“神经网络识别手写数字”报告

181240016 高长江

1. 数据：

MNIST 数据集，在 Nielsen 的 Github 网页上下载。（文件名：data/mnist.pkl.gz）

2. 文件：

（1）network.py 识别手写数字任务的主要文件，主要代码根据书中第 2 章改写。
（2）mnist_loader.py 用于加载 MNIST 数据集的程序，在 Github 上下载，做了改动。运行时需要将此文件复制到 python 运行环境的 Lib 文件夹下。
（3）prototype.py 神经网络的原型文件，根据书中第 2 章代码改写，存在一些问题，不能直接运行
（4）data/mnist.pkl.gz 即 MNIST 数据集

3. 简述

主程序采用的学习方法是随机梯度下降，运行训练数据的方式是列表迭代。开始时代价函数选择的是二次函数。
首先通过调整网络规模和超参数，观察运行结果。然后更换代价函数为交叉熵函数，观察效果。

4. 过程记录

（1）改正代码错误

由于原来的代码是 Python2，在修改了语法格式后，仍然存在一些问题。我主要遇到了 3 个问题，在这里简要记录。
一：加载 cPickle 库时报错。这是由于 Python3 将 cPickle 改为了 pickle。改正后错误解决。
二：运行 mnist_loader.py 程序时，数据集文件解码报错。（UnicodeDecodeError: 'ascii' codec can't decode byte 0x90 in position 614: ordinal not in r…）报错原因是 pickle.load() 和 cPickle.load() 的参数设置不同。在 pickle.load() 中增加参数 encoding=’bytes’，问题解决。
三：主程序运行过程中，在 update_mini_batch() 方法类型报错（TypeError: object of type 'zip' has no len()）报错原因是在 Python3 中 zip 不能直接获取长度。解决方法：将 mnist_loader.py 中的所有 zip 类型返回值均转为 list 类型。

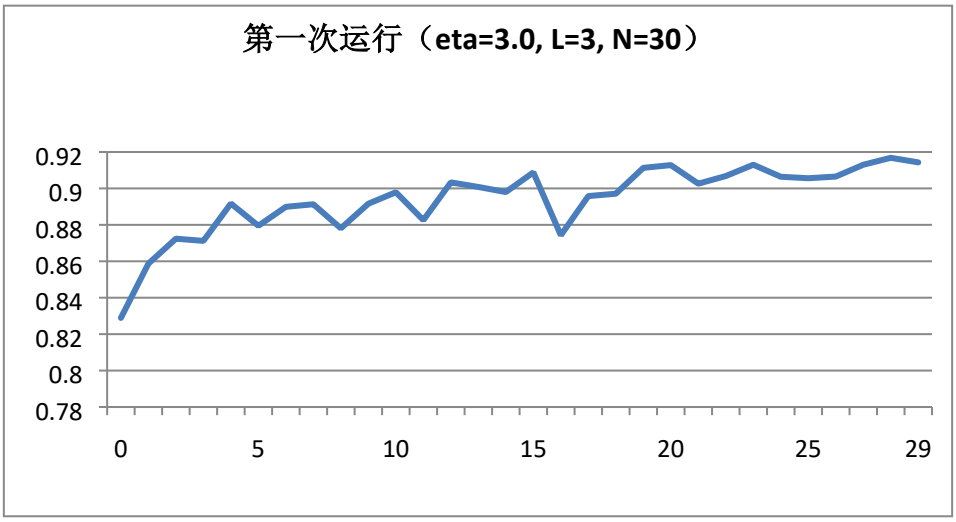
（2）运行神经网络

第一次运行使用和书中一样的超参数：迭代次数（epoch）30，小批量数据大小（mini_batch_size）10，学习率（eta）3.0。同时网络的大小为 [784, 30, 10]。
运行结果如下：

Epoch 0: 8289/10000
Epoch 1: 8587/10000
Epoch 2: 8724/10000

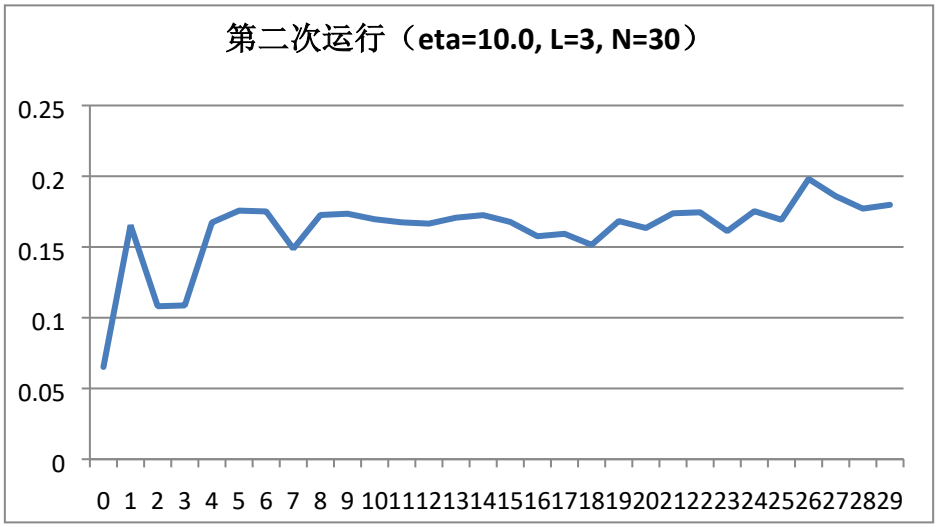
Epoch 3: 8712/10000
Epoch 4: 8917/10000
Epoch 5: 8795/10000
...
Epoch 25: 9056/10000
Epoch 26: 9065/10000
Epoch 27: 9130/10000
Epoch 28: 9168/10000
Epoch 29: 9143/10000

