
目 录

一 项目说明-----	- 1 -
1. 1 项目内容-----	- 1 -
1. 2 Pytorch 框架介绍-----	- 1 -
1. 3 数据集描述-----	- 2 -
二 监督学习 Supervised learning-----	- 3 -
2. 1 逻辑回归 Logistic Regression-----	- 3 -
2. 1. 1 算法解析-----	- 3 -
2. 1. 2 实验分析-----	- 5 -
2. 2 多层感知机 Multilayer perceptron-----	- 7 -
2. 2. 1 算法解析-----	- 7 -
2. 2. 2 实验分析-----	- 10 -
2. 3 深度卷积网络 Deep Convolutional Network-----	- 11 -
2. 3. 1 算法解析-----	- 11 -
2. 3. 2 实验分析-----	- 13 -
2. 4 长短期记忆网络 Long Short Term Memory networks-----	- 17 -
2. 4. 1 算法解析-----	- 17 -
2. 4. 2 实验分析-----	- 18 -
三 无监督学习 Unsupervised learning-----	- 21 -
3. 1 自编码 Auto Encoders-----	- 21 -
3. 1. 1 算法解析-----	- 21 -

3.1.2 实验分析-----	- 22 -
3.2 降噪自编码 Denoising Autoencoders-----	- 23 -
3.2.1 算法解析-----	- 23 -
3.2.2 实验分析-----	- 24 -
3.3 堆叠降噪自编码 Stacked Denoising Auto-Encoders-----	- 25 -
3.3.1 算法解析-----	- 25 -
3.3.2 实验分析-----	- 26 -
3.4 受限波尔兹曼 Restricted Boltzmann Machines-----	- 28 -
3.4.1 算法解析-----	- 28 -
3.4.2 实验分析-----	- 30 -
3.5 深度置信网络 Deep Belief Networks-----	- 32 -
3.5.1 算法解析-----	- 32 -
3.5.2 实验分析-----	- 33 -
四 项目总结-----	- 35 -
4.1 监督学习算法综合分析-----	- 35 -
4.2 无监督学习算法综合分析-----	- 36 -
4.3 总结-----	- 37 -

一 项目说明

1.1 项目内容

使用 Pytorch (<https://pytorch.org/>) 这一深度学习框架实现深度学习的一些基础经典的算法，包括 Logistic Regression、Multilayer perceptron、Deep Convolutional Network 和 Long Short Term Memory network 这四种监督学习算法，以及 Auto Encoders、Denoising Autoencoders、Stacked Denoising Auto-Encoders、Restricted Boltzmann Machines、Deep Belief Networks 这五种无监督学习算法。项目目录如下所示：

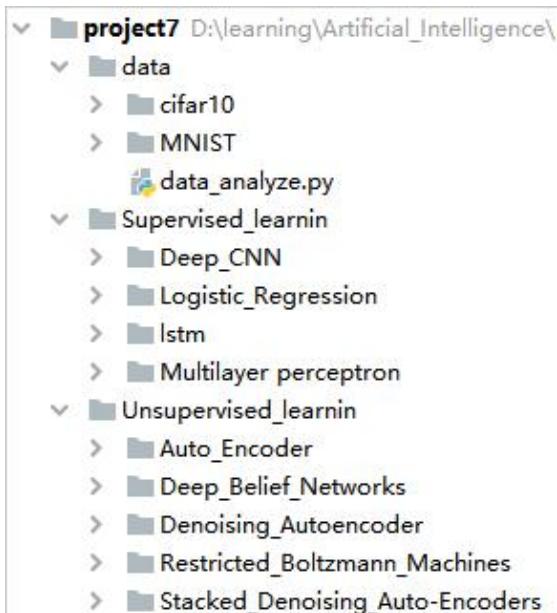


图 1.1

为比较以上四种监督学习算法和五种无监督学习算法间的异同，本项目使用经典的手写数字识别 Mnist 数据集作为统一的数据集进行实验。同时在第二、三章的各个章节中均对各个算法进行了算法解析和实验分析，最后在第四章中对项目中所用的所有算法进行总结分析。

1.2 Pytorch 框架介绍

2017 年 1 月，Facebook 人工智能研究院（FAIR）团队在 GitHub 上开源了 pyTorch (<https://pytorch.org/>)，并很快地占领了 GitHub 热度榜榜首。pytorch 的设计追求最少的封装，尽量避免重复造轮子，pytorch 的设计遵循 tensor->variable(autograd)->nn.Module 三个由低到高的抽象层次，分别代表高维数组（张量）、自动求导（变量）和神经网络（层/模块），这三个抽象之间联系紧密，可以同时进行修改和操作。

