**基于机器学习和强化学习解决gym\_cartpole问题**

**——人工智能课程汇报论文**

姓名：陕杰才

学号：2017202085

**摘要：**本文主要介绍在gym仿真环境下用机器学习和强化学习的方法解决经典控制问题中倒立摆cartpole问题。由于自身能力有限，机器学习上主要采用经典的爬山算法，通过不断地迭代过程，模型的参数不断得到优化，最终得到一组最优的权值作为控制模型的解。强化学习上主要采用Qlearning和DQN，效果均很好。

关键词：机器学习 强化学习 不断优化

一、引言：

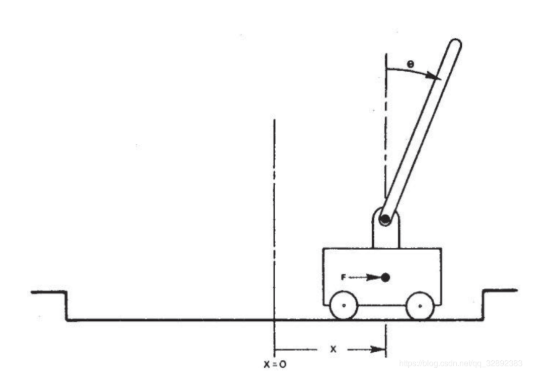
Gym是一个研究和开发强化学习相关算法的仿真平台，无需智能体先验知识，并兼容常见的数值运算库如 TensorFlow、Theano等。OpenAI Gym由以下两部分组成：

Gym开源库：测试问题的集合。当你测试强化学习的时候，测试问题就是环境，比如机器人玩游戏，环境的集合就是游戏的画面。这些环境有一个公共的接口，允许用户设计通用的算法。

OpenAI Gym服务：提供一个站点和API（比如经典控制问题：CartPole-v0），允许用户对他们的测试结果进行比较。

简单来说OpenAI Gym提供了许多问题和环境（或游戏）的接口，而用户无需过多了解游戏的内部实现，通过简单地调用就可以用来测试和仿真。

游戏背景：环境中有一辆小车，在一维无阻力的轨道上运动，车上连接一根活动杆，杆会左右摇摆。游戏模型如下所示：

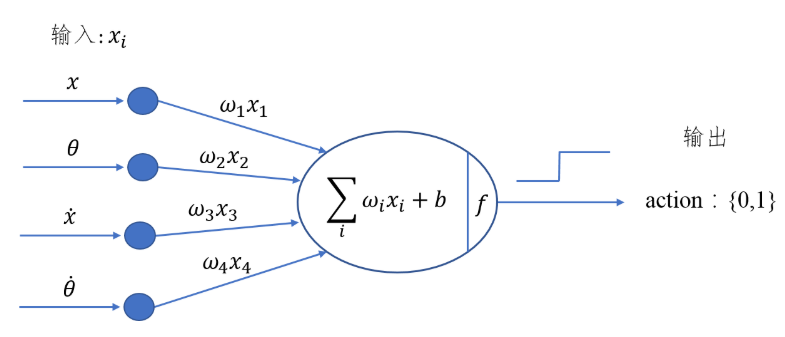


游戏规则很简单，我们要操纵我们的小车左右移动，使它上面的木棒能够保持平衡。当小车偏离中心4.8个单位，或杆的倾斜超过15度，任务失败。每次调用环境中的*env.reset*( )将重新产生一个初始状态observation。observation的四个元素分别表示了小车位置、小车速度、杆子夹角及角变化率。我们需要根据这四个值返回正确的动作action让小车左右移动，来确保杆子不倒。

二、机器学习

1、爬山算法

为了能够有效控制倒立摆首先应建立一个控制模型。明显的，这个控制模型的输入应该是当前倒立摆的状态（*observation*）而输出为对当前状态做出的决策动作（*action*）。决定倒立摆状态的observation是一个四维向量，包含小车位置、杆子夹角、小车速度及角变化率，如果对这个向量求它的加权和，那么就可以根据加权和值的符号来决定采取的动作（action），用sigmoid函数将这个问题转化为二分类问题，从而可以建立一个简单的控制模型。模型如下图所示：



