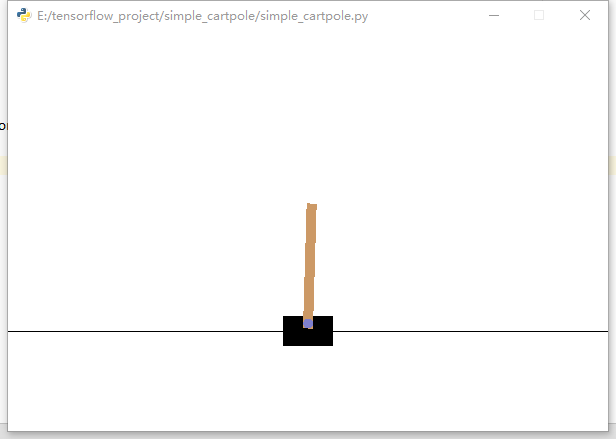
# 基于Q Learning的强化学习

## 原理

在强化学习中，主要有环境enviroment和智能体agent两个元素，智能体根据观察到的环境**状态**采取某种**动作**，环境接受这种动作并反馈**收益**、**下一步状态**及**是否结束**。智能体的决策是基于Q值表，它记录的是(状态,动作)上产生的收益，智能体在某一状态下，选择收益最大的动作作为输出动作。Q值表是通过神经网络学习到的，强化学习的训练过程，就是学习Q值表的过程。

下面结合CartPole游戏说明该技术，该游戏通过对木块左右移动以保持木杆平衡。



### 1.1环境

环境最主要的任务是接受智能体输出的动作，并做出一定的反馈，即下一步状态，该动作获得的收益及是否停止。在CartPole游戏中，返回的状态是一个4元素列表，收益为一个分值。

### 1.2智能体

智能体的主要任务是感知环境，或者说接受环境的反馈信息，作出动作决策。在智能体内部创建一个神经网络称Q值网络，网络的输入为环境状态，在这里是一个四元素的列表，输出是两个动作的Q值，智能体的动作就是基于该网络的输出结果。在智能体感知环境的过程中，智能体收集并保存数据，并每次从中抽取部分数据训练。网络训练的目的，是在给定环境**状态**下，作出一个**动作**，使得该动作的**收益**最大。收益不仅包括**当前动作的收益，**还包括**下一状态的最大收益**，**因为Q值网络是收敛的，所以网络的损失是网络的输出Q值与期待收益（当前收益 + 下一状态最大收益）的均方差**，其中当前收益是看得到的，是从环境反馈得到的，而Q值（即网络估计的收益）和下一状态的最大收益是由网络估计得到的。

## 实现

完整代码在我的github上：https://github.com/hy17003/simple\_cartpole

### 2.1环境

|  |
| --- |
| **class** Environment:  **def** \_\_init\_\_(self, sess):  self.game = gym.make(**'CartPole-v0'**)  self.session = sess   **'''  环境执行一次动作，返回反馈信息  '''  def** step(self,action):  state, reward, done, \_ = self.game.step(action)  **return** state, reward, done   **'''  显示画面  '''  def** show(self):  self.game.render()   **'''  重启  '''  def** restart(self):  **return** self.game.reset() |