《深度学习框架 pytorch 入门与实践》这本书的作者曾说到：“当前开源的框架中，没有哪一个框架能够在灵活性、易用性、速度这三个方面有两个能同时超过 PyTorch。”足以显现 PyTorch 日趋重要的地位，下面是许多研究人员选择 PyTorch 的原因。

- **简洁**: PyTorch 的设计追求最少的封装，尽量避免重复造轮子。不像 TensorFlow 中充斥着 session、graph、operation、name_scope、variable、tensor、layer 等全新的概念，PyTorch 的设计遵循 tensor→variable(autograd)→nn.Module 三个由低到高的抽象层次，分别代表高维数组（张量）、自动求导（变量）和神经网络（层/模块），而且这三个抽象之间联系紧密，可以同时进行修改和操作。

简洁的设计带来的另外一个好处就是代码易于理解。PyTorch 的源码只有 TensorFlow 的十分之一左右，更少的抽象、更直观的设计使得 PyTorch 的源码十分易于阅读。在笔者眼里，PyTorch 的源码甚至比许多框架的文档更容易理解。

- **速度**: PyTorch 的灵活性不以速度为代价，在许多评测中，PyTorch 的速度表现胜过 TensorFlow 和 Keras 等框架。框架的运行速度和程序员的编码水平有极大关系，但同样的算法，使用 PyTorch 实现的那个更有可能快过用其他框架实现的。

- **易用**: PyTorch 是所有的框架中面向对象设计的最优雅的一个。PyTorch 的面向对象的接口设计来源于 Torch，而 Torch 的接口设计以灵活易用而著称，Keras 作者最初就是受 Torch 的启发才开发了 Keras。PyTorch 继承了 Torch 的衣钵，尤其是 API 的设计和模块的接口都与 Torch 高度一致。PyTorch 的设计最符合人们的思维，它让用户尽可能地专注于实现自己的想法，即所思即所得，不需要考虑太多关于框架本身的束缚。

- **活跃的社区** (<https://discuss.pytorch.org/>) : PyTorch 提供了完整的文档，循序渐进的指南，作者亲自维护的论坛 供用户交流和求教问题。Facebook 人工智能研究院对 PyTorch 提供了强力支持，作为当今排名前三的深度学习研究机构，FAIR 的支持足以确保 PyTorch 获得持续的开发更新，不至于像许多由个人开发的框架那样昙花一现。

1.3 数据集描述

Mnist 数据集：每张图是 28 * 28 的大小，在训练集中一共 60000 个样本，在测试集中一共是 10000 个样本，一共是 10 类手写数字。

其中在 pytorch 框架中 torchvision.datasets 中封装了 Mnist 数据集，本项目使用 pytorch 中的 DataLoader 来读取随机读取、显示和处理数据集。主要代码如下图所示。

```
train_dataset = datasets.MNIST(root='../../data/', train=True,
                               transform=transforms.ToTensor(), download=DOWNLOAD)

test_dataset = datasets.MNIST(root='../../data/', train=False,
                             transform=transforms.ToTensor(), download=DOWNLOAD)

train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size, shuffle=False)
```

图 1.2 数据下载以及数据加载

为了更直观的理解数据集，利用 matplotlib 随机显示了 16 个训练集中的手写数字以及其对应标签，如下图 3 所示。

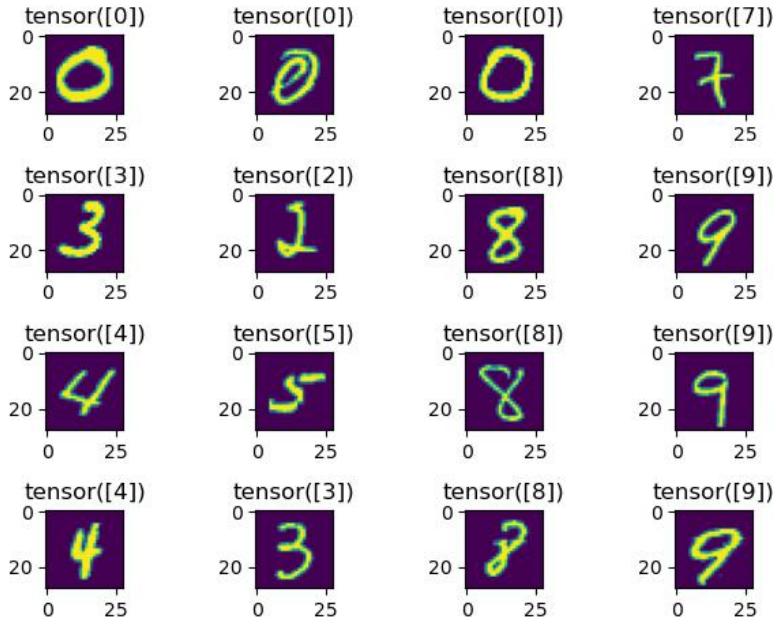


图 1.3 数据集中的手写数字以及其对应的标签

二 监督学习 Supervised learning

2.1 逻辑回归 Logistic Regression

2.1.1 算法解析

1) 算法简介

在本次逻辑回归实验中使用的是 sigmoid 函数作为激励函数，虽然逻辑回归名字中带有回归，但是它是一种分类算法。它无需事先假设数据分布，这样可以避免假设分布不准确所带来的问题；它不仅可以直接对分类结果进行预测，还可以得到属于该类别的概率。而且激励函数 sigmoid 函数是任意阶可导的凸函数，可以直接应用现有数值优化算法取到最优解。

