

# 人工神经网络实验报告 • MLP

周正平 • 2015011314 • zhouzp15@mails.tsinghua.edu.cn

2017 年 10 月 9 日

## Contents

<b>1</b>	<b>实验内容</b>	<b>2</b>
1.1	记号说明	2
1.2	总体设计	2
<b>2</b>	<b>算法实现</b>	<b>2</b>
2.1	全连接层	2
2.1.1	模块描述	3
2.1.2	原理推导	3
2.1.3	算法实现	3
2.2	激活函数	4
2.2.1	模块描述	4
2.2.2	算法实现	4
2.3	损失函数	4
2.3.1	模块描述	5
2.3.2	算法实现	5
<b>3</b>	<b>实验结果</b>	<b>5</b>
3.1	单隐层网络	5
3.2	双隐层网络	6
<b>4</b>	<b>对比分析</b>	<b>7</b>
4.1	激活函数	8
4.2	隐层数	9
4.3	隐层节点数	9
4.4	损失函数	10
4.5	learning_rate	11
4.6	weight_decay	12
4.7	momentum	13

# 1 实验内容

## 1.1 记号说明

为叙述方便起见，以下对报告中出现的各种记号及其语义作一说明：

记号	语义说明
$x_{ik}$ or $x_k$	(第 $i$ 个线性层的) 输入向量的第 $k$ 个分量
$u_{ik}$ or $u_k$	(第 $i$ 个线性层的) 输出向量的第 $k$ 个分量
$y_{ik}$ or $y_k$	(第 $i$ 个线性层的激活函数的) 输出向量的第 $k$ 个分量
$W_k$	$W$ 矩阵的第 $k$ 列
$t_k$	目标输出向量 (正确标签) 的第 $k$ 个分量
$E$	损失函数值

## 1.2 总体设计

本次实验要求设计多层感知器 (MLP) 用于完成 MNIST 手写数字识别任务。以含有一个隐层的 MLP 为例，实验中所用网络结构如下：

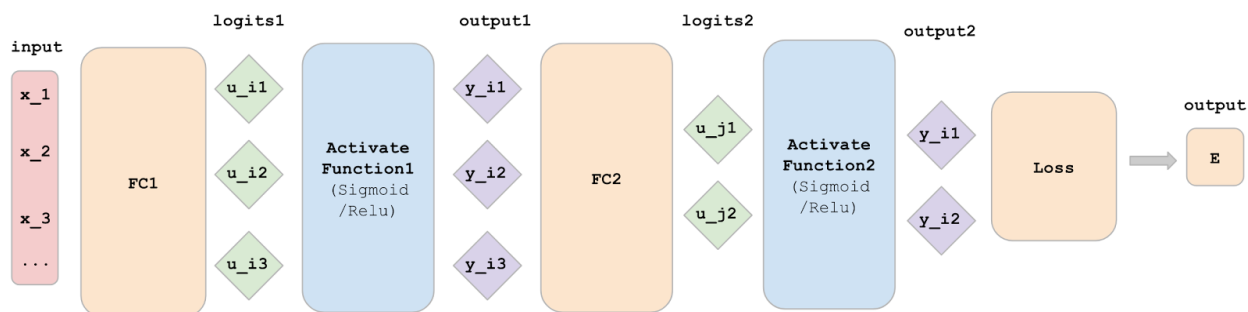


Figure 1: 网络结构总览

总体来说，本次作业需要在给出的框架的基础之上，实现以下几个模块：

- 全连接层：图中的 FC1 和 FC2，公式为  $u = xW + b$ ；
- 激活函数：图中的 Activate Function，本次作业涉及 Relu 和 Sigmoid 两种；
- 损失函数：图中的 Loss，本次作业涉及 EuclideanLoss 和 SoftmaxCrossEntropyLoss 两种。

# 2 算法实现

## 2.1 全连接层

全连接层通过函数  $u = xW + b$  实现仿射变换，将输入向量变换到另一个线性空间。全连接层的导数推导等相对复杂，以下将较为详尽地阐明其推导过程及实现方法。

### 2.1.1 模块描述

对一个 batch 的训练样本而言，该模块基本描述如下（尽管输入输出对单个训练样本而言均为一维向量，但考虑整个 batch，输入输出向量均需加入第一个维度 batch\_size）：

- 输入： $m$  维行向量  $x$  [batch\_size,  $m$ ];
- 输出： $n$  维行向量  $u$  [batch\_size,  $n$ ];
- 参数： $m \times n$  权重矩阵  $W$  [ $m, n$ ]、 $n$  维偏置行向量  $b$  [ $1, n$ ]。