### 2.2智能体

|  |
| --- |
| **class** Agent:   **def** \_\_init\_\_(self, sess):  self.buffer = deque()  self.max\_buffer = 10000  self.state\_dim = 4  self.action\_dim = 2  self.epsilon = 0.1  self.batch\_size = 32  self.session = sess  self.create\_q\_network()  self.create\_train\_method()  self.saver = tf.train.Saver()  self.start\_idx = 0   **'''  决策，根据状态计算Q值，选择最大Q值的动作输出，同时，该动作有一定的随机性  '''  def** get\_greedy\_action(self, state):  q\_value = self.session.run(self.q\_value, feed\_dict={self.state\_input: [state]})  **if** random.random() <= self.epsilon:  **return** random.randint(0, self.action\_dim - 1)  **else**:  **return** np.argmax(q\_value[0])   **'''  决策，根据状态计算Q值，选择最大Q值的动作输出  '''  def** get\_action(self, state):  q\_value = self.session.run(self.q\_value, feed\_dict={self.state\_input: [state]})  **return** np.argmax(q\_value[0])   **'''  感知环境，保存游戏返回的状态及奖励等信息  '''  def** perceive(self, state, action, reward, next\_state, done):  one\_hot\_action = np.zeros(self.action\_dim)  one\_hot\_action[action] = 1  self.buffer.append((state, one\_hot\_action, reward, next\_state, done))  **if** len(self.buffer) > self.max\_buffer:  self.buffer.popleft()  **if** len(self.buffer) > self.batch\_size:  self.train\_q\_network()   **'''  创建Q值网络，输入为当前状态state\_input, 网络输出两个动作的Q值  '''  def** create\_q\_network(self):  self.state\_input = tf.placeholder(**"float"**, [**None**, self.state\_dim])  W1 = self.weight\_variable([self.state\_dim, 20])  b1 = self.bias\_variable([20])  W2 = self.weight\_variable([20, self.action\_dim])  b2 = self.bias\_variable([self.action\_dim])  h\_layer = tf.nn.relu(tf.matmul(self.state\_input, W1) + b1)  self.q\_value = tf.matmul(h\_layer, W2) + b2   **'''  创建训练方法，输入为动作，期望奖励及当前状态，Q值网络根据当前状态估计两个动作Q值，获取当前动作对应的Q值，该Q值与输入期望奖励差的平方即为cost，因为在网络稳定时，因为网络稳定后，Q值表中，每一个(状态，动作)上的值将不变  '''  def** create\_train\_method(self):  self.action\_input = tf.placeholder(**"float"**, [**None**, self.action\_dim])  self.y\_input = tf.placeholder(**"float"**, [**None**, ])  q\_action = tf.reduce\_sum(tf.multiply(self.q\_value, self.action\_input), reduction\_indices=1)  self.cost = tf.reduce\_mean(tf.square(self.y\_input - q\_action))  self.train\_step = tf.train.AdamOptimizer(0.0001).minimize(self.cost)   **'''  训练Q值网络  '''  def** train\_q\_network(self):  minibatch = random.sample(self.buffer, self.batch\_size)  state\_batch = [data[0] **for** data **in** minibatch]  action\_batch = [data[1] **for** data **in** minibatch]  reward\_batch = [data[2] **for** data **in** minibatch]  next\_state\_batch = [data[3] **for** data **in** minibatch]   y\_batch = []  **for** i **in** range(self.batch\_size):  done = minibatch[i][4]  **if** done:  y\_batch.append(reward\_batch[i])  **else**:  reward\_ex = self.session.run(self.q\_value, feed\_dict={self.state\_input: [next\_state\_batch[i]]})  y\_batch.append(reward\_batch[i] + np.max(reward\_ex))  self.session.run(self.train\_step, feed\_dict = {self.state\_input: state\_batch, self.y\_input:y\_batch,  self.action\_input:action\_batch})   **'''  初始化权值  '''  def** weight\_variable(self, shape):  initial = tf.truncated\_normal(shape)  **return** tf.Variable(initial)  **'''  初始化偏置值  '''  def** bias\_variable(self, shape):  initial = tf.constant(0.01, shape=shape)  **return** tf.Variable(initial)  **'''  保存网络模型  '''  def** save\_model(self, path, step):  model\_name = os.path.join(path, **'cartpole.ckpt'**)  self.saver.save(self.session, model\_name, global\_step=step)  **return** 0   **'''  载入网络模型  '''  def** load\_model(self, path):  ckpt = tf.train.get\_checkpoint\_state(checkpoint\_dir=path)  **if** ckpt **and** ckpt.model\_checkpoint\_path:  self.saver.restore(self.session, ckpt.model\_checkpoint\_path)  self.start\_idx=ckpt.model\_checkpoint\_path.split(**'/'**)[-1].split(**'-'**)[-1] |

### 2.3主函数

|  |
| --- |
| MAX\_TIMES = 100000 MAX\_STEP = 300 **def** main(stage):  sess = tf.Session()  env = Environment(sess)  agent = Agent(sess)  sess.run(tf.global\_variables\_initializer())  agent.load\_model(**'model'**)  **if**(stage == **'TRAIN'**):  **for** t **in** range(int(agent.start\_idx), MAX\_TIMES):  cur\_state = env.restart()  score = 0  **for** step **in** range(MAX\_STEP):  *#根据当前状态，智能体作出一种动作* action = agent.get\_greedy\_action(cur\_state)  *#执行这种动作，获得环境反馈* next\_state, reward, done = env.step(action)  score = score + reward  *# 显示画面* **if**(t > 3000):  env.show()  *#智能体感知这些反馈* agent.perceive(cur\_state, action, reward, next\_state, done)  *#更新环境状态* cur\_state = next\_state  **if** done:  **break**;  print(**"t = "**, t, **" Score = "**, score)  **if** t % 2000 == 0:  agent.save\_model(**'model'**, t)  **else**:  **for** t **in** range(100):  cur\_state = env.restart()  score = 0  **for** step **in** range(MAX\_STEP):  *# 根据当前状态，智能体作出一种动作* action = agent.get\_action(cur\_state)  *# 执行这种动作，获得环境反馈* next\_state, reward, done = env.step(action)  score = score + reward  *# 显示画面* env.show()  *# 智能体感知这些反馈* agent.perceive(cur\_state, action, reward, next\_state, done)  *# 更新环境状态* cur\_state = next\_state  **if** done:  **break**;  print(**"t = "**, t, **" Score = "**, score)  sess.close() |

## 其它细节

### 3.1 网络的保存与恢复

保存训练模型，首先定义Saver，在定义完网络后使用 tf.train.Saver()定义一个Saver，在需要保存模型的地方使用以下代码保存模型。

|  |
| --- |
| self.saver.save(self.session, model\_name, global\_step=step) |

载入已训练模型，以下代码检查模型是否存在，若存在，则载入：

|  |
| --- |
| ckpt = tf.train.get\_checkpoint\_state(checkpoint\_dir=path) **if** ckpt **and** ckpt.model\_checkpoint\_path:  self.saver.restore(self.session, ckpt.model\_checkpoint\_path) |

以下代码从模型文件名中提取训练次数：

|  |
| --- |
| self.start\_idx=ckpt.model\_checkpoint\_path.split(**'/'**)[-1].split(**'-'**)[-1] |