第3次作业报告

SA19225404吴语港

1. **尝试对3.5节中的解决方案做如下两种修改。验证修改后的方案，打印出验证集和训练集上的学习曲线；然后，观察分析学习曲线，找到最佳的Epoch取值；最后，设置最佳Epoch值后，在训练集上从头开始训练一个模型，评估该训练好的模型在测试集上的性能。**

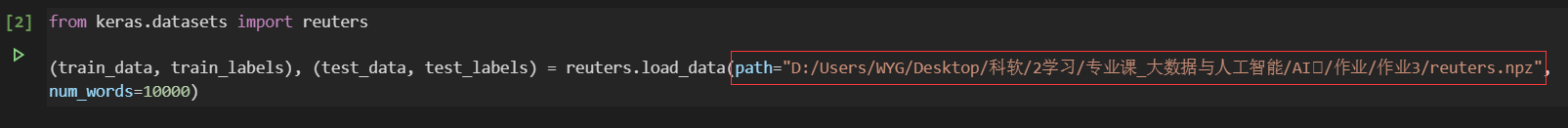
**1）只修改隐藏层的单元数。尝试使用更多或更少的隐藏单元，比如32个、128个等。**

答：

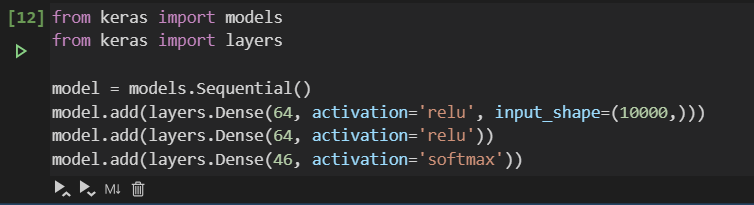
**64个隐藏单元:（**epochs=20**）**

代码的截图：

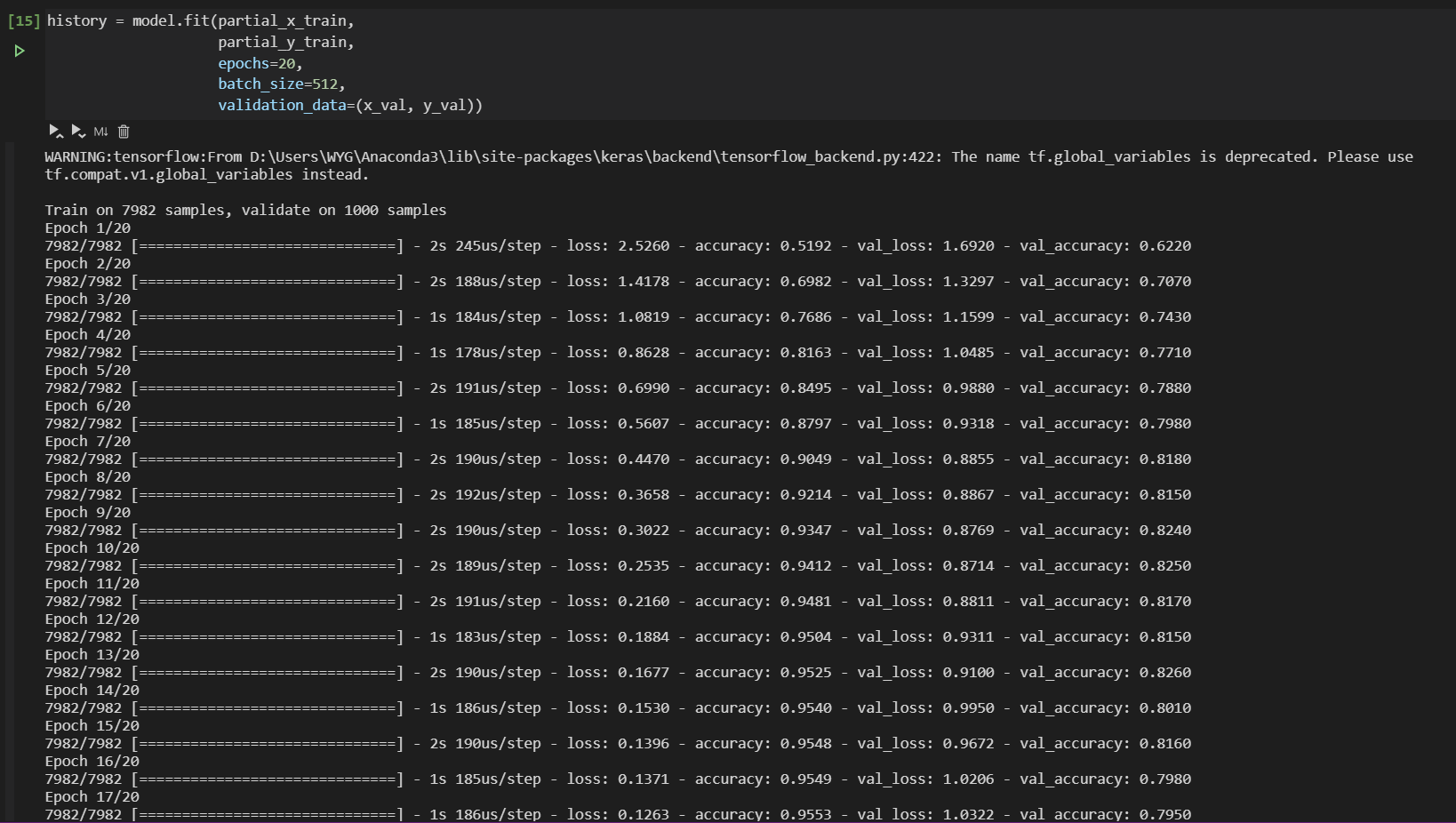
由于官网下载速度太慢了，拷贝了本地的数据集放在源程序文件夹处，并在load数据时注明本地path路径



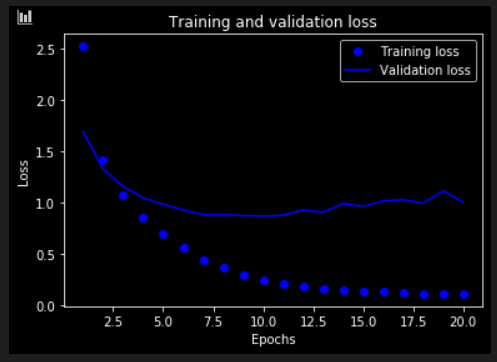
具有两层的神经网络，隐藏层64个神经元，激活函数为relu，层与层之间连接采用全连接dense。



开始了20轮次的神经网络训练



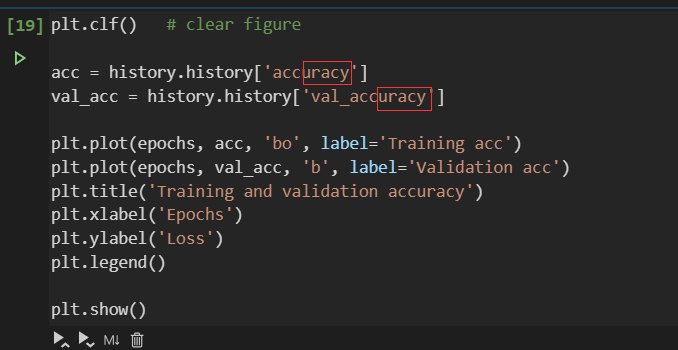
学习曲线的截图：

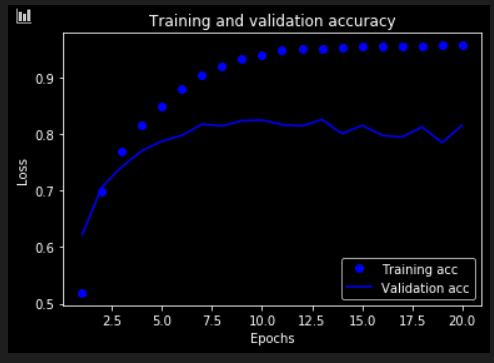


有点稍微过拟合，不是很明显

测试集上的性能:

程序源码出现问题，经过查找发现是history的参数不应该缩写为acc而应该是accuracy



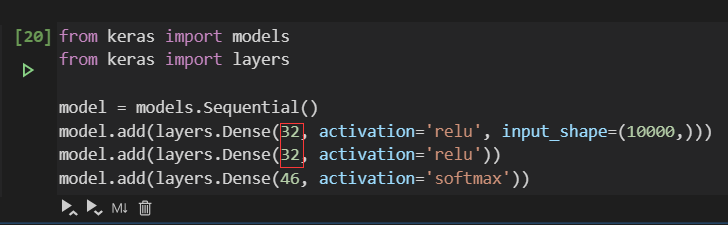


验证集的准确度大概80%左右，最佳epochs大概是10左右

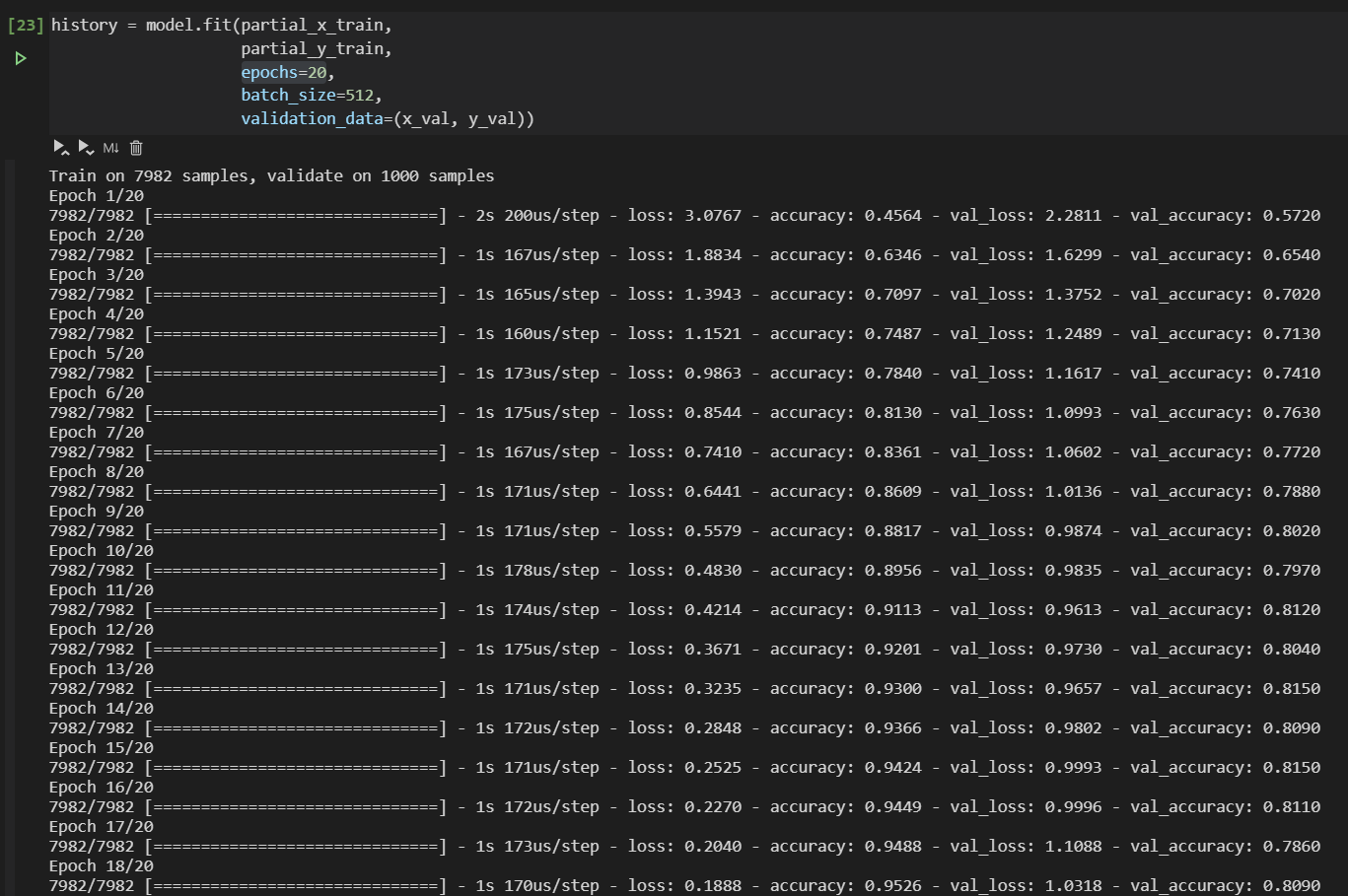
**32个隐藏单元: （**epochs=20**）**

代码的截图：

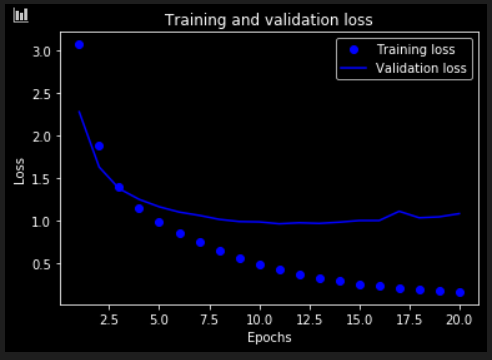
修正了第一个参数（隐藏层神经元个数）为32



开始训练

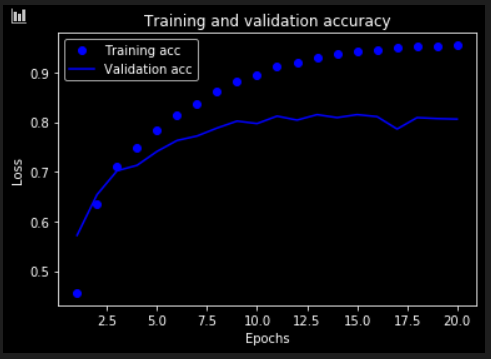


学习曲线的截图：



测试集的效果好了一丢丢

测试集上的性能:

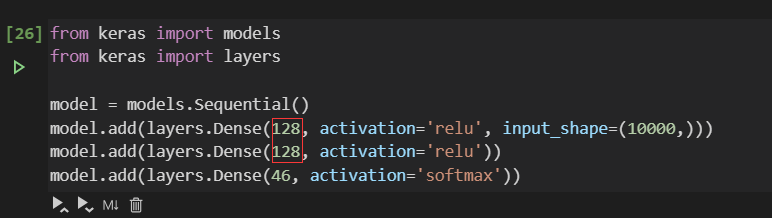


准确度小幅度上升，最佳epochs大概15左右

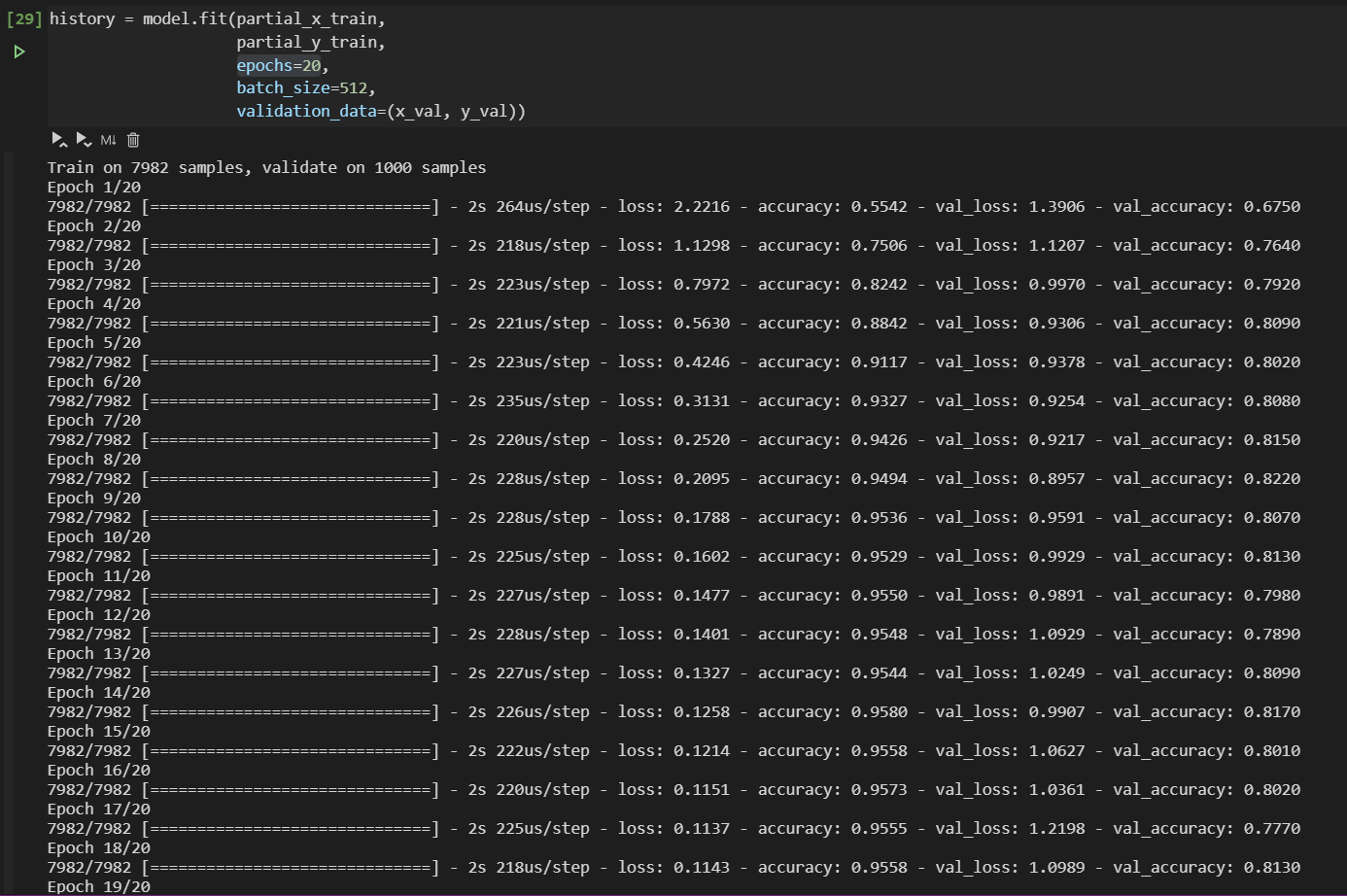
**128个隐藏单元: （**epochs=20**）**

代码的截图：

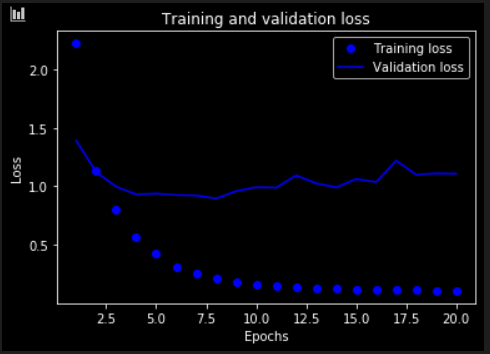
隐藏层神经元个数改为128个



开始训练

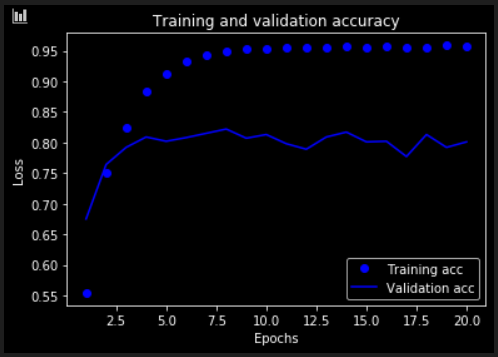


学习曲线的截图：



有轻微过拟合，不过最佳性能好了一些

测试集上的性能:



最佳epochs大概在7.5左右，最佳性能有小幅度下降

对截图中的性能评测结果进行分析，说明其原因。（提示：对比分析：修改后，在测试集上的Accuracy是变好了还是变坏了？为什么？）

修改后发现在一定程度上增加隐藏层神经元个数有助于捕获更多有用的信息，从而增加准确度，在测试集上的Accuracy是变好了一些，不过如果无限制增加隐藏层神经元个数会导致模型太过于复杂从而导致过拟合，影响准确度，需要不断调节这些超参数，让准确率尽可能高一些。

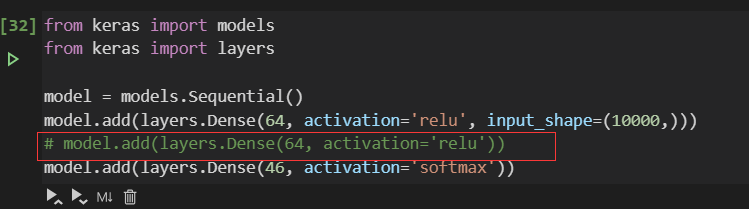
**2）只修改隐藏层的数量。前面使用了两个隐藏层，现在尝试使用一个或三个隐藏层。**