上图的实际功能与神经网络有几分相似，但比神经网络要简单得多。通过加入四个权值，我们可以通过改变权重值来改变决策（*policy*），即有加权和为了得到一组较好的权值从而有效控制倒立摆，我们可以采用爬山算法（*hill climbing algorithm*）进行学习优化。爬山算法是一种启发式方法，是对深度优先搜索的一种改进，它利用反馈信息帮助生成解的决策。

实现思路：

1. 首先随机取权值：best\_reward = 0 # 初始最佳奖励，best\_weights = np.random.rand(5) # 初始权值为随机取值
2. 之后通过爬山算法选取权值：def get\_weights\_by\_hill\_climbing(best\_weights): return best\_weights + np.random.normal(0, 0.1, 5)
3. 获取当前权值的模型控制的奖励和：

def get\_sum\_reward\_by\_weights(env, weights):

# 测试不同权值的控制模型有效控制的持续时间（或奖励）

observation = env.reset() # 重置初始状态

sum\_reward = 0 # 记录总的奖励

for t in range(1000):

# time.sleep(0.01)

# env.render()

action = get\_action(weights, observation) # 获取当前权值下的决策动作

observation, reward, done, info = env.step(action)# 执行动作并获取这一动作下的下一时间步长状态

sum\_reward += reward

# print(sum\_reward, action, observation, reward, done, info)

if done:# 如若游戏结束，返回

print("Episode finished after {} timesteps".format(t + 1))

break

return sum\_reward

1. 更新当前最优权值： if cur\_sum\_reward > best\_reward:

best\_reward = cur\_sum\_reward

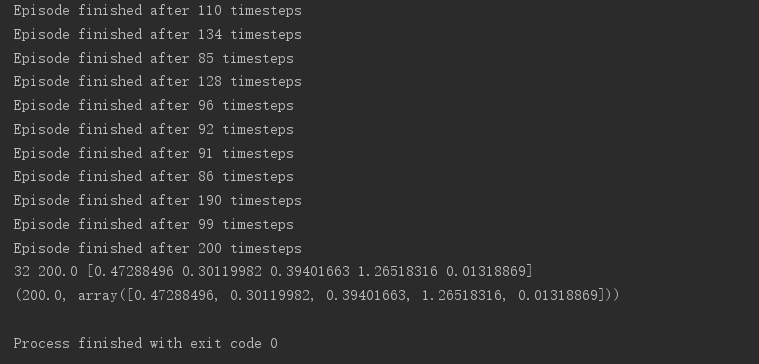
best\_weights = cur\_weights

# 达到最佳奖励阈值后结束

if best\_reward >= 200:

break

结果：



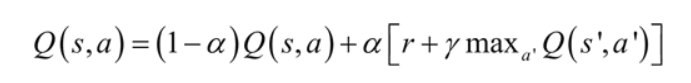
爬山算法本质是一种局部择优的方法，效率高但因为不是全局搜索，所以结果可能不是最优。在这里采用的模型较为简单，如若想要获得更好的学习效果可以考虑更加复杂的模型，如深度神经网络。

三、强化学习

1、Q\_learning

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：

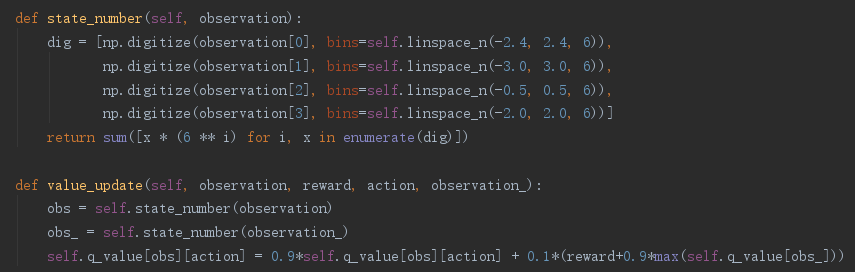
智能体从任意初始状态 Q 开始，选择一个动作 a 并得到奖励 r，然后更新状态为 s'（主要受过去的状态 s 和动作 a 的影响），新的 Q 值用数学公式表示为：



