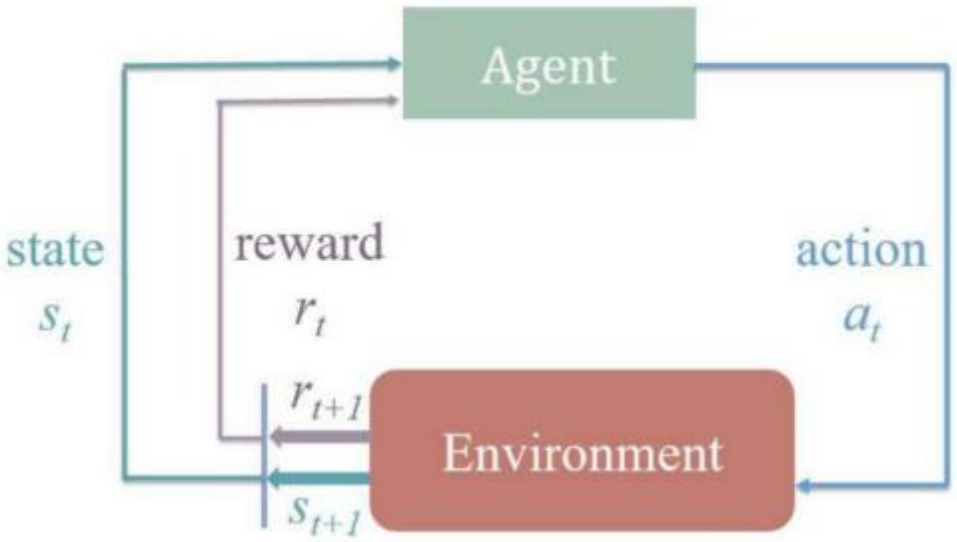
莫烦 强化学习课程代码笔记



本文档为第一版，后续还会修改，修改后的文档会放入 Github 中

本文档是针对莫烦老师深度学习课程(Reinforcement Learning)视频做的笔记

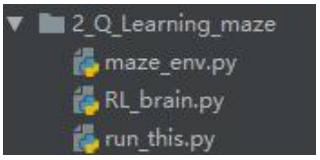
编者：何志强
1603868203@qq.com
https://github.com/18279406017

2018.10.09

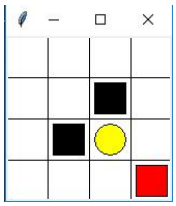
目录

1.Q-Learning

首先放上相关文件代码链接 https://github.com/18279406017/Reinforcement-learning-with-tensorflow/tree/master/contents/2_Q_Learning_maze



Maze_env.py 文件编写的是相当于一个游戏环境，大概长下图这个样子：



Observation：Observation 打印出来也是四个维度的。

Action：Action 的维度是四维的，代表上下左右等等。

Reward：黄色的是天堂（reward 1），黑色的地狱（reward -1），其他的(Reward 0)。

RL_brain.py

在这个文件里面编写的就是 Q-learning 的核心算法。

```
class Qlearningtable:
    def __init__(self, actions, learning_rate = 0.01, r
        self.action = actions
        self.lr = learning_rate
        self.gamma = reward_decay
        self.epsion = e_greedy
        self.q_table = pd.DataFrame(columns=self.action

    def choose_action(self, observation):...
    def learn(self, s, a, r, s_):...
    def check_state_exist(self, state):...
```

里面主要包含的就是两个函数：choose_action 函数接收当前观测值，返回一个动作；Learn 函数接收四个值：当前状态，依据当前状态选择的动作，选择动作加入到环境中后得到的奖励，选择动作后环境切换到的下一个状态。除此之外还有一个函数 check_state_exist，用于检测 Q-learning 的表格中是否含有当前状态。在编写函数之前，我们需要先定义一些参数：

```
def __init__(self, actions, learning_rate = 0.01, reward_decay = 0.9, e_greedy = 0.9 ):
    self.action = actions
    self.lr = learning_rate
    self.gamma = reward_decay
    self.epsion = e_greedy
    self.q_table = pd.DataFrame(columns=self.action, dtype = np.float64)
```

```
1.  self.action = actions  # 代表有多少个 action，方便之后建立 Q-table。
2.  self.lr = learning_rate  # 代表学习率。
3.  self.gamma = reward_decay  # 代表折扣因子。
4.  self.epsion = e_greedy  # 代表贪婪度。
5.  self.q_table = pd.DataFrame(columns=self.action, dtype = np.float64) # 创建 Q 表。
```

Columns：代表表的行标签。行标签代表的是 state 的名称，这里没有写。

我们将 q_table 打印出来的话长下图所示的样子：

	0	1	2	3
[5.0, 5.0, 35.0, 35.0]	0.0	0.00	0.00	0.0
[45.0, 5.0, 75.0, 35.0]	0.0	0.00	0.00	0.0
[5.0, 45.0, 35.0, 75.0]	0.0	0.00	0.00	0.0
[5.0, 85.0, 35.0, 115.0]	0.0	0.00	-0.01	0.0

0，1，2，3：代表的就是动作的标签。

前面的列表：代表的就是状态的值。

接下来我们详细解析一下这里面的几个代码函数：

Check_state_exist:

```
def check_state_exist(self, state):
    if state not in self.q_table.index:
        self.q_table = self.q_table.append(
            pd.Series(
                [0]*len(self.action),
                index=self.q_table.columns,
                name = state
            )
        )
```

这个函数传入一个 state，如果这个 state 不在 q_table 中的话，我们就需要将其加入到 q_table 中去。要完成这个功能的话，我们首先就需要去查看我们原来的 q_table 中是否含有这个 state，也就是需要去查看行标签中是否含有这样一个 state（每一个 state 都有所有动作的值）。我们使用 q_table.index 去查看所有的行的名称，如果行的名称里面没有当前 state 的话，我们就将其加入到 q_table 中去。我们创建的当前 state 的初始值为 0，传入行标签，再用 append 函数将其加入到 q_table 中去。

Choose_action:

```
def choose_action(self, observation):
    self.check_state_exist(observation)
    if np.random.uniform() <self.epsion:
        state_action = self.q_table.loc[observation, :]
        state_action = state_action.reindex(np.random.permutation(state_action.index))
        action = state_action.idxmax()
    else:
        action = np.random.choice(self.action)
    return action
```

在 choose_action 函数中，我们传入的是一个 state，也是 observation，我们需要返回的是一个 action。拿到一个 state 后我们首先就要去检查一下 q_table 中是否含有当前的 state。之后我们需要依据贪婪度来选择动作，np.random.uniform 函数的意思是从 0-1 之间随机采样。Q_table.loc 是依据行名查看这一行的数据值。随机排列一个序列，返回一个排列的序列(同一个 state，可能会有多个相同的 Q action value，所以我们乱序一下)。reindex 使它符合新的索引。Idxmax 返回的是最大的值对应的列标签，也就是我们的动作。