### 2.1.2 原理推导

计算式  $u = xW + b$  即为 Forward 计算公式；关键在于 Backward 公式的推导。由于梯度的反向传播关键在于求出输出  $u$  对于  $x, W, b$  的偏导数，而 numpy 实现了高效的矩阵计算，故以下结合线性代数知识，将梯度计算向量化：

$$\begin{aligned} \because u &= xW + b \\ \therefore u_{k_1} &= \sum_{k_2} x_{k_2} W_{k_2 k_1} + b_{k_1} \end{aligned}$$

- 输出  $u$  对输入  $x$  的导数  $\frac{\partial u}{\partial x}$ ：

$$\begin{aligned} \rightarrow \frac{\partial u_{k_1}}{\partial x_{k_2}} &= W_{k_2 k_1} \\ \rightarrow \frac{\partial u}{\partial x} &= W^T \end{aligned} \tag{1}$$

- 输出  $u$  对权重矩阵  $W$  的导数  $\frac{\partial u}{\partial W}$ ：

$$\begin{aligned} \rightarrow \frac{\partial u_{k_1}}{\partial W_{k_2 k_1}} &= x_{k_1} \\ \rightarrow \frac{\partial u}{\partial W_k} &= \text{diag}(x) \end{aligned} \tag{2}$$

- 输出  $u$  对偏置向量  $b$  的导数  $\frac{\partial u}{\partial b}$ ：

$$\begin{aligned} \rightarrow \frac{\partial u_{k_1}}{\partial b_{k_2}} &= \delta_{k_1 k_2} \\ \rightarrow \frac{\partial u}{\partial b} &= I \end{aligned} \tag{3}$$

### 2.1.3 算法实现

原理推导所得结论总结如下：

Forward（公式）	Backward（导数）	Backward（链式法则）
$u = xW + b$	$\frac{\partial u}{\partial x} = W^T$	$\text{grad\_input} = \text{grad\_output} \cdot W^T$
	$\frac{\partial u}{\partial W_k} = \text{diag}(x)$	$\text{grad\_W} = x^T \cdot \text{grad\_output}$
	$\frac{\partial u}{\partial b} = I$	$\text{grad\_b} = \text{grad\_output}$

结合导数运算的链式法则，Backward 代码中实现了以下逻辑：

- 返回值 **grad\_input**:  $grad\_input = grad\_output \cdot W^T$

```
grad_input = np.matmul(grad_output, np.transpose(self.W))
```

- 权值矩阵梯度 **grad\_W**:  $grad\_W = x^T \cdot grad\_output$

```
self.grad_W = np.matmul(np.transpose(self._saved_input), grad_output)
```

- 偏置向量梯度 **grad\_b**:  $grad\_b = grad\_output$

```
self.grad_b = grad_output
```

## 2.2 激活函数

激活函数为网络引入非线性，本次实验涉及 Sigmoid 及 Relu 两种。

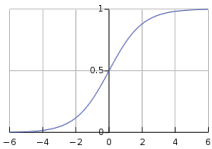
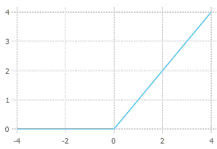
### 2.2.1 模块描述

总体来看，激活函数不改变输入向量的形状，只是逐元素进行函数运算。

- 输入:  $n$  维行向量  $u$   $[batch\_size, n]$ ;
- 输出:  $n$  维行向量  $y$   $[batch\_size, n]$ 。

### 2.2.2 算法实现

激活函数的原理与实现相对简单，在代码实现时，只需注意根据链式法则，将下述导数公式再乘以输出处的梯度值  $grad\_output$  即可。以下进行简要总结：

激活函数	Forward（公式）	Backward（导数）	函数图像
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	
Relu	$f(x) = \max(x, 0)$	$f'(x) = (x > 0)$	

## 2.3 损失函数

损失函数定义了优化的量化指标，本次实验涉及 EuclideanLoss 和 SoftmaxCrossEntropyLoss 两种。

### 2.3.1 模块描述

损失函数比较两个向量（或两个分布），通过某种方法计算其差别程度，得到一个实值标量作为输出。

- 输入： $n$  维行向量  $y, t$  [ $batch\_size, n$ ]，即预测向量与目标向量；
- 输出：一个实值标量  $E$ ，即损失值  $loss$ 。

### 2.3.2 算法实现

以下简要总结损失函数及其导数运算：

损失函数	Forward（公式）	Backward（导数）
EuclideanLoss	$f(x) = \frac{1}{2} \sum_k (y_k - t_k)^2$	$f'(x) = y - t$
SoftmaxCrossEntropyLoss	$f(x) = -\sum_k t_k \log p_k, p_k = \frac{e^{y_k}}{\sum e^y}$	$f'(x) = p - t$

