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。

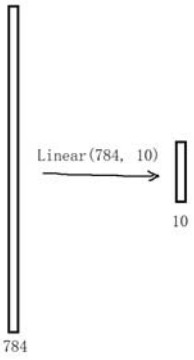
在补全代码时，要充分利用 numpy 提供的向量/矩阵/张量的操作，尽可能的用简短的代码来实现需要的操作，利用 numpy 底层用 C 语言实现的优点尽可能提高运算速度。

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。

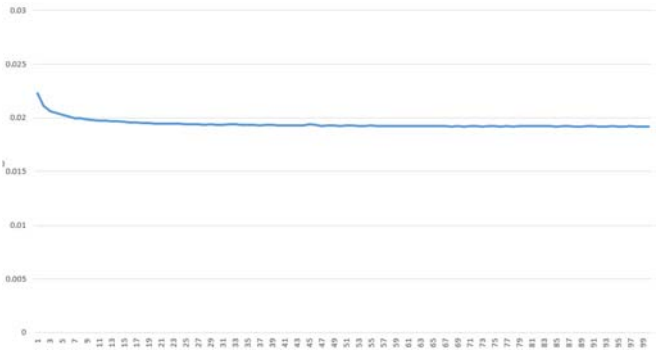
2. 多次尝试

2.1. 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。



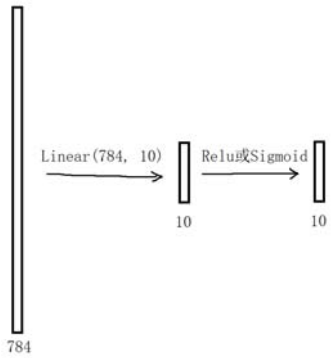
采用 learning_rate=0.01, momentum=0.7 进行训练，每次测试的损失变化过程如下：



