DQN

**Introduction：**

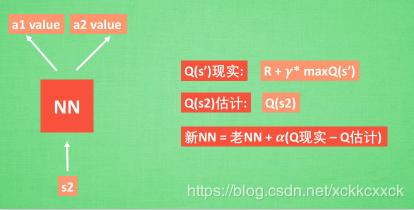
一种融合了神经网络和 Q learning 的方法。

传统的表格形式的强化学习有这样一个瓶颈，用表格来存储(state, action)的Q值。而当今问题是在太复杂, 状态可以多到比天上的星星还多(比如下围棋)。如果全用表格来存储它们, 恐怕我们的计算机有再大的内存都不够, 而且每次在这么大的表格中搜索对应的状态也是一件很耗时的事。

 可以将状态和动作当成神经网络的输入, 然后经过神经网络分析后得到动作的Q值, 这样我们就没必要在表格中记录Q值, 而是直接使用神经网络生成Q值.

也可以只输入状态值, 输出所有的动作值, 然后按照Q-learning的原则, 直接选择拥有最大值的动作当做下一步要做的动作。

Updating method: 基于第二种神经网络来分析, 我们知道神经网络是要被训练才能预测出准确的值。那在强化学习中, 神经网络是如何被训练的呢? 首先, 我们需要 a1, a2正确的Q值, 这个Q值我们就用之前在Q-learning中的Q现实来代替. 同样我们还需要一个Q估计 来实现神经网络的更新。所以神经网络的的参数就是老的NN参数加学习率alpha乘以Q现实和Q估计的差距。也就是下图这样：

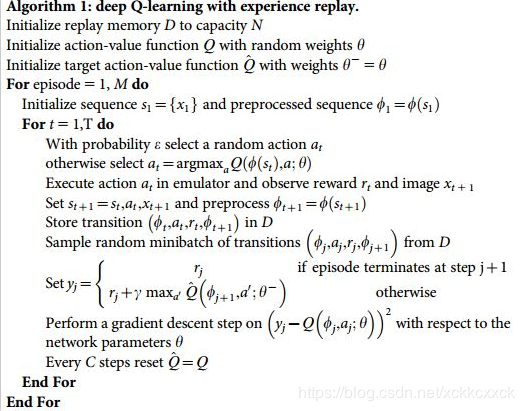


我们通过 NN 预测出Q(s2, a1) 和 Q(s2,a2) 的值, 这就是 Q 估计。然后我们选取Q 估计中最大值的动作来换取环境中的奖励 reward. 而Q现实中也包含从神经网络分析出来的两个Q估计值, 不过这个 Q 估计是针对于下一步在s’的估计。最后再通过刚刚所说的算法更新神经网络中的参数。

**Algorithm:**

Experience replay：DQN 有一个记忆库用于学习之前的经历，Q learning 是一种 off-policy 离线学习法, 它能学习当前经历着的, 也能学习过去经历过的, 甚至是学习别人的经历. 所以每次 DQN 更新的时候, 我们都可以随机抽取一些之前的经历进行学习. 随机抽取这种做法打乱了经历之间的相关性, 也使得神经网络更新更有效率.。

Fixed Q-targets：是一种打乱相关性的机理, 如果使用 fixed Q-targets, 我们就会在 DQN 中使用到两个结构相同但参数不同的神经网络, 预测 Q 估计的神经网络具备最新的参数, 而预测 Q 现实 的神经网络使用的参数则是很久以前的。



**Source Code：**

def run\_maze():

step = 0 # 用来控制什么时候学习

for episode in range(300):

# 初始化环境

observation = env.reset()

while True:

# 刷新环境

env.render()

# DQN 根据观测值选择行为

action = RL.choose\_action(observation)

# 环境根据行为给出下一个 state, reward, 是否终止

observation\_, reward, done = env.step(action)

# DQN 存储记忆

RL.store\_transition(observation, action, reward, observation\_)

# 控制学习起始时间和频率 (先累积一些记忆再开始学习)

if (step > 200) and (step % 5 == 0):

RL.learn()

# 将下一个 state\_ 变为 下次循环的 state

observation = observation\_

# 如果终止, 就跳出循环

if done:

break

step += 1 # 总步数

# end of game

print('game over')

env.destroy()

if \_\_name\_\_ == "\_\_main\_\_":

env = Maze()