如果随机值不在贪婪度里面的话，那么我们就随机选择动作。

Learn:

```
def learn(self, s, a, r, s_):
    self.check_state_exist(s_)
    q_predict = self.q_table.loc[s, a]
    if s_ != 'terminal':
        q_target = r + self.gamma*self.q_table.loc[s_,:].max()
    else:
        q_target = r
    self.q_table.loc[s, a] += self.lr*(q_target - q_predict)
```

Learn 函数的作用就是我们来依据算法更新 q_table 里面的数值(因为我们的动作是依据 q_table 选择出来的，有效的更新算法能够帮助我们更好地玩好这个方格游戏）。输入是四个值：这个时刻的 state s，依据这个时刻的 state 选择的动作 a，在这个状态下选择这个动作获取到的奖励 r，在这个状态 s 下，采取动作 a 之后，状态更新到下一个状态 s_。我们首先依据已知参数[s, a], 就可以得到 q_table 中对应的预测值。我们将其称为 q_predict。如果下个时刻会使得游戏终止(Terminal)的话，我们的目标奖励 q_target 就是当前依据状态 s，采取动作 a 得到的奖励 r， 如果没有终止的话，那么的目标奖励 q_target 就是当前奖励 r，加上折扣因子 gamma 乘以下一个时刻所能得到的最大奖励。最终地话我们更新我们当前 state s，采取动作 a 后所能得到的奖励，也就是更新 q_table。到现在为止的话我们可以通读一下 q_learning 算法的更新公式：

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

Run_this.py

这个文件下面是强化学习的大体思想。

```
def update():
    for episode in range(100):
        observation = env.reset()
        while True:
            env.render()
            action = RL.choose_action(str(observation))
            observation_, reward, done = env.step(action)
            RL.learn(str(observation), action, reward, str(observation_))
            observation = observation_
            if done:
                break
    print('game over')
    env.destroy()
```

首先初始化一个状态 observation；之后依据这个 observation 选择一个动作；将动作加入到环境中后，得到奖励以及环境变化后的下一个状态。这样依次迭代循环，就可以得到 Q-learning 算法的伪代码。

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal

2.Sarsa

首先放上相关文件代码链接：https://github.com/18279406017/Reinforcement-learning-with-tensorflow/tree/master/contents/3_Sarsa_maze

Sarsa 的决策部分和 Q learning 一模一样，因为我们使用的是 Q 表的形式决策，所以我们会在 Q 表中挑选值较大的动作值施加在环境中来换取奖惩. 但是不同的地方在于 Sarsa 的更新方式是不一样的.

具体的不同表现在下面两个方面：

```
def learn(self, s, a, r, s_, a_):
    self.check_state_exist(s_)
    q_predict = self.q_table.loc[s, a]
    if s_ != 'terminal':
        q_target = r + self.gamma * self.q_table.loc[s_, a_]
    else:
        q_target = r
    self.q_table.loc[s, a] += self.lr * (q_target - q_predict)
```

从上图可以看出来，Sarsa 在更新的时候对下一个状态所能得到的奖励是实际的真的能得到的奖励，而在 Q-learning 是选择的最大值，但是实际的时候有可能并不能选择到最大值（存在贪婪度的一个问题）。由于需要得到下一个状态采取的动作，所以我们需要在 Run_this.py 做相应的改变。

```
def update():
    for episode in range(100):
        observation = env.reset()
        action = RL.choose_action(str(observation))
        while True:
            env.render()
            observation_, reward, done = env.step(action)
            action_ = RL.choose_action(str(observation_))
            RL.learn(str(observation), action, reward, str(observation_), action_)
            # swap observation and action
            observation = observation_
            action = action_
            # break while loop when end of this episode
            if done:
                break
    # end of game
    print('game over')
    env.destroy()
```

在上图中 sarsa 相对 q-learning 做的改变就是将下一个 state: observation_，依据 observation_选择出对应的动作，再将其放入 sarsa 算法中对 q_table 进行更新。

3.Sarsa lambda

首先放上相关文件代码链接：https://github.com/18279406017/Reinforcement-learning-with-tensorflow/tree/master/contents/4_Sarsa_lambda_maze

除了与 sarsa 一样之外，sarsa lambda 还多了一张与 q_table 一模一样的表，这里把它叫做 self.eligibility_trace。如下图所示：

```
def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9, trace_decay=0.9):
    super(SarsaLambdaTable, self).__init__(actions, learning_rate, reward_decay, e_greedy)

    # backward view, eligibility trace.
    self.lambda_ = trace_decay
    self.eligibility_trace = self.q_table.copy()
```

由于我们现在有两张表了，所以我们需要稍微改变一下 check_state_exist 函数：

```
def check_state_exist(self, state):
    if state not in self.q_table.index:
        # append new state to q table
        to_be_append = pd.Series(
                [0] * len(self.actions),
                index=self.q_table.columns,
                name=state,
            )
        self.q_table = self.q_table.append(to_be_append)
        # also update eligibility trace
        self.eligibility_trace = self.eligibility_trace.append(to_be_append)
```

主要改动的地方就是在加入新的状态到 q_table 中的时候，也需要将其加入到 self.eligibility_trace 里面。之后我们需要改的就是 sarsa 算法的更新部分。

```
def learn(self, s, a, r, s_, a_):
    self.check_state_exist(s_)
    q_predict = self.q_table.loc[s, a]
    if s_ != 'terminal':
        q_target = r + self.gamma * self.q_table.loc[s_, a_]
    else:
        q_target = r  # next state is terminal
    error = q_target - q_predict
    # increase trace amount for visited state-action pair
    # Method 1:
    # self.eligibility_trace.loc[s, a] += 1
    # Method 2:
    self.eligibility_trace.loc[s, :] *= 0
    self.eligibility_trace.loc[s, a] = 1
    # Q update
    self.q_table += self.lr * error * self.eligibility_trace
    # decay eligibility trace after update
    self.eligibility_trace *= self.gamma*self.lambda_
```

Method 1 的意思就是：每次经历过[s, a]这个状态的话，就将 self.eligibility 表格中对应[s, a]的这个值加 1，并且没有上限。

Method 2 的意思就是：每次经历过[s, a]这个状态的话，就先将行索引中的当前状态 s 的值全部清 0。之后再将 self.eligibility_trace 表格中对应[s, a]的这个值加 1，上限就被限制为 1。

接下来的话再更新 q_table 的值，sarsa lambda 与 sarsa 的不同之处在于 sarsa

是没有乘 self.eligibility_trace 这个表格，而 sarsa lambda 是乘了这个表格。

4.Deep Q Network

首先放上相关文件代码链接：https://github.com/18279406017/Reinforcement-learning-with-tensorflow/tree/master/contents/5_Deep_Q_Network

在学习下面的相关代码之前呢，需要了解一点点的 tensorflow 编写神经网络的知识，这样的话看起来就会比较方便，如果您还没有学习 tensorflow 的话，也没有关系，我之前解析了 tensorflow 官方文档的一些代码，可以在我的 CSDN，也就下面的链接中找到：https://blog.csdn.net/weixin_39059031/article/category/8076812