需注意在 Forward 及 Backward 代码实现时，要将总的  $loss$  除以  $batch\_size$ ：

- **EuclideanLoss:**

```
# Forward
return 0.5 * np.mean(np.square(input - target), axis=1)
```

```
# Backward
return (input - target) / len(input)
```

- **SoftmaxCrossEntropyLoss:**

```
# Forward
cross_entropy = -np.mean(np.sum(target * np.log(self.softmax), axis=1))
```

```
# Backward
return (self.softmax - target) / len(input)
```

## 3 实验结果

### 3.1 单隐层网络

对于单隐层 MLP 网络，经调参优化后，在测试集上获得的最高精度为 98.32%。

网络超参数配置如下：

( $Linear[784, 392] \rightarrow Relu \rightarrow Linear[392, 10] \rightarrow SoftmaxCrossEntropyLoss$ ):

超参数	取值
隐层节点数	392
激活函数	Relu
损失函数	SoftmaxCrossEntropyLoss
learning_rate	0.1
weight_decay	0.0001
momentum	0.0001
batch_size	100
max_epoch	100

绘制损失/精度曲线如下（相邻采样点间隔 50 个 iteration）：

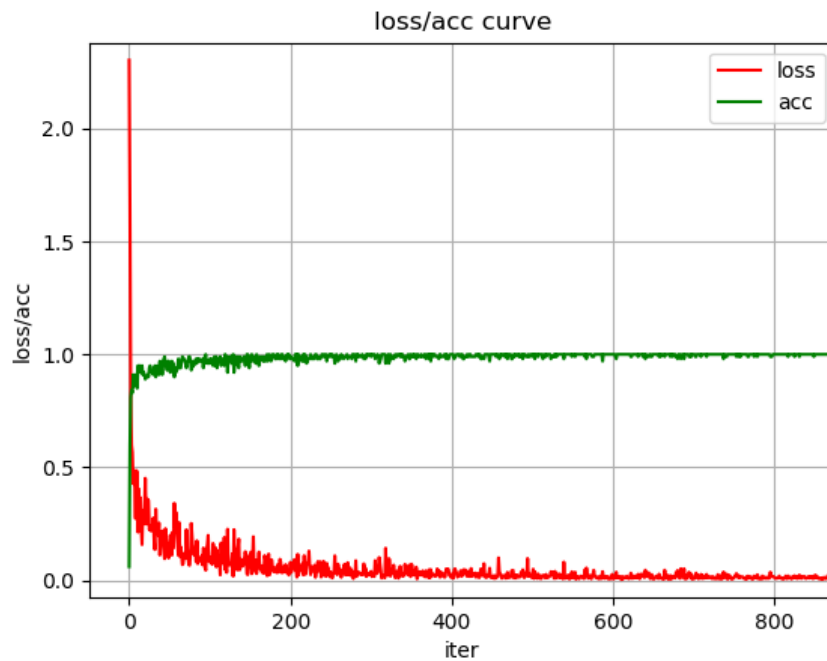


Figure 2: 损失/精度 -迭代次数曲线（单隐层网络）

## 3.2 双隐层网络

对于双隐层 MLP 网络，经调参优化后，在测试集上获得的最高精度为 98.75%。

网络超参数配置如下：

(*Linear*[784, 392]  $\rightarrow$  *Relu*  $\rightarrow$  *Linear*[392, 196]  $\rightarrow$  *Relu*  $\rightarrow$  *Linear*[196, 10]  $\rightarrow$  *EuclideanLoss*)

超参数	取值
隐层 1 节点数	392
隐层 2 节点数	196
激活函数 1	Relu
激活函数 2	Relu
损失函数	EuclideanLoss
learning_rate	0.1
weight_decay	0.0002
momentum	0.0001
batch_size	100
max_epoch	100

