

神经网络实验报告

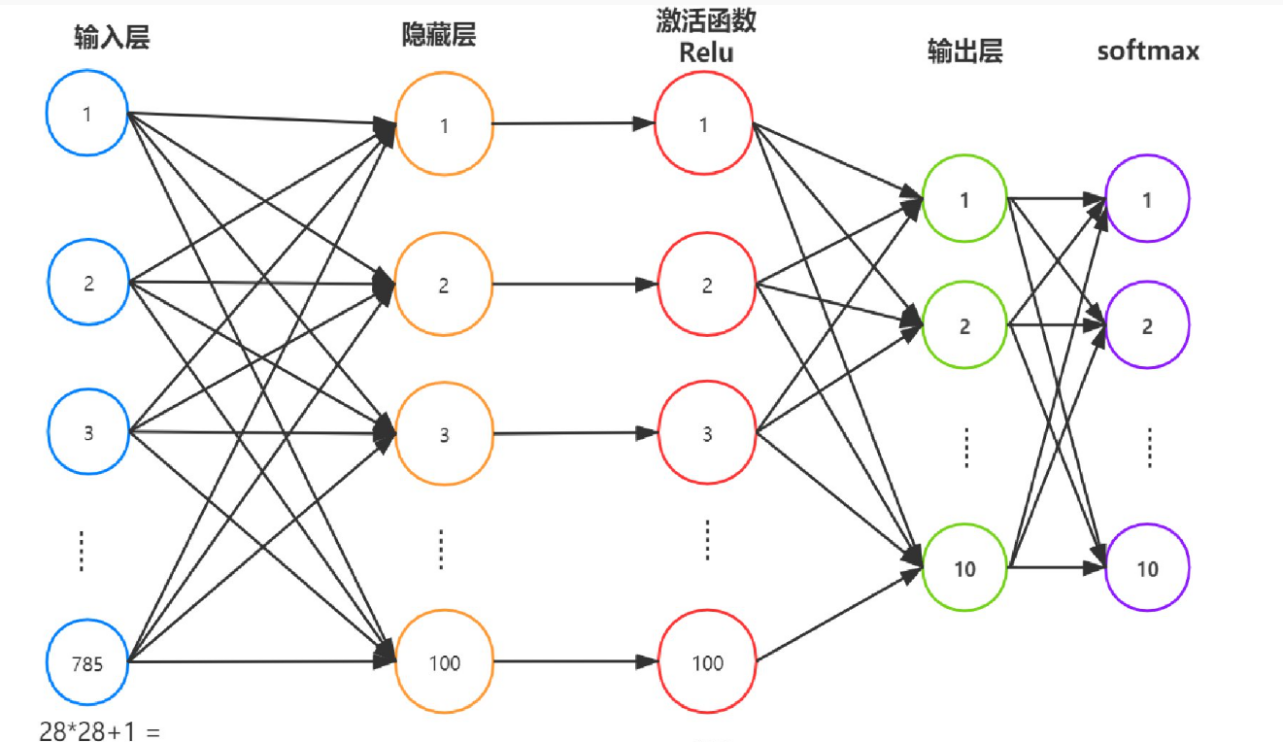
实验一-Numpy实现全连接层

|  |  |
| --- | --- |
| 院系 | 信息学院 |
| 专业班级 | 大数据2001 |
| 姓名 | 李俊松 |
| 学号 | 2020301221205 |
| 指导老师 | 胡学海 |

### 

### 实验原理

应用前馈神经网络，搭建如下模型,完成手写数字识别的任务



该网络只包含一个隐藏层,权重矩阵

使用作为激活函数

使用函数将输出转为概率完成十分类任务

使用交叉熵作为损失函数

### 数据集

本次实验采用的是中下载的数据集,训练集的大小为, 代表有个大小为的矩阵,每一个矩阵代表手写数字图片的灰度矩阵

### 模型优化

模型最初设定了隐藏层为个神经元,学习率为,迭代次数为,按照这个参数我跑了五次,五次的准确率大概在~之间,非常的不稳定.

