搭建人工神经网络对mushrooms数据集分类的实验报告

1. **实验目的**

通过实践加深对于神经网络的理解。

1. **实验任务**

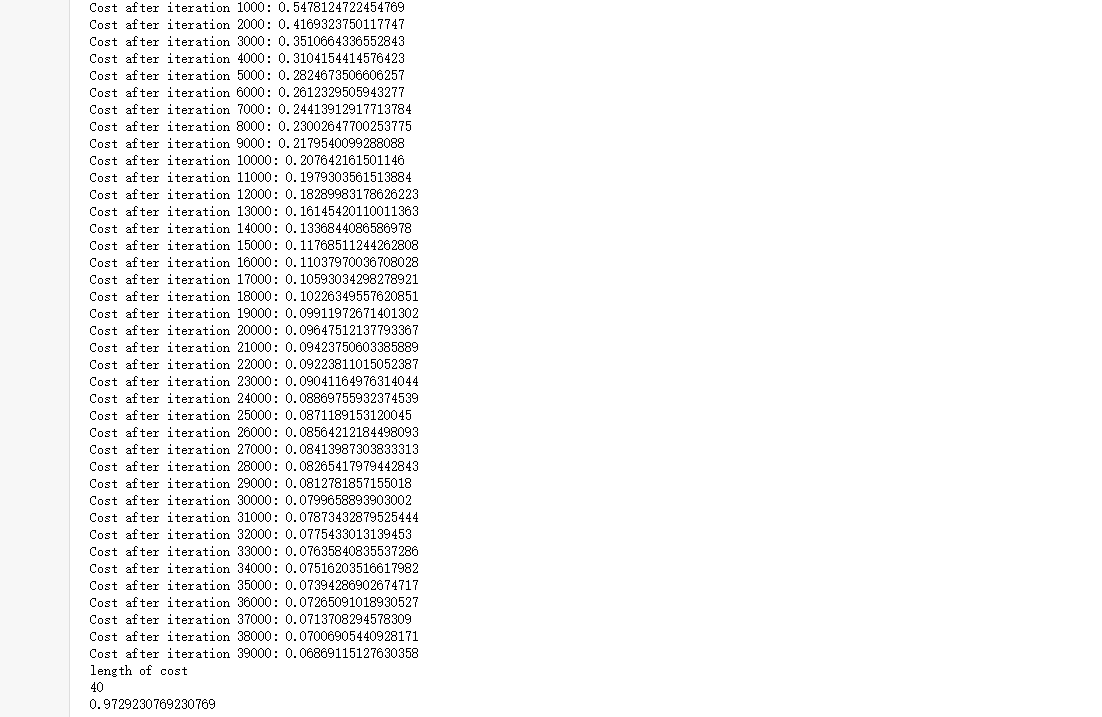
对收集的Mushroom数据进行分类，判断是否有毒。使用自己搭建的人工神经⽹络模型完成Classiﬁcation任务，并且对模型进行不断优化，将准确率提升到极限。不推荐使⽤已有的集成Machine Learning库，⽐如：keras，sklearn等。

