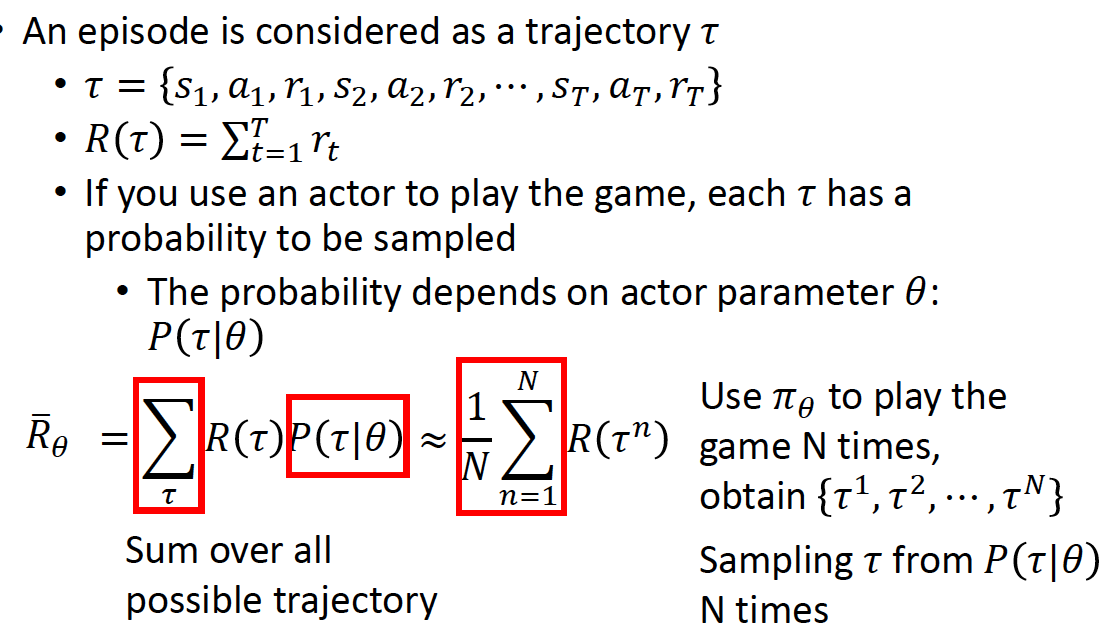
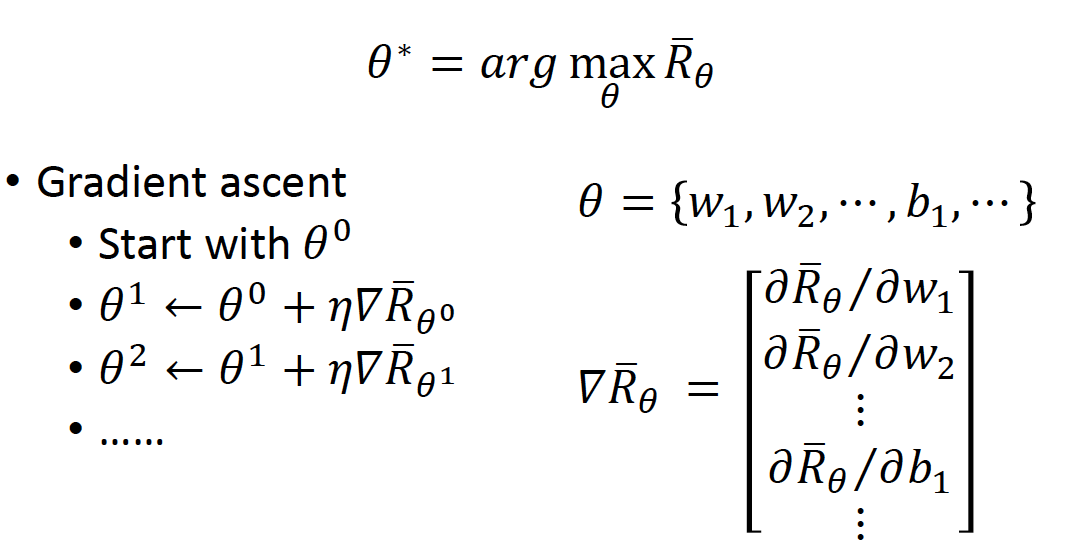
衡量actor的好坏，问题定义