我认为可能是学习的特征不够,于是搭建了第二层隐藏层,权重矩阵为,然而却发现出现了的情况,询问助教之后发现这是由于中途产生了梯度爆炸，调试过程中我发现降低学习率到左右就不会出现的情况,但是这样子就会降的很慢,我本来想通过二分法找到一个出现与否的边界,然而试了十次左右却始终找不到这个合适的值,询问助教得知不能简单理解学习率和的关系,后面有很多优化器就是为了解决梯度爆炸的情况

def \_\_init\_\_(self):  
 self.W1 = np.random.normal(size=[28 \* 28 + 1, 200])  
 # self.W2 = np.random.normal(size=[100, 100])  
 self.W3 = np.random.normal(size=[200, 10])  
 self.mul\_h1 = Matmul()  
 self.relu1 = Relu()  
 self.mul\_h2 = Matmul()  
 # self.relu2 = Relu()  
 self.mul\_h3 = Matmul()  
 self.softmax = Softmax()  
 self.cross\_en = Cross\_entropy()

def forword(self, x, labels):  
 x = x.reshape(-1, 28 \* 28)  
 bias = np.ones(shape=[x.shape[0], 1])  
 x = np.concatenate([x, bias], axis=1)  
 self.h1 = self.mul\_h1.forword(x, self.W1)  
 self.h1\_relu = self.relu1.forword(self.h1)  
 # self.h2 = self.mul\_h2.forword(self.h1\_relu, self.W2)  
 # self.h2\_relu = self.relu2.forword(self.h2)  
 # print(self.W3.shape)  
 self.h3 = self.mul\_h3.forword(self.h1\_relu, self.W3)  
 # print(self.h3.shape)  
 # print(self.h2.shape)  
 self.h3\_soft = self.softmax.forword(self.h3)  
 self.loss = self.cross\_en.forword(self.h3\_soft, labels)

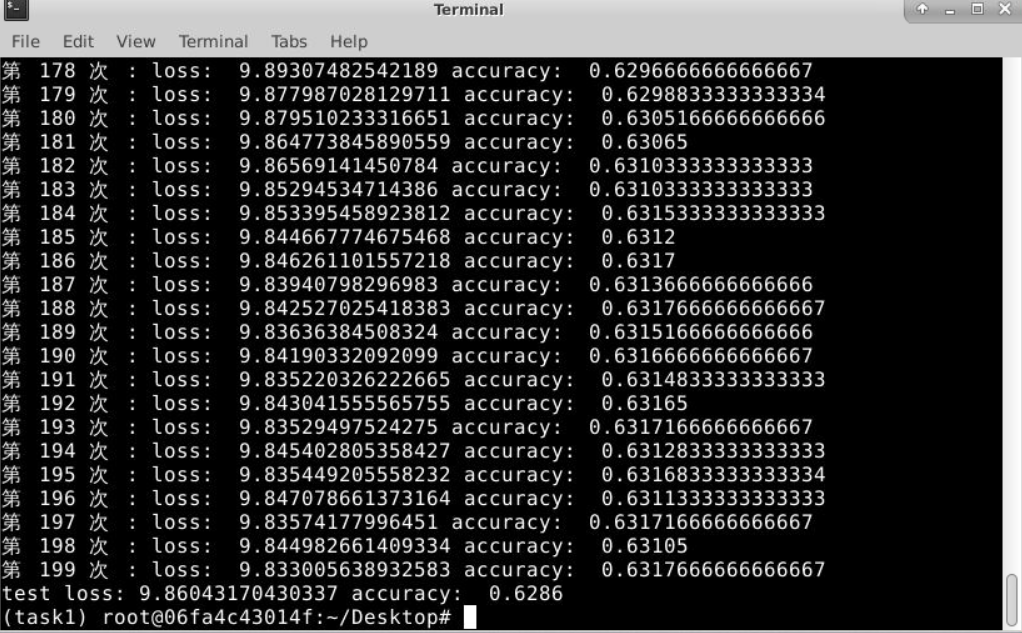
def backword(self, labels):  
 self.loss\_grad = self.cross\_en.backword(labels)  
 self.h3\_soft\_grad = self.softmax.backford(self.loss\_grad)  
 self.h3\_grad, self.W3\_grad = self.mul\_h3.backword(self.h3\_soft\_grad)  
 # print(self.h3\_grad.shape)  
 # self.h2\_relu\_grad = self.relu2.backword(self.h3\_grad)  
 # print(self.h2\_relu\_grad.shape)  
 # self.h2\_grad, self.W2\_grad = self.mul\_h2.backword(self.h2\_relu\_grad)  
 # print(self.h2\_grad.shape)  
 self.h1\_relu\_grad = self.relu1.backword(self.h3\_grad)  
 self.h1\_grad, self.W1\_grad = self.mul\_h1.backword(self.h1\_relu\_grad)