2) 算法模型

设线性回归模型产生的预测值为： $\mathbf{z} = \mathbf{w}^T \mathbf{x} + b$

输出标记： $\mathbf{y} \in \{0,1\}$

(1) 对数几率

一个事件发生的概率为 p ，则该事件发生的几率为 $\frac{p}{1-p}$ ，那么该事件发生的对数几率为： $\ln \frac{p}{1-p}$ 。

(2) 对数几率函数

对数几率函数如图 (2) 所示，又称 sigmoid 函数： $y = \frac{1}{1+e^{-z}}$

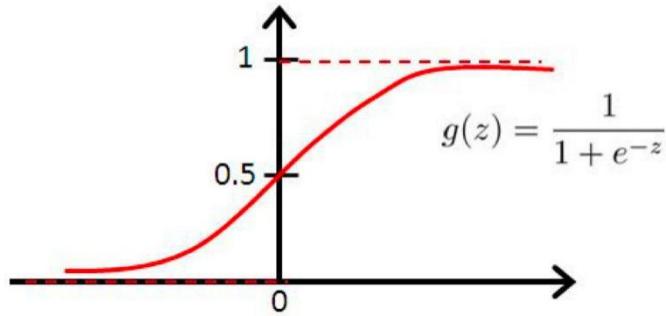


图 2.1 Sigmoid 函数

Sigmoid 函数具有很好的数学性质，是任意阶可导的凸函数，因此可以用来求解最优解，将 z 带进去，得到 $y = \frac{1}{1+e^{-(w^T x+b)}}$ 。

(3) 逻辑回归模型

事件 $y = 1$ 的对数几率为输入 x 的线性函数：即

$$\ln \frac{p(y=1|x)}{1-p(y=1|x)} = \ln \frac{p(y=1|x)}{p(y=0|x)} = w^T x + b$$

上述模型即为逻辑回归，通过利用损失函数，可以得到 w, b 的最优解。对于一个 x ，带入模型即可以得到 $p(y=1|x)$ 的值，若大于 0.5，就认为 $y=1$ ，否则，则认为 $y=0$ 。

(4) 损失函数

在这里使用的是对数损失函数，形式如下：

$$L(w,b) = -\frac{1}{N} \sum_{i=1}^N y_i \ln \hat{y}_i + (1-y_i) \ln (1-\hat{y}_i)$$

其中 \hat{y}_i 是预测值，其输出是介于 0~1 之间的连续概率值， y_i 是 0/1 的真实值，所以损失函数也可以写成如下形式：

$$L(w,b) = \begin{cases} -\frac{1}{N} \sum_{i=1}^N \ln \hat{y}_i & y = 1 \\ -\frac{1}{N} \sum_{i=1}^N (1 - \ln \hat{y}_i) & y = 0 \end{cases}$$

