Project B

实验三报告——基于深度学习

学号：2017202085

姓名：陕杰才

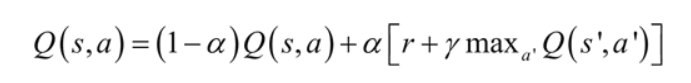
一、Qlearning强化学习

强化学习中有**状态**（state）、**动作**（action）、**奖赏**（reward）这三个要素。智能体（Agent，指小鸟）会根据当前状态来采取动作，并记录被反馈的奖赏，以便下次再到相同状态时能采取更优的动作。

在 Q learning 中，智能体agent计算能够最大化奖励 R 的状态-动作组合，以此学习要采取的动作（策略 π），在选择动作时，智能体不仅要考虑当前的奖励，还要尽量考虑未来的奖励：



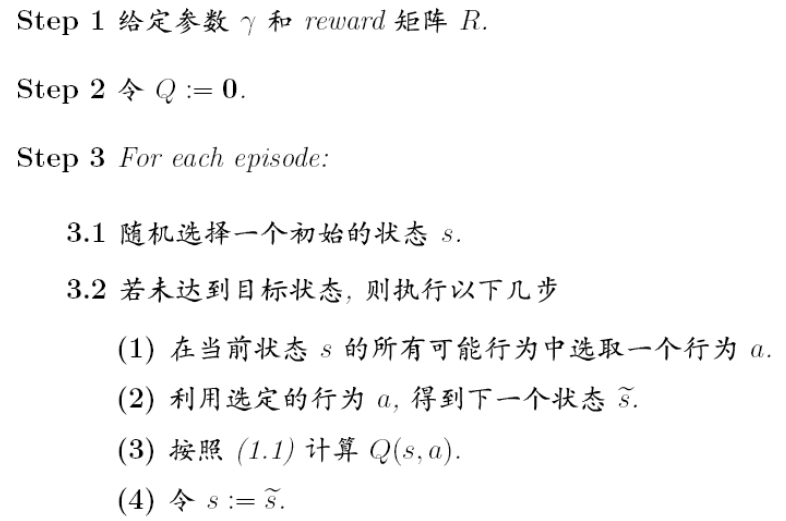
智能体从任意初始状态 Q 开始，选择一个动作 a 并得到奖励 r，然后更新状态为 s'（主要受过去的状态 s 和动作 a 的影响），新的 Q 值为：



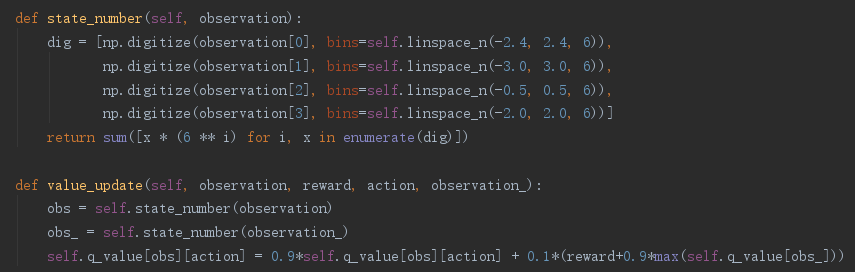
其中，α 是学习率，γ 是折扣因子。第一项保留 Q 的旧值，第二项对 Q 值进行更新估计（包括当前奖励和未来动作的折扣奖励），这会导致在结果状态不满意时降低 Q 值，从而确保智能体在下一次处于此状态时不会选择相同的动作。类似地，当对当前状态满意时，对应的 Q 值将增加。

在无监督的情况下，我们的agent将通过经验进行学习，他不断地从一个状态转至另一状态进行搜索，直到到达目标，我们将agent的每一次探索称为一个episode。

整个Q-learning的计算过程为：



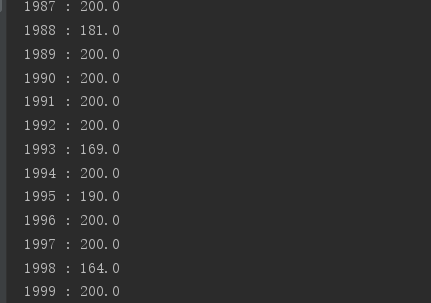
核心函数：



完整代码：

import gym  
import random  
import numpy as np  
  
  
class q\_learning():  
  
