人工神经网络实验报告
多层感知器 (MLP)

张钰晖
2015011372, yuhui-zh15@mails.tsinghua.edu.cn, 185-3888-2881

目录

1 问题描述

机器学习数字分类领域广泛使用 MNIST 训练集，该集合包含 60000 个训练样例和 10000 个测试样例。每一个样例是 784×1 的列向量，该列向量可以被还原为原始的 28×28 的灰度图。下面展示了其中的几张样本。

图 1: MNIST 数据样本

在这次作业中，笔者将使用多层感知器（MLP）来完成数字分类任务，通过不同参数的设置来提高任务的准确率。

2 细节实现

MLP 是最简单的神经网络，通过简单的全连接层和激活函数实现。因为作业提供了较为完美的实验框架，实现了绝大部分代码，本次任务只需完成几个很小的函数即可。

笔者充分使用了机器学习向量化（vectorization）的思想，尽可能最大程度上省略所有的不必要的 for 循环，通过向量化的思想，不仅使得代码极为精简，而且代码效率也较高，运行速度较快。

2.1 激活函数的实现

在给定的框架下，激活函数是按层的方式实现的。

2.1.1 向前传递

- 输入 (input)：$batch\_size * previous\_layer\_size$
- 输出 (output)：$batch\_size * previous\_layer\_size$

在向前传递中，由于仅进行激活操作，输出规模和输入规模相同。这里需要实现 Sigmoid 函数和 Relu 函数。

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

$$Relu(x) = \max(0, x)$$

2.1.2 向后传递

- 输入 (input)：$batch\_size * next\_layer\_size$
- 输出 (output)：$batch\_size * next\_layer\_size$

在向后传递中，由于仅进行激活操作，输出规模和输入规模相同。在向后传递中，需要实现 Sigmoid 函数的导数和 Relu 函数的导数，该导数的实现需要利用之前向前传递时保存的数据。

$$Sigmoid'(x) = Sigmoid(x)(1 - Sigmoid(x))$$

$$Relu'(x) = (x > 0)$$

激活函数层应作为信息的传递者，向后传递时应该乘以前一层的梯度。

2.1.3 代码实现

```
class Sigmoid(Layer):
   def forward(self, input):
      self._saved_for_backward(1 / (1 + np.exp(-input)))
      return self._saved_tensor
   def backward(self, grad_output):
      return grad_output * self._saved_tensor * (1 - self._saved_tensor)
```

```
class Relu(Layer):
   def forward(self, input):
      self._saved_for_backward(input)
      return np.maximum(input, 0)
   def backward(self, grad_output):
      return grad_output * np.array(self._saved_tensor > 0, dtype = float)
```

2.2 全连接层的实现

2.2.1 向前传递

- 输入 (input)：$batch\_size * previous\_layer\_size$
- 输出 (output)：$batch\_size * next\_layer\_size$

利用线性代数相关的知识，可以得到非常优美简洁的表达式，数学推导不再赘述。

$$output = input * W + b$$

2.2.2 向后传递

- 输入 (input)：$batch\_size * next\_layer\_size$
- 输出 (output)：$batch\_size * previous\_layer\_size$

利用线性代数相关的知识，可以得到非常优美简洁的表达式，数学推导不再赘述。

$$grad\_input = grad\_output * W^T$$

$$grad\_W = input^T * grad\_output$$

$$grad\_b = grad\_output$$

2.2.3 代码实现

```
class Linear(Layer):
   def forward(self, input):
      self._saved_for_backward(input)
      return np.dot(input, self.W) + self.b
   def backward(self, grad_output):
      self.grad_W = np.dot(self._saved_tensor.T, grad_output)
      self.grad_b = grad_output
      return np.dot(grad_output, self.W.T)
```

2.3 损失函数的实现

2.3.1 向前传递

- 输入 (input, target)：$batch\_size * output\_layer\_size$
- 输出 (output)：$1 * 1$

EuclideanLoss 和 SoftmaxCrossEntropyLoss 的定义如下：

$$EuclideanLoss = \frac{1}{2 * N} \sum_{k}^{N} (t_k - y_k)^2$$

$$SoftmaxCrossEntropyLoss = -\frac{1}{N} \sum_{k}^{N} t_k \log p_k$$

其中

$$p_k = \frac{e^{y_k}}{\sum_m e^{y_m}}$$

2.3.2 向后传递

- 输入 (input, target)：$batch\_size * output\_layer\_size$
- 输出 (output)：$batch\_size * output\_layer\_size$