(5) 优化求解

当选定损失函数后，需要不断优化模型。即要求 $L^*(w^*, b^*) = \arg \min L(w, b)$ ，采用的是梯度下降的方法。利用链式求导法则：

$$\frac{\partial L}{\partial w} = -\sum_{i=1}^N (y_i - \hat{y}_i) x_i$$

每次使用如下公式来更新 w ，直到损失函数的值收敛为止。

$$w^* = w + \alpha \sum_{i=1}^N (y_i - \hat{y}_i) x_i$$

2.1.2 实验分析

在 Pytorch 中实现逻辑回归模型的关键代码如下，通过 nn.Linear() 实现，其中梯度下降更新参数的过程 pytorch 已自动帮我们实现。

```
class LR(nn.Module):
    def __init__(self, input_dims, output_dims):
        super().__init__()
        self.linear = nn.Linear(input_dims, output_dims, bias=True)

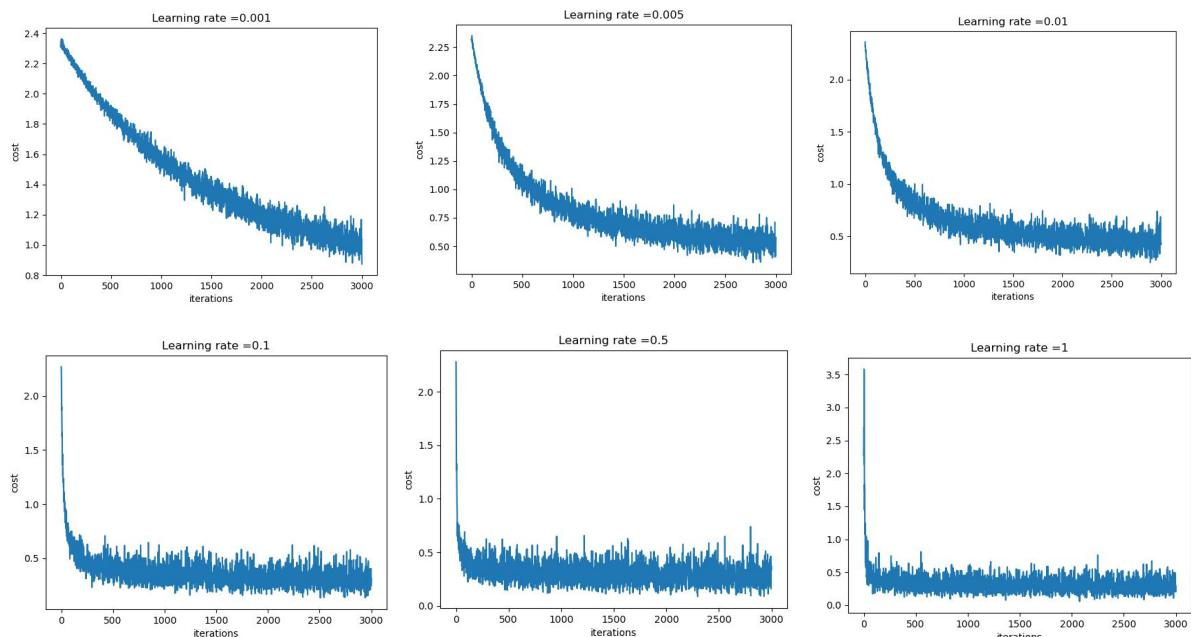
    def forward(self, x):
        x = self.linear(x)
        return x

# 定义逻辑回归模型
class LR(nn.Module):
    def __init__(self, input_dims, output_dims):
        super().__init__()
        self.linear = nn.Linear(input_dims, output_dims, bias=True)

    def forward(self, x):
        x = F.sigmoid(self.linear(x))
        return x
```

图 2.2 pytorch 中 LR 的实现

该部分的实验分为两部分，一部分只使用 nn.Linear() 进行线性回归，另一部分将引入 sigmoid 作为每个神经元的激活函数进行逻辑回归。为了进行对比，这里统一设置 num_epochs = 5，即迭代次数设置为 5。实验结果采用训练中的 loss 变化曲线和对测试集中的数据进行测试后得到的混淆矩阵进行展示。前两行为训练过程中的 loss 变化曲线，从左到右，从上到下的学习率分别是 0.001、0.005、0.01、0.1、0.5、1；后两行为混淆矩阵的可视化显示，从左到右，从上到下的学习率分别是 0.001、0.005、0.01、0.1、0.5、1。