答：

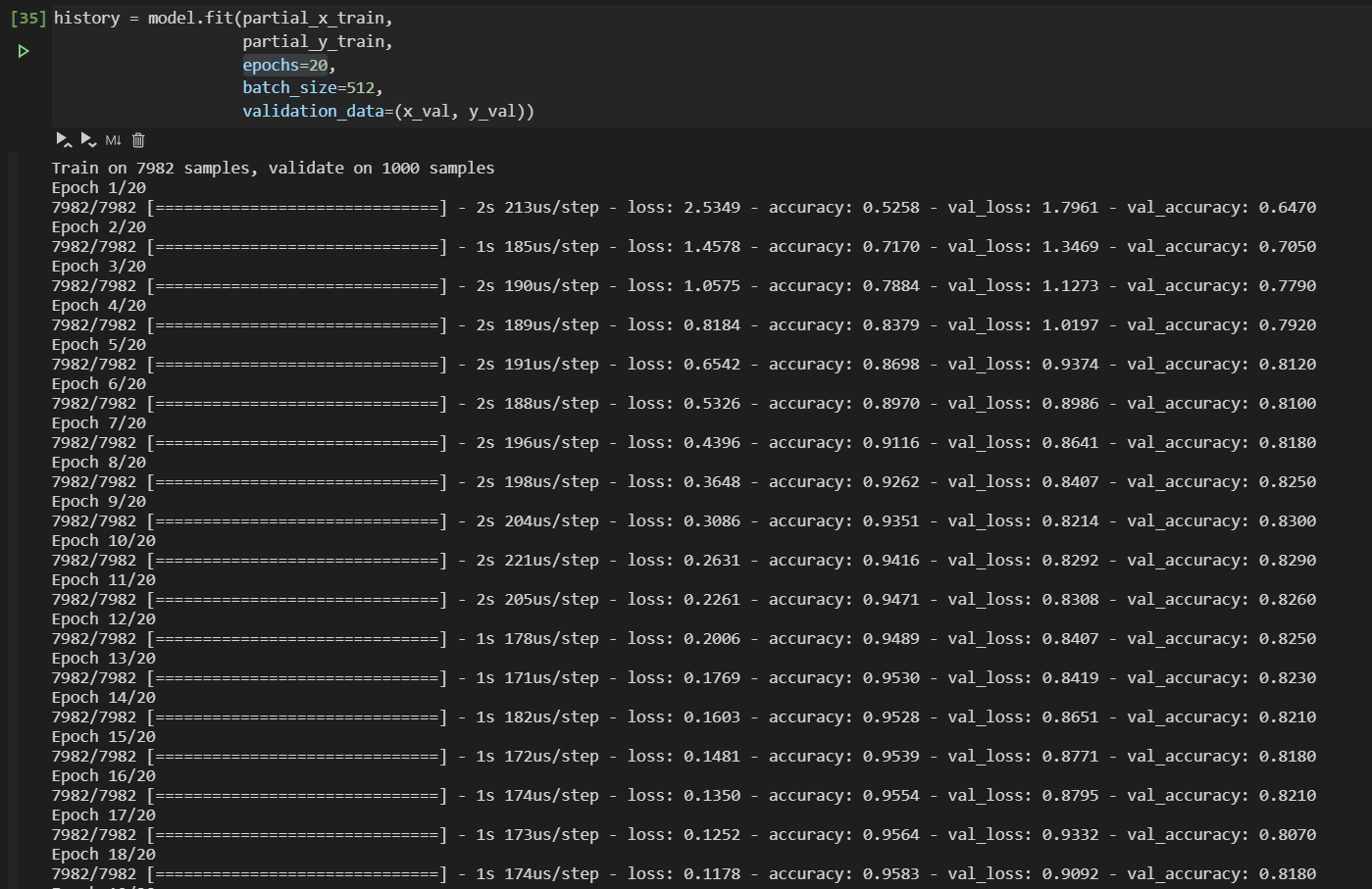
**一个隐藏层: （**epochs=20**）**

代码的截图：

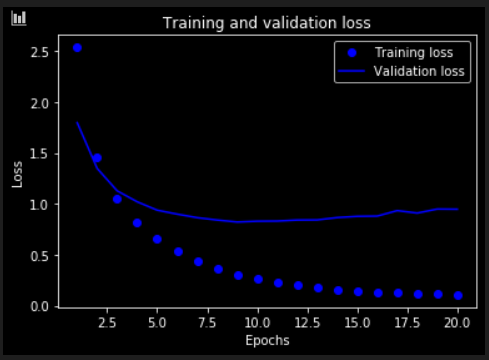
注释掉一层



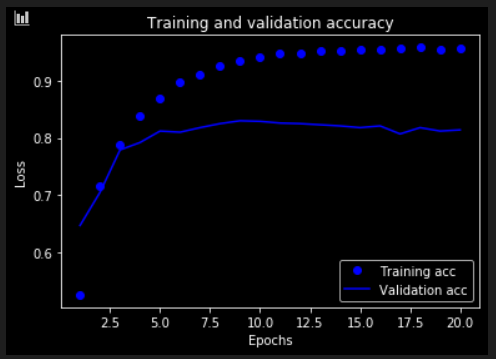
开始训练：



学习曲线的截图：



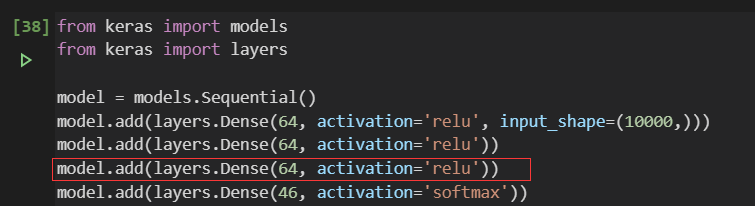
测试集上的性能:

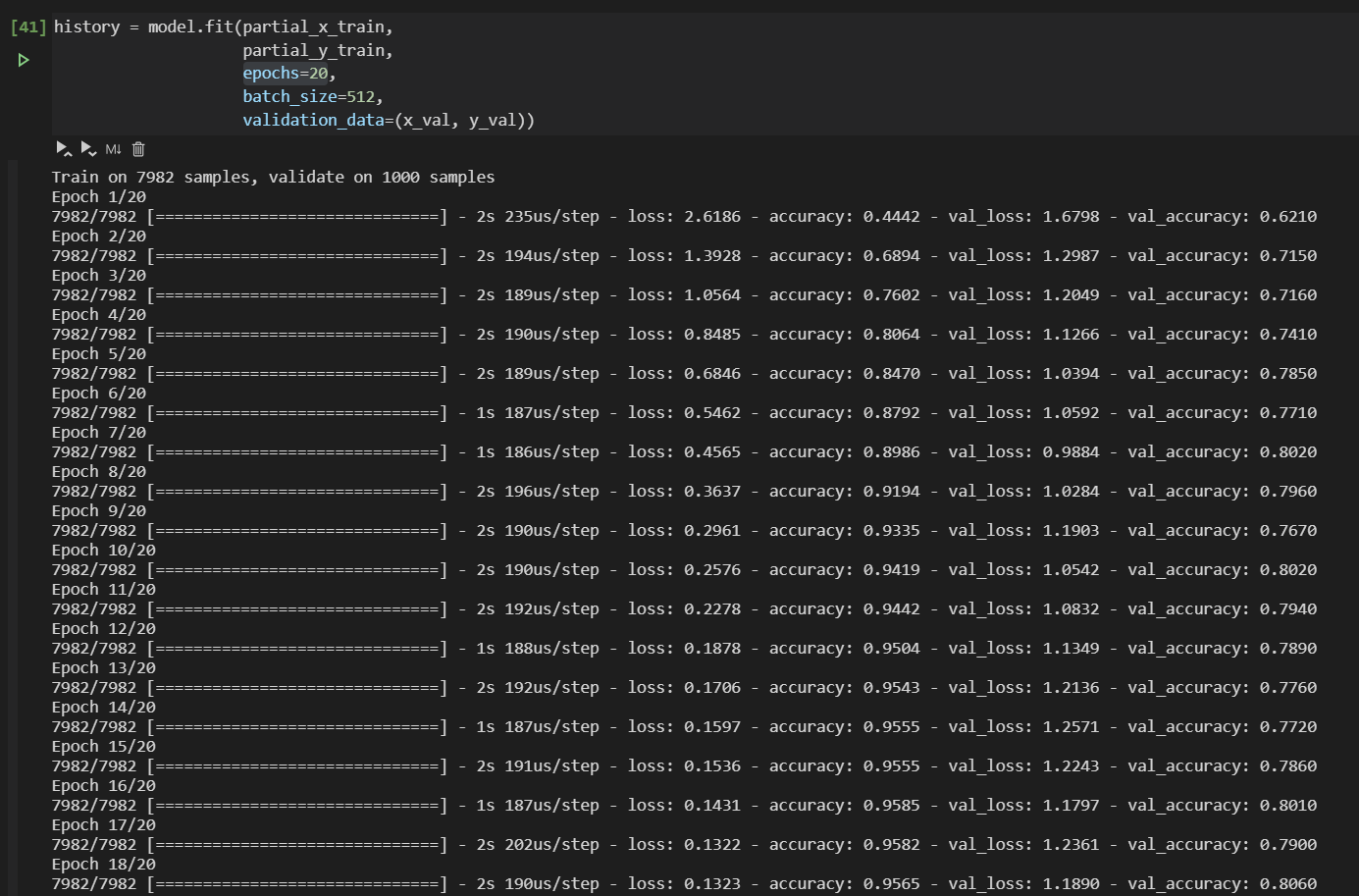


性能较两层的模型有小幅度上升，最佳epochs大概为10

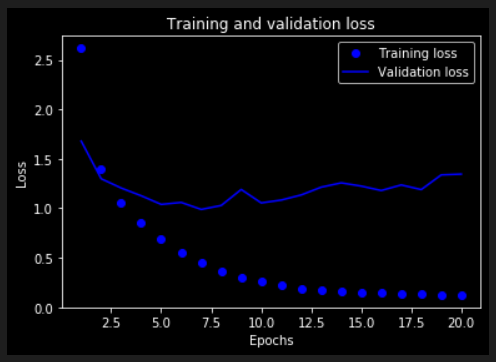
**三个隐藏层: （**epochs=20**）**

代码的截图：

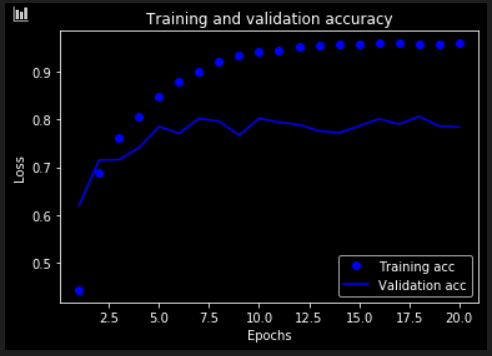




学习曲线的截图：



测试集上的性能:



性能较两层的模型有小幅度上升，最佳epochs大概为10

代码的截图。（代码修改之处用红色框圈出来。）

运行结果的截图。包含学习曲线的截图 和 测试集上的性能评测结果的截图。

对截图中的性能评测结果进行分析，说明其原因。（提示：对比分析：修改后，在测试集上的Accuracy是变好了还是变坏了？为什么？）

修改后发现在虽然做实验感觉一定程度上增加隐藏层个数有助于捕获更多有用的信息，从而增加准确度，但是做完了这个实验发现，其实一层的模型在测试集上的Accuracy居然更好了一些，可能是增加隐藏层个数会导致模型太过于复杂从而导致过拟合，影响准确度，需要不断调节这些超参数，找到最佳层数，让准确率尽可能高一些。