正确率随迭代次数的变化如下图：



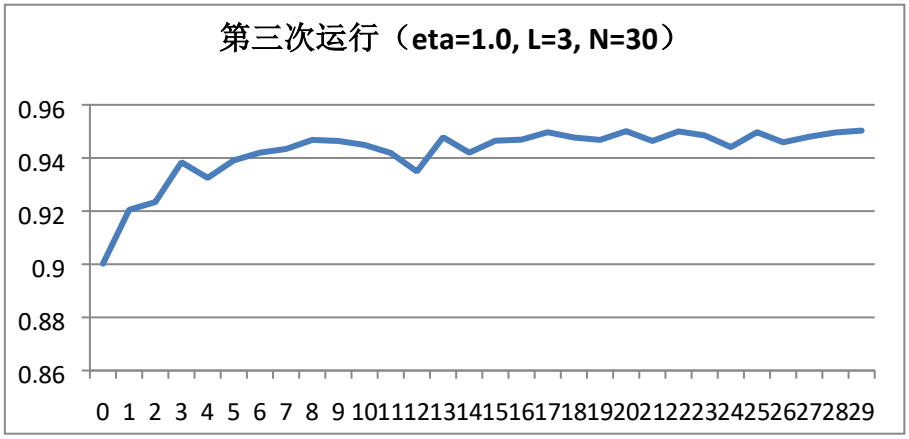
可见正确率不太理想（峰值只有 91.68%），且训练过程中正确率始终在上下波动。同时，每次迭代的时间先长后短，开始时运行速度很慢。正确率总体在提高，速度先快后慢。

第二次，设置学习率为 10.0，运行结果如下图：



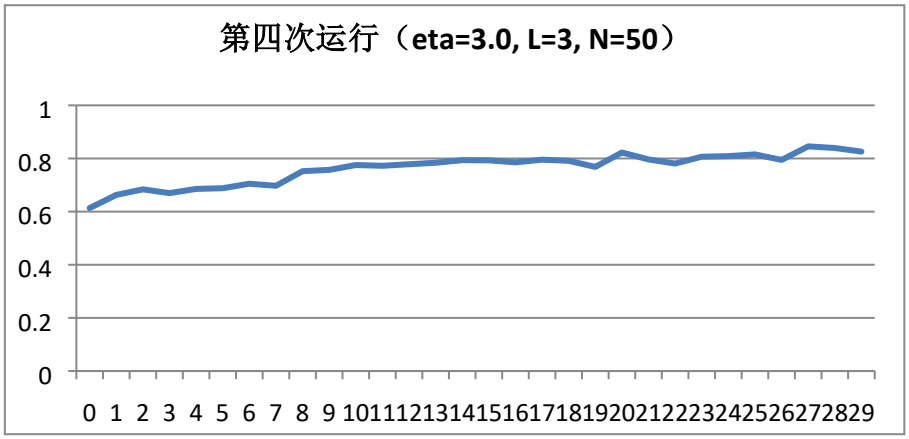
可以明显看出，这样设置的学习效率很低。虽然正确率总体在上升，但始终没超过 20%。说明学习率设置得过大。

第三次，设置学习率为 1.0，运行结果如下图：



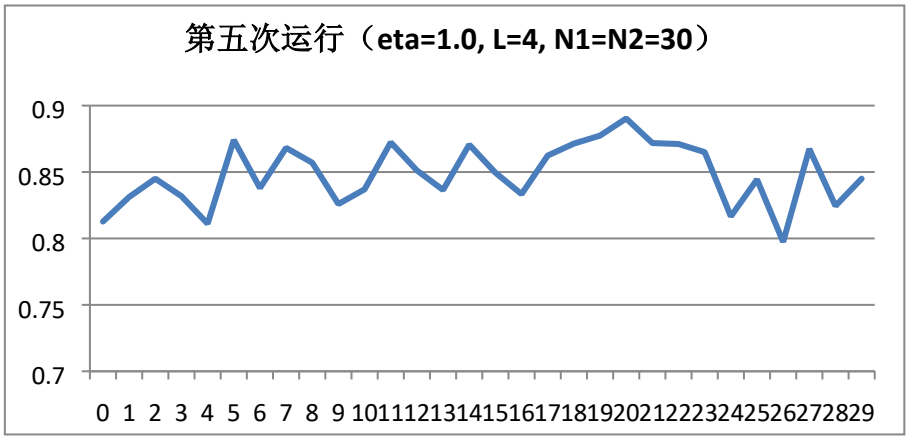
这次正确率一开始就很高（90%），并且持续上升，最后稳定在 95%左右。说明减小学习率可以显著提高训练效果。然而这次的运行速度较慢，正确率提高的速率也较慢。总体来说学习率设为 1.0 比较理想。