绘制损失/精度曲线如下（相邻采样点间隔 50 个 iteration）：

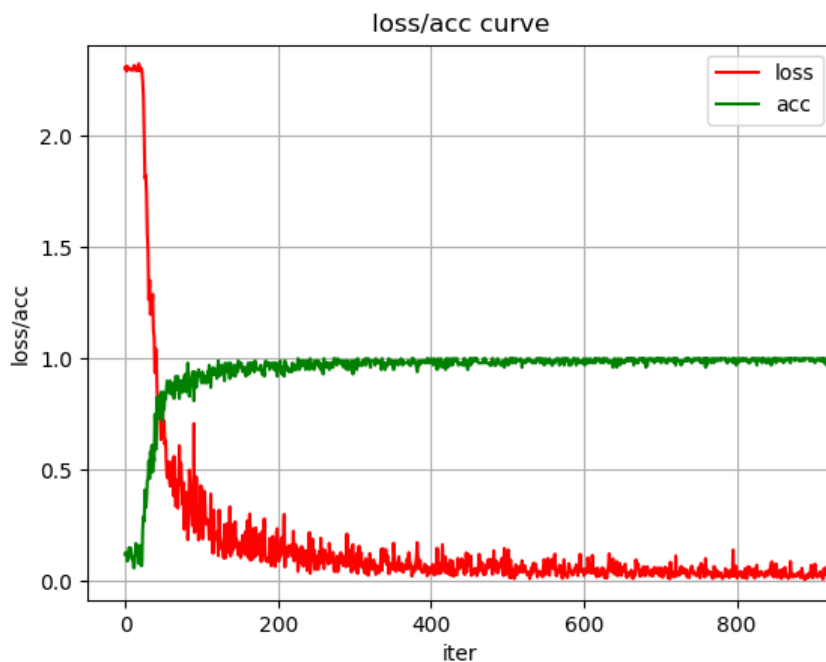


Figure 3: 损失/精度 -迭代次数曲线（双隐层网络）

## 4 对比分析

超参数的调节是优化 MLP 精度的重要因素。以下列出本节对比分析的主要超参数（分析某一超参数的影响时，单、双隐层网络的其他超参数均默认与上一节中的最优值相同）：

超参数	备选值	备注
激活函数	( <i>Sigmoid</i> , <i>Relu</i> )	以单、双隐层网络对比分析
隐层数	(1, 2)	以单、双隐层网络对比分析
隐层节点数	(25%, 50%, 75%) $\times$ 784	以单隐层网络对比分析
损失函数	( <i>EuclideanLoss</i> , <i>SoftmaxCrossEntropyLoss</i> )	以单隐层网络对比分析
learning_rate	(1, 0.1)	以单隐层网络对比分析，分激活函数讨论
weight_decay	(0, 0.0002)	以双隐层网络对比分析
momentum	(0, 0.0001)	以双隐层网络对比分析

## 4.1 激活函数

以单、双隐层网络为例，从训练时间、收敛性、精度三个方面对比分析 Sigmoid、Relu 两种激活函数：

隐层数目	激活函数	训练时间（100 个 epoch）	收敛性	最高测试精度
单	Sigmoid	10.5min	收敛较慢	96.99 %
	Relu	6.5min	收敛较快	98.32 %
双	Sigmoid	14min	收敛极慢	92.30%
	Relu	12min	收敛较快	98.75 %

对单隐层网络绘制测试精度曲线如下（相邻采样点间隔 1 个 epoch）：

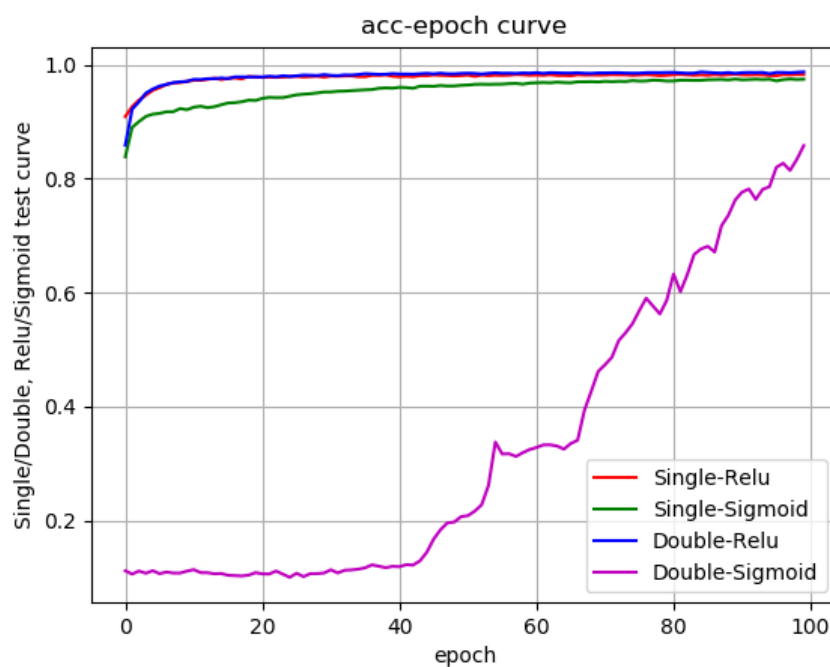


Figure 4: Sigmoid/Relu 精度曲线（单隐层网络）



从图中可以清晰地看出，对单隐层网络而言，Relu 函数相比 Sigmoid 不仅收敛较快（因 Relu 梯度在  $x > 0$  时恒为 1，而 Sigmoid 在  $|x|$  很大时梯度  $\rightarrow 0$ ），训练时间更短（因 Relu 函数不涉及指数及乘除运算），最终测试精度也较高。

