人工神经网络 MLP报告

董又铭 计53 2015011294

1. **代码补全**

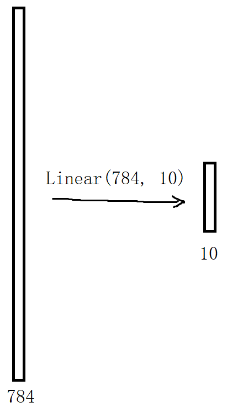
本次作业主要需要补全两个文件：loss.py和layers.py。在loss.py中需要实现欧几里得损失函数的前向和反向传播，在layers.py中需要实现Linear, Sigmoid, Relu三种网络层的前向和反向传播。补全这两个文件之后，可以在run\_mlp.py中修改网络结构、网络参数的初值、训练算法的参数，以此来测试各种情况下模型训练结果的优劣和收敛速度的快慢等性质。

在补全代码时，要充分利用numpy提供的向量/矩阵/张量的操作，尽可能的用简短的代码来实现需要的操作，利用numpy底层用C语言实现的优点尽可能提高运算速度。

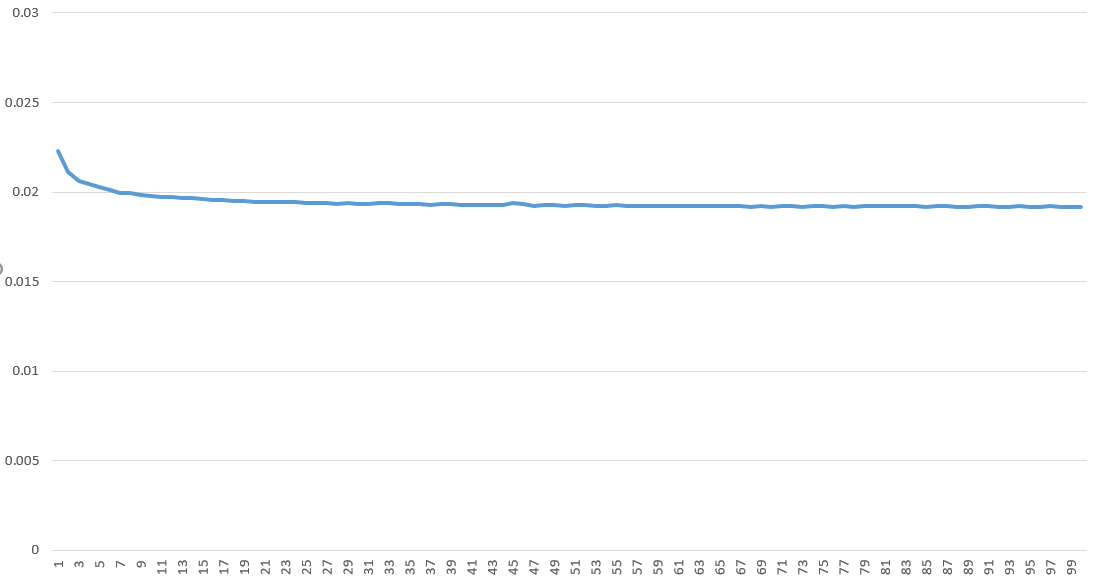
以Relu函数的前向传播为例，最基本的写法就是使用循环语句对输入张量的每个数单独处理得到输出张量；较好的写法是利用Python语法的特性将循环语句省略，但本质上还是以Python的方式在对每个元素进行处理；一种优秀的写法是使用numpy.maximun函数，这样可以用C语言的方式对每个元素进行处理，提升效率。

1. **多次尝试**
   1. 全连接+激活

网络模型中只有1个Linear层，是一个输入数为784输出数为10的全连接层。



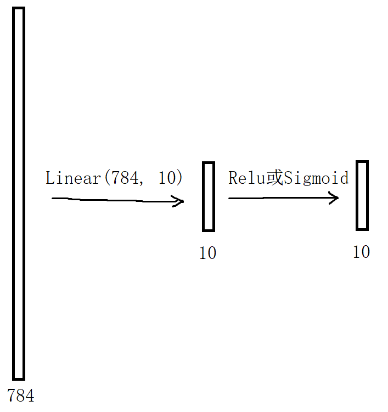
采用learning\_rate=0.01, momentum=0.7进行训练，每次测试的损失变化过程如下：