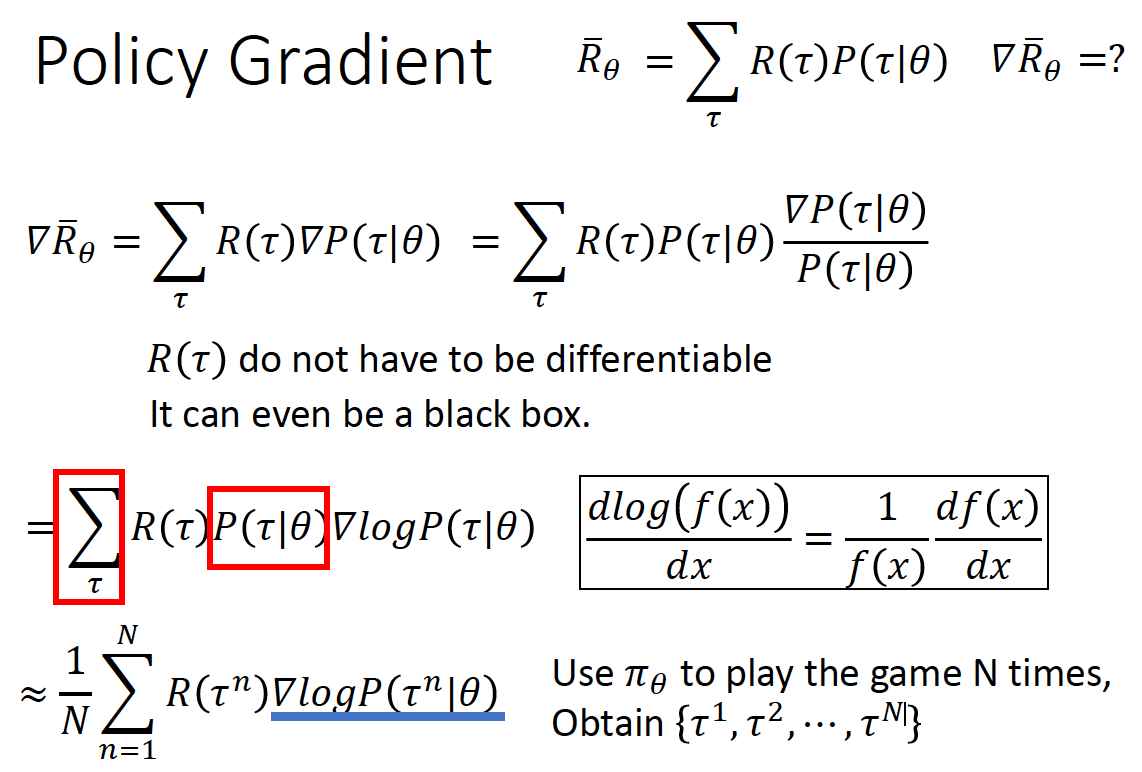
问题陈述：



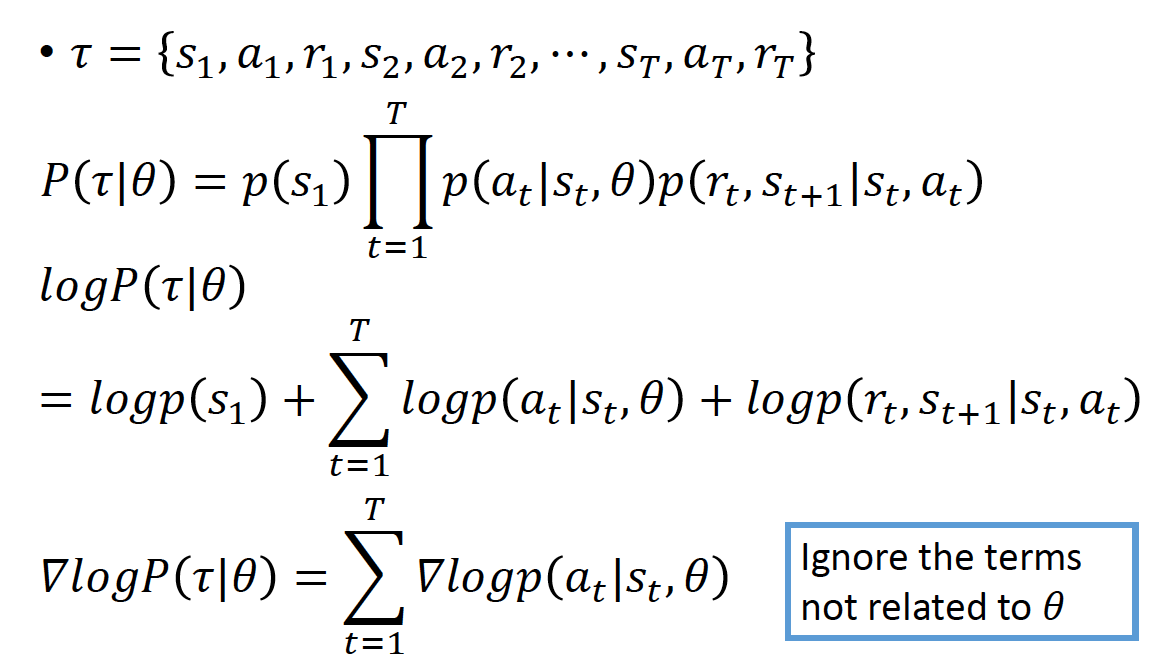
Policy gradient

计算梯度：

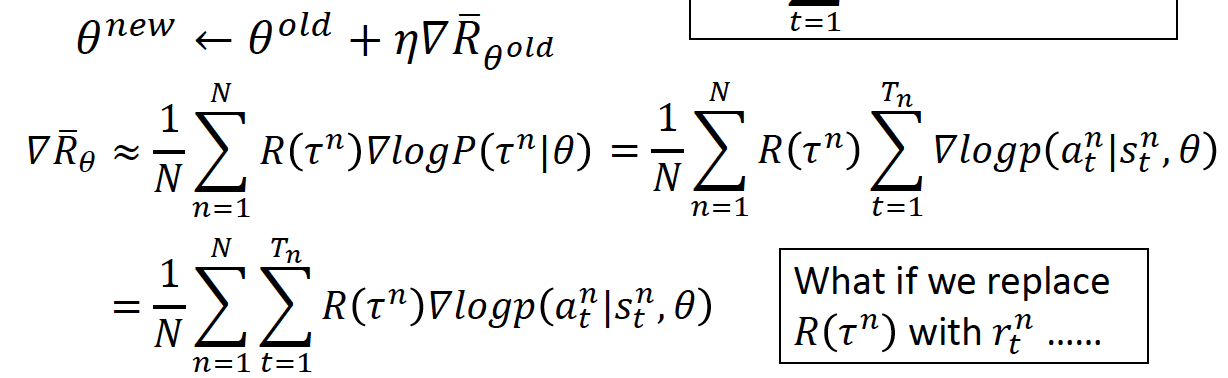