1. **使用环境**

Jupyter Notebook

1. **实验结果**

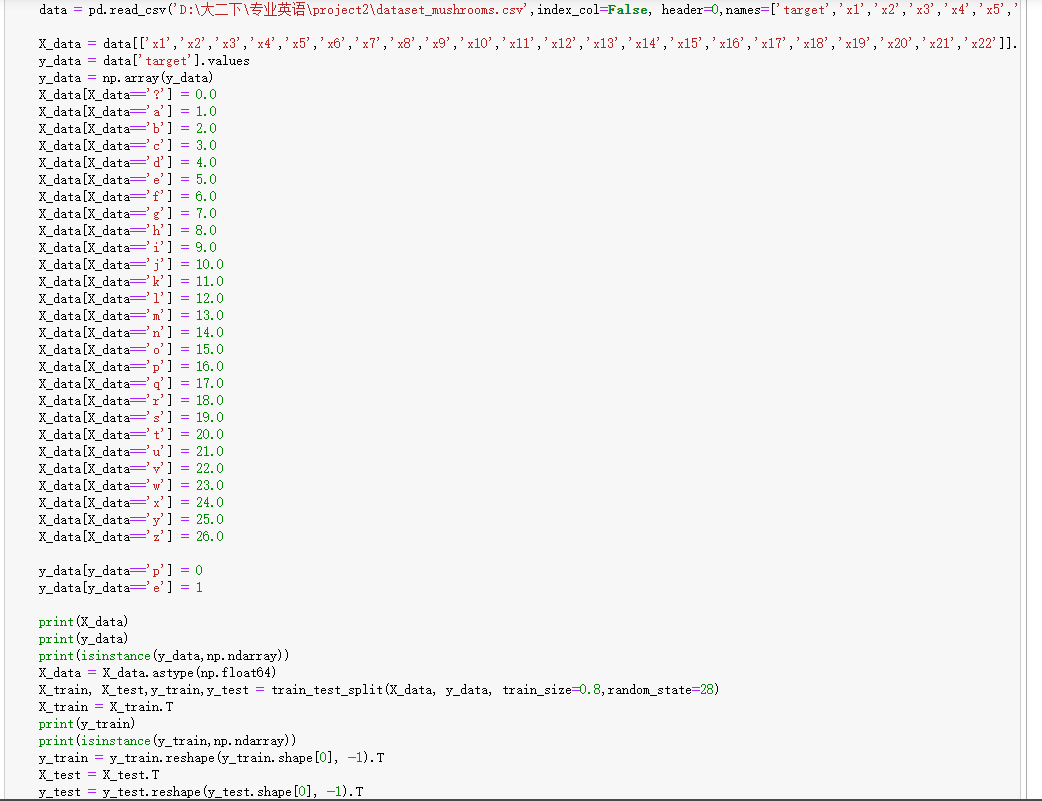
预测准确率约为97.3%



1. **构建过程**

**1.数据预处理**

首先是在main函数中从csv中读取数据。mushrooms数据集描述了蘑菇的22个属性，大体上可分为有毒和可食用。在数据预处理过程中，我将蘑菇的22个属性分别贴标签为x1,x2,……,x22，并将表示这些属性的字母替换为不同的数字；蘑菇是否可食用是我们需要预测的结果，用0表示有毒，1表示可食用。



数据读取我使用了pandas库的pd.read\_csv。pd.read\_csv('D:\大二下\专业英语\project2\dataset\_mushrooms.csv',index\_col=False,header=0,names=['target','x1','x2','x3','x4','x5','x6','x7','x8','x9','x10','x11','x12','x13','x14','x15','x16','x17','x18','x19','x20','x21','x22'],engine='python') 表示读取本地路径的文件并且将列索引改为’target’,’x1’~’x22’。

接下来将数据集划分为输入、输出两个部分，分别表示为X\_data,y\_data。

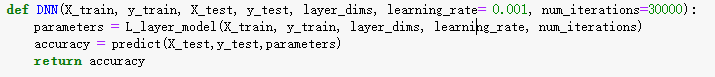
对数据进行预处理，对于X\_data，替换’?’为0.0，替换字母为不同的浮点数；对于y\_data，替换’e’为整型1，替换’p’为整型0。

最后使用函数train\_test\_split将数据划分为训练集和测试集，并做转置和reshape操作，便于后续的处理。

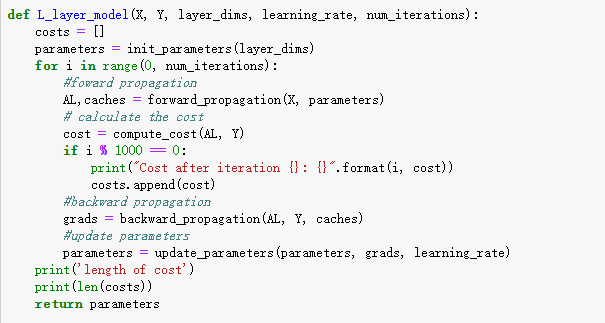
**2.框架**

在main函数中调用DNN(X\_train,y\_train,X\_test,y\_test,[X\_train.shape[0],10,5,1])

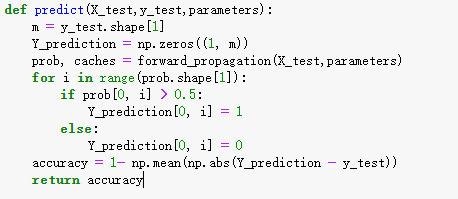
DNN模型如下：



L\_layer\_model函数负责对模型的训练，训练过程包括参数初始化、前向传播过程、计算代价、反向传播过程、参数更新。



predict函数负责测试训练集。根据前向传播过程进行预测，概率大于0.5预测为可食用，概率小于0.5预测为有毒。最后返回预测的准确率accuracy。



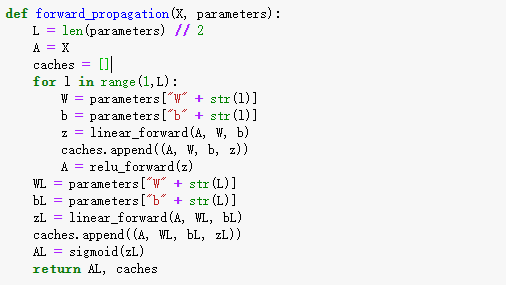
**3.前向传播**

forward propagation的过程首先是随机初始化参数w,b，在函数init\_parameters中完成，将初始化之后的parameters作为参数传入forward\_propagation函数。

parameters["W" + str(l)] = np.random.randn(layer\_dims[l],layer\_dims[l-1])\*0.1

parameters["b" + str(l)] = np.zeros((layer\_dims[l],1))