对于双隐层网络而言，笔者在实验中发现 Sigmoid 函数收敛极慢，而 Relu 则能够较快地收敛到最优值。这可能是因为网络加深后，Sigmoid 介于  $[0, 1]$  间的导数使得反向传播的梯度越来越小，使得收敛缓慢。

## 4.2 隐层数

以单、双隐层两种网络为例，从训练时间、收敛性、精度三个方面对比分析：

隐层数	训练时间（100 个 epoch）	收敛性	最高测试精度
单	6.5min	收敛速度无显著差别	98.32 %
双	12min	收敛速度无显著差别	98.75 %

绘制测试精度曲线如下（相邻采样点间隔 1 个 epoch）：

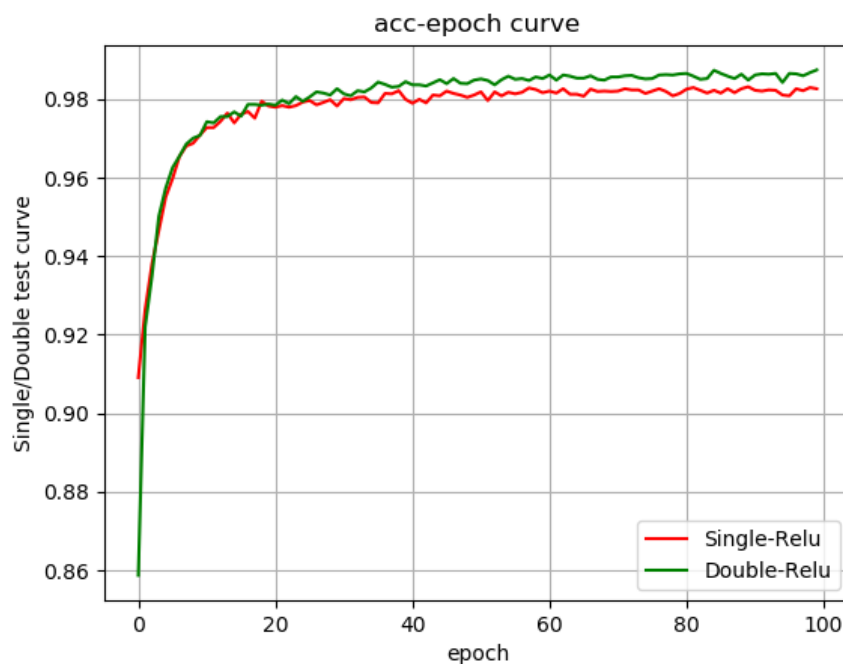


Figure 5: 单隐层/双隐层精度曲线

从图中可以看出，在 MNIST 手写数字分类任务中，双隐层网络的测试精度优于单隐层网络，但网络结构更复杂，参数增多，训练时间较长。故而在实际应用中，需要在测试精度与训练时间之间进行权衡取舍。

## 4.3 隐层节点数

以单隐层网络为例，从训练时间、收敛性、精度三个方面对比分析：

隐层节点数	训练时间（100 个 epoch）	收敛性	最高测试精度
$25\% \times 784 = 196$	6.5min	收敛波动较大	98.30 %
$50\% \times 784 = 392$	12min	收敛最稳定	98.32 %
$75\% \times 784 = 588$	20min	收敛较稳定	98.24 %