对损失函数进行求导，利用线性代数相关的知识，将损失函数的表达式高度向量化，不再赘述，详见代码实现。

2.3.3 代码实现

```
class EuclideanLoss(object):
   def forward(self, input, target):
      return 0.5 * np.sum((target - input) ** 2) / len(input)
   def backward(self, input, target):
      return (input - target) / len(input)
```

```
class SoftmaxCrossEntropyLoss(object):
   def forward(self, input, target):
      self.prob = (np.exp(input).T / np.exp(input).sum(axis=1)).T
      return -np.sum(target * np.log(self.prob)) / len(input)
   def backward(self, input, target):
      return (self.prob - target) / len(input)
```

3 实验结果

3.1 训练曲线

训练曲线刻画了准确率 Accuracy 与损失 Loss 随训练次数变化的曲线。下图展示了一幅经典的训练曲线：

表 1: 网络参数

网络结构	(784, 400, 200, 10) 双隐层网络
激活函数	Relu
损失函数	EuclideanLoss
其它参数	Learning Rate: 1e-1
	Weight Decay: 1e-4
	Momentum: 1e-4
	Batch Size: 100
最高准确率	98.69%

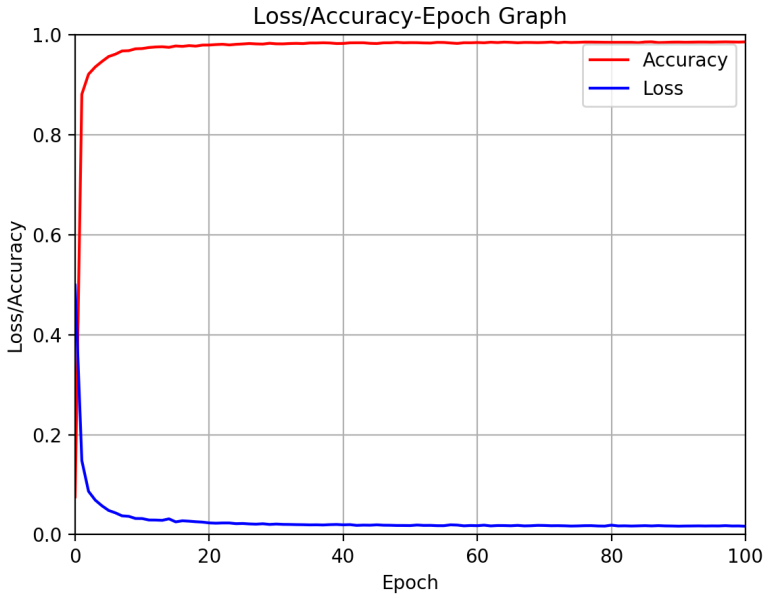

图 2: 训练曲线

这是所有测试的 MLP 中结果最好的一组网络，其最终准确率达到了 98.69%（100 个 Epoch 之内），较好的完成了图像识别任务。

从图中可以看出，MLP 收敛速度很快，迭代次数较少时损失 Loss 下降迅速，准确率 Accuracy 提升迅速。随着迭代次数的增加，损失 Loss 下降速度变得平缓，准确率 Accuracy 提升速度也变得平缓，并逐步收敛，最终稳定在收敛值附近。

3.2 激活函数对比

在这一步中，我们将使用单隐层网络和双隐层网络对比 Sigmoid 激活函数和 Relu 激活函数对结果的影响。

表 2: 网络参数

网络结构	(784, 400, 10) 单隐层网络
	(784, 400, 200, 10) 双隐层网络
激活函数	Sigmoid
	Relu
损失函数	SoftmaxCrossEntropyLoss
其它参数	Learning Rate: 1e-1
	Weight Decay: 1e-4
	Momentum: 1e-4
	Batch Size: 100