 def \_\_init\_\_(self):  
 # q\_table是一个36\*36\*2的二维数组  
 # 离散化后的状态共有2592种可能的取值，每种状态对应一个行动  
 self.q\_value=np.zeros((36\*36+1, 2))  
  
 def choose\_action(self, observation, i):  
 if random.random()>0.1\*(0.99\*\*i):  
 num = self.state\_number(observation)  
 return np.argmax(self.q\_value[num])  
 else:  
 return random.choice([0,1])  
  
 def linspace\_n(self, clip\_min, clip\_max, num):  
 return np.linspace(clip\_min, clip\_max, num + 1)[1:-1]  
  
 def state\_number(self, observation):  
 dig = [np.digitize(observation[0], bins=self.linspace\_n(-2.4, 2.4, 6)),  
 np.digitize(observation[1], bins=self.linspace\_n(-3.0, 3.0, 6)),  
 np.digitize(observation[2], bins=self.linspace\_n(-0.5, 0.5, 6)),  
 np.digitize(observation[3], bins=self.linspace\_n(-2.0, 2.0, 6))]  
 return sum([x \* (6 \*\* i) for i, x in enumerate(dig)])  
  
 def value\_update(self, observation, reward, action, observation\_):  
 obs = self.state\_number(observation)  
 obs\_ = self.state\_number(observation\_)  
 self.q\_value[obs][action] = 0.9\*self.q\_value[obs][action] + 0.1\*(reward+0.9\*max(self.q\_value[obs\_]))  
  
 def game\_start(self):  
 env = gym.make('CartPole-v0')  
 for i in range(2000):  
 count = 0  
 observation = env.reset()  
 while True:  
 action = self.choose\_action(observation, i)  
 observation\_, reward, done, info = env.step(action)  
 count += reward  
 env.render()  
 if done:  
 reward -= 200  
 self.value\_update(observation, reward, action, observation\_)  
 if done:  
 break  
 observation = observation\_  
 print(i,':',count)  
 env.close()  
 print(self.q\_value)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 game = q\_learning()  
 game.game\_start()

结果：



基本可以达到200次

二、DQN

在之前讲到了强化学习求解方法，无论是动态规划DP，蒙特卡罗方法MC，还是时序差分TD，使用的状态都是离散的有限个状态集合S。此时问题的规模比较小，比较容易求解。但是假如我们遇到复杂的状态集合呢？甚至很多时候，状态是连续的，那么就算离散化后，集合也很大，此时我们的传统方法，比如Q-Learning，根本无法在内存中维护这么大的一张Q表。

**1、价值函数的近似表示方法：**

由于问题的状态集合规模大，一个可行的建模方法是价值函数的近似表示。方法是我们引入一个状态价值函数ˆ*v*, 这个函数由参数*w*描述，并接受状态*s*作为输入，计算后得到状态*s*的价值，即我们期望：ˆ*v*(*s*,*w*)≈*vπ*(*s*)

　　类似的，引入一个动作价值函数ˆ*q*，这个函数由参数*w*描述，并接受状态*s*与动作*a*作为输入，计算后得到动作价值，即我们期望：ˆ*q*(*s*,*a*,*w*)≈*qπ*(*s*,*a*)

　　　　价值函数近似的方法很多，比如最简单的线性表示法，用*ϕ*(*s*)表示状态s的特征向量，则此时我们的状态价值函数可以近似表示为：ˆ*v*(*s*,*w*)=*ϕ*(*s*)*Tw*

当然，除了线性表示法，我们还可以用决策树，最近邻，傅里叶变换，神经网络来表达我们的状态价值函数。而最常见，应用最广泛的表示方法是神经网络。

3、DQN算法基本思路

Deep Q-Learning算法的基本思路来源于Q-Learning。但是和Q-Learning不同的地方在于，它的Q值的计算不是直接通过状态值s和动作来计算，而是通过上面讲到的Q网络来计算的。这个Q网络是一个神经网络，我们一般简称Deep Q-Learning为DQN。

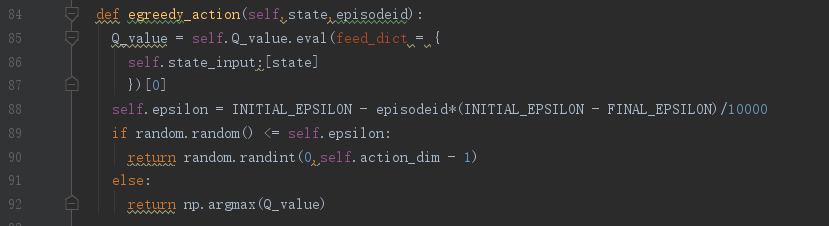
DQN的输入是我们的状态s对应的状态向量*ϕ*(*s*)， 输出是所有动作在该状态下的动作价值函数Q。Q网络可以是DNN，CNN或者RNN，没有具体的网络结构要求。