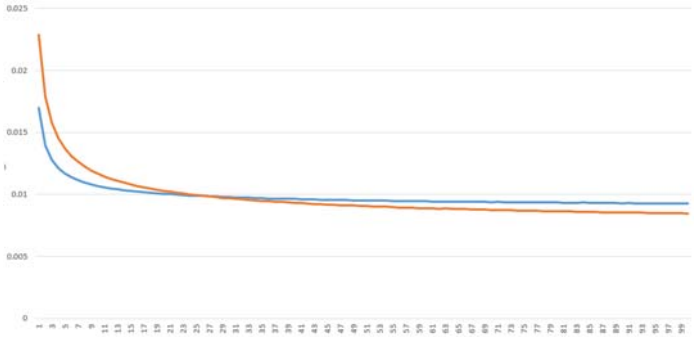
最终的 loss 停留在 0.0192 附近，acc 停留在 0.85 附近。

2.2. 全连接+激活

在 Linear 层后面添加一个激活函数层。



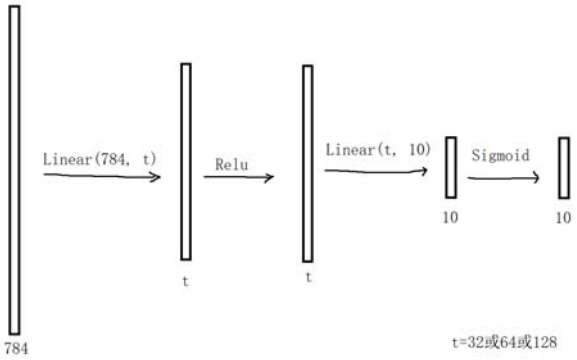
损失数值如下图，其中蓝色是加 Relu 层，橙色是加 Sigmoid 层：



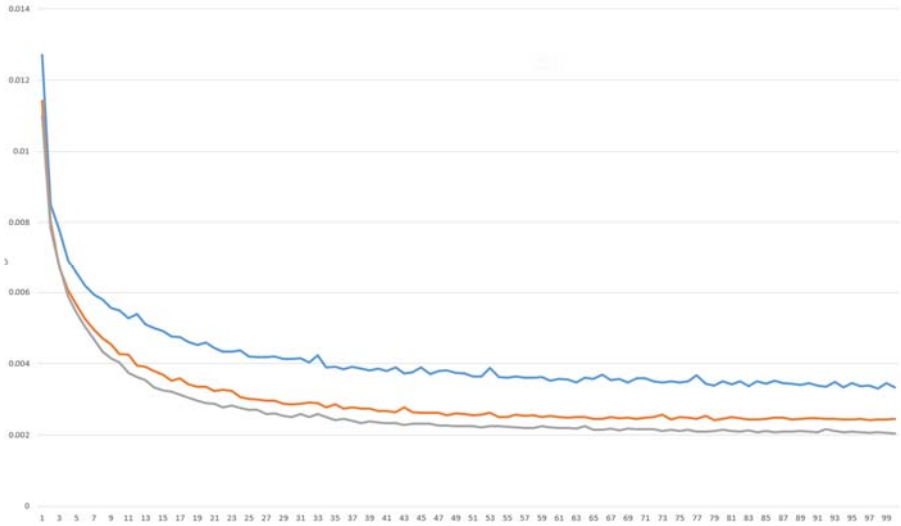
添加 Relu 层的最终 loss 停留在 0.0093 附近，acc 停留在 0.916 附近；添加 Sigmoid 层的最终 loss 停留在 0.0084 附近，acc 停留在 0.912 附近，且仍然有优化的趋势。可以得出初步结论，添加 Relu 层可以非常快的收敛，但最终的收敛效果没有添加 Sigmoid 层的网络好。

2.3. 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。



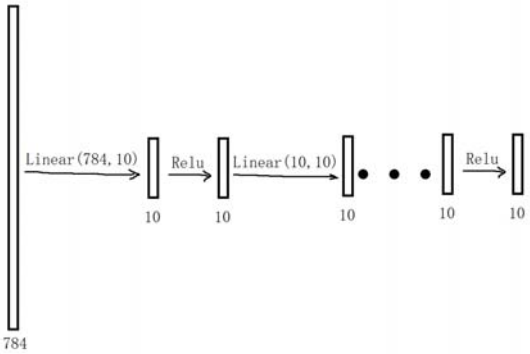
训练时采用 learning_rate=1，momentum=0.7，损失数值如下图，其中蓝色为 t=32，橙色为 t=64，灰色为 t=128：



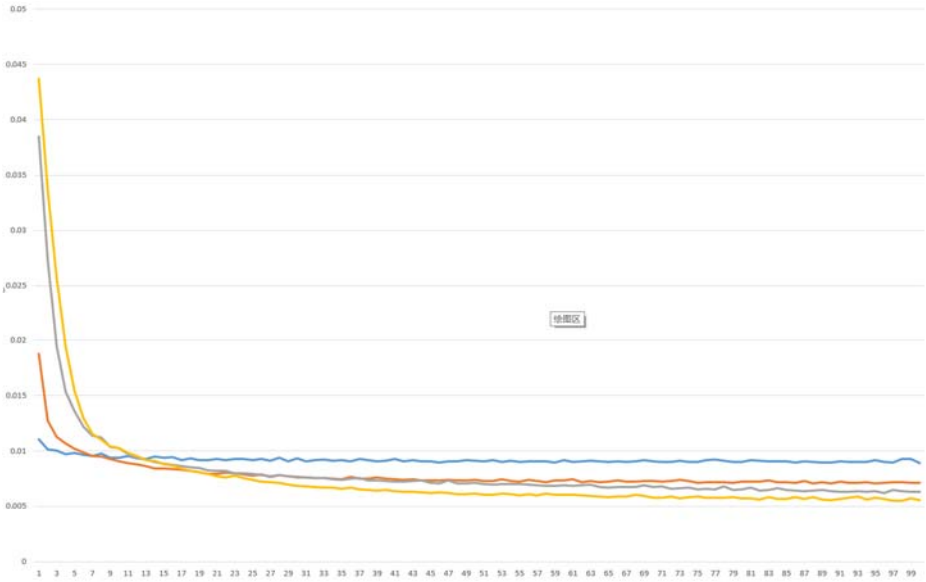
从图中可以看出，t 越大网络模型的效果越好，但训练时的运算量也会相应的增大。

2.4. 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。

3. 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。