下图展示了 4 种模式下的准确率 Accuracy 和损失 Loss：

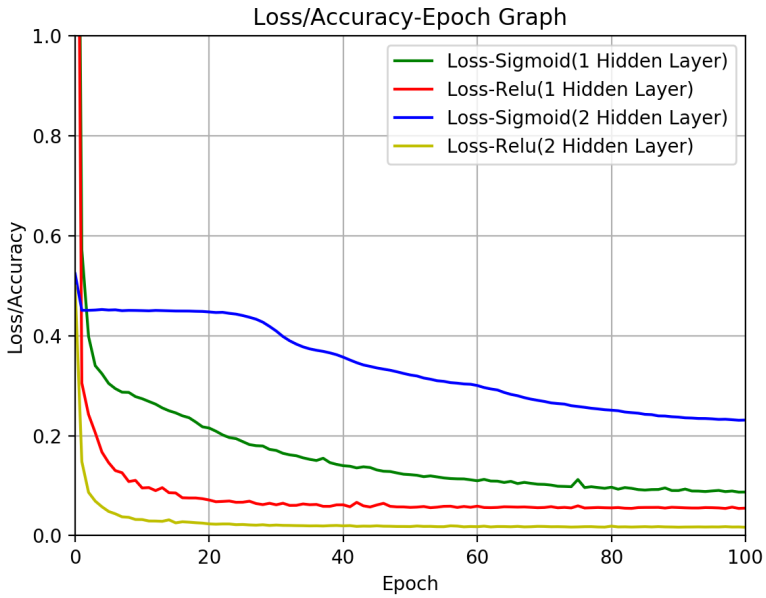
Loss/Accuracy-Epoch Graph
Loss-Sigmoid(1 Hidden Layer)
Loss-Relu(1 Hidden Layer)
Loss-Sigmoid(2 Hidden Layer)
Loss-Relu(2 Hidden Layer)
Loss/Accuracy
Epoch
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100

图 3: 准确率曲线

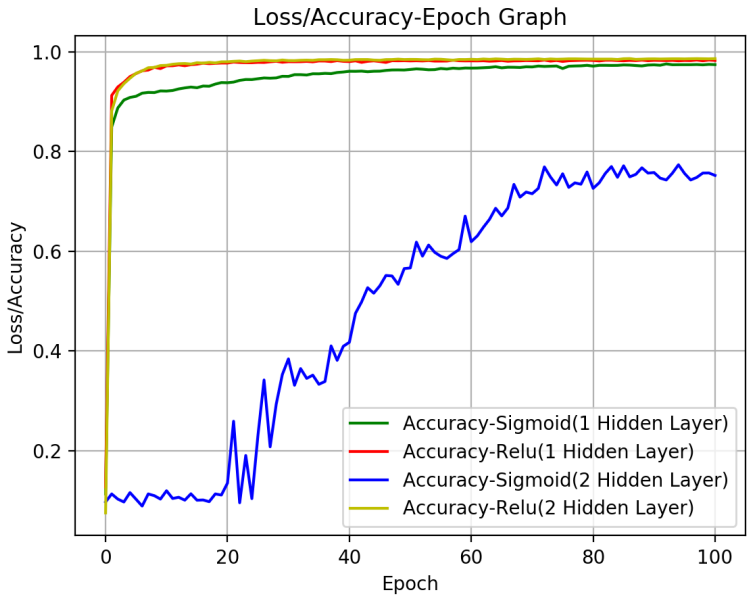

图 4: 损失曲线

表 3: 训练结果

网络结构	**激活函数**	**准确率**	**收敛性**	**收敛速度**
单隐层	Sigmoid	97.54%	好	一般
单隐层	Relu	98.32%	好	很快
双隐层	Sigmoid	77.33%	差	很慢
双隐层	Relu	98.69%	好	很快

从以上图表我们可以清晰的看出，针对数字分类任务，无论是单隐层网络还是双隐层网络，Relu 激活函数都相对于 Sigmoid 激活函数不仅收敛结果好，而且收敛速度快，准确率高。双隐层 Relu 函数激活的网络准确率达到了 98.69%。

笔者猜测这可能是由于 Relu 激活函数的梯度更大，梯度下降速度更快，故收敛更快，且网络模型较为简单，故收敛性也很好。Sigmoid 激活函数梯度较小，有时甚至会梯度消失，双隐层 Sigmoid 函数激活的网络收敛性很差，收敛速度很慢，便可能是这个原因。

