaces for the environments of cart-pole and we do not need to know too much about the internal implementation of the game, which can be used for testing and simulation by simply calling.



Parameter range of cart-pole:

Feature	Value range	
Position:	-2.4	2.4
Velocity:	-inf	inf
Angle:	-41.8	41.8
Angular Velocity:	-inf	inf
Action:	left	right

```
env = gym.make('CartPole-v0')
#setting environment, import cart-pole model from gym
observation = env.reset() #initialize state
env.render() #rebuild environment
observation_new, reward, done, info = env.step(action)
x, x_dot, theta, theta_dot = observation_new
```

- The gym.make() function can build up cart-pole environment
- The env.reset() function needs to be executed to return the initial observation information at the beginning of each episode.
- The env.render() would rebuild environment in each training.
- The env.step() can get key parameters of cart-pole.

Key parameters from environment are Observation, Reward, Done, Info.

Observation: After the current step is executed, the observation of the environment. (position of the cart on the track, cart velocity, angle of the pole with the vertical, angular velocity)

Reward: After executing the previous action, the reward (floating point type) obtained by the agent (agent type) varies in different environments, but the goal of reinforcement learning is to maximize the total reward value

Done: Indicates whether the environment needs to be reset. In most cases, when Done is True, it means that the current episode or experiment) is over.

Info: Diagnostic information for the debugging process.

3. Agent:

a) Build neural network with tensor flow:

There are two networks, which are evaluate network and target network. These two networks have same structure but different function. Evaluate network was used for saving current input and corresponding value, and target network was used for saving or updating evaluate network that has been trained in each trail.

Network structure:

input date (s): [batch_size, n_features] = [32, 4]

First layer:

w1: [n_feature, n_l1] = [4, 10]

b1: [1, n_l1] = [1, 10]

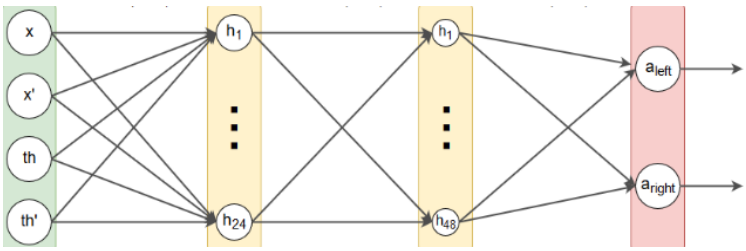
Second layer:

```
l1 = tf.nn.relu(tf.matmul(self.s, w1) + b1)
```

w2: [n_l1, n_actions] = [10, 2]

b2: [1, n_actions] = [1, 2]

```
self.q_eval = tf.matmul(l1, w2) + b2
```



b) Build memory:

The size of the memory is `self.memory = np.zeros((self.memory_size, n_features * 2 + 2))`, which is [500,10]. Store 500 records, 10 elements in each record. Each element is: 4 parameters of current states, one action performed in this state, one reward, and 4 parameters of next state.

```
def store_transition(self, s, a, r, s_):
    if not hasattr(self, 'memory_counter'):
        self.memory_counter = 0

    transition = np.hstack((s, [a, r], s_))

    # replace the old memory with new memory
    index = self.memory_counter % self.memory_size
    self.memory[index, :] = transition

    self.memory_counter += 1
```

c) Deep Q-learning algorithm:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Compared with Q-Learning, DQN makes improvements: one is to use neural networks to approximate the behavior value function, one is to use target Q network to update the target, and the other is to use experience replay Experience replay. Thus, a memory is used to store the experienced data, and each time a parameter is updated, a part of the data is extracted from the Memory for update, so as to break the relationship between the data.

DQN learning process in code:

i. Generate 32 random numbers in 500, as the index value for selecting records from the memory.

ii. Use the records obtained from the memory, through eval_net and target_net to get q_next, q_eval.

```
q_next, q_eval = self.sess.run(
    [self.q_next, self.q_eval],
    feed_dict={
        self.s_: batch_memory[:, -self.n_features:],  # fixed params
        self.s: batch_memory[:, :self.n_features],  # newest params
    })
```

iii. Get q_target with updated q_evalute when learning at first. The current state is (s) from the 32 records that has been obtained. And q_next, q_eval can be obtained. Assuming that q_target = q_eval.copy (), which means using historical data that is records in the memory to find Target. As you can see from the Loss function below, Target = r + gamma * maxQ (s ', a', w). In fact, it is to set the value in Target, and the data format of q_target is (batch_size, n_action) = (32, 2). So in the current state, we only need to assign the Q value that performs the action.

Loss function:

$$L(\omega) = E[\underbrace{(r + \gamma \cdot max_{a'} Q(s', a', \omega)}_{Target} - Q(s, a, \omega)^2)]$$

```
q_target = q_eval.copy()
```

```
q_target[batch_index, eval_act_index] = reward + self.gamma * np.max(q_next, axis=1)
```

iv. Get the current status from memory. Implement the action and the reward obtained after performing the action.