绘制测试精度曲线如下（相邻采样点间隔 1 个 epoch）：

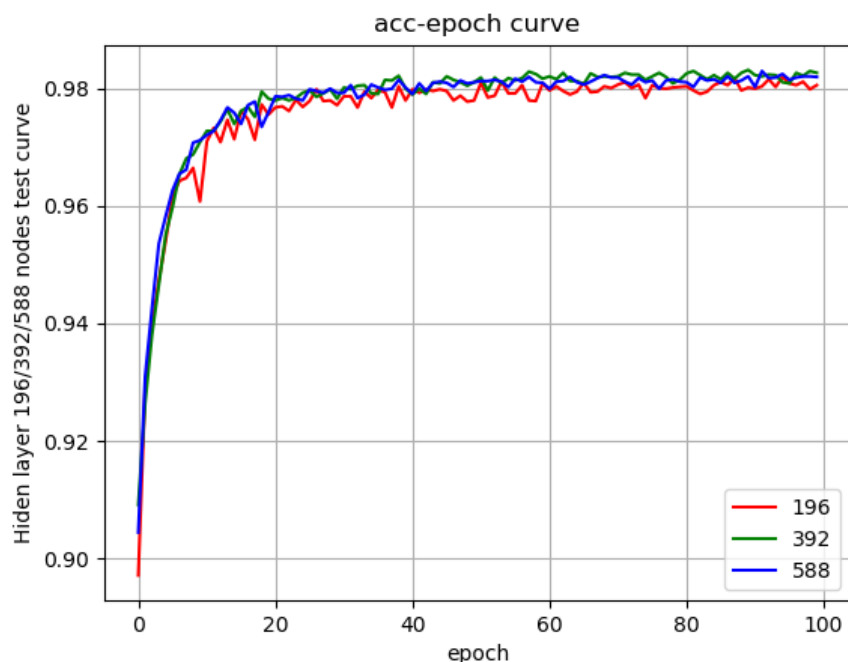


Figure 6: 隐层节点数目 196/392/588 精度曲线（单隐层网络）

从图中可以看出，对于单隐层网络而言，隐层节点数目对测试精度的影响不大，训练曲线甚至极为相近。考虑到训练时长与隐层节点数呈正相关以及测试精度，本次实验选择 392 个隐层节点作为最终的超参数。

类似地，在双隐层的网络中，隐层 1、隐层 2 的节点数分别为 392、196 时，训练精度较高。笔者猜测在设计 MLP 时，也可适当借鉴 CNN 的池化思想，即隐层节点数目依次减少（实验中呈几何级数），保留关键信息（当然，这只是笔者的一点猜测，MLP 与 CNN 在特征提取方面仍存在显著的差异）。

#### 4.4 损失函数

以单隐层网络为例，从训练时间、收敛性、精度三个方面对比 EuclideanLoss 与 SoftmaxCrossEntropyLoss：

损失函数	训练时间（100 个 epoch）	收敛性	最高测试精度
EuclideanLoss	6min	收敛性无显著差别	98.30 %
SoftmaxCrossEntropyLoss	6.5min	收敛性无显著差别	98.32 %

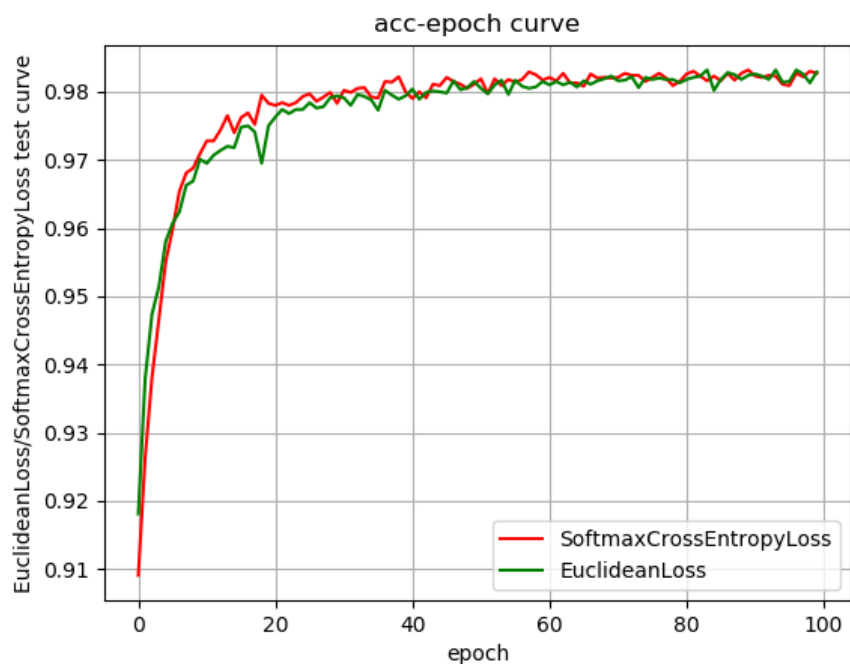


Figure 7: EuclideanLoss/SoftmaxCrossEntropyLoss 精度曲线（单隐层网络）

从图中可以看出，对单隐层网络而言，SoftmaxCrossEntropyLoss 相对 EuclideanLoss 而言，精度上稍高一些；然而 softmax 的计算公式涉及指数运算，在输出层节点数较大时，这将导致较大的计算成本。

#### 4.5 learning\_rate

以单隐层网络对比分析，分 Sigmoid/Relu 两激活函数进行讨论：

激活函数	learning_rate	训练时间（100 个 epoch）	收敛性	最高测试精度
Sigmoid	1	10.5min	收敛较快，波动较大	97.57 %
	0.1	10.5min	收敛较慢，波动较小	96.99 %
Relu	1	-	不收敛	-
	0.1	6.5min	收敛快，稳定	98.32 %