1. **尝试对3.4节中的解决方案做如下三种修改。验证修改后的方案，打印出验证集和训练集上的学习曲线；然后，观察分析学习曲线，找到最佳的Epoch取值；最后，设置最佳Epoch值后，在训练集上从头开始训练一个模型，评估该训练好的模型在测试集上的性能。**
   1. **只修改隐藏层的数量。前面使用了两个隐藏层。你可以尝试使用一个或三个隐藏层，然后观察对验证精度和测试精度的影响。**

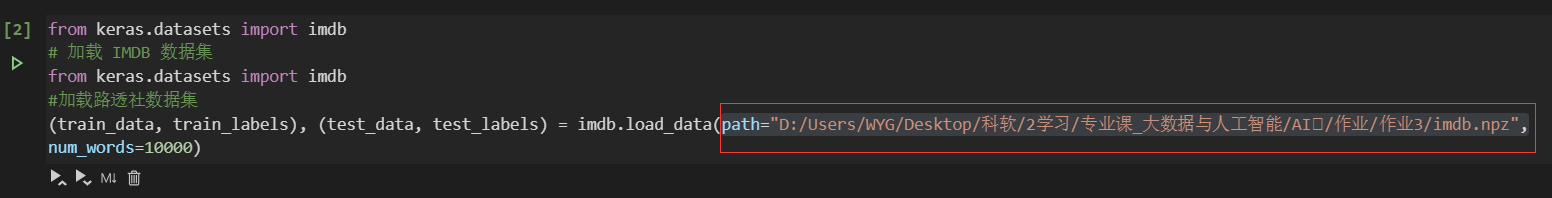
答：

**两个隐藏层：（**epochs=20**）**

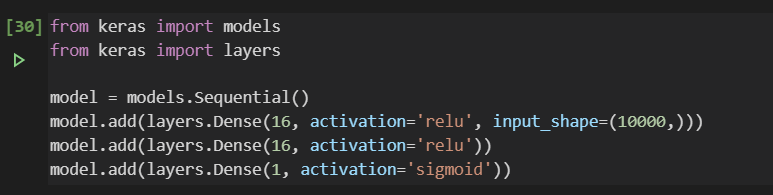
代码的截图：

神经网络

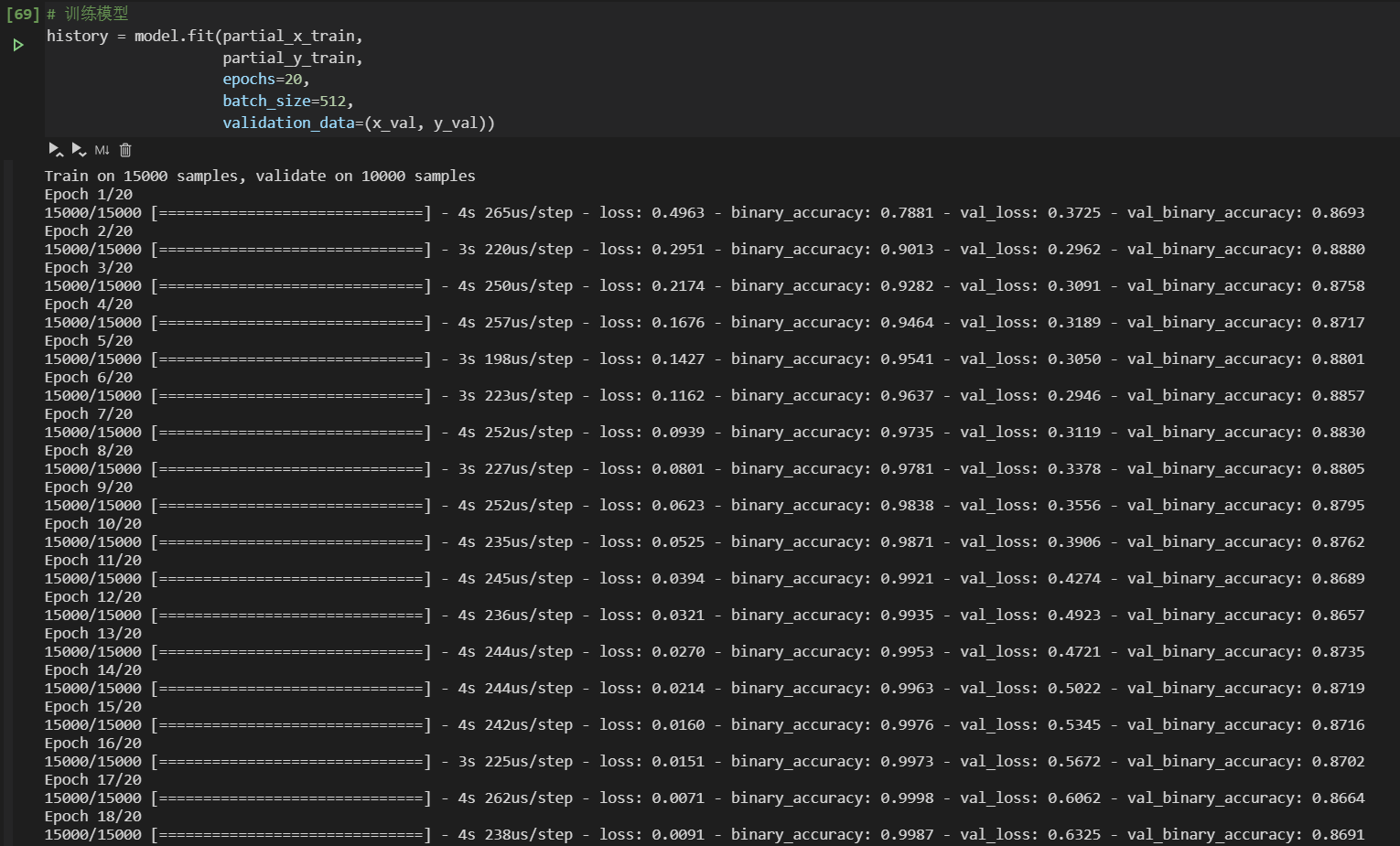
由于官网下载速度太慢了，拷贝了本地的数据集放在源程序文件夹处，并在load数据时注明本地path路径



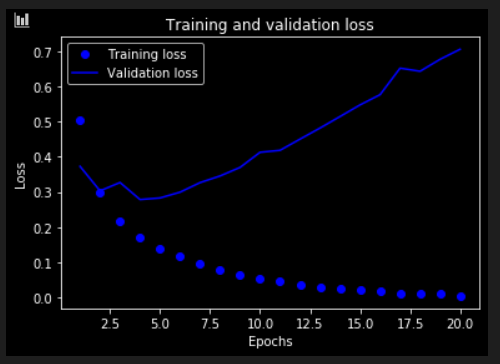
具有两层的神经网络，隐藏层16个神经元，激活函数为relu，层与层之间连接采用全连接dense。