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，直到达到目标状态。

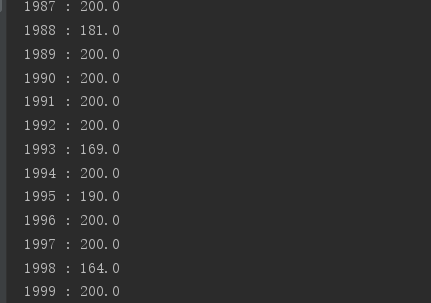
基本思路：

1. 初始化q\_table表：self.q\_value=np.zeros((36\*36+1, 2)) # q\_table是一个36\*36\*2的二维数组
2. 循环迭代获取observasion,若未达到要求，即未达到200次，则返回当前状态在q\_table表中最优的值
3. 达到要求后更新q\_table表，具体更新的核心代码如下：



1. 随着q\_table表的不断更新，结果越来越好，直到完成全部2000次迭代

结果：



2、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。

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。

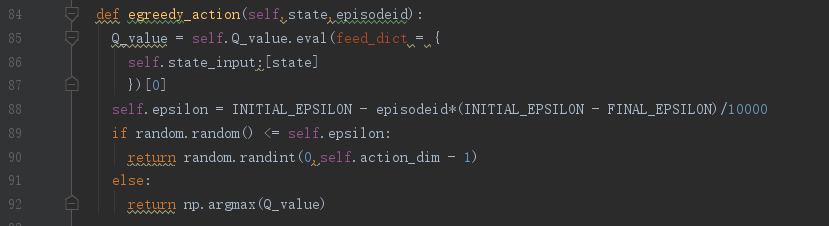
DQN的输入是我们的状态s对应的状态向量*ϕ*(*s*)， 输出是所有动作在该状态下的动作价值函数Q。Q网络可以是DNN，CNN或者RNN，没有具体的网络结构要求。

　　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。

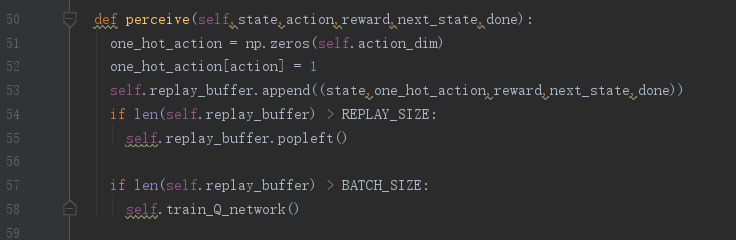
　　通过经验回放得到的目标Q值和通过Q网络计算的Q值肯定是有误差的，那么我们可以通过梯度的反向传播来更新神经网络的参数*w*，当*w*收敛后，我们的就得到的近似的Q值计算方法，进而贪婪策略也就求出来了。

核心代码：

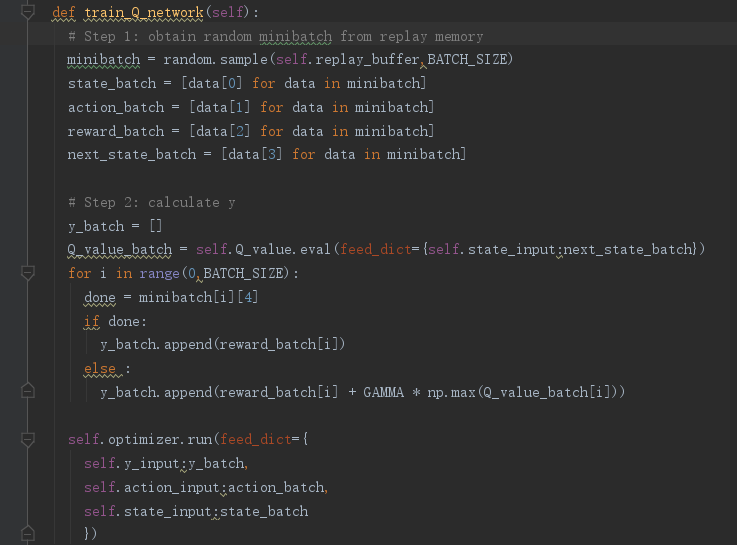
通过*ϵ*−贪婪法选择动作的代码如下，每次我们*ϵ*−贪婪法后都会减小*ϵ*值：