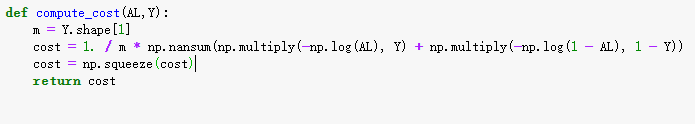
forward\_propagation函数实现如下。对于神经网络的每一层都要用线性函数linear\_forward()处理，将ReLU当做激活函数处理z,得到的A再作为下一层的输入。最后的输出层使用激活函数sigmoid函数。



**4.成本函数**

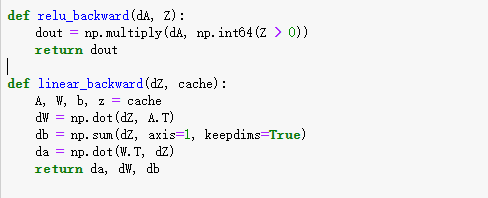
我们的目的是最小化cost function

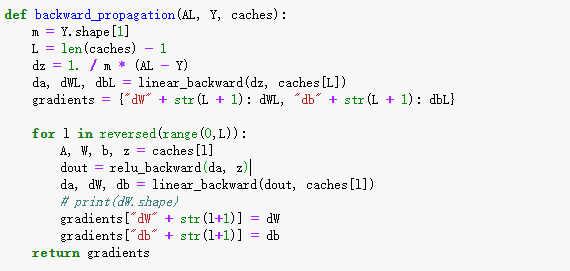
cost function使用交叉熵代价函数，代码如下。np.squeeze(cost)用于从数组的形状中删除单维条目，即把shape中为1的维度去掉。



**5.反向传播**

反向传播过程是前向传播的逆过程。对于函数linear\_forward，relu\_forward等，需要函数linear\_backward，relu\_backward进行求导。而求导得到的结果用于下一轮参数的更新。