第一步：问题化简

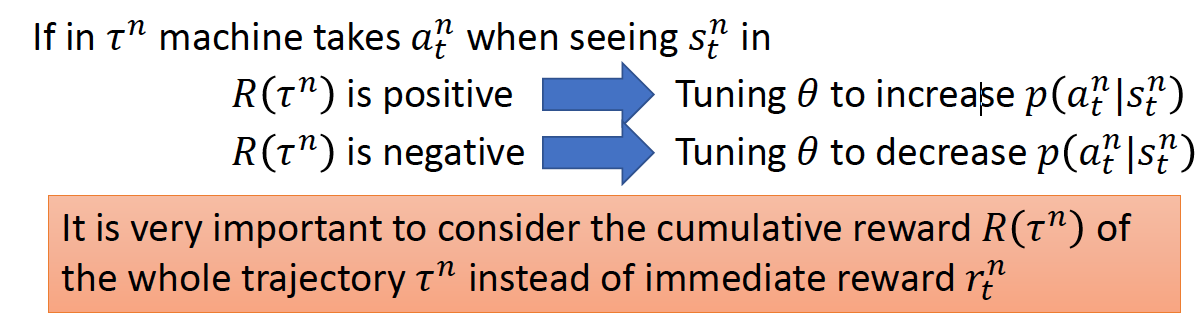


第二步：求梯度

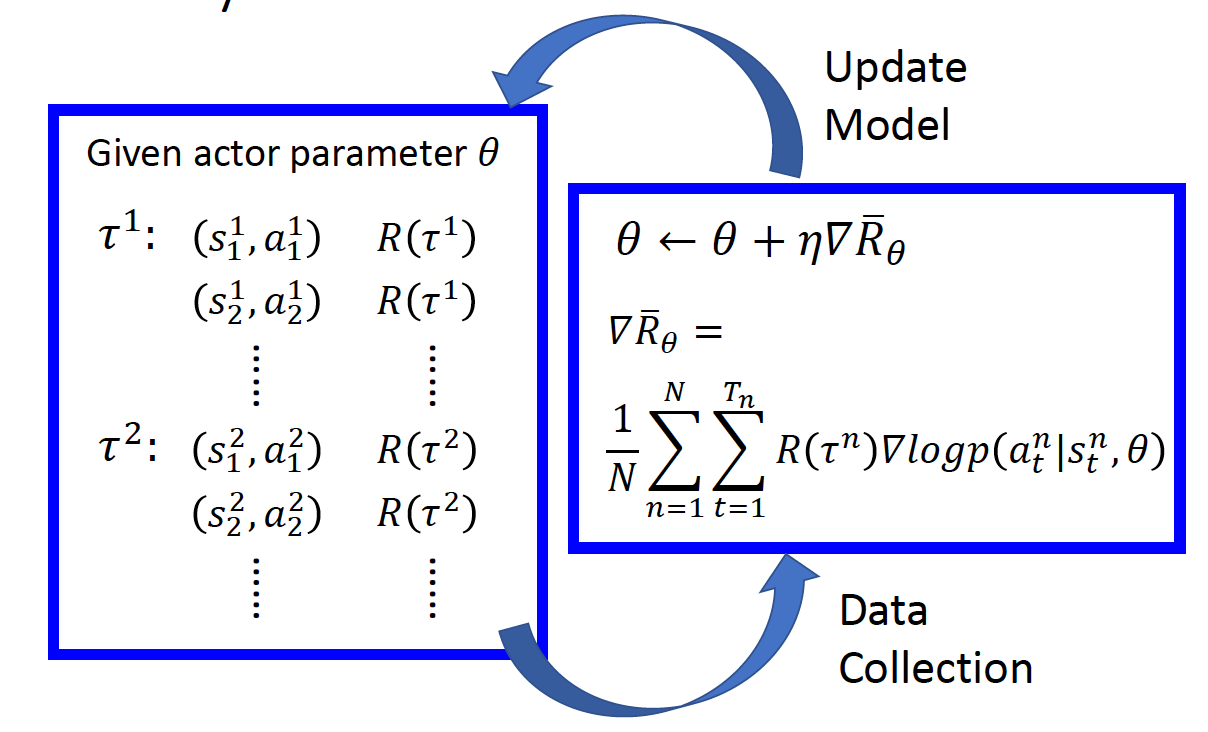


第三步，参数更新