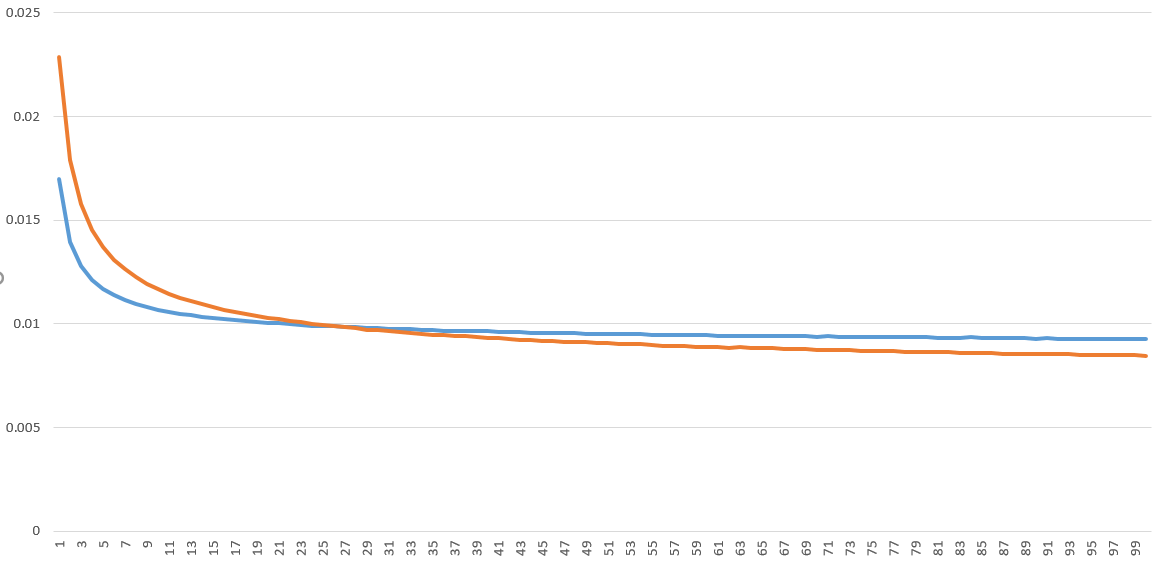
最终的loss停留在0.0192附近，acc停留在0.85附近。

* 1. 全连接+激活

在Linear层后面添加一个激活函数层。



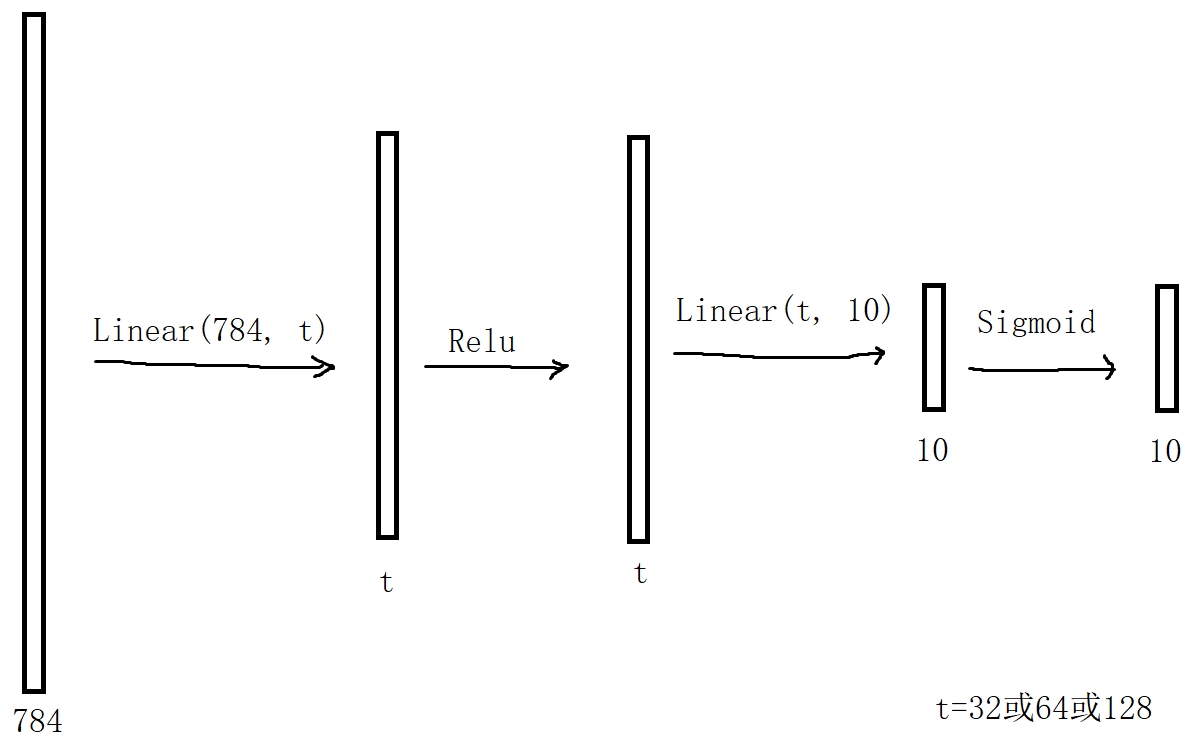
损失数值如下图，其中蓝色是加Relu层，橙色是加Sigmoid层：