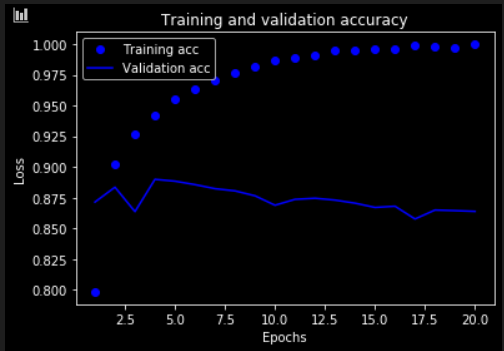
训练模型参数



学习曲线的截图：



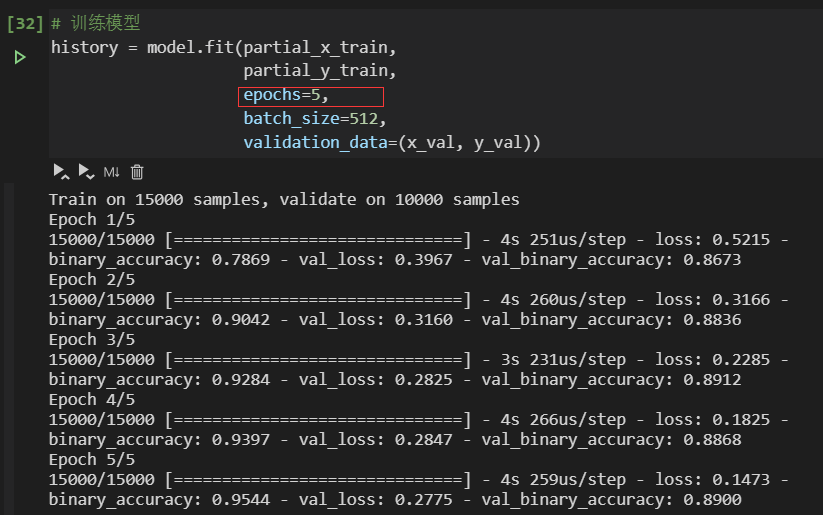
测试集上的性能评测结果的截图：



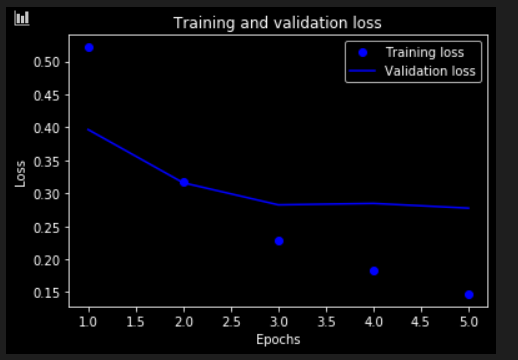
出现了过拟合，现在提前终止，，最佳epochs大概为5，改epochs=5

**两个隐藏层：（**epochs=5**）**

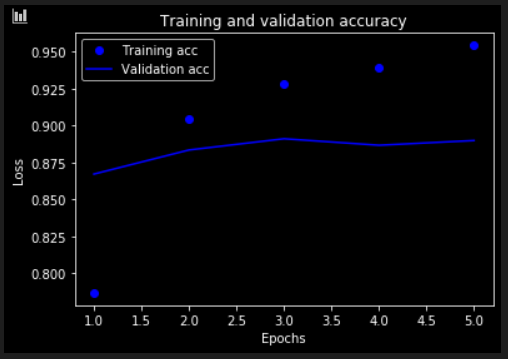
代码的截图：



学习曲线的截图：



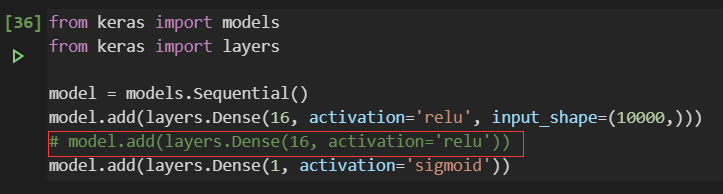
测试集上的性能评测结果的截图：

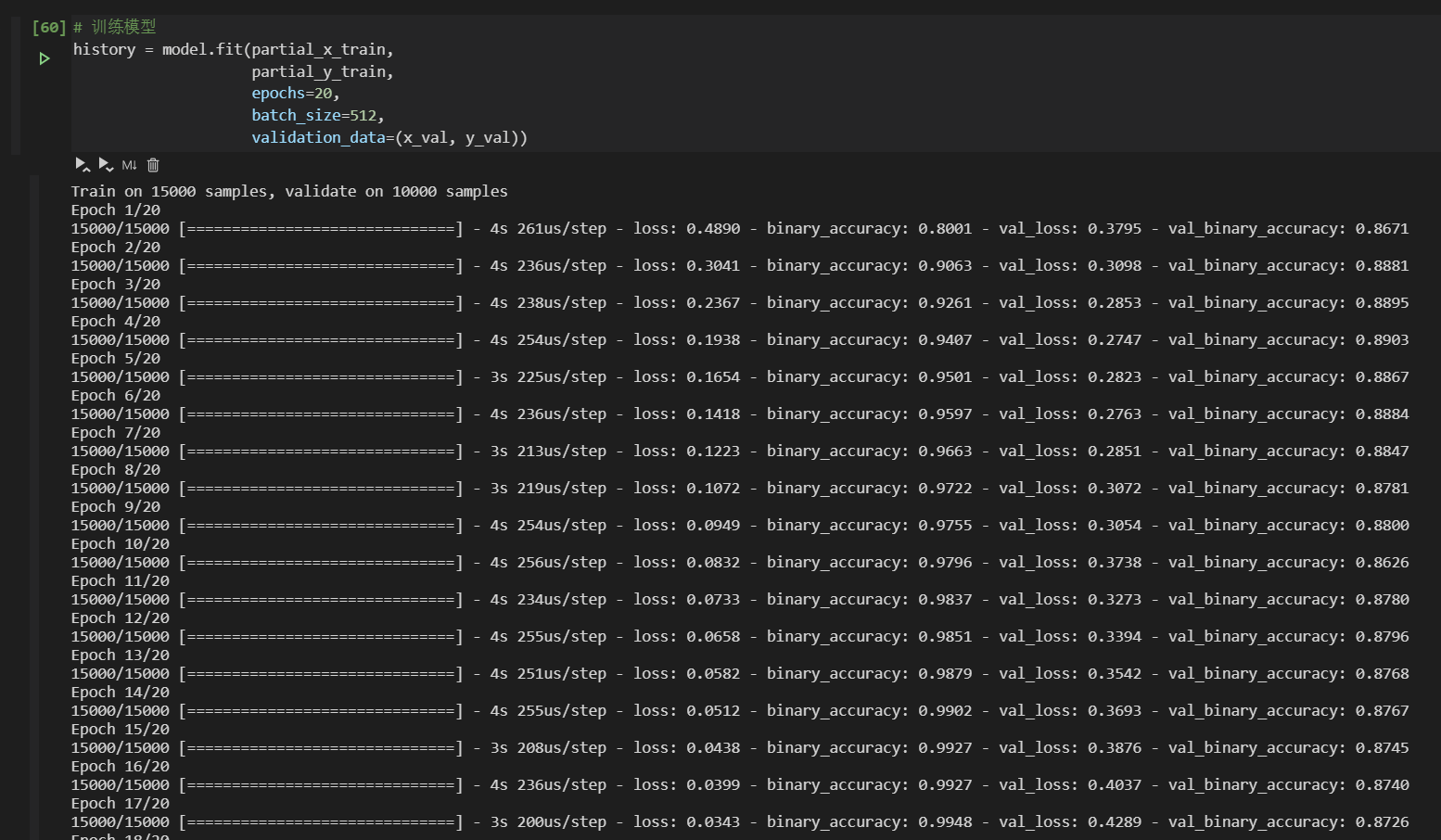


提前终止是应对过拟合的一个好办法，既提高了准确率，又节省了计算量

**一个隐藏层：（**epochs=20**）**

代码的截图：

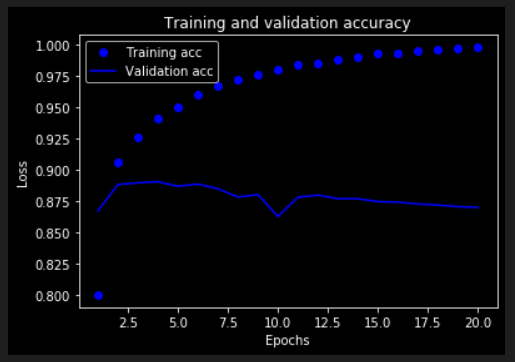




学习曲线的截图：



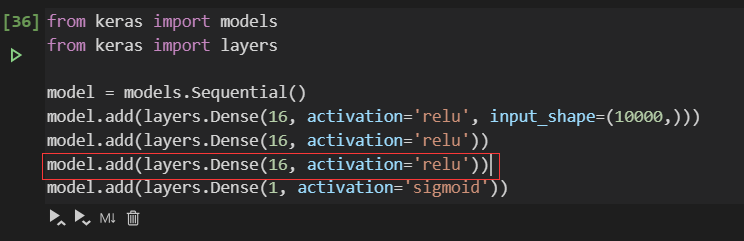
测试集上的性能评测结果的截图：

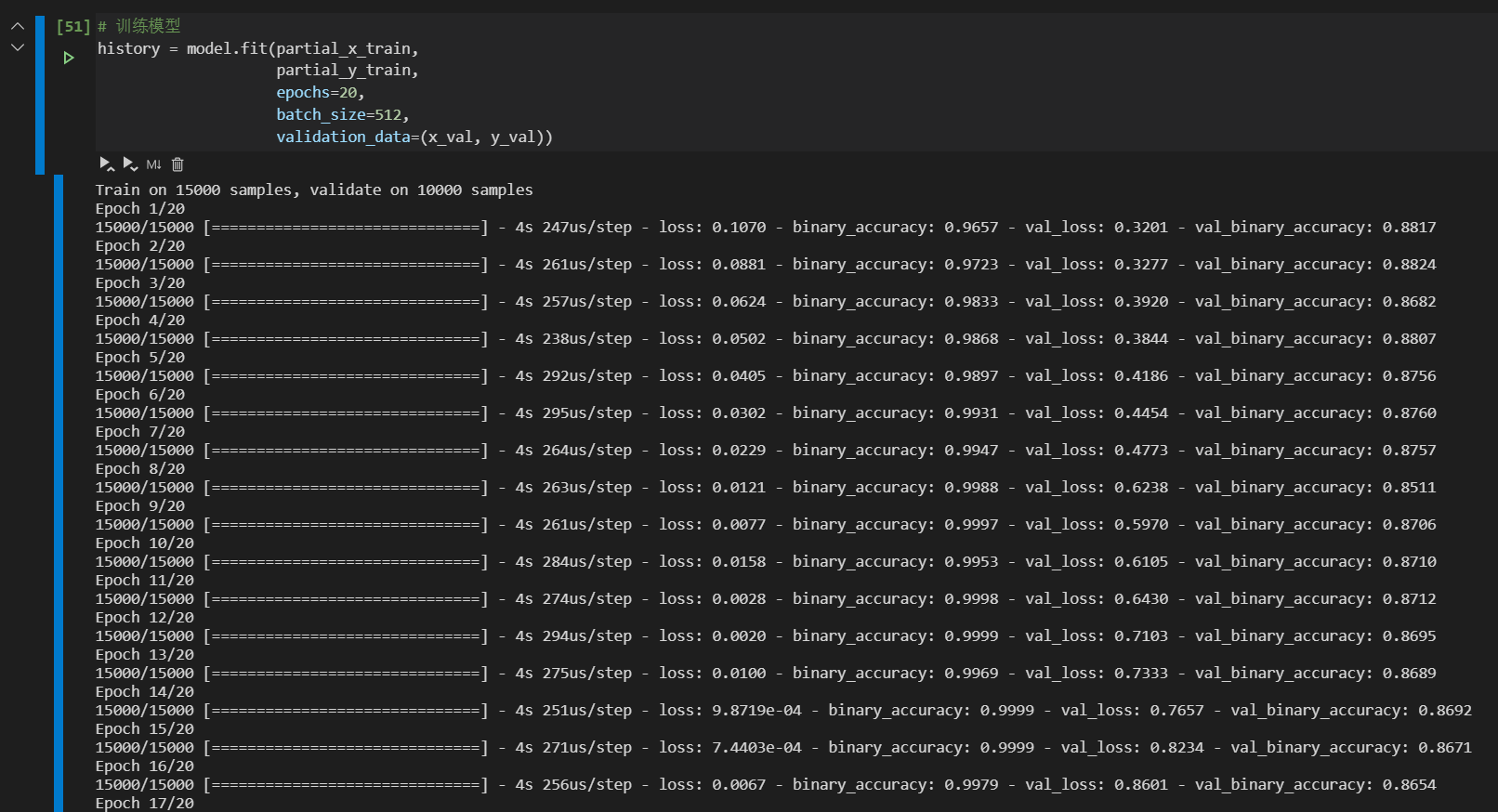


过拟合好了一些，验证集的准确度也好了一些，最佳epochs大概为4

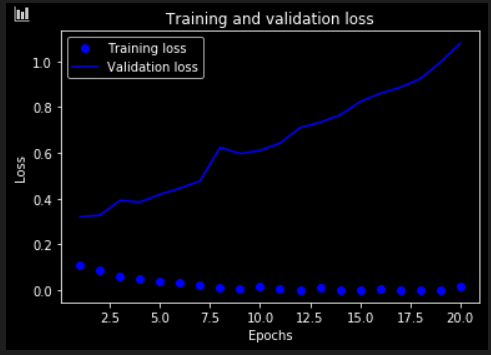
**三个隐藏层：（**epochs=20**）**

代码的截图：

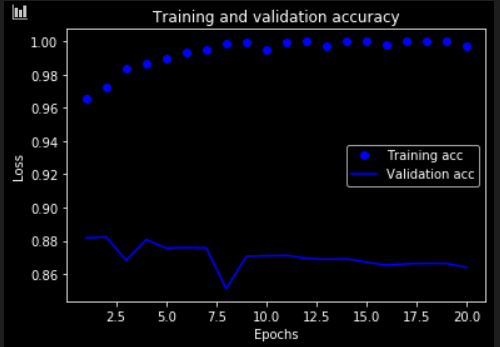




学习曲线的截图：



测试集上的性能评测结果的截图：



过拟合十分严重，准确度也十分差，最佳epochs大概为1

对截图中的性能评测结果进行分析，说明其原因。（提示：对比分析：修改后，在测试集上的Accuracy是变好了还是变坏了？为什么？）

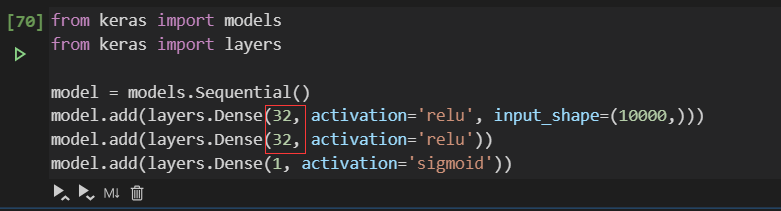
修改后发现在虽然做实验感觉一定程度上增加隐藏层个数有助于捕获更多有用的信息，从而增加准确度，但是做完了这个实验发现，其实一层的模型在测试集上的Accuracy居然更好了一些，可能是增加隐藏层个数会导致模型太过于复杂从而导致过拟合，影响准确度，需要不断调节这些超参数，找到最佳层数，让准确率尽可能高一些。