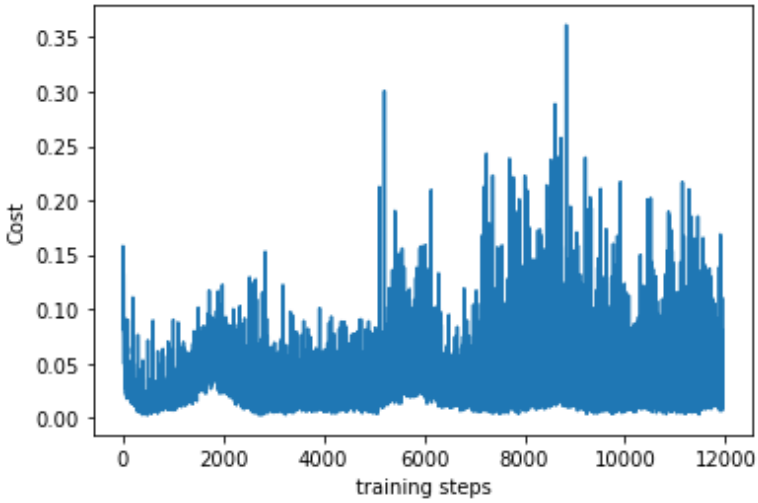
```
batch_index = np.arange(self.batch_size, dtype=np.int32)
eval_act_index = batch_memory[:, self.n_features].astype(int)
reward = batch_memory[:, self.n_features + 1]
```

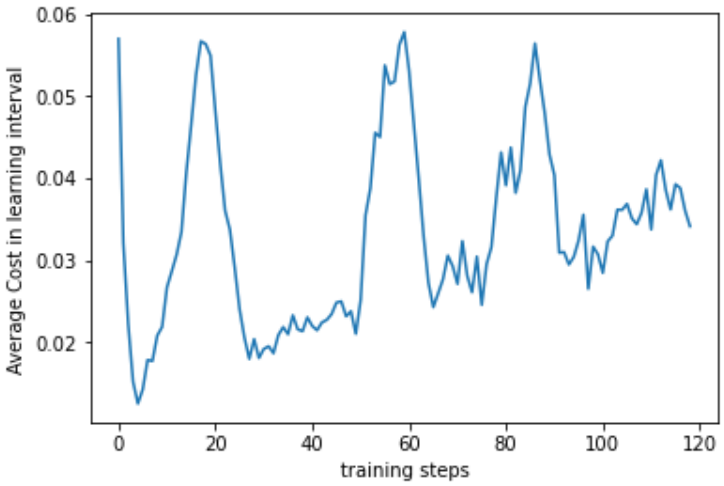
v. Train the evaluate network and record cost. Training updates the parameter values in eval, then updates the parameter values in target net every 300 trainings.

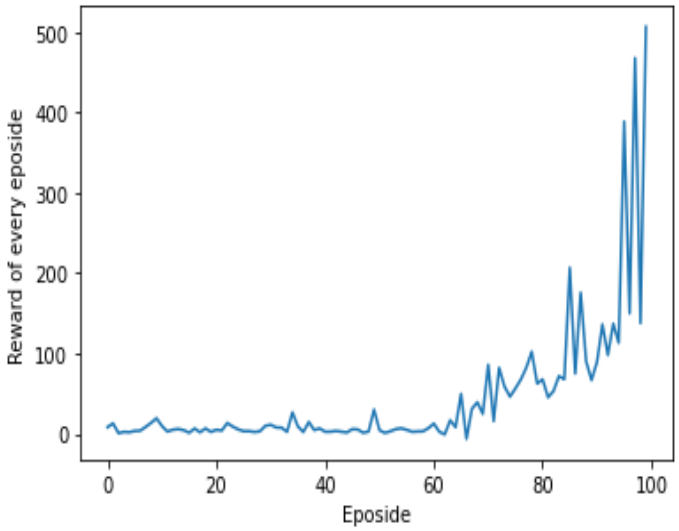
```
_, self.cost = self.sess.run([self._train_op, self.loss],
                             feed_dict={self.s: batch_memory[:, :self.n_features],
                                        self.q_target: q_target})
```