　　DQN主要使用的技巧是经验回放（experience replay）,即将每次和环境交互得到的奖励与状态更新情况都保存起来，用于后面目标Q值的更新。为什么需要经验回放呢？我们回忆一下Q-Learning，它是有一张Q表来保存所有的Q值的当前结果的，但是DQN是没有的，那么在做动作价值函数更新的时候，就需要其他的方法，这个方法就是经验回放。

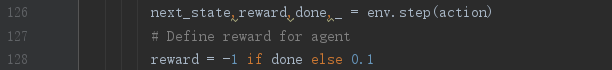
　　通过经验回放得到的目标Q值和通过Q网络计算的Q值肯定是有误差的，那么我们可以通过梯度的反向传播来更新神经网络的参数*w*，当*w*收敛后，我们的就得到的近似的Q值计算方法，进而贪婪策略也就求出来了。

2、代码实现

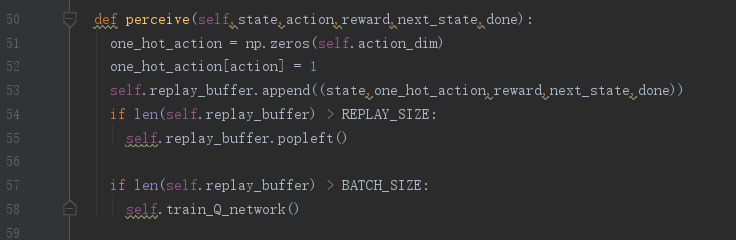
代码使用了一个三层的神经网络，输入层，一个隐藏层和一个输出层。下面我们看看关键部分的代码。