1. **优化过程**

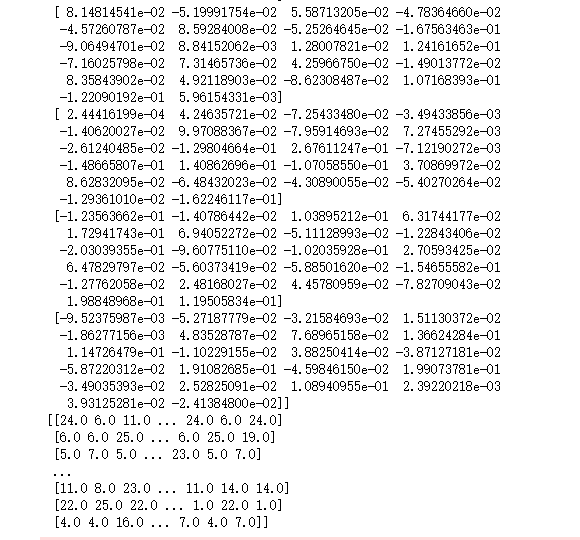
**1.有关debug**

我遇到的大部分报错都与数据的维度、类型有关。对此Jupyter Notebook提供了良好的编译环境，即使存在语法错误也可以输出之前的内容。

比如：

在我修改之前一直存在type error，但是输出矩阵可以看到是正常的可以做乘法的数据。



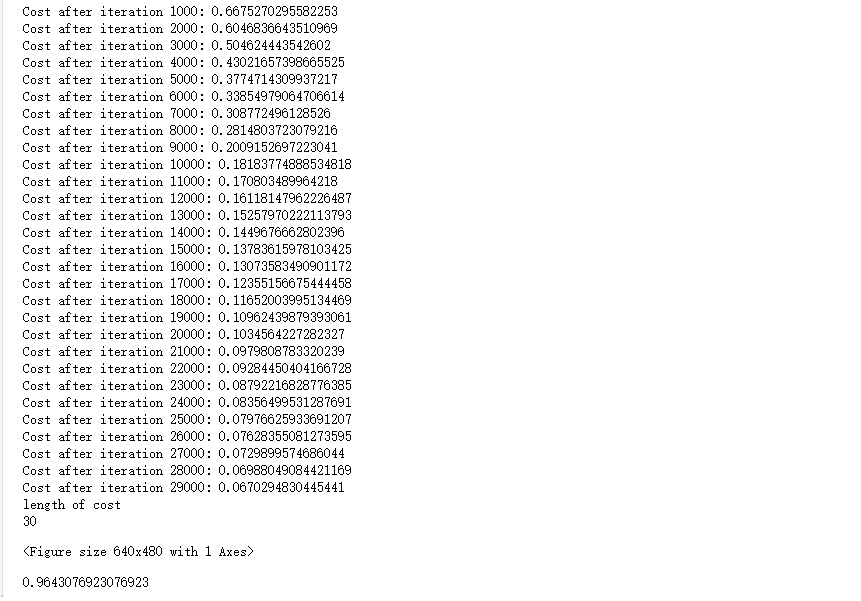


经过一步步的调试，我发现float和np.float64并不是同种类型的数据，需要做类型转换才能解决编译器的报错。

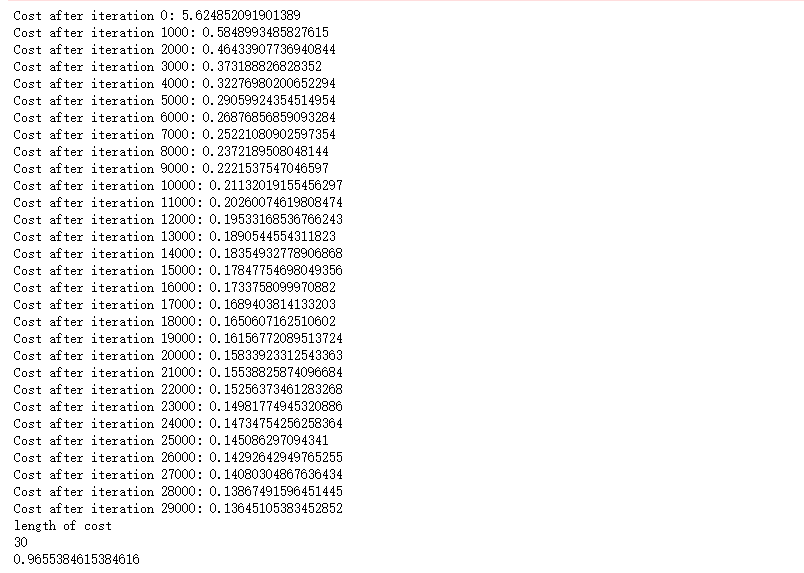
**2.参数初始化方法**

主要尝试了三种初始化方法，其中随机初始化结果需要乘以0.1，防止Z过大。

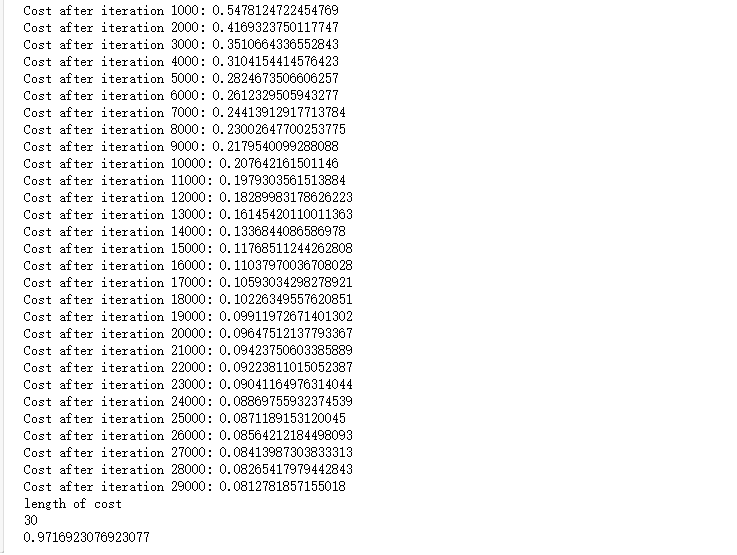
随机初始化结果：0.964



He initialization结果：0.966



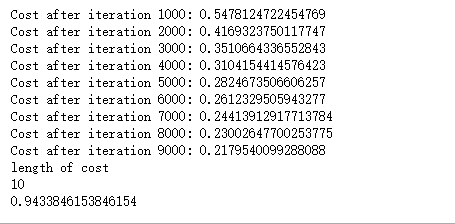
Xavier initialization结果：0.972



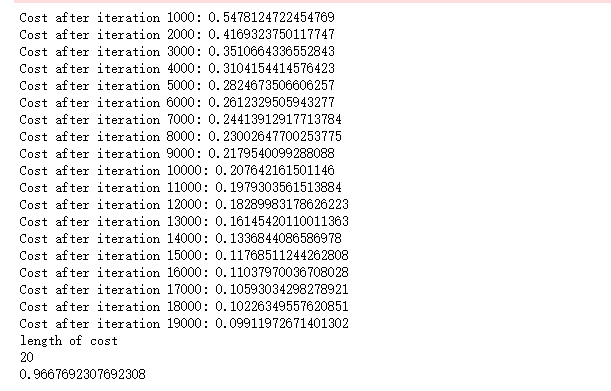
可以看到，Xavier initialization的准确率最高

**3.循环迭代次数**

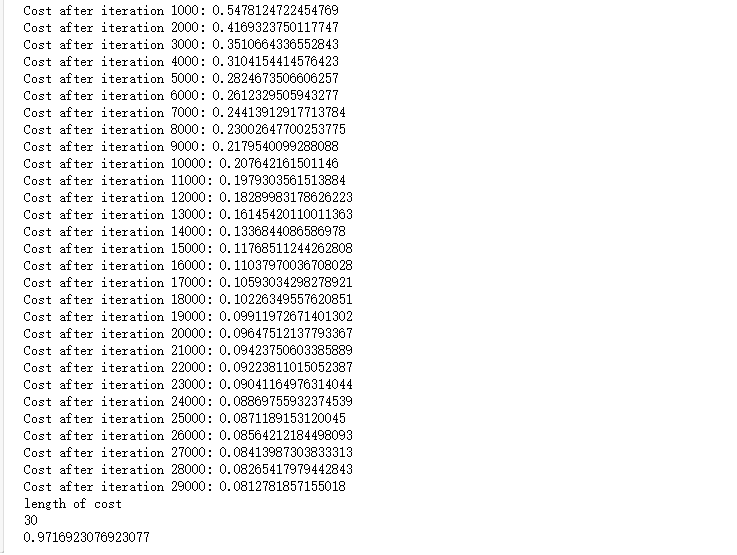
cost after iteration 10000 准确率为



cost after iteration 20000 准确率为



cost after iteration 30000 准确率为



cost after iteration 40000 准确率为



