基于神经网络方法求解RL笔记

Sunday, June 21, 2020 10:04 PM

2020.6.19 晴

知识回顾：

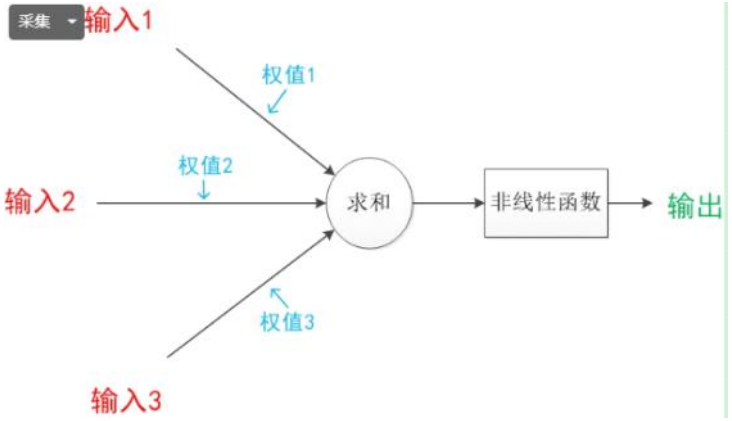
Q表格：状态动作价值矩阵，横轴为actions的维度，纵轴为states的维度，用于描述某个environment中agent在不同states采取不同actions时的总reward

off-policy方法：指行为策略（探索产生数据）和目标策略（利用产生的数据进行训练决策）是分开的方法

QLearning：Qlearning是一种off-policy方法，它的行为策略和目标策略是分开的。其更新函数如下：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

神经网络：是一种非线性统计性数据建模工具，简单来说，神经网络就是一个复杂的函数，通过给定的输入得到预期的输出。神经网络是由一个个神经元连接而成的网络结构，每个神经元包含输入、输出和计算功能，通过对输入进行加权求和后输入一个非线性函数（激活函数）来对某一个问题进行函数拟合。这样可以实现在某个特征维度上对数据的分类，其图形化描述如下，



通常来说一个神经网络包括输入层、隐藏层（中间层）和输出层，每个层级都由多个神经元组成。输入层和输出层的神经元个数是固定的（输入的个数是特征的维度，输出的个数是目标的维度），而隐藏层的神经元个数是可以设置的。对于神经网络的训练，最关键的是找到各神经元之间的连接关系，也就是权重的更新，常使用的方法是梯度下降使损失函数最小（SGD、Adam），因为神经网络的结构相对复杂，为了防止计算梯度的代价过大，常使用反向传播算法来计算梯度（图来自网络），具体可参考 https://www.cnblogs.com/subconscious/p/5058741.html

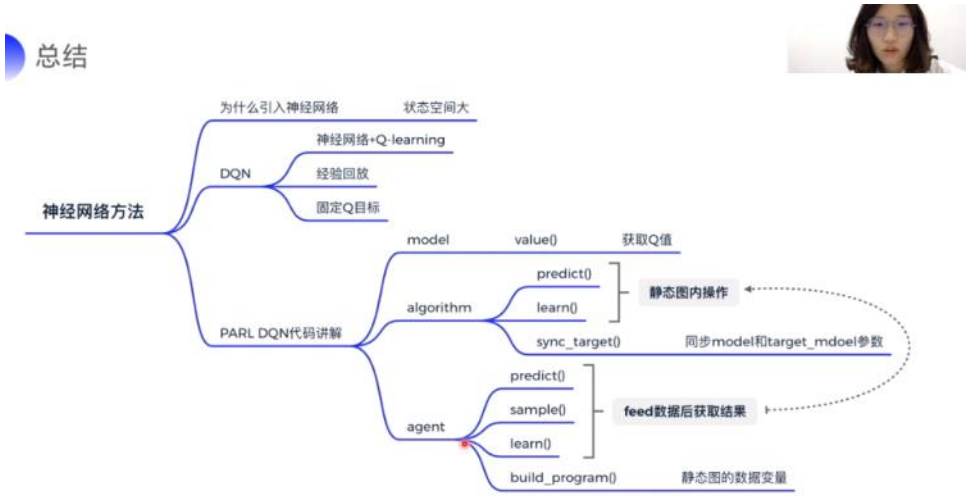
课程结构：

Q表格的缺点

值函数近似的方法

DQN的创新点

DQN的程序解析



Q表格的缺点：

通过上一次的课我们了解到sarsa和qlearning都是通过训练得到一个Q表格来对决策进行指导的，而Q表格的横轴维度为actions的维度，纵轴维度为是states维度。在复杂问题中（围棋、星际大战游戏等）actions和states的维度是非常非常大的，这就会显现出Q表格的两大缺点：