RL = DeepQNetwork(env.n\_actions, env.n\_features,

learning\_rate=0.01,

reward\_decay=0.9,

e\_greedy=0.9,

replace\_target\_iter=200, # 每 200 步替换一次 target\_net 的参数

memory\_size=2000, # 记忆上限

# output\_graph=True # 是否输出 tensorboard 文件

)

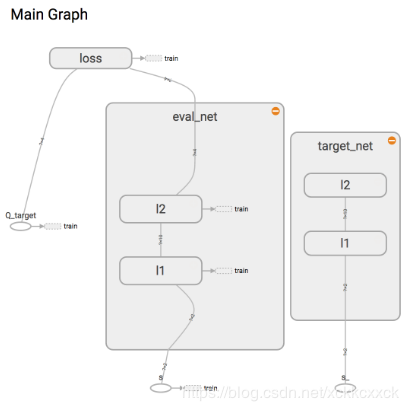
env.after(100, run\_maze)

env.mainloop()

RL.plot\_cost() # 观看神经网络的误差曲线

为了使用 Tensorflow 来实现 DQN, 比较推荐的方式是搭建两个神经网络, target\_net 用于预测 q\_target 值, 他不会及时更新参数. eval\_net 用于预测 q\_eval, 这个神经网络拥有最新的神经网络参数. 不过这两个神经网络结构是完全一样的, 只是里面的参数不一样

两个神经网络是为了固定住一个神经网络 (target\_net) 的参数, target\_net 是 eval\_net 的一个历史版本, 拥有 eval\_net 很久之前的一组参数, 而且这组参数被固定一段时间, 然后再被 eval\_net 的新参数所替换. 而 eval\_net 是不断在被提升的, 所以是一个可以被训练的网络 trainable=True. 而 target\_net 的 trainable=False.



Network Code：

class DeepQNetwork:

def \_build\_net(self):

# -------------- 创建 eval 神经网络, 及时提升参数 --------------

self.s = tf.placeholder(tf.float32, [None, self.n\_features], name='s') # 用来接收 observation

self.q\_target = tf.placeholder(tf.float32, [None, self.n\_actions], name='Q\_target') # 用来接收 q\_target 的值, 这个之后会通过计算得到

with tf.variable\_scope('eval\_net'):

# c\_names(collections\_names) 是在更新 target\_net 参数时会用到

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) # config of layers

# eval\_net 的第一层. collections 是在更新 target\_net 参数时会用到

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)

# eval\_net 的第二层. collections 是在更新 target\_net 参数时会用到

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'): # 求误差

self.loss = tf.reduce\_mean(tf.squared\_difference(self.q\_target, self.q\_eval))

with tf.variable\_scope('train'): # 梯度下降

self.\_train\_op = tf.train.RMSPropOptimizer(self.lr).minimize(self.loss)

# ---------------- 创建 target 神经网络, 提供 target Q ---------------------

self.s\_ = tf.placeholder(tf.float32, [None, self.n\_features], name='s\_') # 接收下个 observation

with tf.variable\_scope('target\_net'):

# c\_names(collections\_names) 是在更新 target\_net 参数时会用到

c\_names = ['target\_net\_params', tf.GraphKeys.GLOBAL\_VARIABLES]

# target\_net 的第一层. collections 是在更新 target\_net 参数时会用到

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)

# target\_net 的第二层. collections 是在更新 target\_net 参数时会用到

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

DDQN

DDQN和Nature DQN一样，也有一样的两个Q网络结构。在Nature DQN的基础上，通过解耦目标Q值动作的选择和目标Q值的计算这两步，来消除过度估计的问题。

在上一节里，Nature DQN对于非终止状态，其目标Q值的计算式子是：

yj=Rj+γmaxa′Q′(ϕ(S′j),A′j,w′)

　　在DDQN这里，不再是直接在目标Q网络里面找各个动作中最大Q值，而是先在当前Q网络中先找出最大Q值对应的动作，即

amax(S′j,w)=argmaxa′Q(ϕ(S′j),a,w)

然后利用这个选择出来的动作amax(S′j,w)在目标网络里面去计算目标Q值。即：

yj=Rj+γQ′(ϕ(S′j),amax(S′j,w),w′)

　　综合起来写就是：

yj=Rj+γQ′(ϕ(S′j),argmaxa′Q(ϕ(S′j),a,w),w′)

除了目标Q值的计算方式以外，DDQN算法和Nature DQN的算法流程完全相同。

Dueling DQN:

<https://www.cnblogs.com/pinard/p/9923859.html>

<https://www.jianshu.com/p/b421c85796a2>