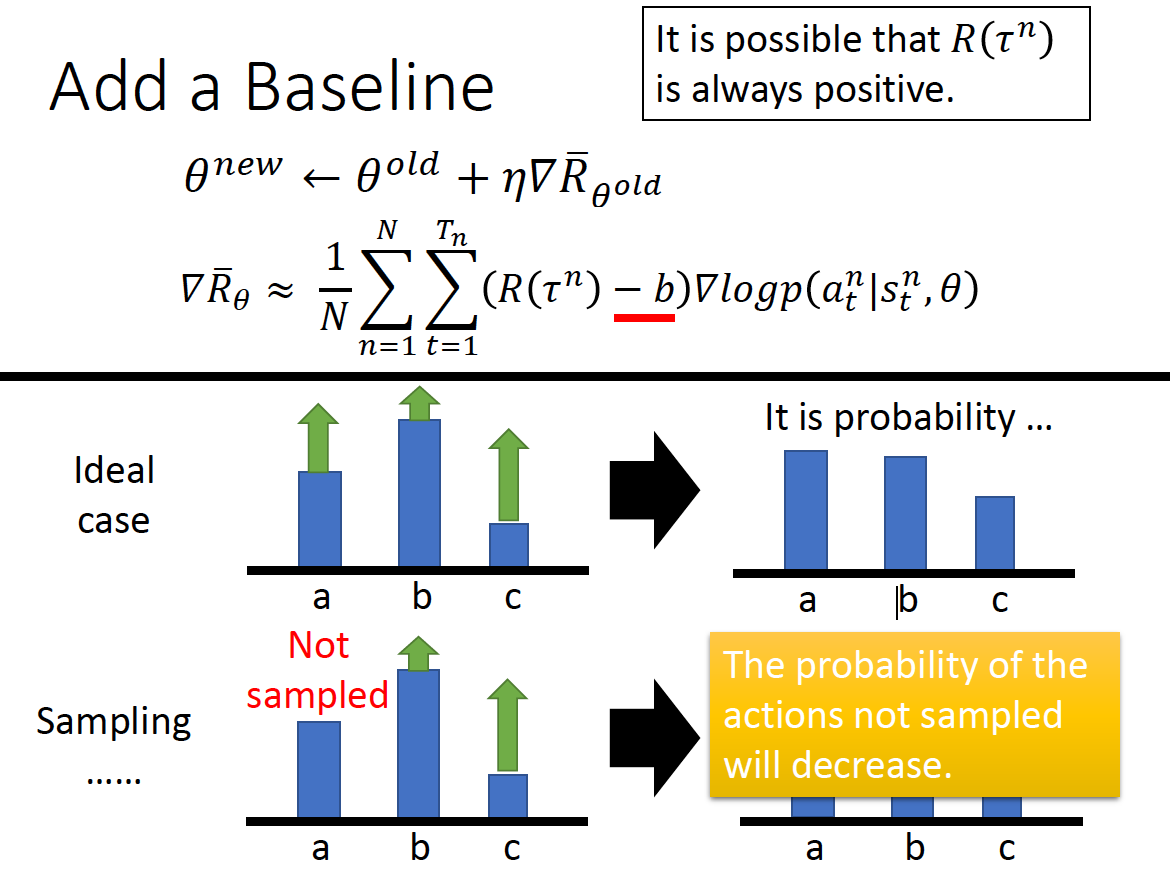




具体的更新步骤：



有个问题，就是序列τ的reward永远为正值，这样当采样到的action被选择的概率就会增加，相对应的，没有被采样到的action的概率其实在变小；因此将R（τ）的值减掉一个平均值，使得其有正有负。



