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。

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    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 D

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$

  End For

End For

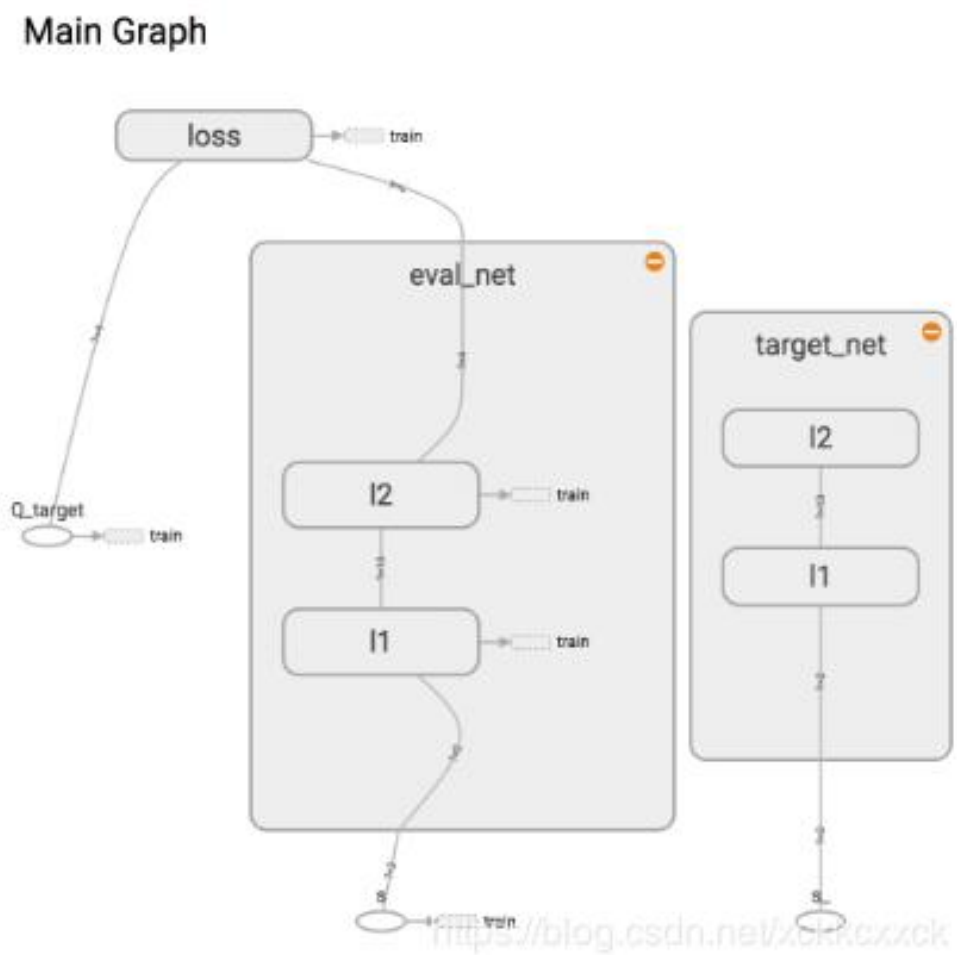
Source Code：

```
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：

$$yj=Rj+\gamma maxa′Q′(\phi(S′j),A′j,w′)$$

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

$$amax(S'j,w)=argmaxa'Q(\phi(S'j),a,w)$$

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：

$$yj=Rj+\gamma Q'(\phi(S'j),amax(S'j,w),w')$$

综合起来写就是：

$$yj=Rj+\gamma Q'(\phi(S'j),argmaxa'Q(\phi(S'j),a,w),w')$$

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