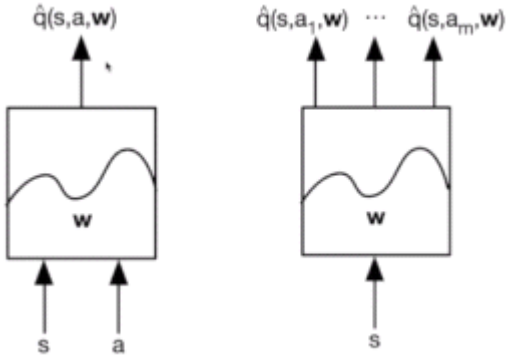
占用极大的内存

查表效率低下

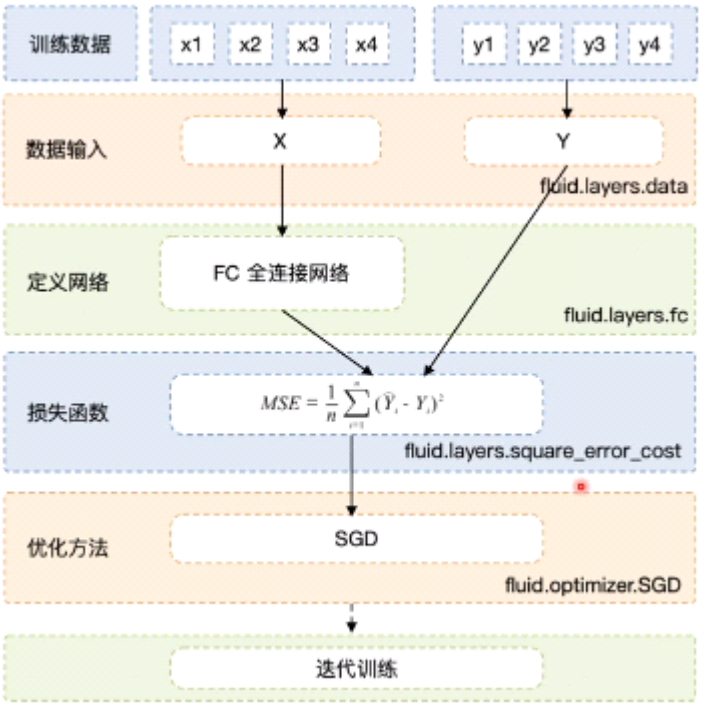
为了解决这个问题，2015年deepmind团队提出了DQN算法（https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf）。DQN算法本质上仍然是QLearning算法，但不再生成一个确定的Q表格，而是通过值函数逼近来近似的描述Q表格。值函数近似方法主要有：多项式函数、神经网络等。

值函数近似的方法：

值函数近似，实际上是指我们建立一个数学模型，给定一定的输入（可以是一个state、一个action也可以是一个state），通过计算，得到一定的输出（可以是一个action对应的q，也可以是所有actions对应的q）



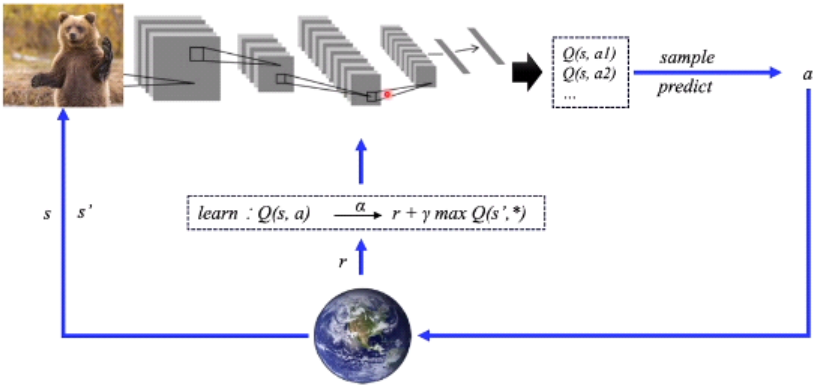
它的主要优点是可以存储有限的参数，并且使状态泛化，令相似的状态可以得到一样的输出（Q表格中没有见过的状态只能从0开始训练）。DQN算法使用的是神经网络。由于神经网络中引入了非线性函数（激活函数），所以DQN是不可证明最终收敛的。神经网络的建模过程如下