3.3 隐层层数对比

由于 Relu 激活函数效果明显好于 Sigmoid 激活函数，在这一步中我们统一使用 Relu 激活函数，对比隐层层数对结果的影响。

表 4: 网络参数

网络结构	(784, 400, 10) 单隐层网络
	(784, 400, 200, 10) 双隐层网络
	(784, 300, 150, 50, 10) 三隐层网络
激活函数	Relu
损失函数	EuclideanLoss
其它参数	Learning Rate: 1e-1
	Weight Decay: 1e-4
	Momentum: 1e-4
	Batch Size: 100

下图展示了隐层数为 1，2，3 三种模式下的训练曲线：

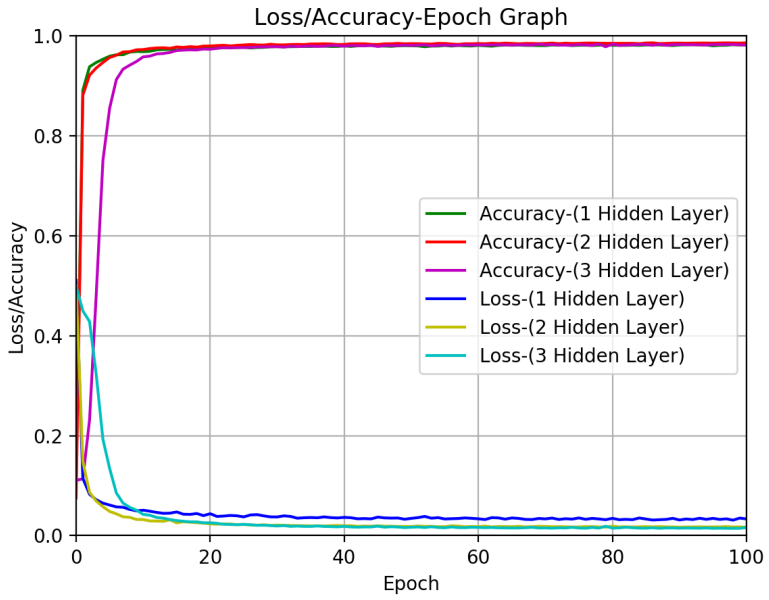

图 5: 训练曲线

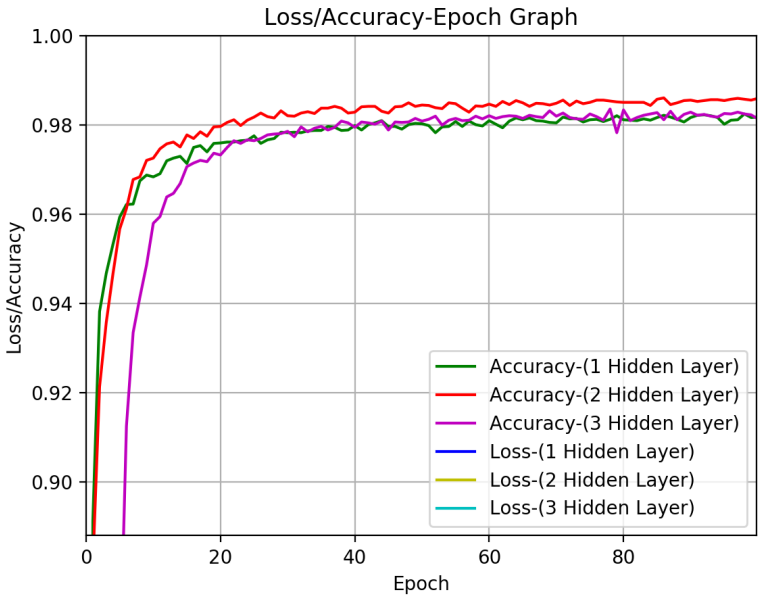

图 6: 放大的准确率曲线

表 5: 训练结果

网络结构	**准确率**	**收敛性**	**收敛速度**	**训练速度**
单隐层	98.25%	好	很快	快
双隐层	98.69%	好	很快	中
三隐层	98.36%	好	较快	慢

从以上图表我们可以清晰的看出，尽管不同的训练层数之间训练曲线较为相似，收敛性都较好，准确率都较高，但通过放大的准确率曲线可以看出细微差别。在本次测试中，准确性最好的是双隐层网络，其次是三隐层网络，最后是单隐层网络。

理论上，隐层数量越高，其准确率期望应该越高，所以三隐层理论上应比双隐层结果较好。但实际中双隐层结果更好，笔者猜测是由于训练次数还不够，Bias 较大，存在欠拟合等问题。