和之前一样，我们有三个文件 **Maze_env.py：** 强化学习交互的环境对象，**RL_brain.py ：** 强化学习核心算法部分，**run_this.py ：**强化学习与环境交互迭代部分。

Maze_env.py

这里的环境和之前的一样，也是九宫格的那个，就不说了。

RL_brain.py

在这个文件下面主要是写了强化学习的大脑部分，内部有五个函数:

```
def _build_net(self):...
def store_transition(self, s, a, r, s_):...
def choose_action(self, observation):...
def learn(self):...
def plot_cost(self):...
```

def _bulid_net(): 构建神经网络架构，里面有两个结构一模一样的神经网络，被称为目标网络(Target network)，和评估网络(Eval network)。

def _store_transition(): 这个函数里面主要是用于存储强化学习与环境交互之后得到的数据，也叫做强化学习智能体的记忆。

choose_action():选择动作，依据观测的状态作为输入，然后再将输入传入神经网络中计算应该采取哪一步动作。

Learn(): 强化学习策略的学习过程，其实就是神经网络如何进行迭代更新的部分。

plot_cost():绘制强化学习的 loss 曲线。

在开始之前，我们需要定义先导入一些模块，然后定义一些强化学习需要的参数，如学习率，折扣因子等等这些。在这里我们采用 class 类的编写方式：

```
class DeepQNetwork:
    def __init__(
            self,
            n_actions,
            n_features,
            learning_rate = 0.01,
            reward_decay = 0.9,
            e_greedy  = 0.9,
            replace_target_iter = 300,
            memory_size = 500,
            batch_size = 32,
            e_greedy_increment = None,
            output_graph = False
    ):
        self.n_actions = n_actions
        self.n_features = n_features
        self.lr = learning_rate
        self.gamma = reward_decay
        self.epsilon_max = e_greedy
        self.replace_target_iter = replace_target_iter
        self.memory_size = memory_size
        self.batch_size = batch_size
        self.epsilon_increment = e_greedy_increment
        self.epsilon = 0 if e_greedy_increment is not None else self.epsilon_max
        self.learn_step_counter = 0
```

n_action 代表强化学习智能体的动作维度，n_features 代表强化学习所接收到的状态的维度，也就是你给强化学习看的 state 的维度。Learning_rate 代表学习率，reward_decay 代表折扣因子，e_greedy 代表贪婪度，也可以说是探索度，就是强化学习智能体有多大的概率随机输出动作，以此来表示智能体的探索度。Replace_target_iter 代表多久我们替换一次目标网络，由于这里是含有两个神经网络：目标网络，评估网络，但是在与环境做互动的一直都是评估网络，所以我们目标网络的参数更新就要依靠评估网络了，就是我们隔一段时间将评估网络的参数替换目标网络的参数，这样去更新目标网络。Memory_size 代表我们存储记忆的大小，说白了就是一个多大的数组，来存储一些数据。Batch_size 代表一次取多少数据出来训练。e_greedy_increment 代表是否开启贪婪度吧？Learning_step_counter 代表学习的次数，方便之后知道什么时候用评估网络替换一次目标网络。

之后我们需要定义记忆库的大小、形状：

```
self.memory = np.zeros((self.memory_size, n_features*2 + 2))
```

我们定义记忆库初始时是一个全 0 的矩阵，有 self.memory_size 这么多行，有 n_fearure*2+2 这么多列。至于为什么是这么多列，我们可以算一下，我们首先想一下我们记忆库里面需要存什么东西：[s, a, r, s_]我们一个 s 的维度是 n_feature，所以是 2*n_feature，还有一个动作选取的是啥，一个奖励是多少，因此就是 n_fearure*2+2 这么多列了。

做好这么多准备工作的话，我们就可以来编写强化学习智能体的网络架构：评估网络和目标网络了：

```python
def _build_net(self):
    # evaluate net
    self.s = tf.placeholder(tf.float32, [None, self.n_features], name='s')
    self.q_target = tf.placeholder(tf.float32, [None, self.n_actions], name='Q_target')
    with tf.variable_scope('eval_net'):
        c_names, n_l1, w_initializer, b_initializer = \
            ['eval_net_params', tf.GraphKeys.GLOBAL_VARIABLES], 10, \
            tf.random_normal_initializer(0, 0.3), tf.constant_initializer(0.1)
        with tf.variable_scope('l1'):
            w1 = tf.get_variable('w1', [self.n_features, n_l1], initializer=w_initializer, collections=c_names)
            b1 = tf.get_variable('b1', [1, n_l1], initializer=b_initializer, collections=c_names)
            l1 = tf.nn.relu(tf.matmul(self.s, w1) + b1)
        with tf.variable_scope('l2'):
            w2 = tf.get_variable('w2', [n_l1, self.n_actions], initializer=w_initializer, collections=c_names)
            b2  = tf.get_variable('b2', [1, self.n_actions], initializer=b_initializer, collections=c_names)
            self.q_eval = tf.matmul(l1, w2) + b2
    with tf.variable_scope('loss'):
        self.loss = tf.reduce_mean(tf.squared_difference(self.q_target, self.q_eval))
    with tf.variable_scope('train'):
        self._train_op = tf.train.RMSPropOptimizer(self.lr).minimize(self.loss)
```

我们先来在 **def _bulid_net()**函数里面构建评估网络，我们先来想一下这个网络的功能，我们希望将我们的状态 s 输入进去，得到我们动作的输出 a，由于这个网络是需要进行迭代更新的，更新的目标，也就是我们希望的输出是由我们实际得到的奖励和目标网络对接下来奖励的估计，所以这里的话我们还要预留出一个位置给我们计算得到的目标值（实际得到的奖励和目标网络对接下来奖励的估计）。由此的话我们就需要两个 placeholder 来给他们占个位置。接下来的话我们就开始写神经网络了：

由于我们这里有两个一模一样的网络，因此会有很多变量一样的名称，为了代码写起来方便，我们需要定义一个大的框架名，这样的话就会允许在两个大框架名下面的小框架名可以相同。可以采用 with tf.variable_scope()这样的方式。

tf.variable_scope 可以让变量有相同的命名，包括 tf.get_variable 得到的变量，还有 tf.Variable 的变量

tf.name_scope 可以让变量有相同的命名，只是限于 tf.Variable 的变量

之后的话我们可以先定义神经网络的一下参数：

c_names: 我们评估神经网络的名字叫做“eval_net_params” tf.Graphkey.GLOBAL_VARIABLES 应该是吧变量跟这个名称联系起来(我猜的，嘻嘻)，方便之后通过神经网络的名称来替换目标网络。

n_l1:神经网络的层数，也就是隐藏层第一层，这里定义为 10 个神经元。

w_initializer：权重 w 的初始值为 tf.random_normal_initializer(0, 0.3)

b_initializer: 偏置 b 的初始化值为 tf.constant_initializer(0.1)