第四次，重新设置学习率为 3.0，增加中间层的神经元个数为 50。预计正确率会上升。然而运行结果如下：



不仅学习的速度、正确率的提升速率都变得很慢，正确率也变低了，峰值只有 84.55%。可见增加中间神经元数量并不一定能提高神经网络的性能。分析原因，可能是初始的权重和偏置错误太大，导致学习缓慢。

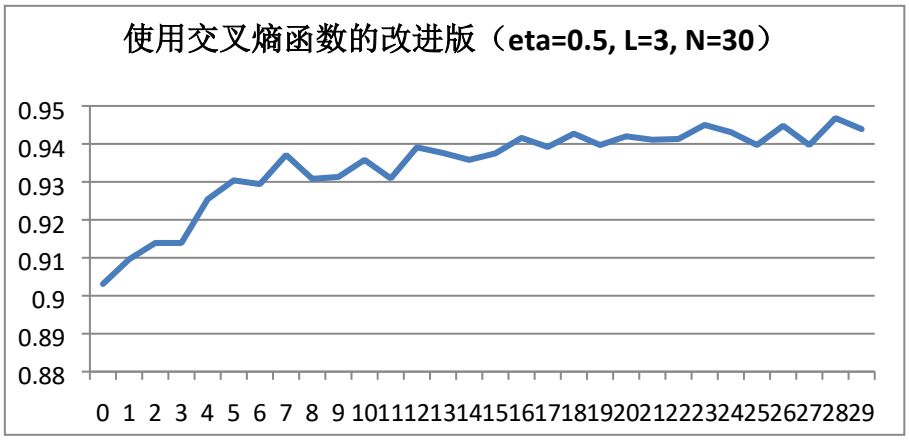
第五次，学习率 3.0，增加一层隐藏层，神经元数量均为 30。运行结果如下：



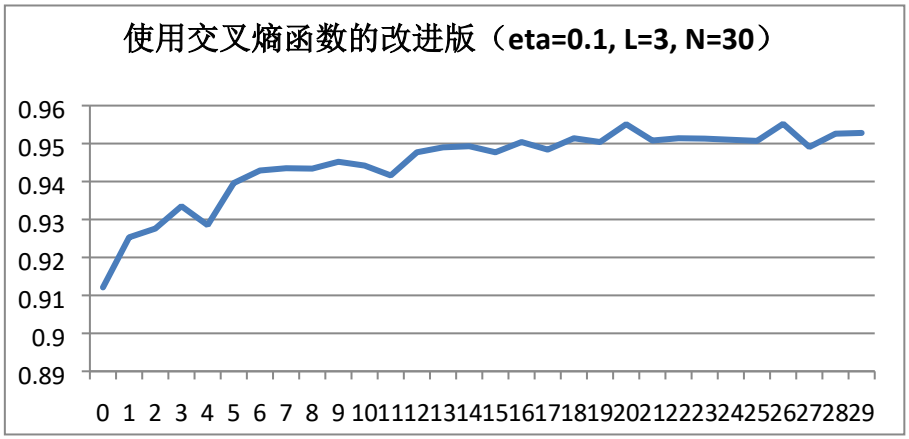
同样，增加一层神经元并没有起到很好的效果，学习效率不高，正确率没有显著的改观。虽然正确率峰值达到了 89%，但随后正确率竟快速下降。看起来可能是增加一层隐藏层后，没有改变算法，导致学习率的问题。看来深度学习时算法的改进十分重要。

（3）改进

将代价函数改为交叉熵函数，除学习率调整成 0.5 以外，其他超参数与第一次相同。此时需要修改一下反向传播算法中的输出层误差代码，防止损失函数的导数出现除零错误。运行结果如下：



这次的运行结果比较令人满意。首先，正确率相比第一次得到了较大提高，峰值达到了 94.67%，相比第一次运行提高了 3 个百分点。同时，可以明显看出开始时学习效率较高，体现出交叉熵函数的特点。然而，可能是由于学习率设置得较大，在正确率接近 94%以后学习效果不明显。于是，将学习率设为 0.1，再次运行，结果如下：



果然，减小学习率以后，正确率不仅达到了最高的一次（峰值 95.5%），而且在后面的迭代中保持在较高水平。只不过，前几次迭代的运行速度比较慢，这算是一个缺点。