在实际应用中，还应考虑收敛速度和训练速度，前者指的是训练曲线接近收敛值的快慢，后者指的是每训练一个完整的 Epoch 时间。从收敛曲线上可以看出，单隐层和双隐层网络比三隐层网络收敛速度更快，训练初期损失 Loss 下降更快。同时从实际测试中，由于隐层数越多计算量越大，可以明显感受到三隐层训练时间大于双隐层大于单隐层。

综上所述，针对数字分类任务，双隐层网络做到了准确率和速度之间的权衡取舍，表现最为优异。

3.4 隐层规模对比

由于 Relu 激活函数效果明显好于 Sigmoid 激活函数，在这一步中我们统一使用 Relu 激活函数，对比隐层规模对结果的影响。

表 6: 网络参数

网络结构	(784, 400, 10) 单隐层网络
	(784, 200, 10) 单隐层网络
激活函数	Relu
损失函数	SoftmaxCrossEntropyLoss
其它参数	Learning Rate: 1e-1
	Weight Decay: 1e-4
	Momentum: 1e-4
	Batch Size: 100

下图展示了隐层规模为 400（约输入层规模 50%）和 200（约输入层规模 25%）时的训练曲线：

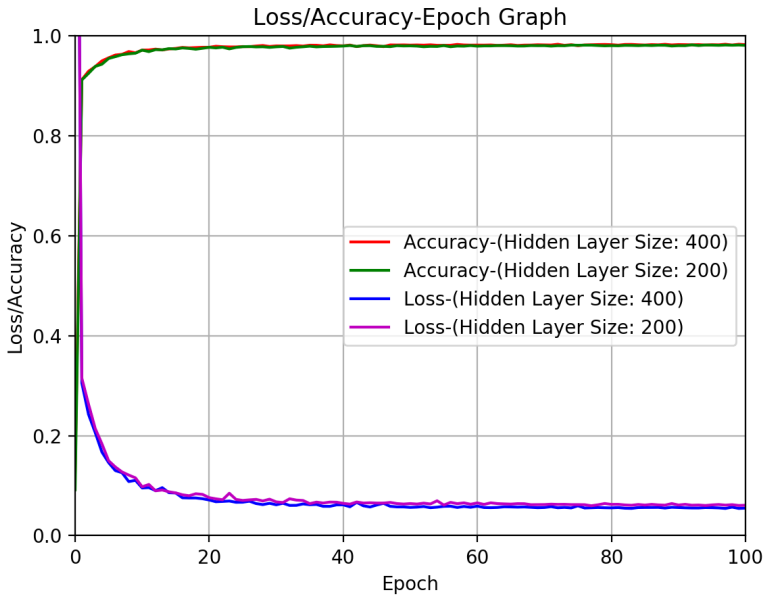
Loss/Accuracy-Epoch Graph
Loss/Accuracy
Epoch
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Accuracy-(Hidden Layer Size: 400)
Accuracy-(Hidden Layer Size: 200)
Loss-(Hidden Layer Size: 400)
Loss-(Hidden Layer Size: 200)