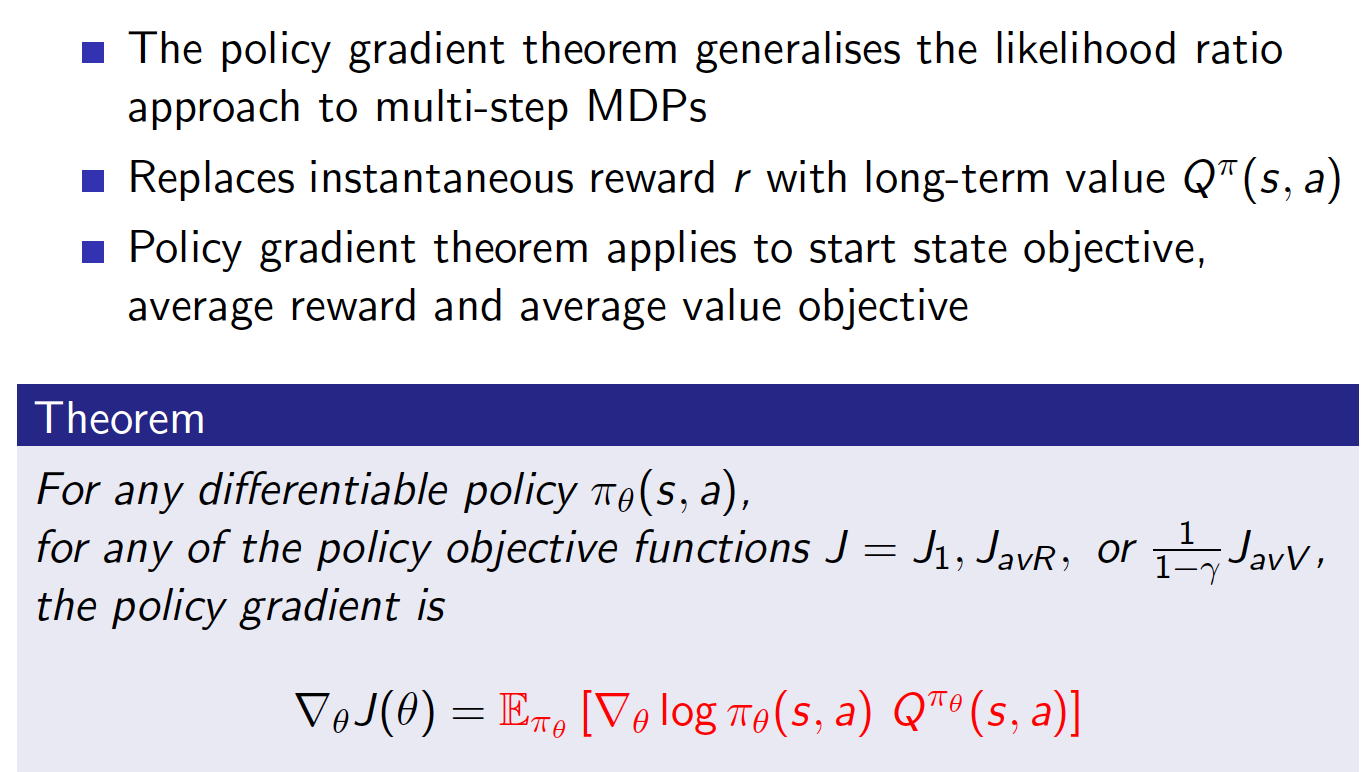
Policy gradient的目标：输出是action被选取的概率分布

希望好的对话被选择的概率越大，不好的对话被选择的概率越低

当action不好时，得到的反馈是负反馈，值为负，则R\*P的值也是越大越好；

当action很好时，得到的反馈是正反馈，值为正，则R\*P的值也是越大越好；

理论：



模型步骤：

M1为初始model，参数固定

M2为训练的model

1. 首先输入当前状态s，然后通过M1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），真正的模型训练数据
4. 在训练模型之前，需要对reward进行进一步的处理，需要考虑后续步骤带来的reward，即模型训练中使用的reward为：r\_model = r + gamma\*r1+gamma^2\*r2….

没有将这个模型使用的reward保存下来的原因应该是有考虑到gamma可以变，这样不会影响到原始数据。

1. 模型输入（s，a，r\_model），输出为action的概率分布

模型拟合目标是：当前action被选择的概率\*reward的值越大越好，乘以-1后，对应的就是loss越小越好。

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)

运行步骤代码示意：

随机选一个action

action = RL.choose\_action(observation)  
函数得出下一个状态和reawrd  
observation\_, reward, done, info = env.step(action)  
保存这个数据对，以后可以直接选用训练  
RL.store\_transition(observation, action, reward)

模型训练过程

def learn(self):  
 # discount and normalize episode reward

模型不会直接使用之前得到的reward，那个只是即时的reward，因为是序列，所以需要考虑之后的状态带来的reward  
 discounted\_ep\_rs\_norm = self.\_discount\_and\_norm\_rewards()  
  
 # train on episode  
 self.sess.run(self.train\_op, feed\_dict={  
 self.tf\_obs: np.vstack(self.ep\_obs), # shape=[None, n\_obs]  
 self.tf\_acts: np.array(self.ep\_as), # shape=[None, ]  
 self.tf\_vt: discounted\_ep\_rs\_norm, # shape=[None, ]  
 })  
  