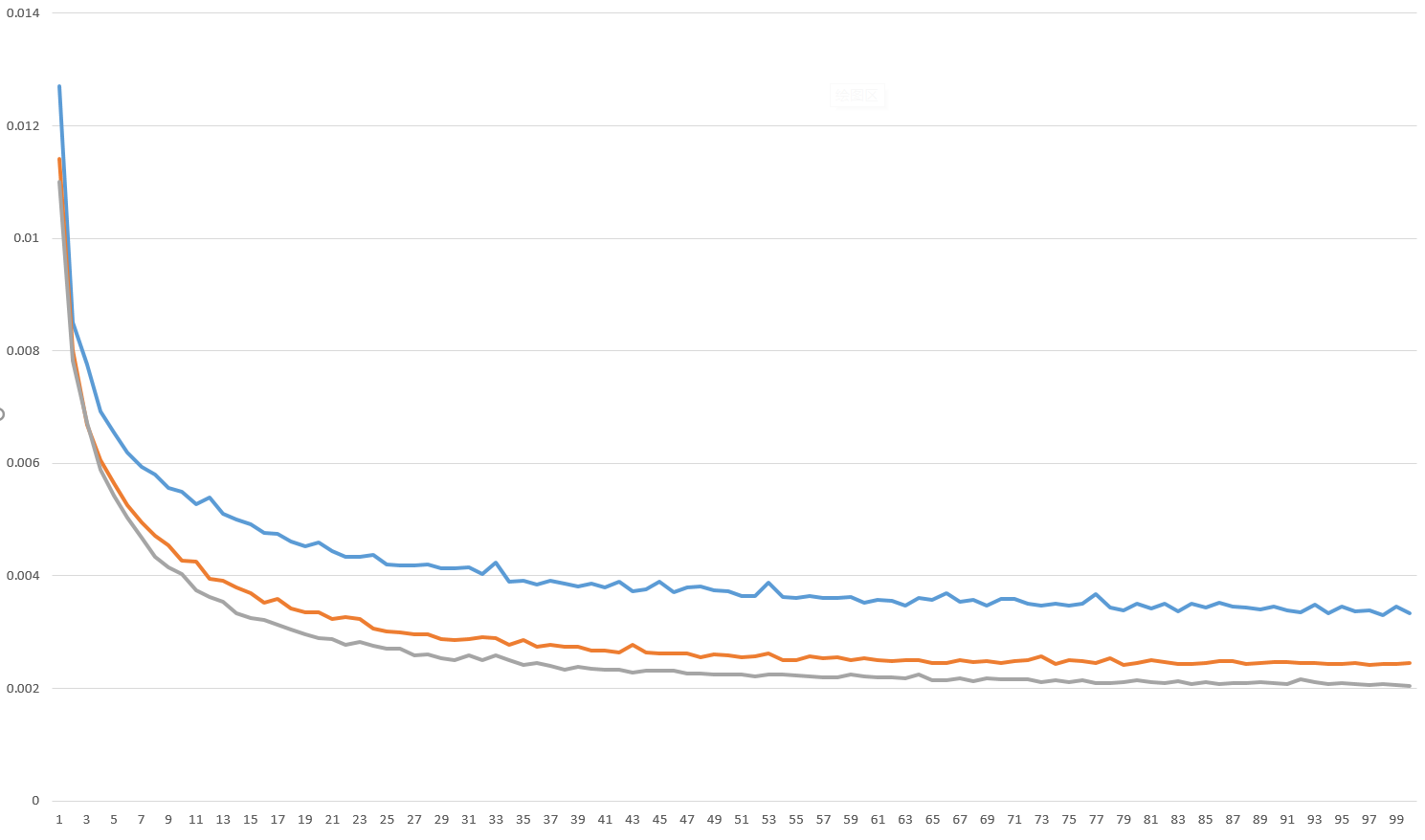
添加Relu层的最终loss停留在0.0093附近，acc停留在0.916附近；添加Sigmoid层的最终loss停留在0.0084附近，acc停留在0.912附近，且仍然有优化的趋势。可以得出初步结论，添加Relu层可以非常快的收敛，但最终的收敛效果没有添加Sigmoid层的网络好。