图 7: 训练曲线

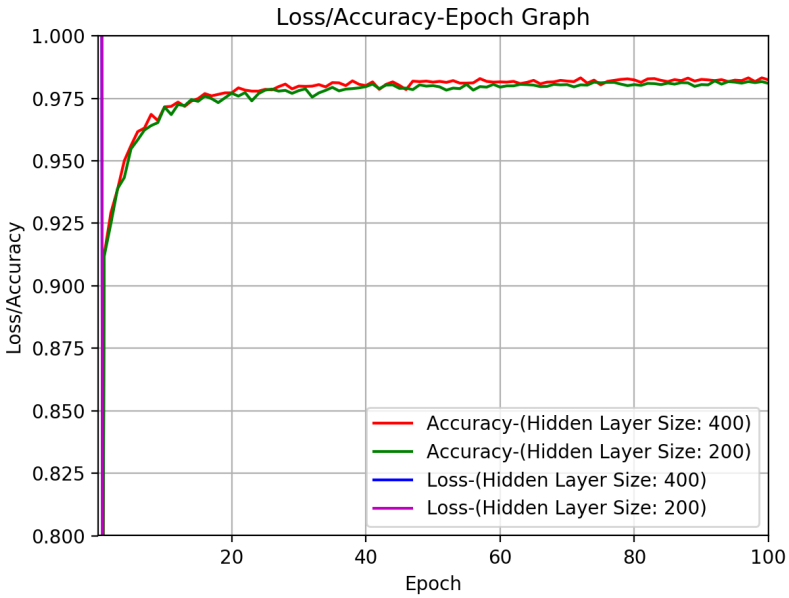

图 8: 放大的准确率曲线

表 7: 训练结果

隐层规模	**准确率**	**收敛性**	**收敛速度**	**训练速度**
400	98.32%	好	很快	较快
200	98.20%	好	很快	快

从以上图表我们可以看出，针对数字分类任务，隐层规模对训练结果影响似乎不大，至少相比激活函数和网络层数的影响小很多，两组数据均收敛性较好，收敛速度较快，有可能只是笔者调整的幅度较小，对比数据较少。从结果来看，隐层规模稍大一些结果会更好，但是由于计算量变大，训练速度会变慢。

3.5 损失函数对比

在这一步中我们统一使用 Relu 激活函数，对比不同损失函数对结果的影响。

表 8: 网络参数

网络结构	(784, 400, 10) 单隐层网络
激活函数	Relu
损失函数	EuclideanLoss
	SoftmaxCrossEntropyLoss
其它参数	Learning Rate: 1e-1
	Weight Decay: 1e-4
	Momentum: 1e-4
	Batch Size: 100

下图展示了损失函数分别为 Euclidean 和 SoftmaxCrossEntropy 时的训练曲线：

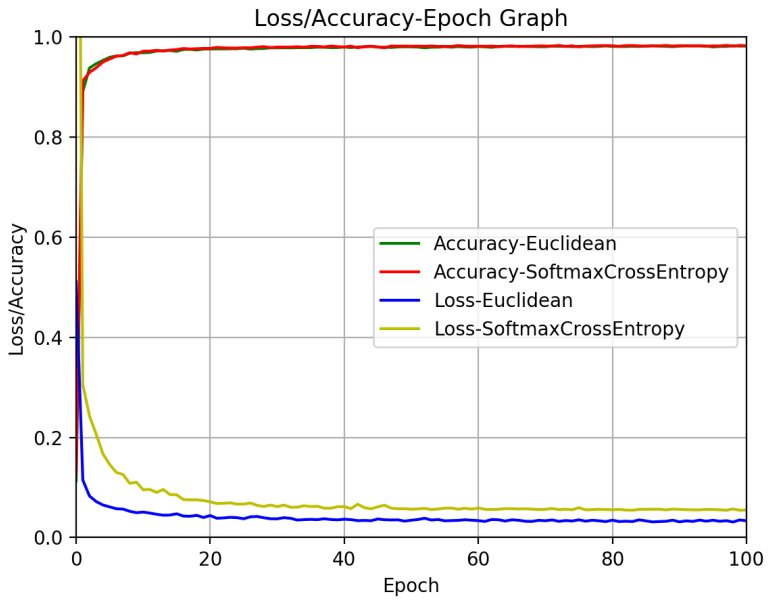

图 9: 训练曲线

表 9: 训练结果

损失函数	准确率	收敛性	收敛速度	训练速度
Euclidean	98.25%	好	很快	较快
SoftmaxCrossEntropy	98.32%	好	很快	较快

从以上图表我们可以清晰的看出，针对数字分类任务，使用 EuclideanLoss 比 SoftmaxCrossEntropyLoss 效果都很好，准确率都较高，收敛性都很好，收敛速度都很快。由于损失函数计算方法不同，对比损失 Loss 的数值没有意义，从准确性 Accuracy 来看，相差很小，故损失函数对该任务影响不大。

3.6 其它参数选择

在这一步中我们统一使用 Relu 激活函数，网络结构均采用单隐层，对比其它超参数对结果的影响。

表 10: 网络参数

网络结构	(784, 200, 10) 单隐层网络
激活函数	Relu
损失函数	SoftmaxCrossEntropyLoss
其它参数	Learning Rate: 1e-2, 1e-1, 1e0
	Weight Decay: 1e-4, 0
	Momentum: 1e-4, 0
	Batch Size: 100