之后的话就是将输入传入神经网络与权重和偏置做一些运算，这样的话就可以得到 self.q_eval 了。简而言之就是传入状态 s 到神经网络后，得到输出 q_eval。

接下来的话就是计算 loss，tf.squared_difference(x, y)返回的是(x-y)(x-y); tf.reduce_mean()返回的是这个数组所有元素相加的和再除以元素的个数。这里的 q_target 并不是评估神经网络的直接输出值，这里要注意，评估神经网络的输出值只是组成 q_target 的一部分，还有一部分是真实的奖励值。之后再以 loss 为基础训练整个神经网络。至此评估神经网络就构建完毕了。

目标神经网络：也就是 target 目标神经网络，先看代码吧：

```
# target network
self.s_ = tf.placeholder(tf.float32, [None, self.n_features], name='s_')
with tf.variable_scope('target_net'):
    c_names = ['target_net_params', tf.GraphKeys.GLOBAL_VARIABLES]

    with tf.variable_scope('l1'):
        w1 = tf.get_variable('w1', [self.n_features, n_l1], initializer=w_initializer, collections=c_names)
        b1 = tf.get_variable('b1', [1, n_l1], initializer=b_initializer, collections=c_names)
        l1 = tf.nn.relu(tf.matmul(self.s_, w1) + b1)
    with tf.variable_scope('l2'):
        w2 = tf.get_variable('w2', [n_l1, self.n_actions], initializer=w_initializer, collections=c_names)
        b2  = tf.get_variable('b2', [1, self.n_actions], initializer=b_initializer, collections=c_names)
        self.q_next = tf.matmul(l1, w2) + b2
```

这个神经网络的输入也是一个状态 s，输出也是一个动作的分布，整个神经网络架构与之前的估计网络一模一样，只不过这个状态是下一个状态 s_，这个时候可能会有点懵逼，为啥还要一个神经网络来做相同的事情，这个神经网络做了什么事情呢？

我们再来看看 q-learning 的更新公式：

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

之前我们的估计神经网络输出的是 Q(s, a),也是我们迭代更新的，而目标神经网络的输出是 Q(s', a')，原论文中说这样用两个神经网络可以打破数据之间的关联性，至于数学中到底是怎么样的，我以后再了解了解补充一下吧，嘻嘻嘻嘻。上面这个是 q-learning 的更新公式，DQN 跟这个还有一点点不同，之后代码写到的部分再说一下吧。到这里的话目标神经网络也说完了，它是没有什么梯度下降更新这样的操作的，参数更新全靠估计网络的参数替换。

接下来看一下存储模块，也就是记忆函数怎么写的：

```
def store_transition(self, s, a, r, s_):
    if not hasattr(self, 'memroy_counter'):
        self.memory_counter = 0
    transition = np.hstack((s, [a, r], s_))
    index = self.memory_counter % self.memory_size
    self.memory[index, :] = transition
    self.memory_counter += 1
```

可以看到输入是 s, a, r, s_。其中 s 是当前状态，a 是采取了的动作，r 是动作采取之后获得的奖励，s_是动作采取之后状态转移为下一个的状态。第一行代码的意思就是如果 self 里面没有 memroy_counter 的话，就创建一个 self.memory_counter，并且它的值等于 0。np.hstack()函数的意思是把传入的数据横向铺平，至于里面为啥要用[ ]我也不知道，把它去掉了的话也可以跑通的(我试过了，嘻嘻)，之后的话就是将其存入到数组当中去就好了。

选择动作函数：np.random.randint(0, self.n_action)从 0-self.action 选一个值。

```
def choose_action(self, observation):
    observation = observation[np.newaxis, :]
    if np.random.uniform() < self.epsilon:
        actions_value = self.sess.run(self.q_eval, feed_dict={self.s:observation})
        action = np.argmax(actions_value)
    else:
        action = np.random.randint(0, self.n_actions)
    return action
```

这里要注意的是，这里的动作是离散的，np.argmax()选区的是最大值的索引。

接下来就是最重要的 learn 函数了：

在开始 learn 函数之前，我们需要编写一下目标网络的参数替换节点，就是在开始学习之前，我们先判断一下目标网络的参数是否需要替换，如果达到我们的要求的话，我们就需要去替换一下。下面是替换参数的代码：

```
self._build_net()
t_params = tf.get_collection('target_net_params')
e_params = tf.get_collection('eval_net_params')
self.replace_target_op = [tf.assign(t, e) for t, e in zip(t_params, e_params)]
```

tf.assign(A, new_number): 这个函数的功能主要是把 A 的值变为 new_number

接下来看一下 learn 函数：

```
def learn(self):
    if self.learn_step_counter % self.replace_target_iter == 0:
        self.sess.run(self.replace_target_op)
        print('\ntarget_params_replaced\n')
    if self.memory_counter > self.memory_size:
        sample_index = np.random.choice(self.memory_size, size=self.batch_size)
    else:
        sample_index = np.random.choice(self.memory_counter, size=self.batch_size)
    batch_memory = self.memory[sample_index, :]
```

开始的话我们需要做一些准备工作，比如像是否更新目标网络，从记忆库中取出 batch_size 个记忆来。

```
q_next, q_eval = self.sess.run(
    [self.q_next, self.q_eval],
    feed_dict={
        self.s_: batch_memory[:, -self.n_features:],
        self.s: batch_memory[:, :self.n_features],
    })
q_target = q_eval.copy()
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]
q_target[batch_index, eval_act_index] = reward + self.gamma * np.max(q_next, axis=1)
_, self.cost = self.sess.run([self._train_op, self.loss],
                             feed_dict={self.s: batch_memory[:, :self.n_features],
                                        self.q_target: q_target})
```

我们先将记忆库中的 s_,和 s 分别输入到目标神经网络和估计神经网络当中去，得到 q_next 和 q_eval 这两个矩阵，这里要注意的是这里的 q_next 和 q_eval 都是 self.n_action 个维度的，并不是我们实际取值的那个。

eval_act_index 取出来的就是记忆中 action 那一列的值，行数由实际取的记忆样本个数决定。Reward 是记忆中奖励的那一列。我们之前编写的训练函数是：

```
    self.loss = tf.reduce_mean(tf.squared_difference(self.q_target, self.q_eval))
    self._train_op = tf.train.RMSPropOptimizer(self.lr).minimize(self.loss)
```

我们是通过 q_target,和 q_eval 两者做差来对其进行比较的，所以做法应该是我们先复制一个 q_eval 一样的数组作为 q_target，然后再将里面我们选择了的动作的值依据实际获得奖励来进行更新。eval_act_index 对应的正好就是 0-3 列，因此依据 DQN 的算法就可以对其进行更新训练了。

plot_cost()画 loss 曲线图：

```
def plot_cost(self):
    import matplotlib.pylab as plt
    plt.plot(np.arange(len(self.cost_his)), self.cost_his)
    plt.ylabel('Cost')
    plt.xlabel('training step')
    plt.show()
```