 self.ep\_obs, self.ep\_as, self.ep\_rs = [], [], [] # empty episode data

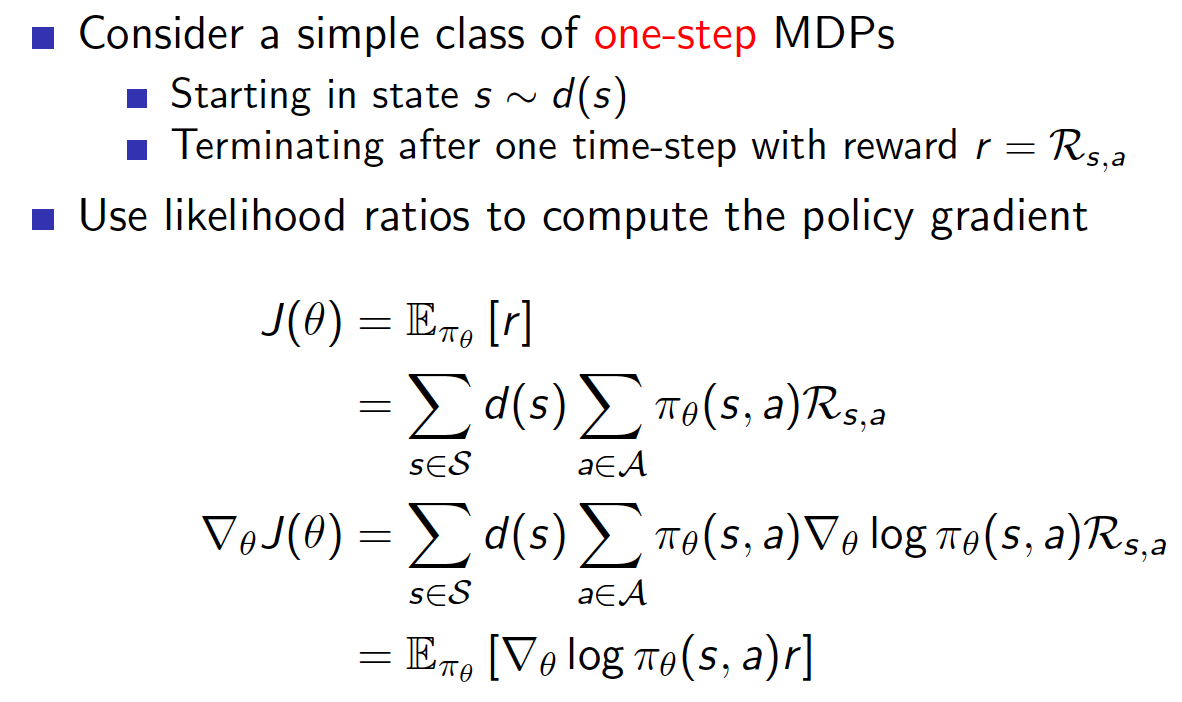
Reward = r(t)+gamma\*r2+gamma^2\*r3+…

def \_discount\_and\_norm\_rewards(self):  
 # discount episode rewards  
 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  
  
 # normalize episode rewards  
 discounted\_ep\_rs -= np.mean(discounted\_ep\_rs)  
 discounted\_ep\_rs /= np.std(discounted\_ep\_rs)  
 return discounted\_ep\_rs