3.6.1 学习率 Learning Rate

下图展示了学习率 Learning Rate 分别为 1e0，1e-1，1e-2 时的训练曲线：

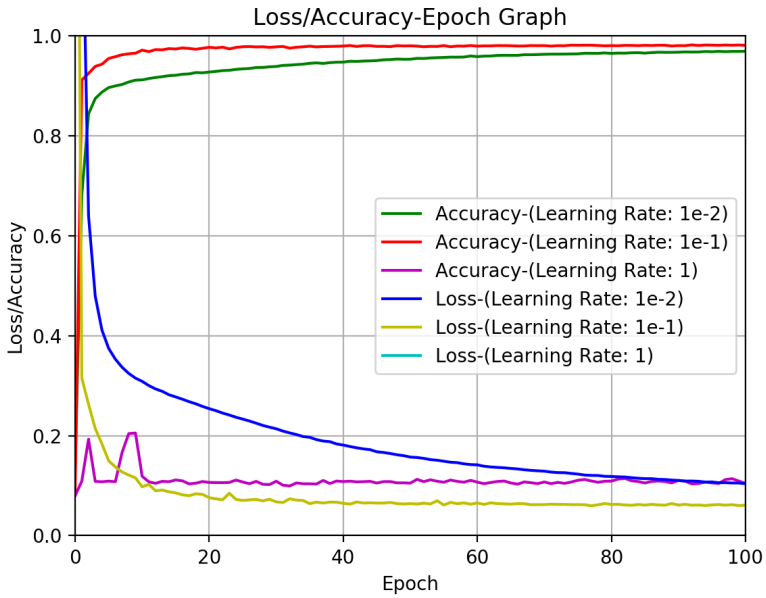

图 10: 训练曲线

表 11: 训练结果

Learning Rate	**准确率**	**收敛性**	**收敛速度**
1e-2	96.92%	较好	一般
1e-1	98.20%	好	很快
1e-0	20.53%	不收敛	不收敛

学习率决定了梯度下降的速度。从以上图表我们可以清晰的看出，学习率对收敛性影响很大，故对训练结果影响很大。

学习率太小（参考学习率为 1e-2 的曲线），由于每步结果改变较小，收敛较慢，到达收敛值需要较长的训练时间。学习率太大（参考学习率为 1e0 的曲线），由于每步结果改变太大，可能不收敛或收敛很慢。故实际应用中，必须选择一个适当的学习率，学习率适中时（参考学习率为 1e-1 的曲线），不仅收敛性好，而且收敛速度快。

3.6.2 权衰减 Weight Decay

下图展示了权衰减分别为 1e-4 和 0 时的训练曲线：

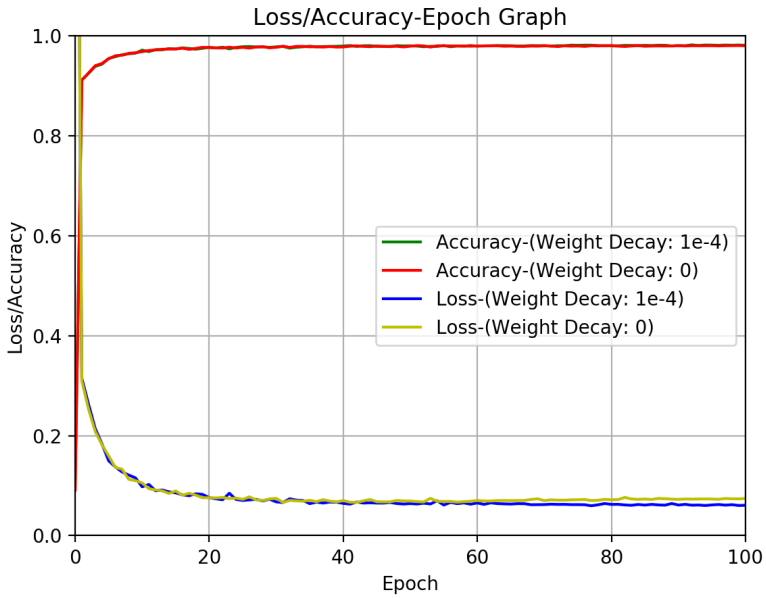

图 11: 训练曲线

表 12: 训练结果

Weight Decay	**准确率**	**收敛性**	**收敛速度**
1e-4	98.20%	好	快
0	98.06%	好	快

通过微积分可以看出，权衰减实际上是正则项之前的系数，正则项一般反映了模型的复杂程度，其最终目的是避免过拟合，即过度拟合训练集，在测试集上表现反而变差。从以上图表我们可以看出，权衰减对收敛性等影响不大，对结果影响也不大。