绘制测试精度曲线如下（相邻采样点间隔 1 个 epoch）：

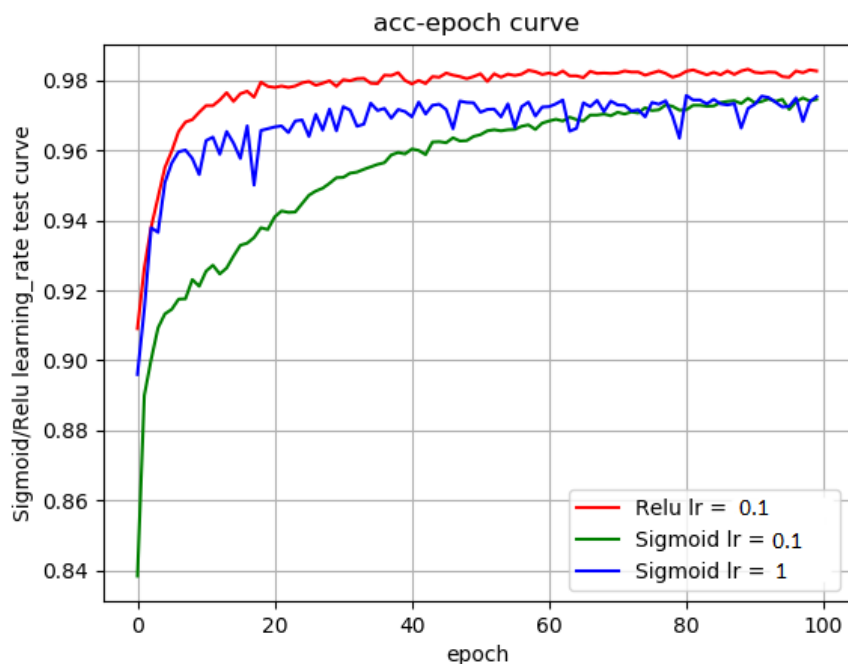


Figure 8: Sigmoid/Relu 在不同 learning\_rate 下的精度曲线（单隐层网络）

从图中可以看出，对于梯度值较大的 Relu 函数，在 learning\_rate 较大（1）时不能收敛；对于梯度值较小的 Sigmoid 函数，较大的 learning\_rate 可以加快收敛速度，但同时也使得测试准确率在整个训练过程中不太稳定，可能是因为这个 learning\_rate 在接近收敛时显得太大，导致在极小值附近出现震荡现象。

综上所述，learning\_rate = 0.1 的 Relu 函数收敛快，较为稳定，同时测试精度较高，对单隐层网络来说效果最佳。

#### 4.6 weight\_decay

在数学上，weight\_decay 本质上等价于平方正则项：将损失函数由  $E(W) = E_0(W)$  扩展为  $\hat{E}(W) = E_0(W) + \frac{\lambda}{2}W^2$ ，从而  $W$  的更新公式也由  $W_k \leftarrow W_k - \eta \frac{\partial E}{\partial W_k}$  变为  $W_k \leftarrow W_k - \eta (\frac{\partial E}{\partial W_k} + \lambda W_k)$ （其中  $\lambda$  为 weight\_decay， $\eta$  为 learning\_rate）。

以双隐层网络为例进行对比分析：

weight_decay	训练时间（100 个 epoch）	收敛性	最高测试精度
0	6.5min	收敛速度无显著差别，波动很大	98.14 %
0.0002	12min	收敛速度无显著差别，比较稳定	98.75 %