附 DQN 算法伪代码：

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

附 DQN 论文阅读笔记链接：待加

run_this.py

在这个文件下面编写的是算法的迭代更新部分，和之前的迭代更新也差不多：

```
def run_maze():
    step = 0
    for episode in range(300):
        observation = env.reset()
        while True:
            env.render()
            action = RL.choose_action(observation)
            observation_, reward, done = env.step(action)
            RL.store_transition(observation, action, reward, observation_)
            if (step > 200) and (step % 5 == 0):
                RL.learn()
            observation = observation_
            if done:
                break
            step += 1
    print('game over')
    env.destroy()
```

可以看出也就是初始化环境，依据初始的环境选择动作，将动作加入到环境当中去就会得到下一个环境状态，奖励等信息，我们再将这一回合的数据存到记忆中去，如果强化学习智能体玩的步数大于 200 步并且是五的倍数的话，我们就更新学习一下。至此 DQN 到这里就说的差不多了。关于 DQN 的代码莫烦大神最后还有一个写的更加清晰一点的代码。以后再说吧。

5.Double DQN

首先放上相关文件代码链接：https://github.com/18279406017/Reinforcement-learning-with-tensorflow/tree/master/contents/5.1_Double_DQN

在学习下面的相关代码之前呢，需要了解一下 gym，由于之前的智能体交互的环境对象是我们自己写的，但是有很多环境别人都已经写好了，我们只需要将其拿过来用就好了，关于 gym 的简介以及详细安装使用教程可以在我的 CSDN 上面找到，连接如下：https://blog.csdn.net/weixin_39059031/article/details/82152857

RL_brain.py

同样的，在强化学习大脑这个文件里面，除了一些基本的参数之外，我们有四个和之前一样的函数：

```
def _build_net(self):...

def store_transition(self, s, a, r, s_):...

def choose_action(self, observation):...

def learn(self):...
```

功能的话，和之前在 DQN 中讲的是一样的，接下来的话，我们主要说一下其中不一样的部分：

我们知道，之前在 def _build_net()的时候，我们将两个神经网络的架构写了两遍。在这里，采用的是将其封装为一个函数，然后调用两次，效果如下：

```
def build_layers(s, c_names, n_l1, w_initializer, b_initializer):
    with tf.variable_scope('l1'):
        w1 = tf.get_variable('w1', [self.n_features, n_l1], initializer=w_initializer, collections=c_names)
        b1 = tf.get_variable('b1', [1, n_l1], initializer=b_initializer, collections=c_names)
        l1 = tf.nn.relu(tf.matmul(s, w1) + b1)

    with tf.variable_scope('l2'):
        w2 = tf.get_variable('w2', [n_l1, self.n_actions], initializer=w_initializer, collections=c_names)
        b2 = tf.get_variable('b2', [1, self.n_actions], initializer=b_initializer, collections=c_names)
        out = tf.matmul(l1, w2) + b2
    return out
```

可以看到，函数的输入是状态 s，网络的名称，隐藏层第一层的神经元个数，权重初始化值，偏置初始化值，输出的就是神经网络的输出(一个 self.n_action 维度的向量)。之后的话，我们构建评估网络：

```
# ------------------ build evaluate_net ------------------
self.s = tf.placeholder(tf.float32, [None, self.n_features], name='s')  # input
self.q_target = tf.placeholder(tf.float32, [None, self.n_actions], name='Q_target')  # for calculating loss

with tf.variable_scope('eval_net'):
    c_names, n_l1, w_initializer, b_initializer = \
        ['eval_net_params', tf.GraphKeys.GLOBAL_VARIABLES], 20, \
        tf.random_normal_initializer(0., 0.3), tf.constant_initializer(0.1)  # config of layers

    self.q_eval = build_layers(self.s, c_names, n_l1, w_initializer, b_initializer)

with tf.variable_scope('loss'):
    self.loss = tf.reduce_mean(tf.squared_difference(self.q_target, self.q_eval))
with tf.variable_scope('train'):
    self._train_op = tf.train.RMSPropOptimizer(self.lr).minimize(self.loss)
```

这里的代码和之前的一样，没啥区别，主要就是传入状态 s 得到 q_eval，然后和 q_target 做 loss 进行反向传播训练。target_net 就和之前是一样的了，就不说了。store_transition()函数和之前的也一样，用于存储记忆。

接下来的话就是 choose_action 函数了：

```
def choose_action(self, observation):
    observation = observation[np.newaxis, :]
    actions_value = self.sess.run(self.q_eval, feed_dict={self.s: observation})
    action = np.argmax(actions_value)

    if not hasattr(self, 'q'):  # record action value it gets
        self.q = []
        self.running_q = 0
    self.running_q = self.running_q*0.99 + 0.01 * np.max(actions_value)
    self.q.append(self.running_q)

    if np.random.uniform() > self.epsilon:  # choosing action
        action = np.random.randint(0, self.n_actions)
    return action
```

和之前不一样的地方就是我们记录了神经网络输出的最大 q 值。

接下来就是最最重要的核心算法部分，我们先来简单看一下两者算法之间的不同：

DQN：

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-) .$$

Double DQN：

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t') .$$

我们先看 DQN 的算法，目标 q 值就是等于当前拿到的奖励，加上后续可能拿到的奖励，后续的奖励是由目标网络依据下一个状态 s_ 而估计可以拿到的最大奖励，而下一个状态想要拿到奖励肯定得采取动作，这个动作还是依据目标网络的最大奖励选取的动作。

在 Double DQN 中，目标 q 值等于当前拿到的奖励，加上下一个状态所能得到的最大奖励，不同与 DQN 之处在于，我们选取的这个动作不是依据目标网络哪个奖励大就选取哪个动作，而是依据评估网络中哪个奖励大，才选取哪个动作。

```
q_next, q_eval4next = self.sess.run(
    [self.q_next, self.q_eval],
    feed_dict={self.s_: batch_memory[:, -self.n_features:],    # next observation
               self.s: batch_memory[:, -self.n_features:]})    # next observation
q_eval = self.sess.run(self.q_eval, {self.s: batch_memory[:, :self.n_features]})

q_target = q_eval.copy()

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]

if self.double_q:
    max_act4next = np.argmax(q_eval4next, axis=1)        # the action that brings the highes
    selected_q_next = q_next[batch_index, max_act4next]  # Double DQN, select q_next dependi
else:
    selected_q_next = np.max(q_next, axis=1)    # the natural DQN

q_target[batch_index, eval_act_index] = reward + self.gamma * selected_q_next
```