笔者猜测一方面可能是因为所设权衰减数值较小，另一方面也可能是因为针对数字分类任务，所用网络都较为简单，并不容易造成过拟合，这也可以通过之前的训练曲线看出来，训练曲线基本都没有发生过拟合现象。在实际应用中，适度设置正则项可以避免过拟合，从而提高准确率。

3.6.3 动量 Momentum

下图展示了动量分别为 1e-4 和 0 时的训练曲线：

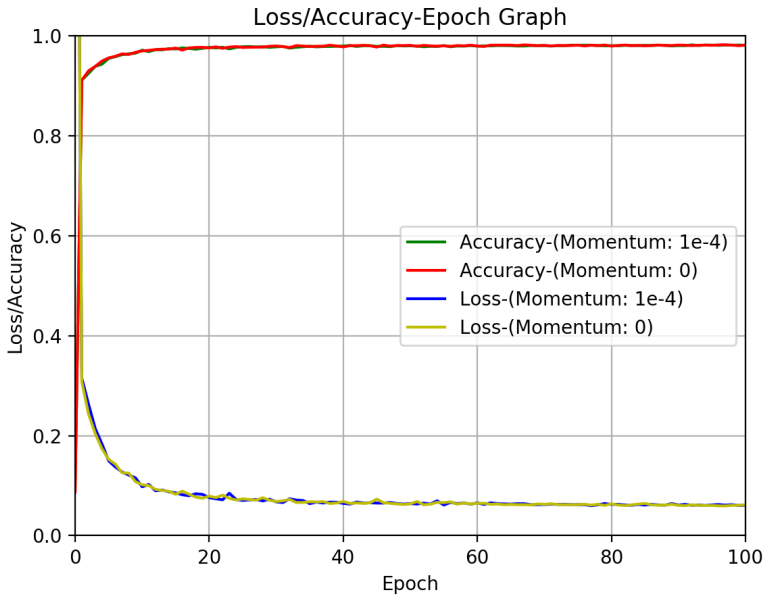

图 12: 训练曲线

表 13: 训练结果

Momentum	**准确率**	**收敛性**	**收敛速度**
1e-4	98.20%	好	快
0	98.21%	好	快

动量是是梯度下降法中一种常用的加速技术，如果上一次的动量与这一次的负梯度方向是相同的，那这次下降的幅度就会加大，所以这样做能够达到加速收敛的过程。由训练曲线也可以看出，动量的设置对结果影响很小。

笔者猜测这可能是因为针对数字分类任务，本身网络收敛速度就已经很快了，很难体现出差别。

4 实验心得

通过本次实验，我大大加深了对多层感知器（MLP）的理解，初步掌握了 MLP 的搭建与参数选择技巧，进入了机器学习世界的大门。

在这次实验中，我充分感受到了 MLP 的神奇之处，通过这样极为简单的连接，识别准确率竟然可以达到 98.69%，接近 100%，而且是在噪声较高的数据集上进行的测试，实际表现应该会更好。

在这次实验中，笔者就训练曲线、激活函数、隐层层数、隐层规模、损失函数、其他超参数进行了系统的对比与分析，更加深入的理解与掌握了这些概念。

同时在本次实验中，提供的代码框架让我对 Python 的 OOP 有了更为深刻的理解，并被框架简洁与优雅的代码风格所叹服，希望今后自己也能写出如此优质的代码。

但是这只是一个入门，更高层次的神经网络和参数设置技巧我还没有完全掌握，还需要进一步的练习与实践。

5 后期工作

由于时间和精力有限，我认为可以通过以下方式进行改进，留给以后的工作。

(1) 对训练集进行处理，例如适当旋转一定角度，适当放大或缩小，从而获得更多的训练数据，使得训练更为充分。

(2) 对测试集进行处理，例如去噪声，比如有的测试数据有一个黑点，可以采用深度优先搜索和设置阈值的方法得以实现。

(3) 尝试更多的参数，对比不同参数对训练效果的影响。

(4) 采用 CNN、RNN、LSTM 等不同的神经网络进行训练与测试，对比结果。

参考文献

[1] 人工神经网络, 黄民烈, 2017.