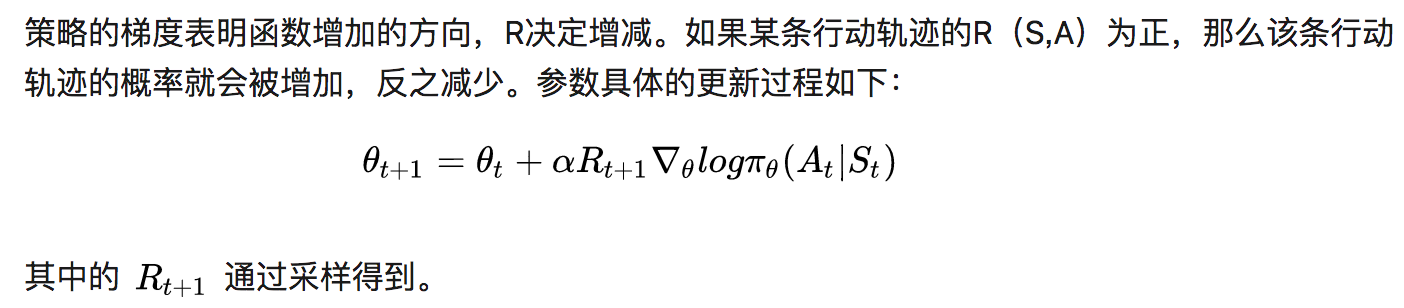
Policy的理论推导

1. one-step MDP

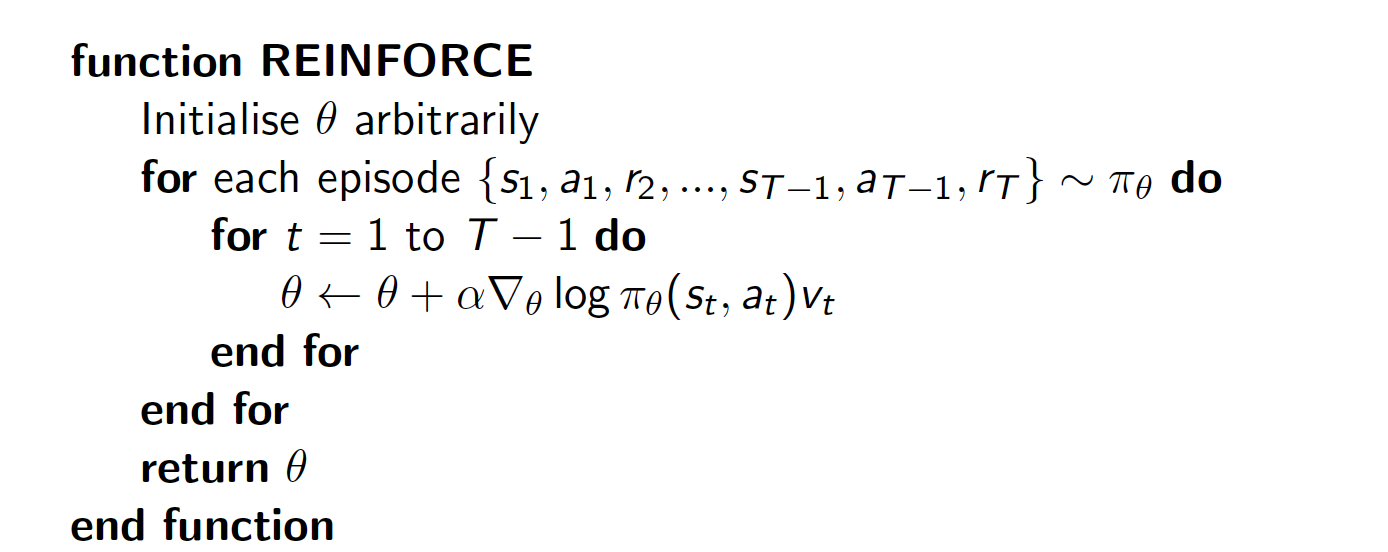
推导过程：



对目标的求导最终转换为求解策略的梯度，其余的值则可以通过采样从数据集中得到：

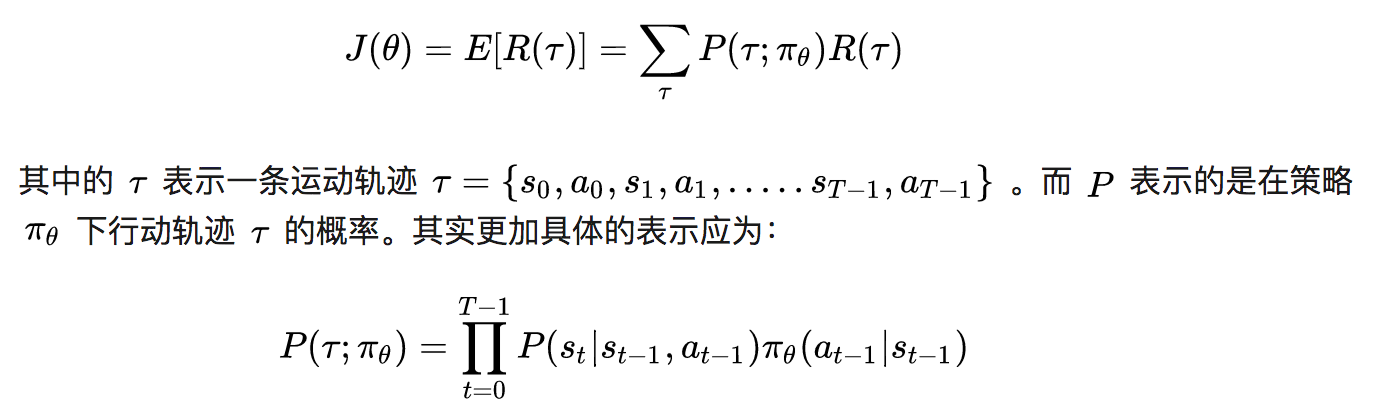


参数更新步骤(monte-carlo policy )：

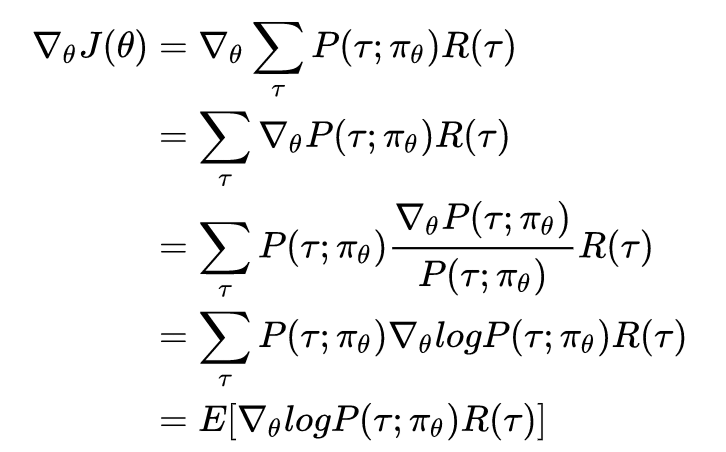


1. 整个序列的平均值 multi-step

不同与上面的是：其中的**每一个奖励函数都是由一条一条的行动轨迹构成的**。因此，这里可以表示为:



P表示的意义为产生的这条轨迹的各个状态下采取该动作的概率和该状态转移的乘积。了解了各个部分的定义之后我们回到performance函数 ，同样对其求导：



因此对函数J(theta)求导转化为对序列出现的概率函数进行求导。这个问题仍然是复杂的，原因在于你大多数的模型是你不知道的，也就是其中的状态转移你并不清楚。因此，按照 P的定义将其中的带入进行分解得到：