* 1. 全连接+激活+全连接+激活

这时涉及到一个问题：第一个全连接的输出数据有多少维。凭直觉可以知道维数应该在10到784之间，于是我选择了32, 64, 128进行了测试。根据多次测试发现，第一个激活函数使用Relu，第二个激活函数使用Sigmoid效果最佳，于是我全都采用了这种配置。



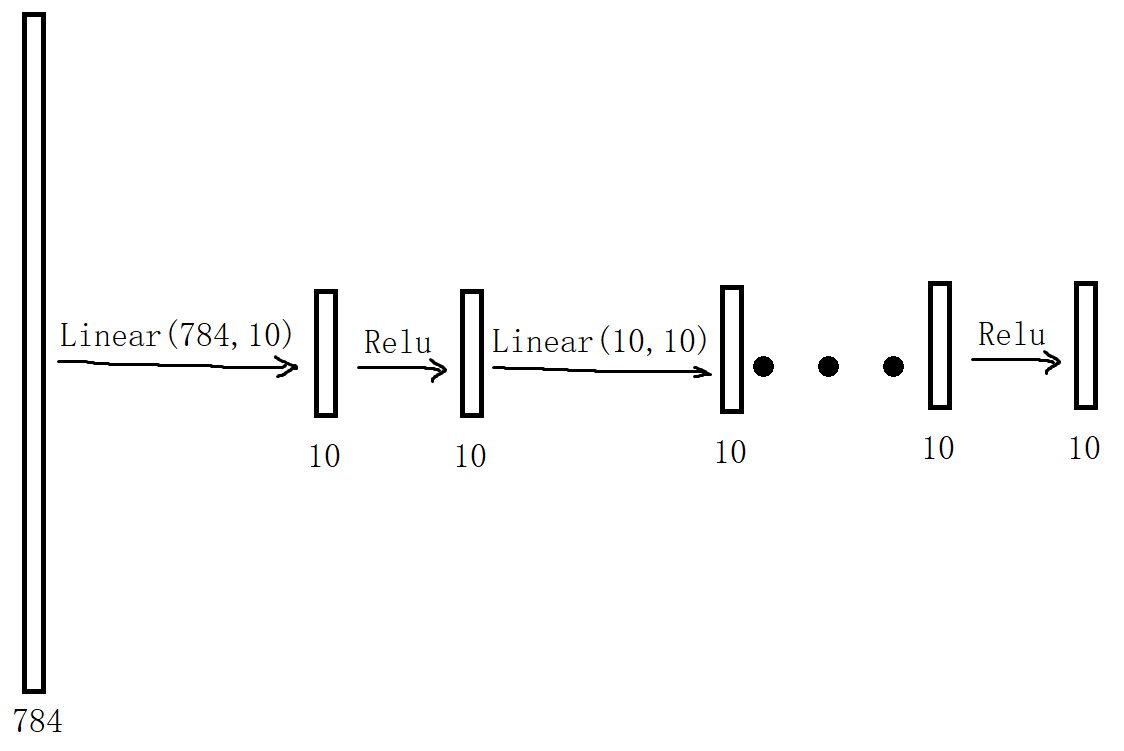
训练时采用learning\_rate=1，momentum=0.7，损失数值如下图，其中蓝色为t=32，橙色为t=64，灰色为t=128：