通过*ϵ*−贪婪法选择动作的代码如下，每次我们*ϵ*−贪婪法后都会减小*ϵ*值

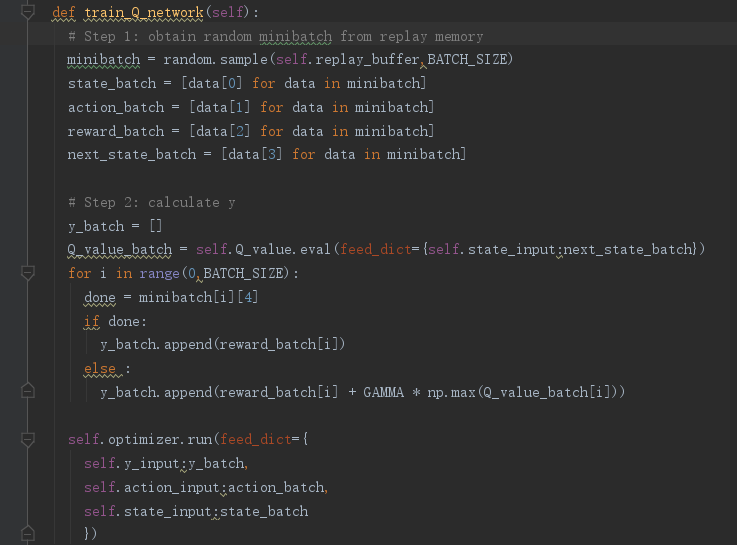
执行当前动作，这个交互由gym完成：



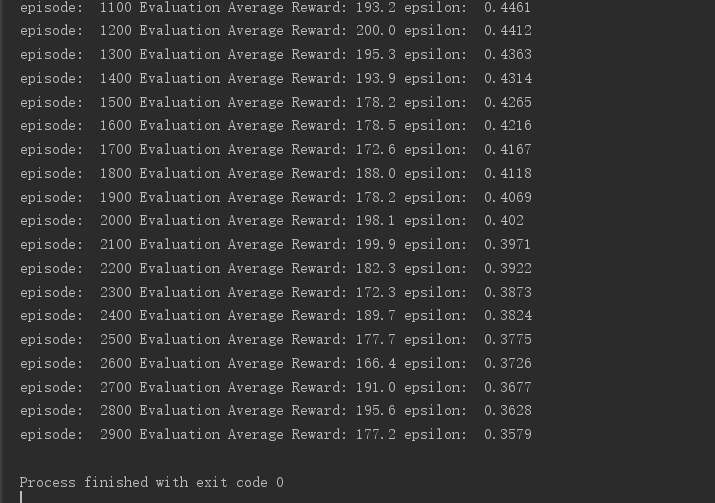
保存经验回放数据的代码如下：



计算目标Q值，并更新Q网络的代码如下：



3000轮迭代的输出如下：



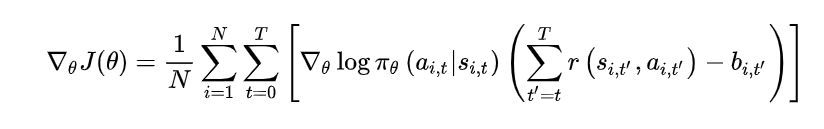
大概到第1200次迭代后，算法已经收敛，达到最高的200分。当然由于是*ϵ*−探索，每次前面的输出可能不同，但最后应该都可以收敛到200的分数。当然由于DQN不保证绝对的收敛，所以可能到了200分后还会有抖动。

三、策略梯度A2C

DQN算法缺陷：对连续动作的处理能力不足。DQN之类的方法一般都是只处理离散动作，无法处理连续动作。

1. AC算法

梯度下降公式：

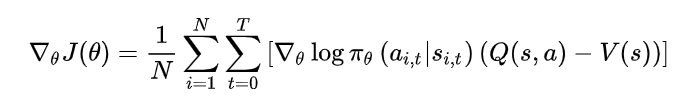


用轨迹的回报表示整个序列的价值，这个表示是无偏的，但是在真实的训练过程中，由于交互次数的有限，方差相对较大，为了模型的稳定，可以牺牲一定的偏差来使方差变小，Actor-Critic 算法即是这样的一种方法。

AC 算法的主要特点就是用一个独立的模型估计轨迹的长期回报，而不再直接使用轨迹的真实回报，在估计时使用模型估计轨迹价值，在更新时利用轨迹的回报得到目标价值，然后将模型的估计值和目标值进行比较，从而改进模型。使用TD-Error 估计轨迹的回报，此时梯度公式变为：  
由于引入了状态价值模型， 算法整体包含了两个模型，一个是策略模型，一个是价值模型，所以这个算法被称为Actor-Critic ， 其中Actor 表示策略模型， Critic 表示价值模型。

2、A2C算法

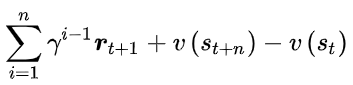
A2C算法直接使用优势函数估计轨迹的回报，由此梯度公式变为：



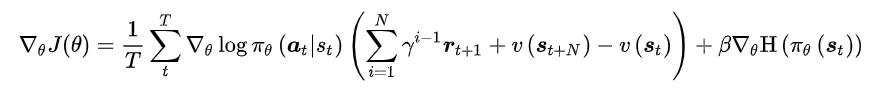
这样的话就需要有两个网络分别计算状态-动作价值Q和状态价值V，此时的Critic变为估计状态价值V的网络。因此Critic网络的损失变为实际的状态价值和估计的状态价值的平方损失。

3、A3C算法

A3C 算法使用了多步回报估计法，即

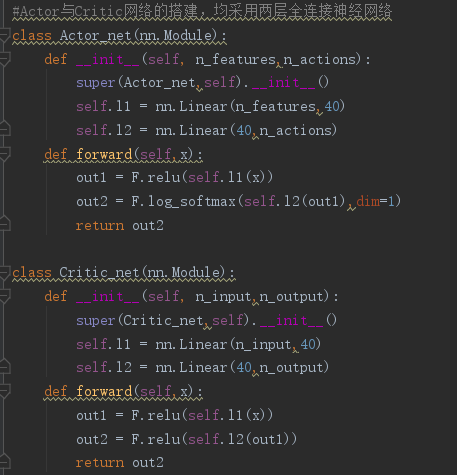


这个方法可以在训练早期更快地提升价值模型，为了增加模型的探索性，模型的目标函数中加入了策略的熵。由于熵可以衡量概率分布的不确定性，所以我们希望模型的情尽可能大一些，这样模型就可以拥有更好的多样性。这样，完整的策略梯度计算公式就变为



4、使用A2C方法实现CartPole游戏过程

核心函数：Actor与Critic网络的搭建，均采用两层全连接神经网络



结果：