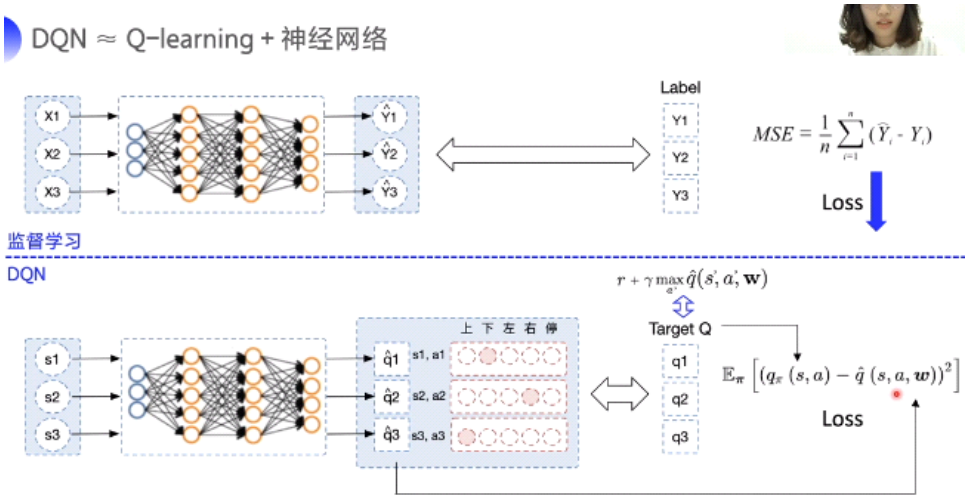
通过给定一定的训练数据（特征和目标），定义一个全连接网络，根据损失函数（均方差），利用SGD（随机梯度下降）方法不断的进行迭代训练得到最优的网络参数（权重）。

为了更简单的理解DQN算法，我们仍然拿人遇到熊的情景举例，我们取消了之前Q表格代表的一些确定的遇熊经验（具体的遇熊状态，该状态下可以采取的相对最优行为），而是通过一个网络计算的结构来拟合这些经验

使用神经网络拟合Q表格



那么DQN实际上引入的神经网络具体是怎么实现的呢？通过对比监督学习我们可以看到，DQN中输入的是qlearning函数的learn函数输入值（state、action、next_state、reward、done），输出的是一个q值向量（不同动作的q值），为了计算损失函数，我们根据Qlearning 的预测公式，先计算一个target，通过两者求均方差得到损失函数。