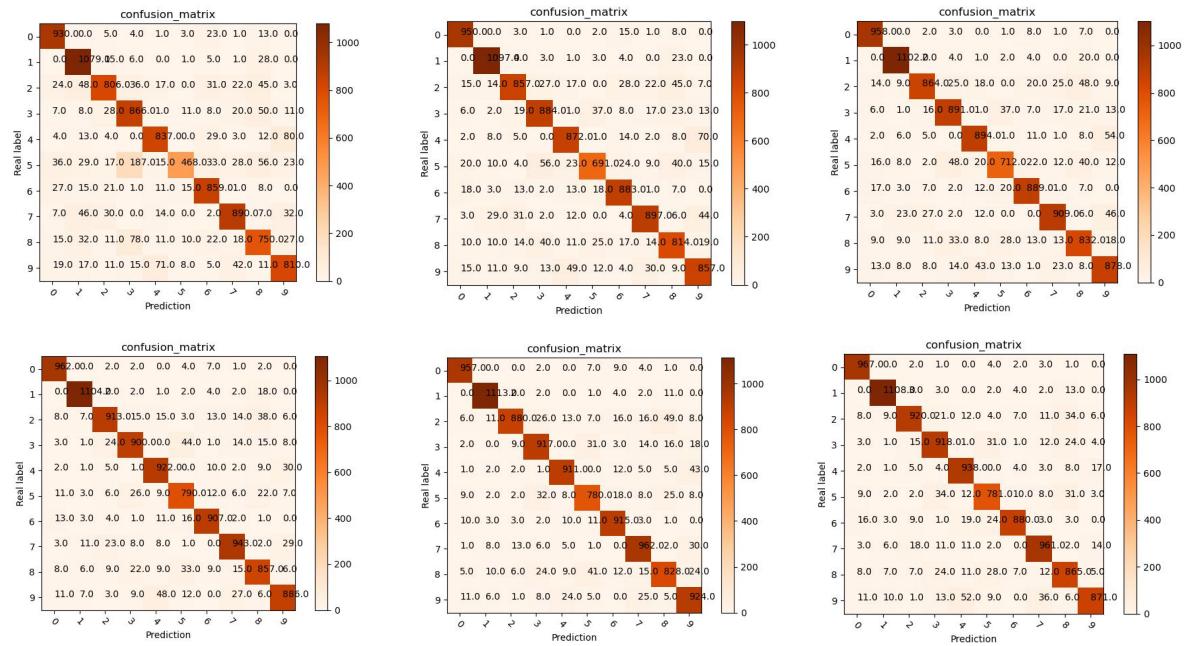
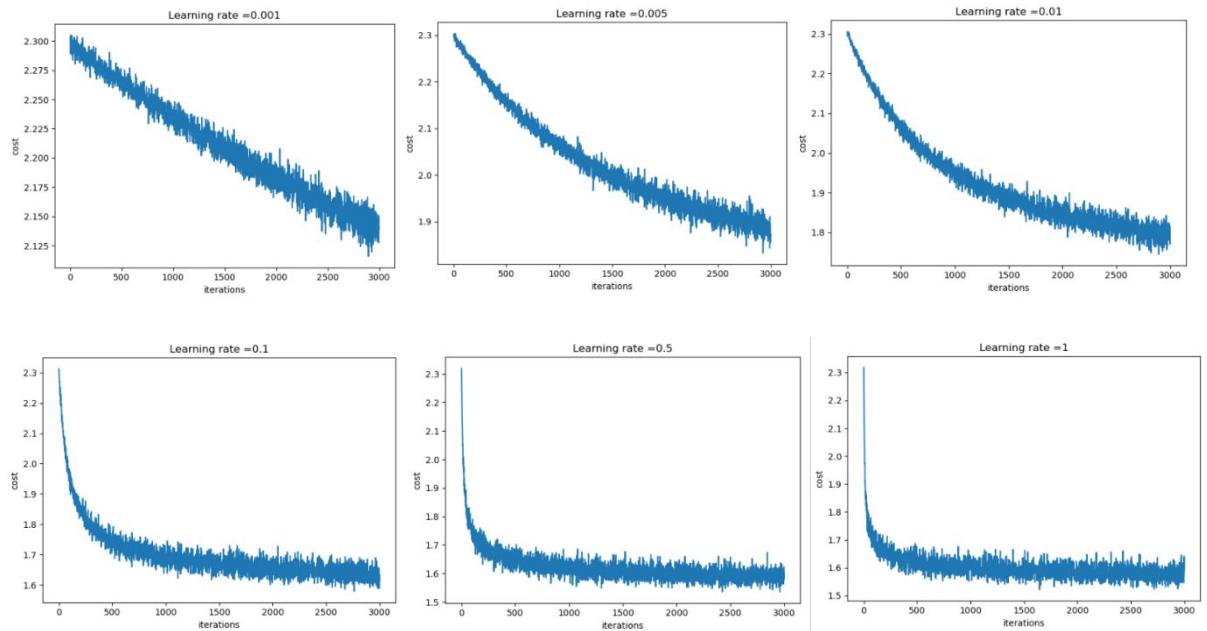


图 2.3 未加 sigmoid 时模型的训练误差曲线和测试的混淆矩阵

由于上述结果的实现并没有增加激活函数，于是接下来加入 sigmoid 函数来进行激活，结果如下所示。前两行为训练过程中的 loss 变化曲线，从左到右，从上到下的学习率分别是 0.001、0.005、0.01、0.1、0.5、1；后两行为混淆矩阵的可视化显示，从左到右，从上到下的学习率分别是 0.001、0.005、0.01、0.1、0.5、1。