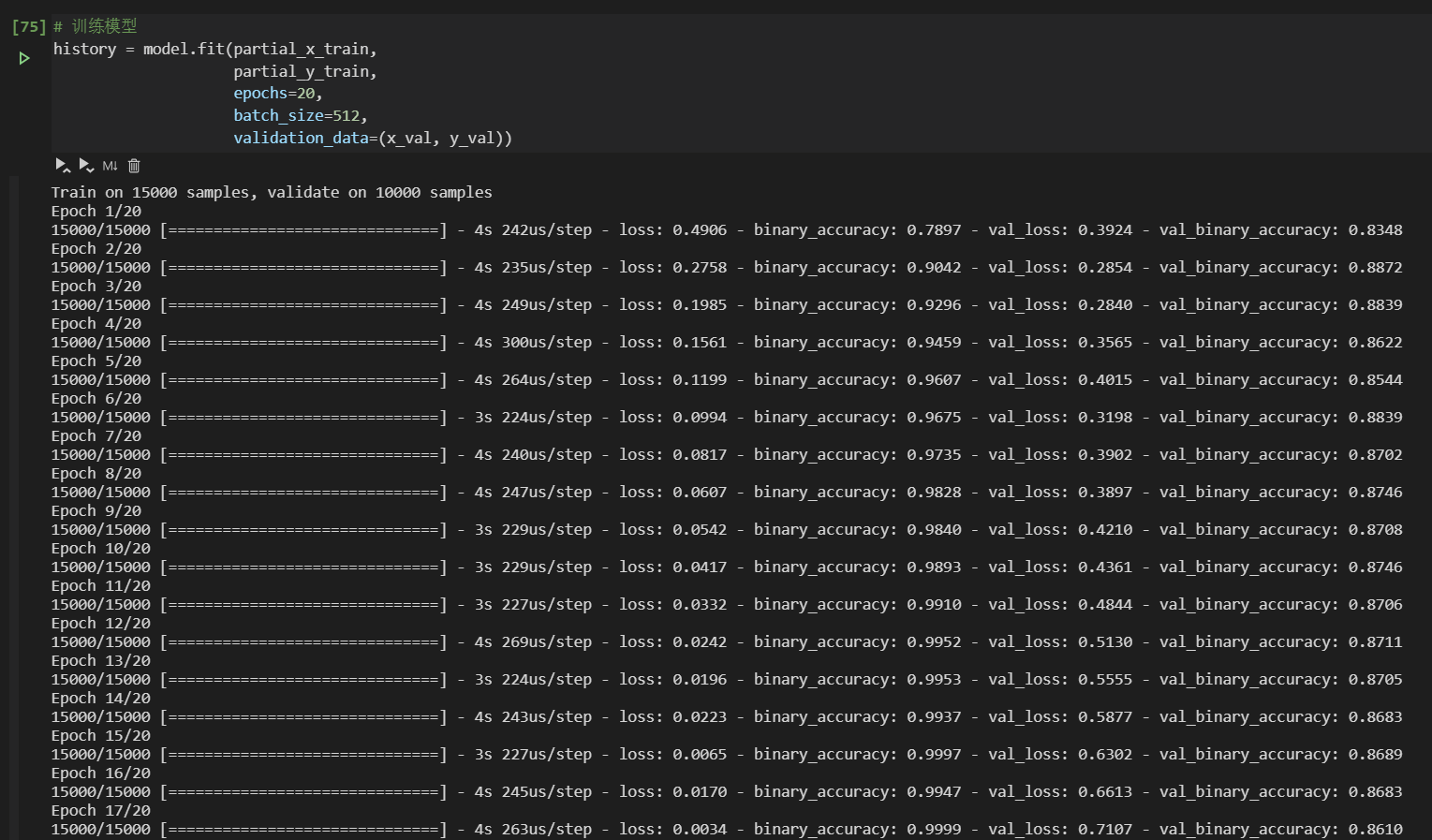
* 1. **只修改隐藏层的单元数。尝试用更多或更少的隐藏单元，比如32个、64个等。**