为了实现上述功能，我们需要将下一个状态 s_，传入到目标网络和评估网络中去，传入到目标网络中去是为了计算 DQN 算法中的最大 Q 值，传入到评估网络中去是为了获取在评估网络中最大 q 值对应的 action 是啥，然后再拿这个 action 去目标网络中索引 Q 值。再加上奖励，就构建了两种算法。

run_Pendulum.py

在这个文件夹下面的话主要就是一些强化学习的主循环，基本上都一样：

```
def train(RL):
    total_steps = 0
    observation = env.reset()
    while True:
        env.render()
        # if total_steps - MEMORY_SIZE > 8000: env.render()
        action = RL.choose_action(observation)

        f_action = (action-(ACTION_SPACE-1)/2)/((ACTION_SPACE-1)/4)   # convert to [-2 ~ 2] float a
        observation_, reward, done, info = env.step(np.array([f_action]))
        reward /= 10     # normalize to a range of (-1, 0). r = 0 when get upright
        # the Q target at upright state will be 0, because Q_target = r + gamma * Qmax(s', a') = 0
        # so when Q at this state is greater than 0, the agent overestimates the Q. Please refer to
        RL.store_transition(observation, action, reward, observation_)
        if total_steps > MEMORY_SIZE:   # learning
            RL.learn()
        if total_steps - MEMORY_SIZE > 20000:   # stop game
            break
        observation = observation_
        total_steps += 1
    return RL.q
```

其中有一步是将动作缩放了一下，缩放到-2 到 2 之间，其他的都好理解。

到这里的话，DQN 可能就讲的差不多了，当然除了 Double DQN 外还有一些变种，但是都是局部的一些小改进。如 Prioritized Experience Replay 中引入 TD-error 作为我们从记忆库采样的依据。Dueling DQN 将输出 q 值的神经网络拆分，最后再合并。这些小技巧以后再补充吧。

6.Policy Gradients

首先放上相关文件代码链接：https://github.com/18279406017/Reinforcement-learning-with-tensorflow/tree/master/contents/7_Policy_gradient_softmax

我们知道之前所讲的是基于状态然后给出当前状态下所有动作的值，然后再依据这个值来选择动作，q-learning 的话就不用解释了，就是一张 q 表，动作肯定是离散的，在 DQN 中，神经网络的输入是一个状态，输出的维度是动作的维度，输出的值对应与每个动作的值。神经网络拟合的是值函数。但是基于值的就有一个问题，就是动作空间是离散的，应为连续的话维度就太大了，做不下去。因此，先驱们就提出了基于策略的强化学习方法，直接输出动作的值，依据奖励的大小来决定更新幅度的大小，这样的话就可以实现其更新：

这里由于莫烦老师没有说过多的原理上的理解，后续的话我将给出详细的原理解释，这里还是主要说一下算法的代码实现。

RL_brain.py

```
class PolicyGradient():
    def __init__(
            self,
            n_actions,
            n_features,
            learning_rate = 0.01,
            reward_decay = 0.95,
            output_graph = False,
    ):
        self.n_actions = n_actions
        self.n_feature = n_features
        self.lr = learning_rate
        self.gamma = reward_decay
        self.ep_obs, self.ep_as, self.ep_rs = [], [], []
        self._build_net()
        self.sess = tf.Session()
        if output_graph:
            tf.summary.FileWriter("logs/", self.sess.graph)
        self.sess.run(tf.global_variables_initializer())
```

在开始之前我们首先需要 import 一些模块，这里就不详细说明了。之后就跟大部分强化学习算法的写法一样，我们首先要告诉智能体，我们的动作维度是多少的，我们的特征输入又是多少的，然后定义学习率，折扣因子和决定是否在 tensorflow 中输出神经网络架构的流程图。与上文基于值的方法不同的是，在这里我们定义了三个空的列表：self.ep_obs, self.ep_as, self.ep_rs。这三个空的列表用于存储记忆，分别存储强化学习与环境交互过程中产生的状态，动作，奖励。

```
def _build_net(self):...
def choose_action(self, observation):...
def store_transition(self, s, a, r):...
def learn(self):...
def _discount_and_norm_rewards(self):...
```

其中定义了五个函数，其中的四个都是我们的老朋友了，这第五个是干啥的？这是由于 Policy Gradient 在原始的公式推到的时候，学习算法的更新就是回

合更新的，因此我们需要将这个奖励函数做一些小小的处理。那我们就从这个小小的处理开始吧：

_discount_and_norm_rewards

```
def _discount_and_norm_rewards(self):
    # np.zeros_like()生成一个全零的类似的矩阵
    discounted_ep_rs = np.zeros_like(self.ep_rs)
    running_add = 0
    for t in reversed(range(0, len(self.ep_rs))):
        running_add = running_add * self.gamma + self.ep_rs[t]
        discounted_ep_rs[t] = running_add
    discounted_ep_rs -= np.mean(discounted_ep_rs)
    discounted_ep_rs /= np.std(discounted_ep_rs)
    return discounted_ep_rs
```

在开始处理之前，我们先说一下我们手中的奖励函数是什么样子的，也就是self.ep_rs 中的奖励。self.ep_rs 是一个列表，列表中存储的是与环境交互过程中每次动作选取所产生的奖励。而我们需要的是从当前状态到回合结束所产生的奖励的总和，也就是回合更新的算法。由此我们通过一个 for 循环，利用 reversed 函数，将 for 循环产生的数组颠倒过来，数据从 len(self.ep_rs)依次到 0。然后我们将 self.ep_rs 数组里面的数据从后面往前依次递加。然后将其减去均值，除以标准差。

_build_net

```
def _build_net(self):
    with tf.name_scope('inputs'):
        self.tf_obs = tf.placeholder(tf.float32, [None, self.n_feature], name="observations")
        self.tf_acts = tf.placeholder(tf.int32, [None, ], name="actions_num")
        self.tf_vt = tf.placeholder(tf.float32, [None, ], name="actions_value")
    layer = tf.layers.dense(
        inputs = self.tf_obs,
        units = 10,
        activation = tf.nn.tanh,
        kernel_initializer=tf.random_normal_initializer(mean=0, stddev=0.3),
        bias_initializer=tf.constant_initializer(0.1),
        name="fc1"
    )
    all_act = tf.layers.dense(
        inputs = layer,
        units=self.n_actions,
        activation=None,
        kernel_initializer=tf.random_normal_initializer(mean=0, stddev=0.3),
        bias_initializer=tf.constant_initializer(0.1),
        name="fc2"
    )
    self.all_act_prob = tf.nn.softmax(all_act, name="act_prob")
```

在神经网络中，我们传入的是状态 self.tf_obs，这个是由当前回合所有的状态构成的。输出是一个动作的概率，维度在之前就已经定义好了。

```
with tf.name_scope("loss"):
    # neg_log_prob = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=all_act, labels=self.tf_acts)
    neg_log_prob = tf.reduce_sum(-tf.log(self.all_act_prob)*tf.one_hot(self.tf_acts, self.n_actions), axis=1)
    loss = tf.reduce_mean(neg_log_prob*self.tf_vt)
with tf.name_scope("train"):
    self.train_op = tf.train.AdamOptimizer(self.lr).minimize(loss)
```

