DQN：

Why DQN?

如果您将Q-learning视为更新二维数组（动作空间\*状态空间）中的数字，则实际上它类似于动态编程。这表明，对于Q学习代理以前从未见过的状态来说，它不知道要采取什么行动。换句话说，Q学习代理人没有能力估计看不见的状态的价值。为了解决这个问题，DQN通过引入神经网络来摆脱二维阵列。

DQN利用神经网络来估计Q值函数。网络的输入是电流，输出是每个动作的相应Q值。



1. DQN中数据特点：

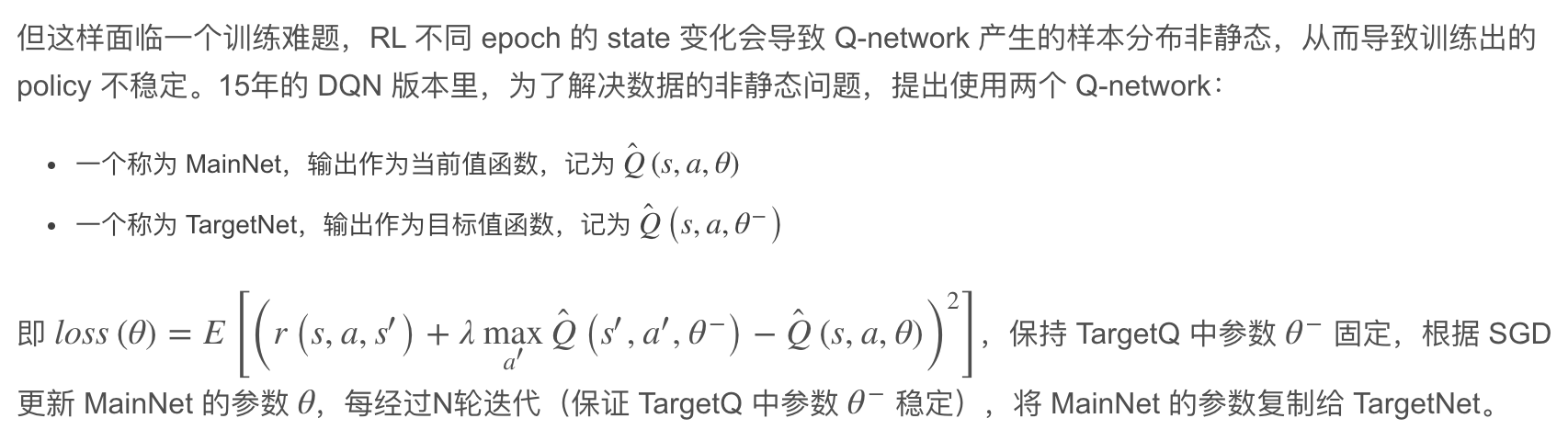
（1）RL 产生的前后 state 数据具有相关性，数据效率较低，不符合 ML 基本假设（DL 要使用 SGD），导致网络难以收敛。

解决方案：experience replay

解决样本分配问题的一种方法是采用经验回放。本质上，样本转换被存储，随后将从“转换池”中随机选择以更新知识。即每次DQN更新的时候，都会从之前的经历进行学习。随机抽取的做法打乱了经历的之间的相关性，使得神经网络的更新更有效率。

（2）RL 在不同 epoch 的 state 分布可能不同，导致 state 数据具有非静态特性，使得训练不稳定。

解决方案：两个网络结构相同但是参数不同的神经网络



2. DQN存在的问题

DQN的动作空间是离散的，不能处理连续action空间的问题。

3. DQN算法





4． 训练过程

记忆单元：S, s1,action\_id,r

S1和s的结构一样，但是参数不一样，target 网络的参数不训练

Target network: s1 -> q\_target

Train network: s -> q\_train

实际s对应的target为 q\_train[action\_id] = reward + gamma\*q\_target

将s和target作为输入输出对，训练train网络

S, s1, reward(根据用户回复判断，符合预期+1，不符合0)

步骤：

两个网络Q(eval) Q(target)

1. 首先输入当前状态s，然后通过Q(eval)和s得到动作的概率分布，依此概率分布，随机的选出一个action（a）来参与训练；
2. 函数F：输入a，输出是下一个状态s1，以及得到的实时reward（r）
3. 保存数据（s, a, r，s1），真正的模型训练数据-memory
4. 每次训练时随机从memory中选择部分数据来作为训练数据
5. 通过状态s和Q(eval)得到q\_eval，通过状态s和Q(target)得到q\_next，然后令

q\_target = q\_eval，接着通过q\_next和r来更新q\_target中a对应的值‘

1. 调整模型参数：输入状态s，q\_target来调参

模型优化目标：q\_eval为通过状态s和Q(eval)得到。

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

模型训练代码过程：

# RL choose action based on observation  
action = RL.choose\_action(observation)  
  
# RL take action and get next observation and reward  
observation\_, reward, done = env.step(action)

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

随机选取样本进行训练

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, :]  
  
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  
 })

# change q\_target w.r.t q\_eval's action  
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)