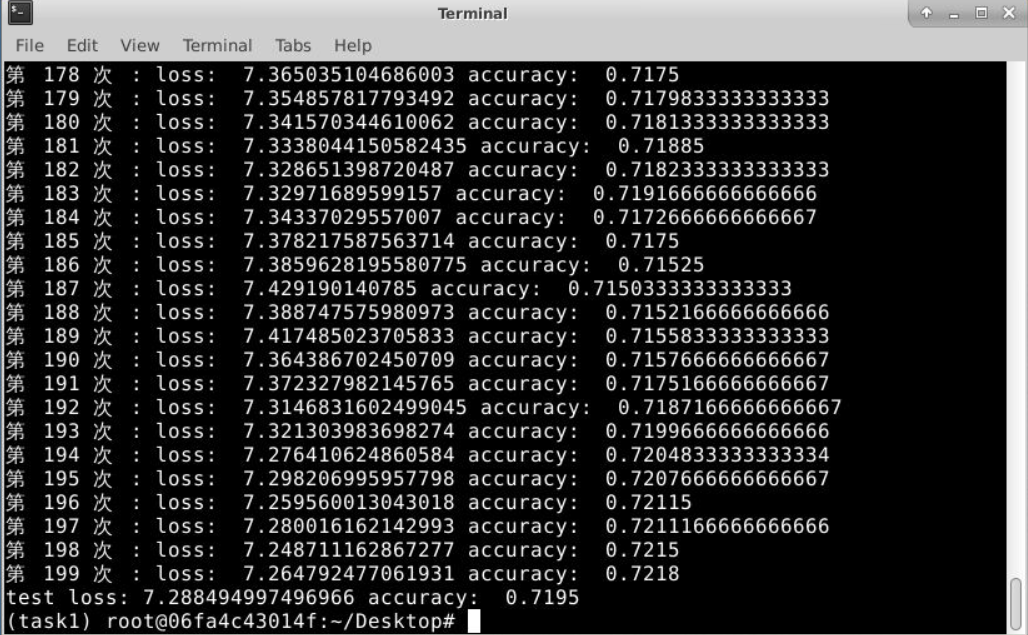
注释的部分为第二个隐藏层相关代码

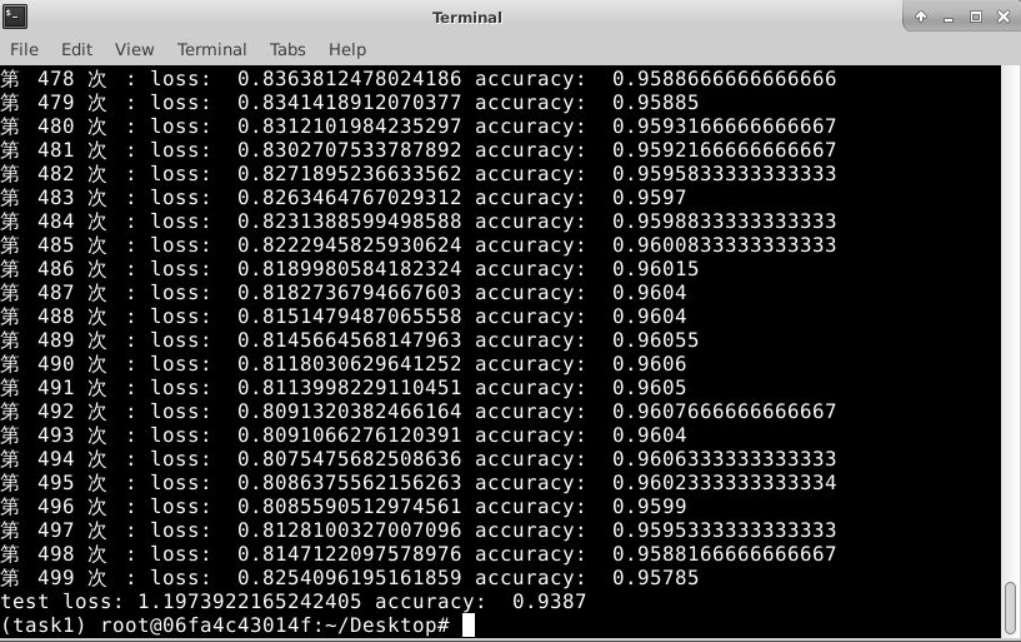
**于是想到调整隐藏层神经元数量，学习率和迭代次数**

调大学习率可以使每次权重矩阵沿梯度方向下降的更多，加大迭代次数同样可以使网络更好地学习特征

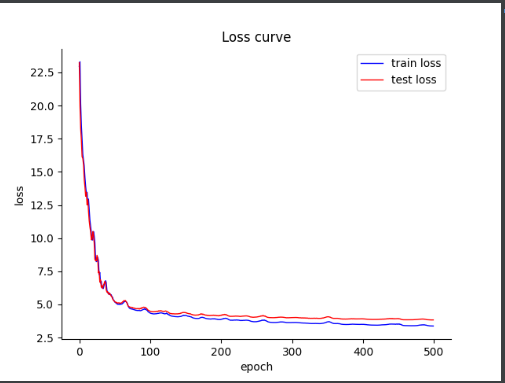
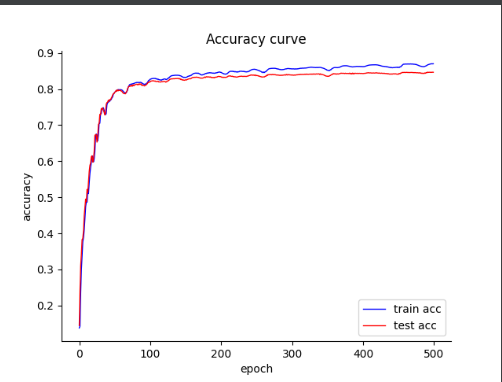
我首先调大了迭代次数,不变,还是,准确率稍微高了一些,但依然不太稳定





这是学习率神经元数量为200，,时得到的结果,可以看到效果还是比较好的,我运行了几次,在训练集上的准确率有左右,在测试集的准确率有左右,但运行途中有几次也出现了的情况

下图为上述参数的某一次计算结果



### 具体实现

载入数据集,并对样本进行标准化

def mnist\_dataset():  
 (x, y), (x\_test, y\_test) = datasets.mnist.load\_data()  
 x = x / 255.0  
 x\_test = x\_test / 255.0  
 return (x, y), (x\_test, y\_test)

#### **矩阵乘法**

实现输入层到隐藏层的前向传播和反向传播更新梯度

为传到此处的梯度

,由矩阵求导得:

class Matmul():  
 def \_\_init\_\_(self):  
 self.men = {}  
  
 def forword(self, x, W):  
 h = np.matmul(x, W)  
 self.men = {'x': x, 'W': W}  
 return h  
  
 def backword(self, grad\_y):  
 x = self.men['x']  
 W = self.men['W']  
 grad\_x = np.matmul(grad\_y, W.T)  
 grad\_W = np.matmul(x.T, grad\_y)  
 return grad\_x, grad\_W

#### 实现激活函数

那么在反向传播的过程中

若,梯度为

若,梯度为1

class Relu():  
 def \_\_init\_\_(self):  
 self.men = {}  
  
 def forword(self, x):  
 self.men['x'] = x  
 return np.where(x > 0, x, np.zeros\_like(x))  
  
 def backword(self, grad\_y):  
 x = self.men['x']  
 return (x > 0).astype(np.float32) \* grad\_y

#### **函数**

class Softmax():  
 def \_\_init\_\_(self):  