绘制测试精度曲线如下（相邻采样点间隔 1 个 epoch）：

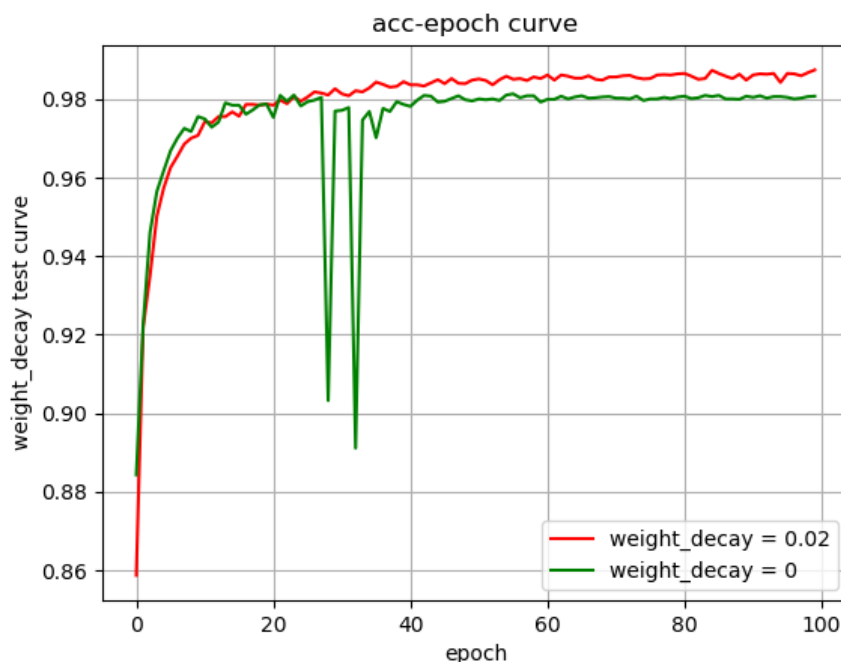


Figure 9: 不同 `weight_decay` 下的精度曲线（双隐层网络）

从图中可以看出，对双隐层网络而言，`weight_decay = 0.0002` 在收敛稳定程度上优于 `weight_decay = 0`，而在精度上前者稍占优势。笔者猜测，这可能是由于正则项惩罚过大的权重值，从而反向传播时的梯度不致过大，故每次迭代网络中各参数向量的变化量相对较小，因此测试精度不致有大的波动，稳定性得到增强。

## 4.7 momentum

`momentum` 的数学表达式为  $\Delta W_k \leftarrow \gamma \Delta W_k + \frac{\partial \hat{E}}{\partial W_k}$ ，其物理意义可以类比惯性，在神经网络的训练中则起到“在一定程度上保持原有更新方向”的作用。例如当训练陷入局部极小值时，梯度项几乎为 0，而动量项维持了原有更新方向，有助于借“惯性”跃出局部极小值；而当训练到达全局最小值时，尽管动量项仍维持原有更新方向，这股“惯性”不足以支撑它跃出真正的最小值谷底。此外，由于在收敛过程中更新方向基本保持不变，`momentum` 在理论上可以加速收敛的过程。

以双隐层网络为例进行对比分析：

<code>momentum</code>	训练时间（100 个 epoch）	收敛性	最高测试精度
0	12min	收敛速度无显著差别，波动大	98.30 %
0.0001	12min	收敛速度无显著差别，较稳定	98.75 %

绘制测试精度曲线如下（相邻采样点间隔 1 个 epoch）：

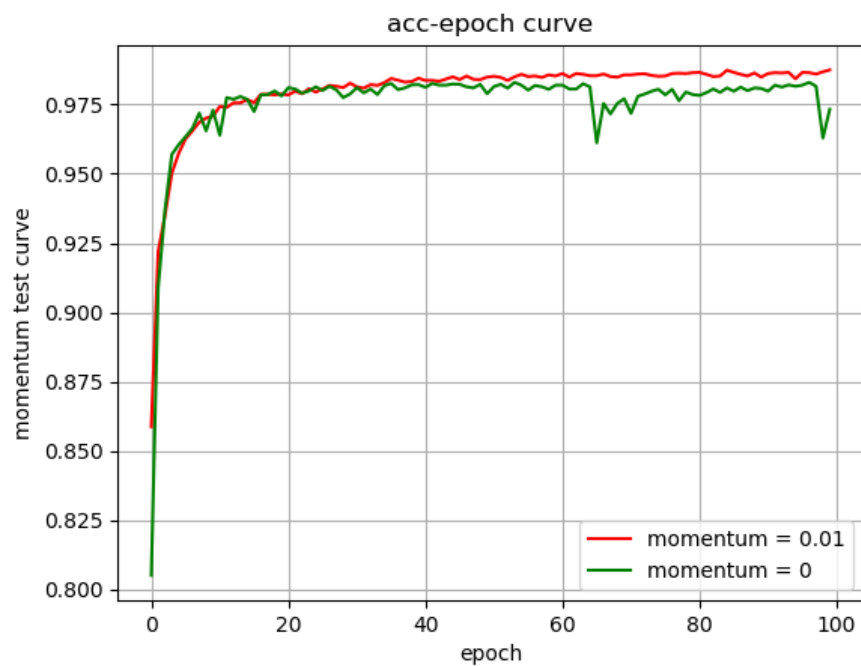


Figure 10: 不同 momentum 下的精度曲线（双隐层网络）

从图中可以看出，由于笔者设置的 momentum 值相差不大，在收敛速度上二者无显著差别；而在稳定性上，加入 momentum 似乎减小了收敛过程中的波动程度，笔者猜测可能是因为各个 batch 之间存在差别，加入动量项之后前一 batch 的更新方向会对本次 batch 产生影响，从而减小了各个 batch 之间更新方向的差别，使得收敛稳定性提高。