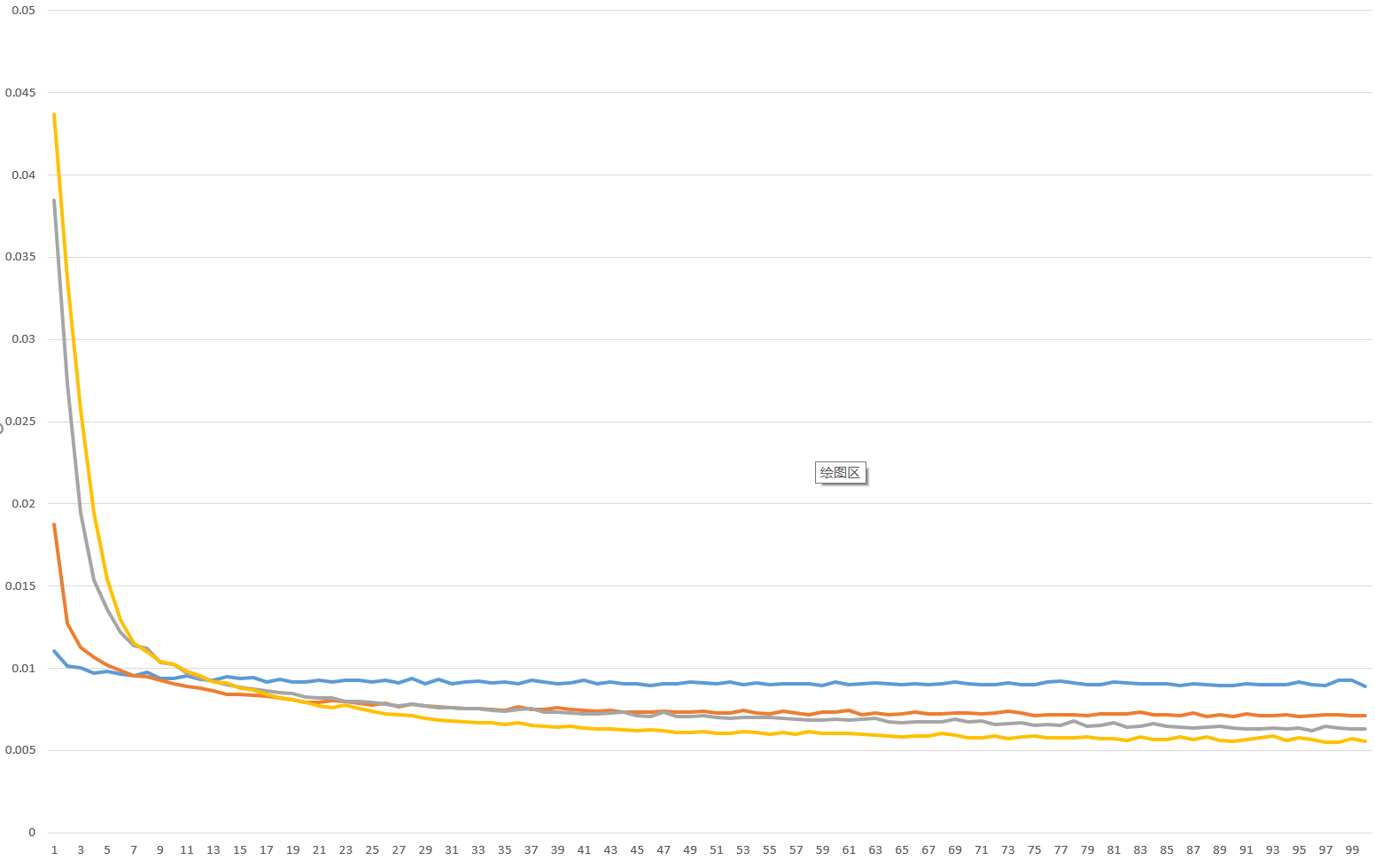
从图中可以看出，t越大网络模型的效果越好，但训练时的运算量也会相应的增大。

* 1. 多个Linear层

激活函数统一使用Relu，网络模型分别是784→10，784→10→10，784→10→10→10，也就是如下图所示的模型



训练结果如图，蓝色为784→10，橙色为784→10→10，灰色为784→10→10→10，黄色为784→10→10→10→10：



可以看出，网络层数越多，初始loss越大，但最终的loss越小，即最终效果越好。

1. **总结**

本次作业让我们了解到如何手动实现神经网络的训练框架，以及在调参尝试的过程中体会到了神经网络的玄学性，全都是玄学，瞎改一个参数之后说不定就强的一匹，根本没法解释。真不知道那些天天搞研究的人是怎么玩的。