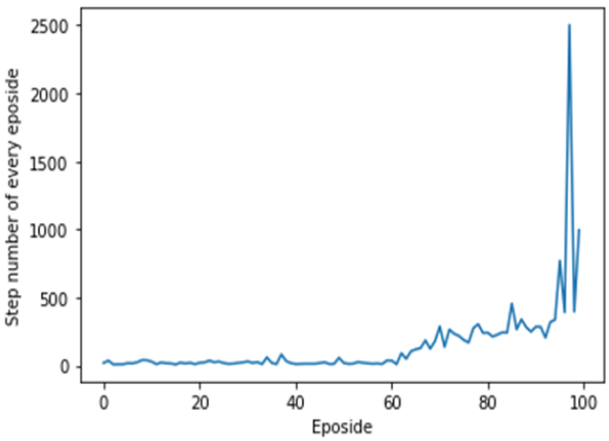
```
self.ch.append(self.cost)
```

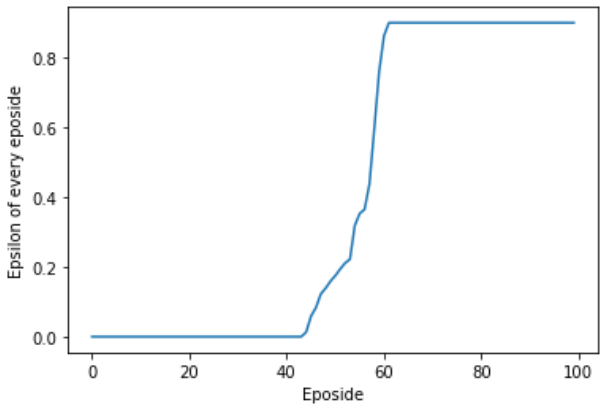
4. Result and conclusion:



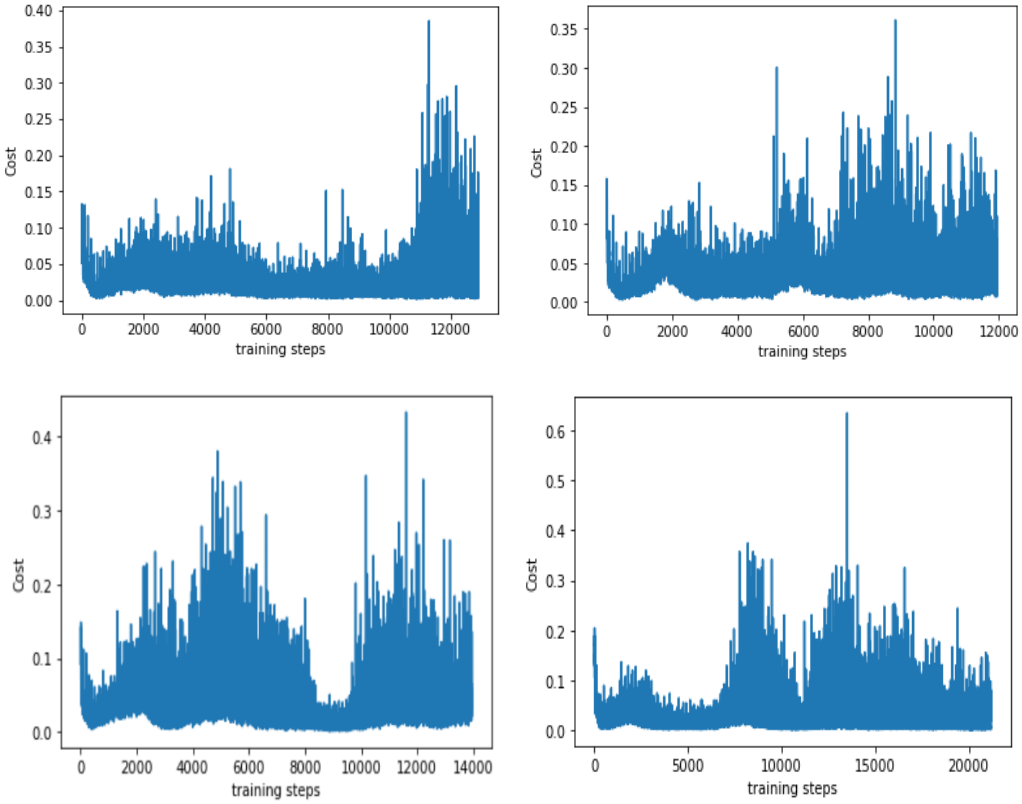








Comparison with extra trials of cost ：



The learning process is also the process of the DQN algorithm. Each action performed (random moving the car to the left or right or keeping it Change), as long as the straight bar does not fall, you will get a reward of size 1. The parameters of the training neural network stop looping until the average reward of the last 100 times is greater than 200 (This algorithm has been define in environment).At this time, the trained agent is considered to have a high level in the game, and it starts to learn. The average reward is greater than 0.02, indicating that the trained agent has a very high level in this game. Network collects experience in each episode and store it in the experience pool. Each episode is trained until the straight rod of the cart falls down (falling to the end state). If the rod of the cart does not fall, it trains to the maximum allowed in an episode Steps.

According to the graphs above, the reward of every episode increases with training, which means the cart-pole behave better gradually, and achieve a learning process because of getting higher points. And the steps of every episode also explained same phenomenon. Because the cart-pole can keep more and more steps in every episode, which is to say it starts to learn and have probability to behave better. But for the cost of every training in every trail, due to the process of a random trying and learning good policy, cost and training step is also unsure and could be different from each playing of cart-pole. Sometimes, it had good learning effects, and it also had low learning effects. But whatever the effects of learning, it always can control the cart-pole good and keep learning to behave better.