 self.men = {}  
 self.epsilen = 1e-12  
  
 def forword(self, x):  
 # print(x.shape)  
 # print(x)  
 x\_exp = np.exp(x)  
 sum = np.sum(x\_exp, axis=1, keepdims=True)  
 out = x\_exp / (sum + self.epsilen)  
 self.men['out'] = out  
 self.men['x\_exp'] = x\_exp  
 return out  
  
 def backford(self, grad\_y):  
 s = self.men['out']  
 sisj = np.matmul(np.expand\_dims(s, axis=2), np.expand\_dims(s, axis=1))  
 g\_y\_exp = np.expand\_dims(grad\_y, axis=1)  
 tmp = np.matmul(g\_y\_exp, sisj)  
 tmp = np.squeeze(tmp, axis=1)  
 softmax\_grad = -tmp + grad\_y \* s  
 return softmax\_grad

#### **交叉熵**

,是维向量,其中

class Cross\_entropy():
  
 def \_\_init\_\_(self):
  
 self.epsilen = 1e-12
  
 self.men = {}
  
  
 def forword(self, x, labels):
  
 log\_prob = np.log(x + self.epsilen)
  
 sum = np.sum(-log\_prob \* labels, axis=1)
  
 loss = np.mean(sum)
  
 self.men['x'] = x
  
 return loss
  
  
 def backword(self, labels):
  
 x = self.men['x']
  
 return -1 / (x + self.epsilen) \* labels

**实验心得**

本次实验通过numpy库实现全连接层，代码实现不仅需要对于库函数语法的熟悉，更需要对于各层正向与反向传播原理的熟练掌握，将数学理论推导通过代码实现，我加深了对神经网络的理解，同时通过对神经网络拓扑结构、激活规则和学习算法的更改，比较前后模型学习能力，与理论知识相结合，有很大的收获。

当然在实验的过程中也遇到了问题，当加深神经网络或增加神经元个数时，输出层Softmax函数在计算时会发生上溢出，但是输入时，像素值已经进行了标准化，通过查阅资料以及和同学讨论后发现，可能是因为权重初始化时值较大，随着神经网络层数的增加，向前传播不断累积，导致最后输出时值较大，做指数运算后数值太大导致溢出，解决方案有：减小初始化权重值并适当增大学习率，从Softmax函数计算做出改变。