显然循环次数越多，cost越小，预测准确率越高，相应的花费时间也越多;但循环次数逐渐增多到一定程度准确率提升的效果却不大。需要在准确率和运行时间权衡下选择合适的循环次数。

1. **心得体会**

(1)神经网络的计算量一般很大，加上数据集规模较大，代码运行时间将会非常长。当然python的numpy库已经帮我们解决了一部分的计算量。比如线性函数中z=wx+b的计算，如果使用循环计算需要写成：

z=0

​for i in range(n\_x)

z+=w[i]\*x[i]

z+=b

如果一个线性函数的计算就需要一个循环，前向传播过程又需要更新系数，在外面还需要套一个迭代的循环，神经网络的代码将会非常复杂，代码运行也会耗费很长时间。为此python的numpy库提供了向量化计算和广播的机制，简化了计算。使用z=np.dot(w,x)+b即可。

(2)反向传播过程需要相关的数学知识。从输出层到隐藏层到输入层反向求导，应用了微分公式和链式法则。

(3)初始化需要随机初始化或者其他初始化方法。如果系数都初始化为0，隐含层开始计算同一个函数，一次迭代后同样的表达式结果仍然是相同的。这样多次之后的迭代就没有意义了。

(4)我使用的模型是深层神经网络DNN。深层的网络隐藏单元数量相对较少，隐藏层数目较多。比如一个三层的神经网络，第一层用**ReLU**激活函数，第二层用**ReLU**激活函数，第三层则是**sigmoid**函数（应用于二分类）；向后迭代进行反向传播求导来求dw3，db3 ，dw2 ，db2 ，dw1 ，db1。在计算的时候，缓存会把z1,z2,z3 传递过来，然后回传da2，da1 ，可以用来计算da0，但不需要使用它。前向传播只需要一次显式**for**循环，l从1到L，然后一层接着一层去计算。

(5)使用梯度下降算法找全局最优解，也就是代价函数（成本函数）这个凸函数的最小值点。迭代的过程也就是根据导数更新参数的过程。

(6)在整个代码实现中应该多用reshape，把不方便用于计算的一维数组转化为列向量或行向量。