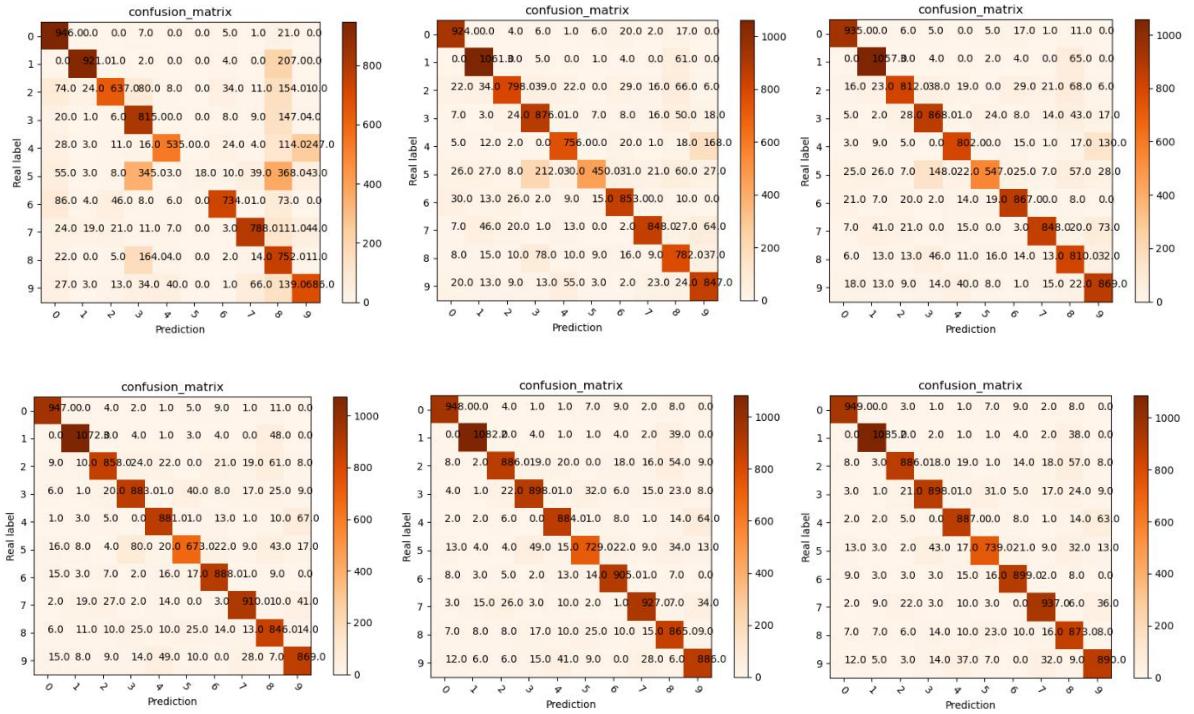


图 2.4 添加 sigmoid 时模型的训练误差曲线和测试的混淆矩阵

Accuracy of the model on the 10000 test images:

learning_rate对LR的影响

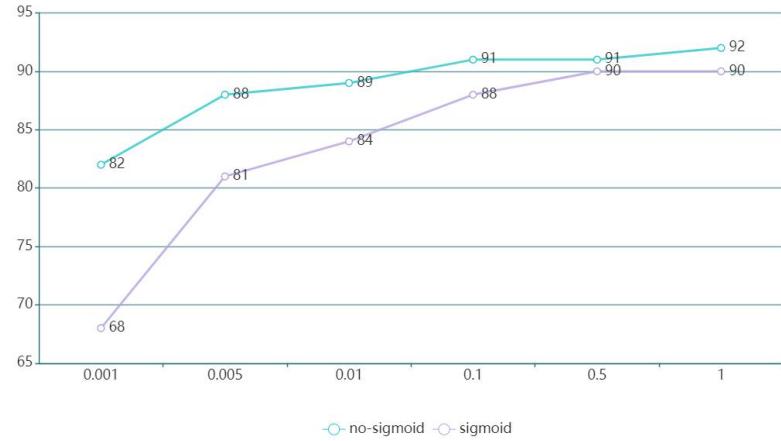


图 2.5 未加 sigmoid 和添加 sigmoid 时模型随学习率的变化曲线

【结果分析】可以看出，在两种情况下，模型的精度均随着学习率的增加而有所提升，但在加入了 sigmoid 激活函数中却没有不加的效果好，这里分析可能是因为没有对数据进行进一步的归一化，导致可能有部分数据在经过激活函数之前，落在 sigmoid 的梯度消失或梯度爆炸的区间，使得模型不能够很好的学习到数据的特征。更多的有关激活函数的分析，将在 2.3 节进行体现。

2.2 多层感知机 Multilayer perceptron

2.2.1 算法解析

这里以 BP 神经网络为例进行解释。首先介绍单层感知机。

1. M-P 神经元模型

在神经网络中最基本的信息处理单元就是 M-P 神经元模型，一个 M-P 神经元模型表示如下图所示：

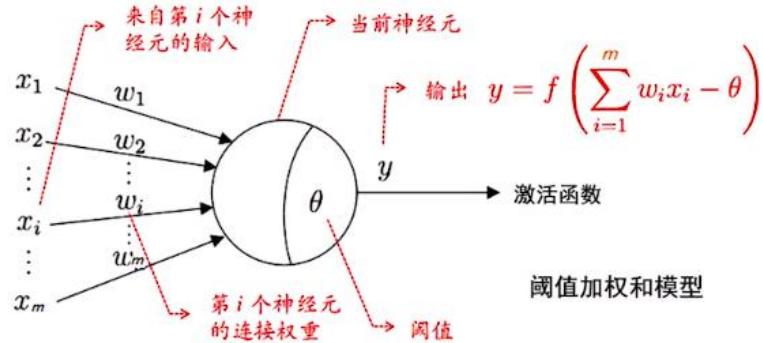


图 2.6 M-P 神经元模型

- [1] 输入：来自其他 m 个神经元传递过来的输入信号。
- [2] 处理：输入信号通过带权重的连接进行传递，到达神经元，神经元对其进行加权和得到总输入值，然后将其与神经元的阈值进行比较，大于阈值则激活神经元。
- [3] 输出：通过激活函数的映射形成输出。