DQN的创新点：

DQN算法中使用了两个技巧来更好的训练神经网络：

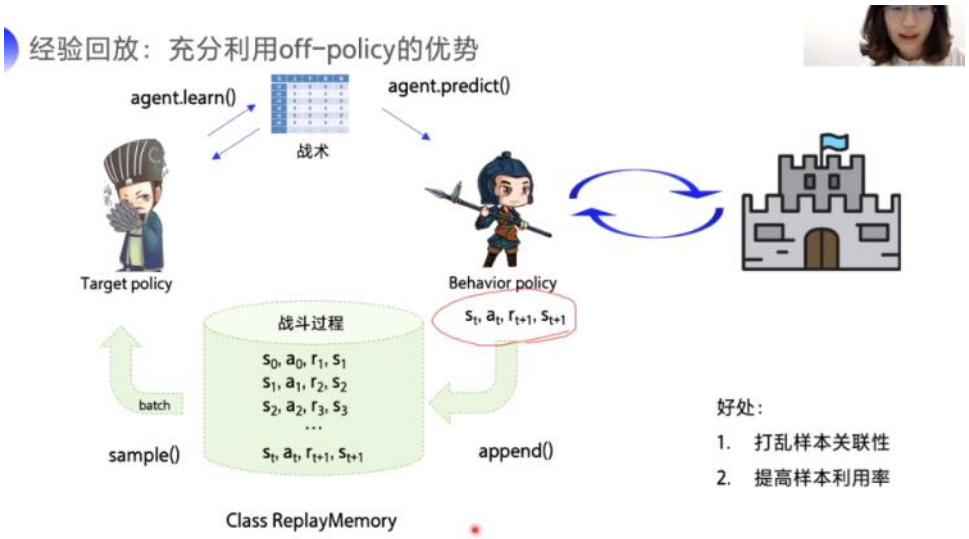
经验回放：

解决样本关联性问题（序列决策的样本关联，样本利用率低）

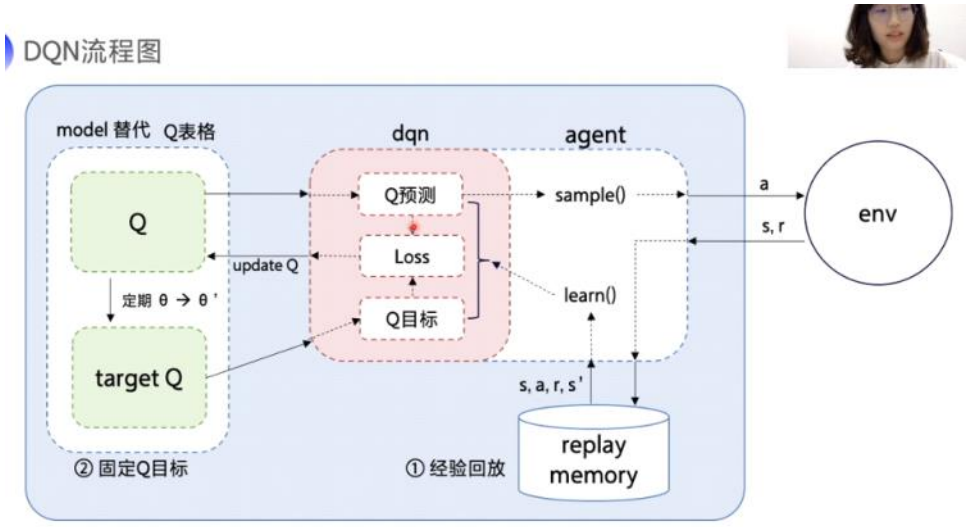
固定Q目标：

解决非平稳性问题（算法非平稳）

样本关联性：在神经网络的训练中，输入的样本是不相关的，但是强化学习中，神经网络的输入是一系列的states状态，而状态间的转移是相关联的，为了解决这个问题，DQN引入了经验池的概念，利用qlearning是off-policy的算法性质，先生成一些数据，暂存在经验池中，神经网络从经验池中随机的抽取数据，打乱他们之间的顺序后进行训练。同一个数据可以被多次训练利用到。举例为士兵攻城：



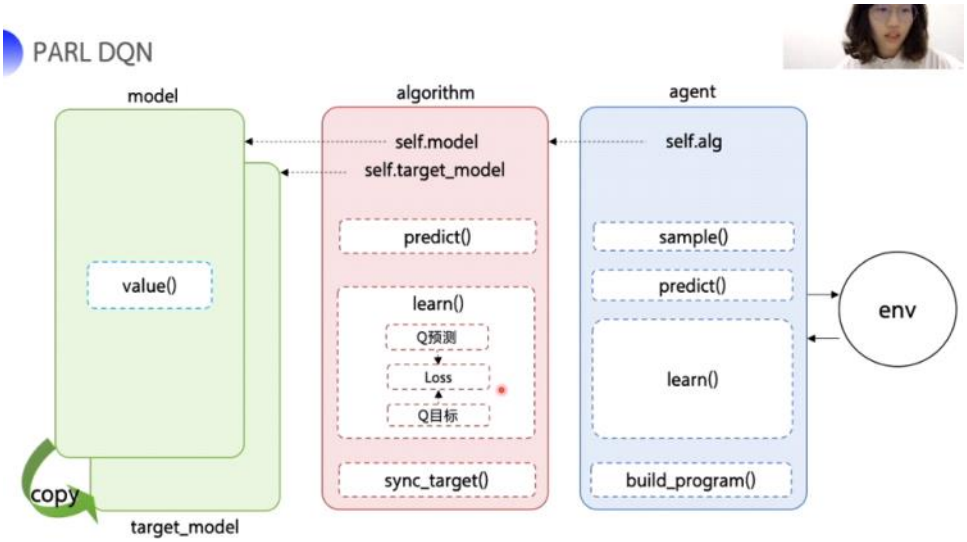
固定的Q目标：在神经网络的训练中，label是标注的确定的真实数据，而强化学习中的target是不真实的，可变化的（qlearning函数预测的q值）这样会导致算法是非平稳的，为了解决这个问题，DQN在一定时间内固定生成target的神经网络（对应QLearning中的Q表格）不更新，这样得出的target在一定时间内也就是固定的，一定程度上解决了算法不平稳的问题，其流程图如下