答：

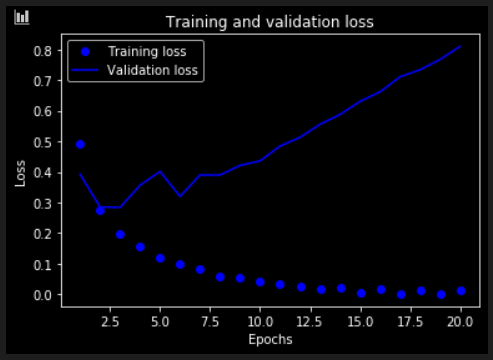
**32个隐藏单元：**

代码的截图：

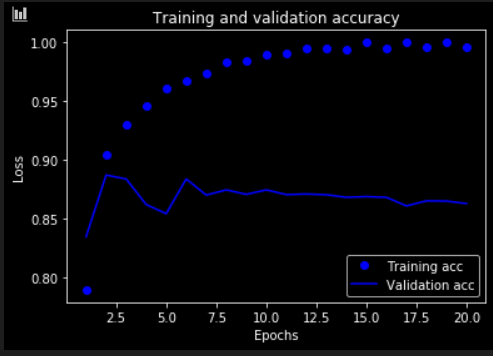




学习曲线的截图：



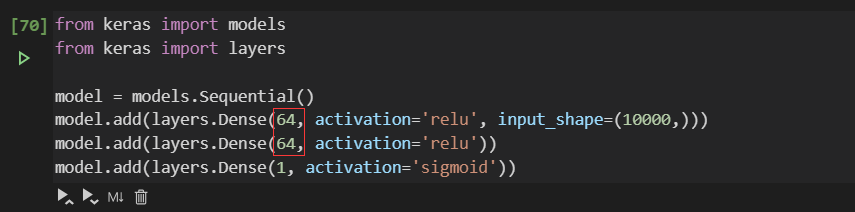
测试集上的性能评测结果的截图：

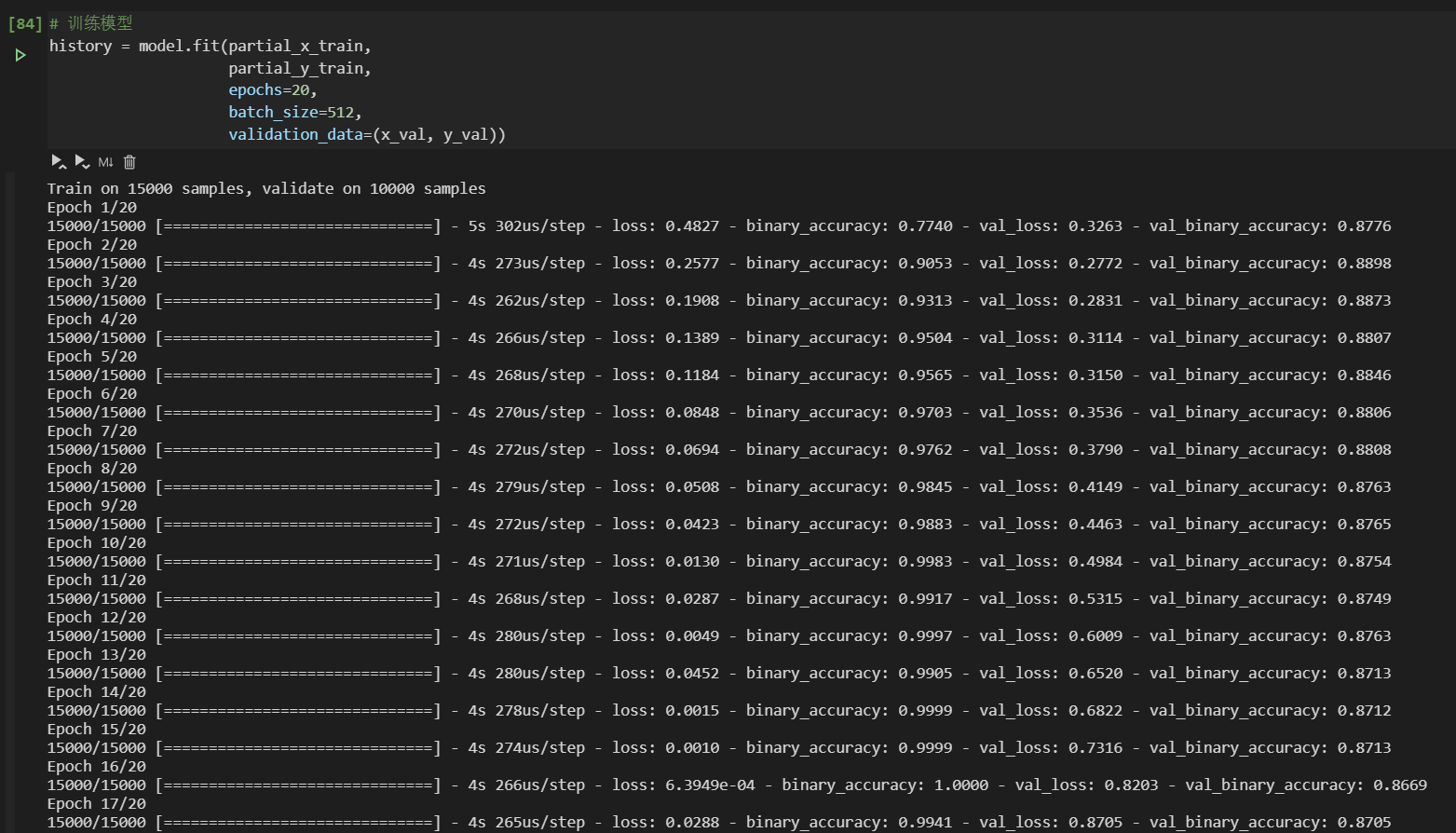


性能并没有多大改善，训练曲线有锯齿状现象，最佳epochs大概为2

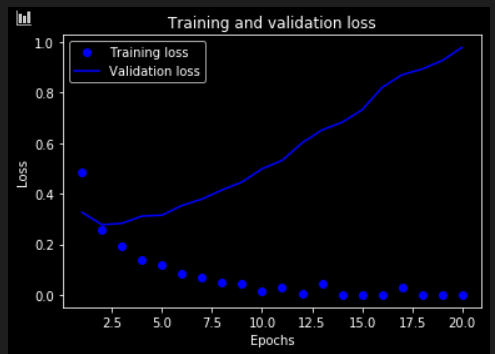
**64个隐藏单元：**

代码的截图：

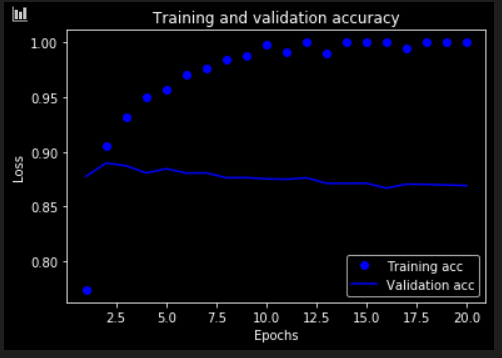




学习曲线的截图：



测试集上的性能评测结果的截图：



性能变化不大，最佳epochs大概为2，训练曲线有锯齿状现象消失了，神奇，是因为神经元多，考虑周全了吗。

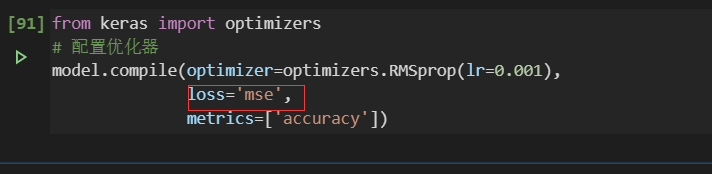
对截图中的性能评测结果进行分析，说明其原因。（提示：对比分析：修改后，在测试集上的Accuracy是变好了还是变坏了？为什么？）

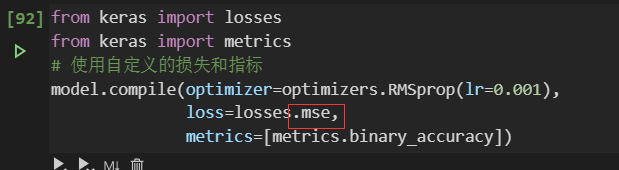
修改后发现在一定程度上增加隐藏层神经元个数有助于捕获更多有用的信息，从而增加准确度，在测试集上的Accuracy是变好了一些，不过如果无限制增加隐藏层神经元个数会导致模型太过于复杂从而导致过拟合，影响准确度，需要不断调节这些超参数，让准确率尽可能高一些。

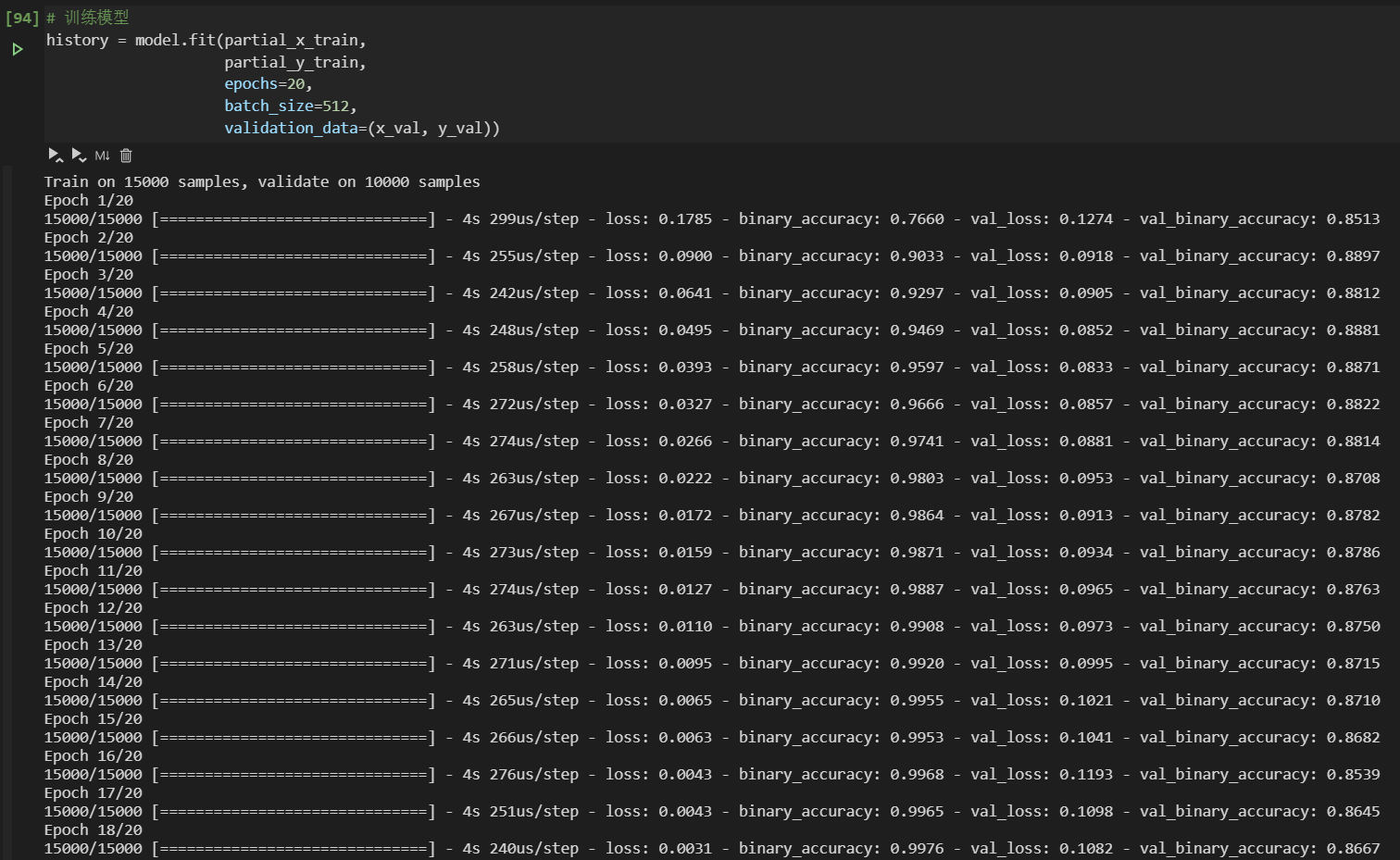
* 1. **只修改损失函数。尝试使用mse损失函数代替binary\_crossentropy。**

答：

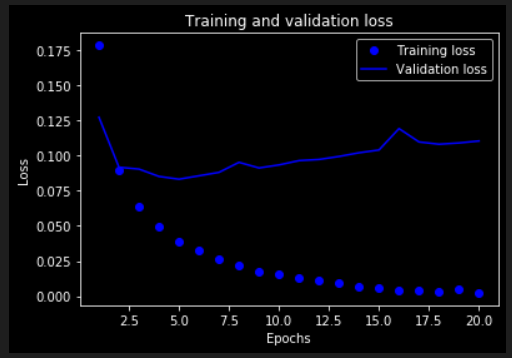
代码的截图：



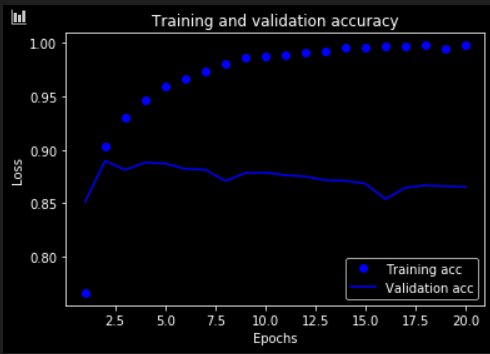




学习曲线的截图：



测试集上的性能评测结果的截图：



验证集的预测准确度下降了，最佳epochs大概为3

对截图中的性能评测结果进行分析，说明其原因。（提示：对比分析：修改后，在测试集上的Accuracy是变好了还是变坏了？为什么？）

常用的损失函数有均方误差：MSE

是最经典也是最简单的损失函数，几乎万能，但是不太准确。

binary\_crossentropy交叉熵损失函数，一般用于二分类：

这个是针对概率之间的损失函数，你会发现只有yi和y^i是相等时，loss才为0，否则loss就是为一个正数。而且，概率相差越大，loss就越大。这个神奇的度量概率距离的方式称为交叉熵。

categorical\_crossentropy分类交叉熵函数：

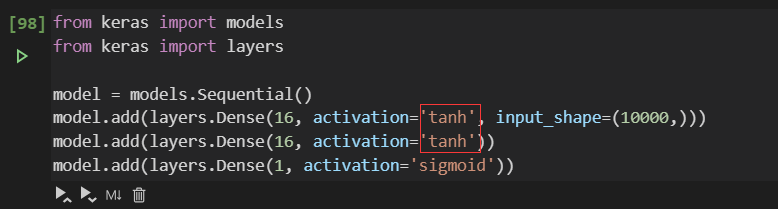
n是样本数，m是分类数，注意，这是一个多输出的loss的函数，所以它的loss计算也是多个的。