M-P 模型的本质是一个广义的线性回归模型，这里的激活函数相当于广义线性回归模型中的 g 函数（联结函数）。

2. 激活函数

激活函数是对神经元所获得的网络输入进行映射，激活函数通常使用的有线性函数、非线性的斜面函数、阈值函数和逻辑斯蒂函数，我们在这里使用的是 sigmoid 函数，如上图 3 所示。

3. 单层感知机

单层感知机是只有一层 M-P 神经元的神经网络模型，即只包含输入层与输出层，输入层接受外界信号后传递给输出层，输出层是一个或多个 M-P 神经元进行激活处理。

学习过程：

- 1) 初始化，给定训练数据集，权重 $\omega_i (i = 1, 2, \dots, m)$ 和阈值 θ 。
- 2) 权重更新，对于训练样例 (x, y) ，假设当前感知机输出为 \hat{y} ，则感知机的权重调整规则为： $\omega_i \leftarrow \omega_i + \Delta\omega_i$ ，其中 $\Delta\omega_i = \eta(y - \hat{y})$ ， η 为学习率，是一个 0 到 1 之间的数。若感知机对训练样例预测正确，即 $y = \hat{y}$ ，则感知机的权重不发生变化，否则根据错误的程度进行权重调整。

单层感知机学习能力有限，只能解决线性可分问题，对于非线性可分问题就要用到多层感知机。多层感知机由以下几部分组成。

1. 多层前馈神经网络

多层前馈神经网络中多层是指除了输入层与输出层以外，还存在一个或多个隐含层；前馈是指外界信号从输入层，经过隐含层到达输出层，不存在信号的逆向传播；且每层神经元与下层神经元全互连结，神经元之间不存在同层连接，也不存在跨层连接。即输入层接受外界输入，隐含层与输出层神经元对信号进行加工，最终结果由输出层神经元输出。

2. BP 算法

误差逆传播算法（Error Back Propagation，简称 BP）是多层前馈神经网络最具代表性的学习算法，在这个实验中我们采用的是标准 BP 算法，标准 BP 学习算法的过程如下：

1) 初始化，输入层到隐含层的权重矩阵 ω_{ij} ，隐含层到输出层的权重矩阵 ω_{jk} ，隐含层每个 M-P 神经元的阈值 θ_j ，输出层每个 M-P 神经元的阈值 θ_k 。

2) 反复调整：

信号的前向传播，计算每一个神经元的网络净输入和网络输出，其中每个隐含层的输出是输出神经元的输入。

a) 计算每个隐含层节点的输入： $I_j = \sum \omega_{ji}x_i - \theta_j$ ；

b) 计算每个隐含层节点的输出： $O_j = \frac{1}{1+e^{-I_j}}$ ；

c) 计算每个输出层节点的输入： $I_k = \sum \omega_{kj}O_j - \theta_k$ ；

d) 计算每个输出层节点的输出： $O_k = \frac{1}{1+e^{-I_k}}$ ；

误差的逆向传播，计算每个神经元的误差，其中输出层既有实际输出又有理想输出，但隐含层只有实际输出，把输出层的误差逆向传播给隐含层，从而获得隐含层神经误差。

a) 计算每个输出层节点的误差： $Err_k = O_k(1 - O_k)(T_k - O_k)$ ，其中 T_k 是输入样本与之对应的理想输出。

b) 计算每个隐含层节点的误差： $Err_j = O_j(1 - O_j) \sum \omega_{jk} Err_k$ 。

权值和阈值更新，更新所有神经元连接权值和 M-P 神经元的阈值，其中 η 是学习率

a) 更新输入层到隐含层的权值： $\omega_{ij} = \omega_{ij} + \eta x_i Err_j$ ；

b) 更新隐含层到输出层的权值： $\omega_{jk} = \omega_{jk} + \eta O_j Err_k$ ；

c) 更新每个隐含层节点的阈值： $\theta_j = \theta_j - \eta Err_j$ ；

d) 更新每个输出层节点的阈值： $\theta_k = \theta_k - \eta Err_k$ ；

3) 判断终止条件是否满足, 满足则停止, 否则重复执行步骤 2)。

2.2.2 实验分析

为进行统一的对比实验, 这里设置迭代次数 epoch 为 5 轮, 以下为 MLP 模型分别在不同的学习率下的训练 loss 变化曲线和在测试集中的所呈现的混淆矩阵。从左到右, 从上到下, 学习率依次为 Learning rate=[0.001、0.005、0.01、0.1、0.5、1]。