由上图可见，DQN在实现上需要一个存放经验的队列（固定大小，每次更新数据通过出队入队的方式进行更新），两个神经网络（一个用于生成实际的q，每次都需要更新，一个用于生成target，隔段时间更新一次（从生成q的神经网络中复制权重参数））

DQN的程序解析：

程序结构如下



科老师的视频中有完整的代码讲解，这里就不再赘述，我们直接贴出该算法的关键代码

model：

```
class Model(parl.Model):
    def __init__(self, act_dim):
    hid1_size = 128
    hid2_size = 128 # 3层全连接网络
    self.fc1 = layers.fc(size=hid1_size, act='relu')
    self.fc2 = layers.fc(size=hid2_size, act='relu')
    self.fc3 = layers.fc(size=act_dim, act=None)
def value(self, obs): # 定义网络 # 输入state，输出所有action对应的Q，[Q(s,a1), Q(s,a2), Q(s,a3)...]
```

```
        h1 = self.fc1(obs)
        h2 = self.fc2(h1)
        Q = self.fc3(h2)
        return Q
```

神经网络结构，建立一个三层的全连接网络，每层的激活函数为relu，隐藏层神经元个数为128，预测神经网络的输入层维度为5（state、act、reward、next_act、done），输出层维度为action的维度（act_dim），target生成神经网络的输入层维度为1（state），输出层维度为action的维度（act_dim）

DQN的learn函数：

```
def learn(self, obs, action, reward, next_obs, terminal):
""" 使用DQN算法更新self.model的value网络 """
        # 从target_model中获取 max Q' 的值，用于计算target_Q
        next_pred_value = self.target_model.value(next_obs)
        best_v = layers.reduce_max(next_pred_value, dim=1)
        best_v.stop_gradient = True # 阻止梯度传递
        terminal = layers.cast(terminal, dtype='float32')
        target = reward + (1.0 - terminal) * self.gamma * best_v  # qlearning计算公式
        pred_value = self.model.value(obs) # 取得predict值，此时是q向量，接下里找到action对应的q值
        # 获取Q预测值 # 将action转onehot向量，比如：3 => [0,0,0,1,0]
        action_onehot = layers.one_hot(action, self.act_dim)
        action_onehot = layers.cast(action_onehot, dtype='float32')
        # 下面一行是逐元素相乘，拿到action对应的 Q(s,a)
        # 比如：pred_value = [[2.3, 5.7, 1.2, 3.9, 1.4]], action_onehot = [[0,0,0,1,0]]
        # ==> pred_action_value = [[3.9]]
        pred_action_value = layers.reduce_sum( layers.elementwise_mul(action_onehot,
        pred_value), dim=1)
        # 计算 Q(s,a) 与 target_Q的均方差，得到loss
        cost = layers.square_error_cost(pred_action_value, target)
        cost = layers.reduce_mean(cost)
        optimizer = fluid.optimizer.Adam(learning_rate=self.lr) # 使用Adam优化器
        optimizer.minimize(cost) return cost
```

分三步：计算target_Q（与Qlearning 的learn算法一致，注意生成target的神经网络参数不更新，而只在固定的时间点复制训练网络的参数，所以要设置阻止梯度传递）,获取Q预测值（将actions写为onehot向量，与预测神经网络中取得的q向量进行位运算得到对应的q值），计算loss（计算均方差，使用adam优化更新网络参数）

举例：平衡车

PS： DQN算法文章写的相对久了一点，截止发稿时，强化学习的课程已经全部结课了，后期的作业非常耗时，基本上每次都要跑10个小时以上，调参会困难很多，之后有时间会写一个专门的作业调参总结~因为上一篇的排版问题，这次摸索了一下好的模板，当然文章本身还存在很多问题，在叙述上不是特别有逻辑，希望这次可以有所改进~