之后我们将选中的动作的概率取 log 值，再乘以回合奖励，我们期望这个值越大越好。将其取个负号就是使其越小越好。

```
def choose_action(self, observation):
    prob_weight = self.sess.run(self.all_act_prob, feed_dict={self.tf_obs:observation[np.newaxis, :]})
    action = np.random.choice(range(prob_weight.shape[1]), p=prob_weight.ravel())
    return action
```

range(prob_weight.shape[1])中 prob_weight.shape[1]代表的是神经网络输出的列数，这里是两列，代表动作的维度为 2。加个 range 那么生成的就是 0 或者 1。prob_weight.ravel()是将概率数组给打平。然后以概率 p 选择动作 0，或者 1。这里的动作还是离散的，这是由于智能体交互环境动作的离散导致的，这种方法是可以做连续的。

```
def store_transition(self, s, a, r):
    self.ep_obs.append(s)
    self.ep_as.append(a)
    self.ep_rs.append(r)
def learn(self):
    discounted_ep_rs_norm = self._discount_and_norm_rewards()
    self.sess.run(self.train_op, feed_dict={
        # 垂直地将数组堆叠起来
        self.tf_obs: np.vstack(self.ep_obs),
        self.tf_acts: np.array(self.ep_as),
        self.tf_vt: discounted_ep_rs_norm,
    })
    self.ep_obs, self.ep_as, self.ep_rs = [], [], []
    return discounted_ep_rs_norm
```

之后的存储记忆和学习训练。这里要注意，学习完了之后就需要将记忆清空。

7.Actor Critic

首先放上相关文件代码链接：https://github.com/18279406017/Reinforcement-learning-with-tensorflow/blob/master/contents/8_Actor_Critic_Advantage/AC_CartPole.py

总的 actor-critic 思想就是将 policy gradient 中的奖励驱动算法更新的那部分的奖励用基于值方法的 Q 值，或者 Q 值的一些变种(总的思想方法就是用一个神经网络去预测奖励，从而实现单步更新，而不用回合更新)，替代以往的回合奖励。在开始之前我们除了导入相关的模块之外，我们还需要定义一些超参数。

```
OUTPUT_GRAPH = False
MAX_EPISODE = 3000
DISPLAY_REWARD_THRESHOLD = 200
MAX_EP_STEPS = 1000
RENDER = False
GAMMA = 0.9
LR_A = 0.001
LR_C = 0.01
```

OUTPUT_GRAPH 表示的是：是否在 tensorboard 中输出图。MAX_EPISON 表示的是玩 3000 个回合。DISPLAY_REWARD_THRESHOLD 表示奖励大于 200 之后开始渲染环境。MAX_EP_STEPS 表示的是每个回合内部最多玩 1000 次。开始的时候不渲染环境，折扣因子等于 0.9，actor 的学习率是 0.001，critic 的学习率是 0.01。

```
class Actor(object):
    def __init__(self, sess, n_features, n_actions, lr=0.001):...

    def learn(self, s, a, td):...

    def choose_action(self, s):...


class Critic(object):
    def __init__(self, sess, n_features, lr=0.01):...

    def learn(self, s, r, s_):...
```

从上图中可以知道 actor-critic 算法中核心的东西就是一个 actor 类和一个 critic 类。在 actor 类中，我们需要定义一个神经网络，输入是状态，输出是动作的概率；然后我们需要定义一个函数去更新算法；还要一个函数定义如何依据输出的动作概率值来选择动作。在 critic 类中，我们同样需要定义一个神经网络，输入是状态值，输出是这个状态的价值估计，并以此状态估计来估计对未来的奖励值。

```
def __init__(self, sess, n_features, n_actions, lr=0.001):
    self.sess = sess

    self.s = tf.placeholder(tf.float32, [1, n_features], "state")
    self.a = tf.placeholder(tf.int32, None, "act")
    self.td_error = tf.placeholder(tf.float32, None, "td_error")  # TD_erro

    with tf.variable_scope('Actor'):...

    with tf.variable_scope('exp_v'):
        log_prob = tf.log(self.acts_prob[0, self.a])
        self.exp_v = tf.reduce_mean(log_prob * self.td_error)  # advantage

    with tf.variable_scope('train'):
        self.train_op = tf.train.AdamOptimizer(lr).minimize(-self.exp_v)
```

我们的”Actor”中输入的是当前状态 s，输出是一个有定义好的维度的动作分布概率。如果我们想训练的话，我们需要定义一下变种的 loss，也就是“exp_v”，是对我们选取的那个动作取 log 之后乘以 critic 的输出的能代表未来奖励的值：TD-error。然后最大化那个变种的 loss，就是将其加个负号就可以了。

```
def learn(self, s, a, td):
    s = s[np.newaxis, :]
    feed_dict = {self.s: s, self.a: a, self.td_error: td}
    _, exp_v = self.sess.run([self.train_op, self.exp_v], feed_dict)
    return exp_v
```

在 actor 学习函数里面，我们主要是将 actor 的神经网络训练通过给他数据，让它跑一遍。

```
def choose_action(self, s):
    s = s[np.newaxis, :]
    probs = self.sess.run(self.acts_prob, {self.s: s})   # get probabilit
    return np.random.choice(np.arange(probs.shape[1]), p=probs.ravel())
```

在选择动作的函数里面，主要是依据输入状态之后得到的 actor 神经网络输出概率进行动作选择。Actor 网络到这里就说的差不多了，主要就是输入状态得到输出动作概率；还有一个就是自身的训练更新，除了将自己选择的那个动作的概率值取 log 之外，还要 critic 来提供 TD-error 来对其进行更新。

```
def __init__(self, sess, n_features, lr=0.01):
    self.sess = sess

    self.s = tf.placeholder(tf.float32, [1, n_features], "state")
    self.v_ = tf.placeholder(tf.float32, [1, 1], "v_next")
    self.r = tf.placeholder(tf.float32, None, 'r')

    with tf.variable_scope('Critic'):...

    with tf.variable_scope('squared_TD_error'):
        self.td_error = self.r + GAMMA * self.v_ - self.v
        self.loss = tf.square(self.td_error)    # TD_error = (r+gamma*V_n
    with tf.variable_scope('train'):
        self.train_op = tf.train.AdamOptimizer(lr).minimize(self.loss)
```

在 critic 网络当中，我们输入到神经网络 Critic 里面的是当前状态 s，期望输出是当前状态的价值：q 值(可以想象成能获得奖励的优势)。整个神经网络需要更新，因此，我们需要将之前我们基于值的方法中的更新算法拿过来对其进行更新。这里的 self.v_我们在之后会通过 tensorflow 中的 placeholder 的方法传入，它代表的是下一个状态的价值：q 值。我们通过计算奖励：r，神经网络估计出的下一个状态的值：v_，神经网络估计当前状态的值：v，来构建 TD-error。然后构建 loss 对神经网络进行训练。

```
def learn(self, s, r, s_):
    s, s_ = s[np.newaxis, :], s_[np.newaxis, :]

    v_ = self.sess.run(self.v, {self.s: s_})
    td_error, _ = self.sess.run([self.td_error, self.train_op],
                                {self.s: s, self.v_: v_, self.r: r})
    return td_error
```