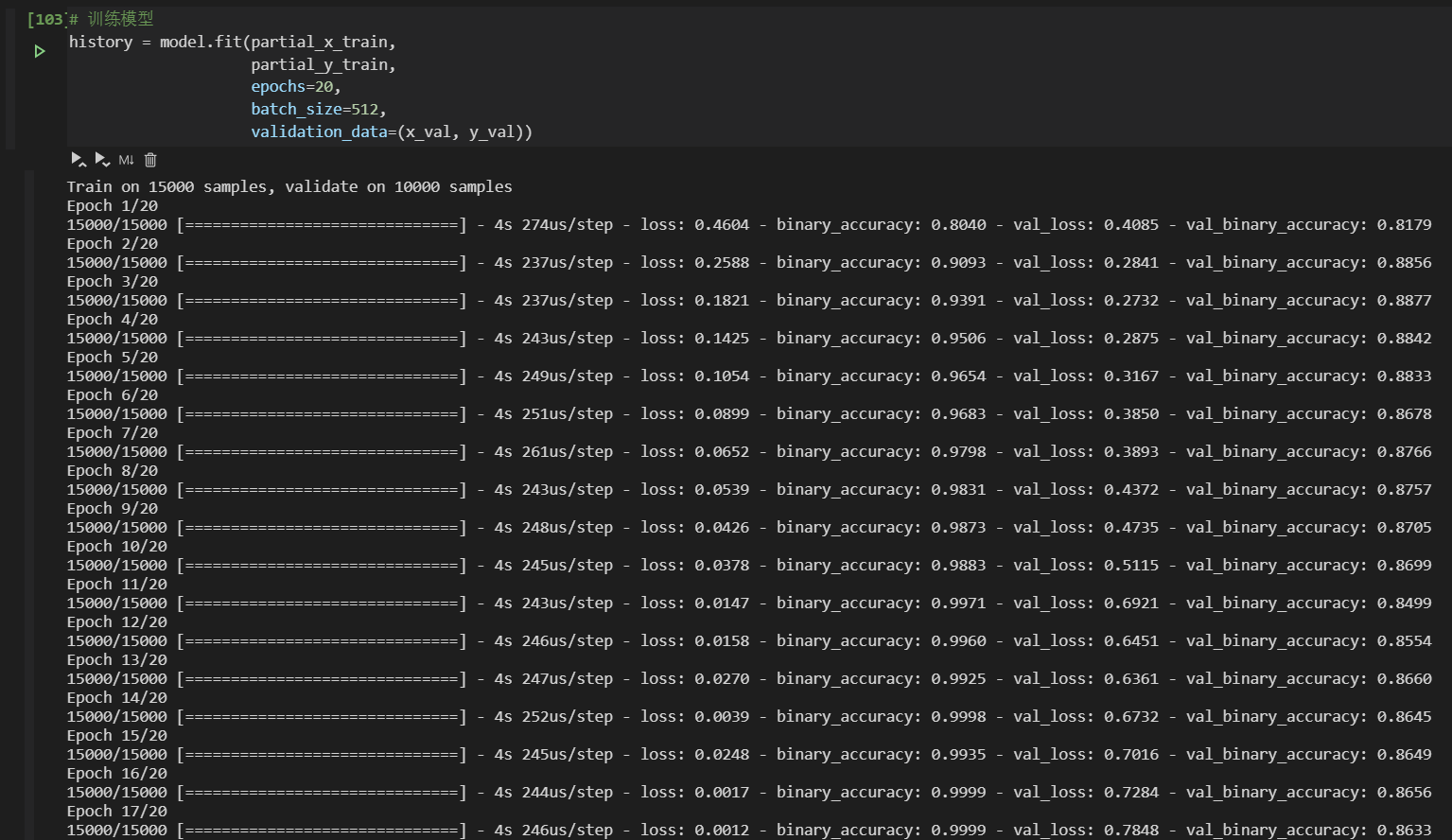
在本例中使用MSE损失函数的效果比binary\_crossentropy交叉熵损失函数的效果差了一些，看来还是要专业的损失函数对应专业的模型。

* 1. **只修改激活函数。尝试使用tanh激活（这种激活在神经网络早期非常流行）代替relu。**

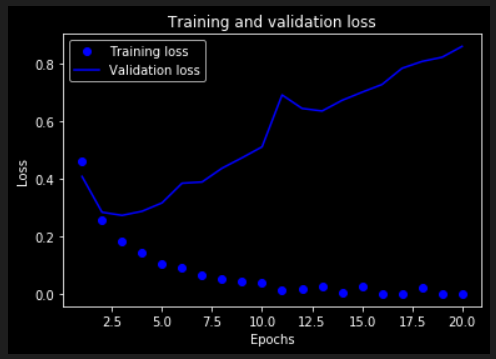
答：

代码的截图：

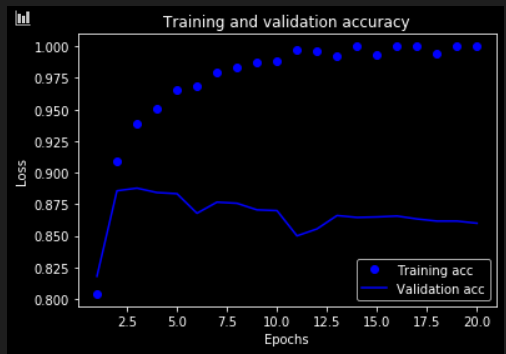




学习曲线的截图：



测试集上的性能评测结果的截图：



效果变差了，最佳epochs大概为3

对截图中的性能评测结果进行分析，说明其原因。（提示：对比分析：修改后，在测试集上的Accuracy是变好了还是变坏了？为什么？）

修改后在测试集上的效果变差了，可能是因为tanh存在梯度消失和指数计算问题

附：各种激活函数对比

早期研究神经网络主要采用sigmoid函数或者tanh函数，输出有界，很容易充当下一层的输入。

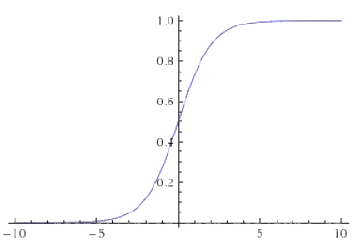
近些年Relu函数及其改进型（如Leaky-ReLU、P-ReLU、R-ReLU等）在多层神经网络中应用比较多。下面我们来总结下这些激活函数：

Sigmoid函数

Sigmoid 是常用的非线性的激活函数，它的数学形式如下：



Sigmoid的几何图像如下：



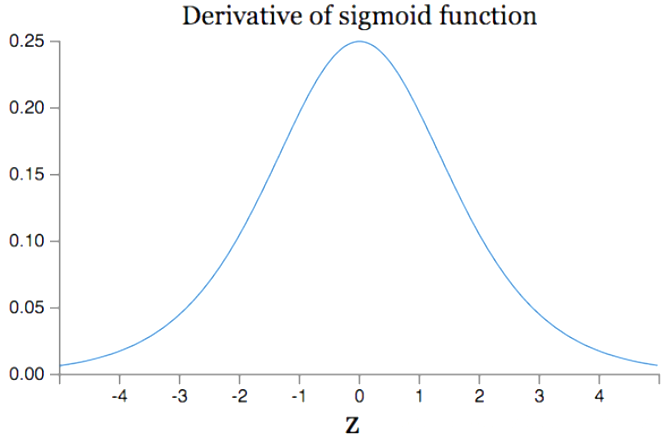
特点：

它能够把输入的连续实值变换为0和1之间的输出，特别的，如果是非常大的负数，那么输出就是0；如果是非常大的正数，输出就是1.

缺点：

sigmoid函数曾经被使用的很多，不过近年来，用它的人越来越少了。主要是因为它固有的一些 缺点。

缺点1：在深度神经网络中梯度反向传递时导致梯度爆炸和梯度消失，其中梯度爆炸发生的概率非常小，而梯度消失发生的概率比较大。首先来看Sigmoid函数的导数，如下图所示：



如果我们初始化神经网络的权值为 [0,1][0,1][0,1] 之间的随机值，由反向传播算法的数学推导可知，梯度从后向前传播时，每传递一层梯度值都会减小为原来的0.25倍，如果神经网络隐层特别多，那么梯度在穿过多层后将变得非常小接近于0，即出现梯度消失现象；当网络权值初始化为 (1,+∞)(1,+∞)(1,+∞) 区间内的值，则会出现梯度爆炸情况。

缺点2：Sigmoid 的 output 不是0均值（即zero-centered）。这是不可取的，因为这会导致后一层的神经元将得到上一层输出的非0均值的信号作为输入。 产生的一个结果就是：如x>0, f=wTx+bx&gt;0, \ f= w^Tx+bx>0, f=wTx+b,那么对w求局部梯度则都为正，这样在反向传播的过程中w要么都往正方向更新，要么都往负方向更新，导致有一种捆绑的效果，使得收敛缓慢。 当然了，如果按batch去训练，那么那个batch可能得到不同的信号，所以这个问题还是可以缓解一下的。因此，非0均值这个问题虽然会产生一些不好的影响，不过跟上面提到的梯度消失问题相比还是要好很多的。

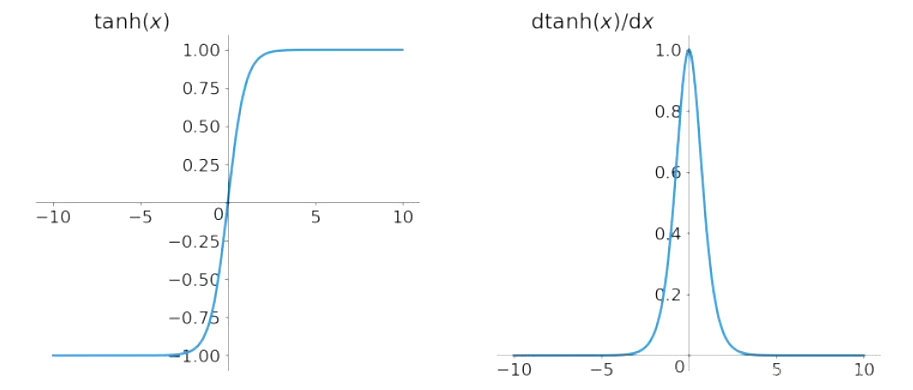
缺点3：其解析式中含有幂运算，计算机求解时相对来讲比较耗时。对于规模比较大的深度网络，这会较大地增加训练时间。

tanh函数

tanh函数解析式：



tanh函数及其导数的几何图像如下图：



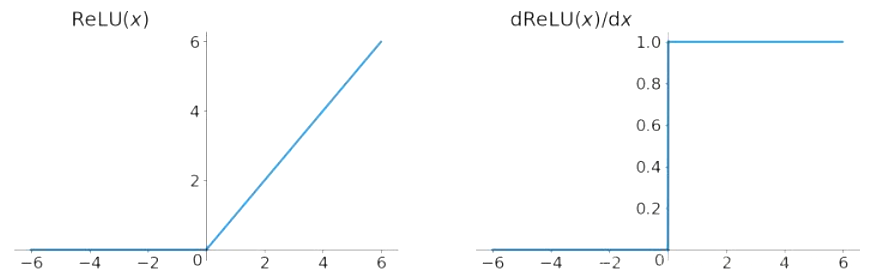
tanh读作Hyperbolic Tangent，它解决了Sigmoid函数的不是zero-centered输出问题，然而，梯度消失（gradient vanishing）的问题和幂运算的问题仍然存在。

Relu函数

Relu函数的解析式：



Relu函数及其导数的图像如下图所示：



ReLU函数其实就是一个取最大值函数，注意这并不是全区间可导的，但是我们可以取sub-gradient，如上图所示。ReLU虽然简单，但却是近几年的重要成果，有以下几大优点：

1） 解决了gradient vanishing问题 (在正区间)

2）计算速度非常快，只需要判断输入是否大于0

3）收敛速度远快于sigmoid和tanh

ReLU也有几个需要特别注意的问题：

1）ReLU的输出不是zero-centered

2）Dead ReLU Problem，指的是某些神经元可能永远不会被激活，导致相应的参数永远不能被更新。有两个主要原因可能导致这种情况产生: (1) 非常不幸的参数初始化，这种情况比较少见 (2) learning rate太高导致在训练过程中参数更新太大，不幸使网络进入这种状态。解决方法是可以采用Xavier初始化方法，以及避免将learning rate设置太大或使用adagrad等自动调节learning rate的算法。

尽管存在这两个问题，ReLU目前仍是最常用的activation function