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：

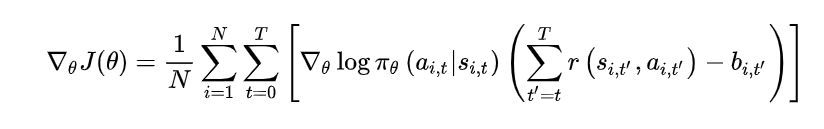


大概到第1200次迭代后，算法已经收敛，达到最高的200分。

3、策略梯度A2C

（1）AC算法

梯度下降公式：

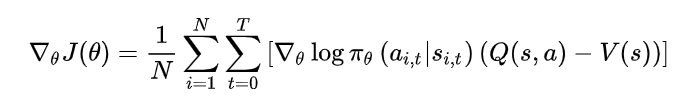


用轨迹的回报表示整个序列的价值，这个表示是无偏的，但是在真实的训练过程中，由于交互次数的有限，方差相对较大，为了模型的稳定，可以牺牲一定的偏差来使方差变小，Actor-Critic 算法即是这样的一种方法。

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

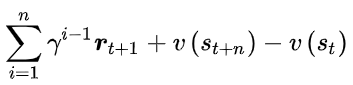
A2C算法直接使用优势函数估计轨迹的回报，由此梯度公式变为：



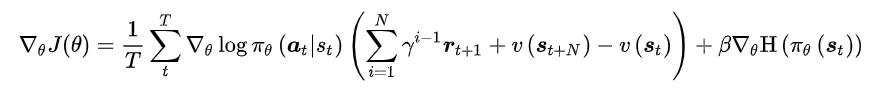
这样的话就需要有两个网络分别计算状态-动作价值Q和状态价值V，此时的Critic变为估计状态价值V的网络。因此Critic网络的损失变为实际的状态价值和估计的状态价值的平方损失。

（3）A3C算法

A3C 算法使用了多步回报估计法，即

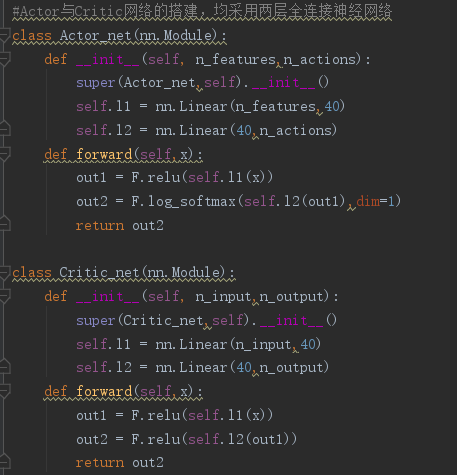


这个方法可以在训练早期更快地提升价值模型，为了增加模型的探索性，模型的目标函数中加入了策略的熵。由于熵可以衡量概率分布的不确定性，所以我们希望模型的情尽可能大一些，这样模型就可以拥有更好的多样性。这样，完整的策略梯度计算公式就变为

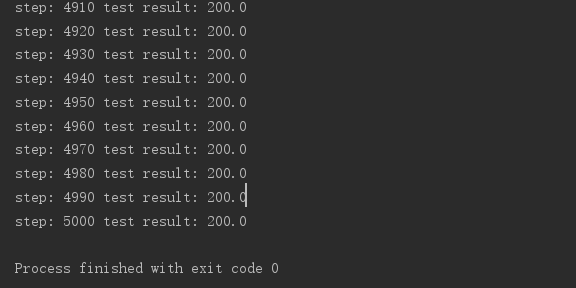


（4）使用A2C方法实现CartPole游戏过程

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：



四、遇到的问题：

1、机器学习我一开始想的是调用现成的函数来实现常见的svm，贝叶斯等机器学习方法，但是跑出来的效果都不是很好，结果最高只能达到100-120左右，达不到要求，数据集我才用的是之后爬山算法得出的数据，我觉得有可能和数据集有关，这部分我还需要再考虑一下

2、爬山算法相当于是通过不断的迭代更新权值来达到的过程，，一开始我对这个算法的思想不熟，随机值的设置较大，效果一直不是很好，后来不断更改参数后才得到较好的结果

3、Qlearning算法最主要的是设置q\_table的大小，因为所有的observasion状态有很多，全部保存会发生维度灾难，我通过查找找到了np.digitize()分别对各个连续特征值进行离散化（分箱处理）来将所有的情况存在一个36\*36\*2的表中。

五、总结

通过这次的课程作业，我学习了爬山算法，Q\_learning强化学习，DQN深度强化学习等方法，更加熟练的掌握了机器学习和强化学习的基本思想，为以后的继续学习打下了基础