我们对其学习更新的时候先将下一个状态的估计值计算出来，然后运行之前写好的更新图，就可以得到 TD-error，这样就可以放入 actor 中，对 actor 进行更新。之后的主循环和之前的算法都差不多，这里就不过多阐述了。

8.DDPG

首先放上相关文件代码链接：https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/blob/master/contents/9_Deep_Deterministic_Policy_Gradient_DDPG/DDPG_update2.py

```
MAX_EPISODES = 200
MAX_EP_STEPS = 200
LR_A = 0.001    # learning rate for actor
LR_C = 0.002    # learning rate for critic
GAMMA = 0.9     # reward discount
TAU = 0.01      # soft replacement
MEMORY_CAPACITY = 10000
BATCH_SIZE = 32
RENDER = False
ENV_NAME = 'Pendulum-v0'
```

在开始之前，我们同样需要定义一些超参数，像最大循环步数MAX_EPISODES为200步,每次大循环里面最大迭代200次(MAX_EP_STEPS)，actor的学习率为0.001，critic的学习率为0.002，折扣因子GAMMA为0.9，TAU为指数加权平均的参数，后面会介绍到。记忆库的最大容量MEMORY_CAPACITY为10000，batch_size为32，渲染环境参数RENDER为False，环境交互对象为Pendulum-v0。

定义好这些参数之后，接下来的话就是DDPG的主算法了：

```
class DDPG(object):
    def __init__(self, a_dim, s_dim, a_bound):...

    def choose_action(self, s):...

    def learn(self):...

    def store_transition(self, s, a, r, s_):...

    def _build_a(self, s, reuse = None, custom_getter = None):...

    def _build_c(self, s, a, reuse = None, custom_getter = None):...
```

也与基本的强化学习算法一样，有神经网络构建模块，选择动作模块，学习模块，存储记忆模块。我们先来看一下构建神经网络模块的函数：

```
def _build_a(self, s, reuse = None, custom_getter = None):
    trainable = True if reuse is None else False
    with tf.variable_scope("Actor", reuse=reuse, custom_getter=custom_getter):
        net = tf.layers.dense(s, 30, activation=tf.nn.relu, name="l1", trainable=trainable)
        a = tf.layers.dense(net, self.a_dim, activation=tf.nn.tanh, name="a", trainable=trainable)
        return tf.multiply(a, self.a_bound, name="scaled_a")
```

Actor网络输入是一个状态s，输出是动作值，其维度是开始定义的动作维度。这里需要注意的是神经网络的最后一层输出是通过了激活函数的，幅值是被限制掉了的。因此我们乘以动作边界后就能够将动作放大。

```
def _build_c(self, s, a, reuse = None, custom_getter = None):
    trainable = True if reuse is None else False
    with tf.variable_scope("Critic", reuse = reuse, custom_getter=custom_getter):
        nl1 = 30
        w1_s = tf.get_variable("w1_s", [self.s_dim, nl1], trainable=trainable)
        w2_a = tf.get_variable("w2_a", [self.a_dim, nl1], trainable=trainable)
        b1 = tf.get_variable("b1", [1, nl1], trainable=trainable)
        net = tf.nn.relu(tf.matmul(s, w1_s) + tf.matmul(a, w2_a) + b1)
        return tf.layers.dense(net, 1, trainable=trainable)
```

在 critic 网络里面，输入的是状态 s 和动作 a，输出是对状态 s 下采取动作 a 的一个评估 q 值。

但是我们知道，整个 DDPG 是由四层神经网络架构所组成，那么还有两层由于其网络架构一模一样，并且参数的更新是依据原有评估网络的参数，其本身并不做 loss 梯度下降。这里的参数 reuse 如果等于 True 的话，那么我们就使用之前定义过的 name_scope 和其中的变量，通过这种方法我们就能实现两层目标网络的构建。

```
def __init__(self, a_dim, s_dim, a_bound):
    self.memory = np.zeros((MEMORY_CAPACITY, 2*s_dim+a_dim+1), dtype=np.float32)
    self.pointer = 0
    self.sess = tf.Session()

    self.a_dim, self.s_dim, self.a_bound = a_dim, s_dim, a_bound
    self.S = tf.placeholder(tf.float32, [None, self.s_dim], "s")
    self.S_ = tf.placeholder(tf.float32, [None, self.s_dim], "s_")
    self.R = tf.placeholder(tf.float32, [None, 1], "r")

    self.a = self._build_a(self.S)
    q = self._build_c(self.S, self.a)
```

在这里的话我们首先定义了一下记忆库，动作维度，动作边界这些。之后我们输入状态 s 得到 action，再将其送入 critic 网络，得到状态动作估计值 q。

```
a_params = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='Actor')
c_params = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope='Critic')
ema = tf.train.ExponentialMovingAverage(decay=1 - TAU)  # soft replacement

# ema.average()这一步只是取出值
def ema_getter(getter, name, *args, **kwargs):
    return ema.average(getter(name, *args, **kwargs))

target_update = [ema.apply(a_params), ema.apply(c_params)]  # soft update operation

a_ = self._build_a(self.S_, reuse=True, custom_getter=ema_getter)
q_ = self._build_c(self.S_, a_, reuse=True, custom_getter=ema_getter)

a_loss = - tf.reduce_mean(q)
self.atrain = tf.train.AdamOptimizer(LR_A).minimize(a_loss, var_list=a_params)
# tf.control_dependebcies()给tensorflow计算图中的某些计算指定顺序，这里的话我们获取的就是更新
with tf.control_dependencies(target_update):
    q_target = self.R + GAMMA*q_
    td_error = tf.losses.mean_squared_error(labels=q_target, predictions=q)
    self.ctrain = tf.train.AdamOptimizer(LR_C).minimize(td_error, var_list=c_params)

self.sess.run(tf.global_variables_initializer())
```

至此的话我们还需要将下一个状态送入目标网络中得到动作，再将状态、动作，送入到 critic 网络中，得到对下个状态的估计值。再去构建 td-error 更新参数，actor 的 loss 只用了 critic 估计网络中的输出 q 值(我还得研究一下，嘻嘻)。

tf.train.ExponentialMovingAverage 是指数加权平均的求法，具体的公式是 total=a*total+(1-a)*next。tf.control_dependencies()设计是用来控制计算流图的，给图中的某些计算指定顺序，这里的话，我们就是先计算 target_update，再计算接下来的图。

好吧，就说到这里吧，之后还有一些 A3C, PPO 等算法涉及到一点点多线程，但是主要还是 actor-critic 的网络架构，做了点小小改动。等我学完算法推导之后再来好好补充一下吧。后续版本会放入我的 github 中。