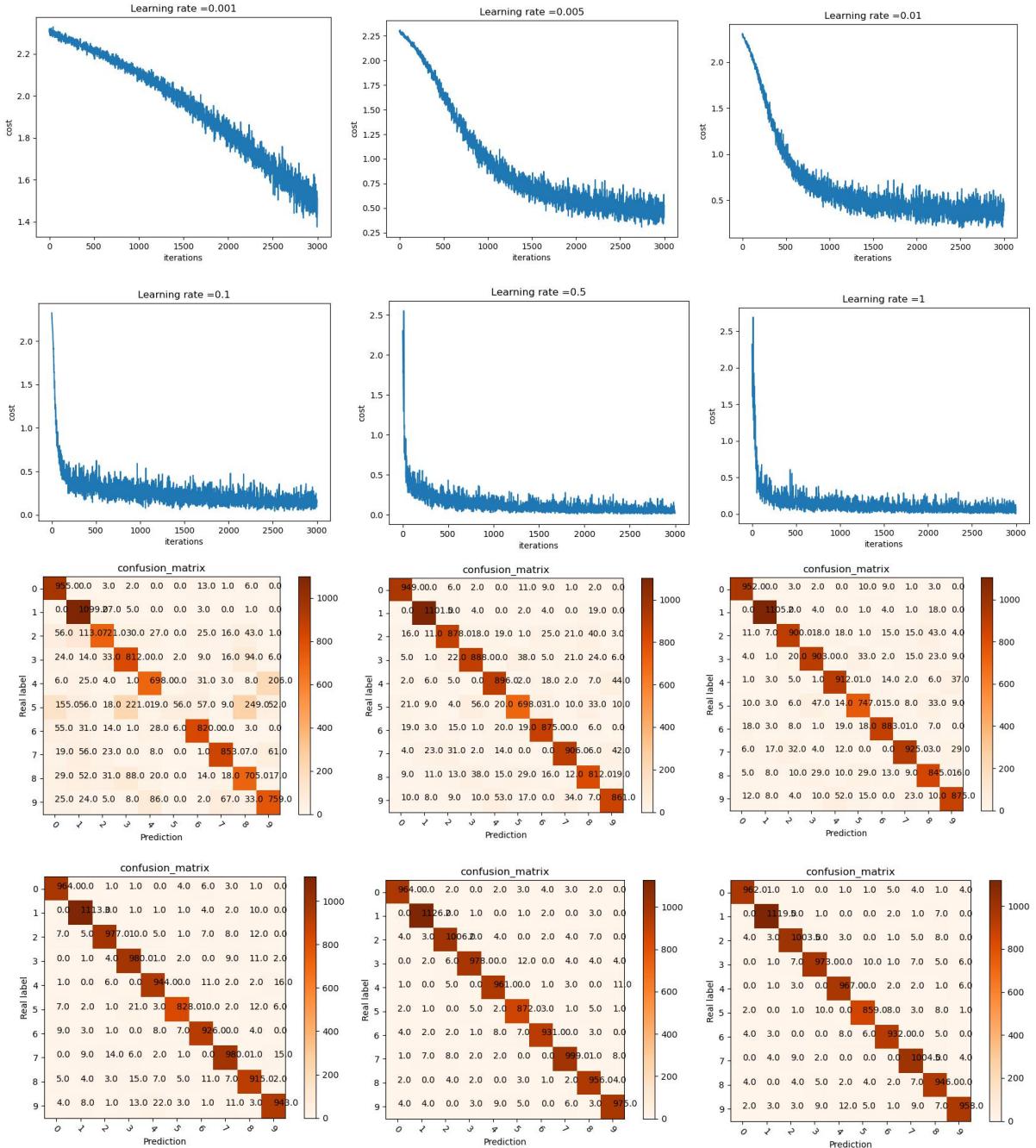


图 2.7 模型的训练误差曲线和测试的混淆矩阵

Accuracy of the model on the 10000 test images:



图 2.8 模型随学习率的变化曲线

【结果分析】从测试集中的混淆矩阵的表现，可以很明显的看出，随着学习率的增加，对角线上的颜色逐渐变的越来越深，被分类错误的数字越来越少，同时从折线图中也可以很明显的看出，模型的精度在逐渐上升。

2.3 深度卷积网络 Deep Convolutional Network

2.3.1 算法解析

这里以经典的 LeNet-5 为例，进行算法的解析。首先 LeNet-5 的主要结构如下图所示。

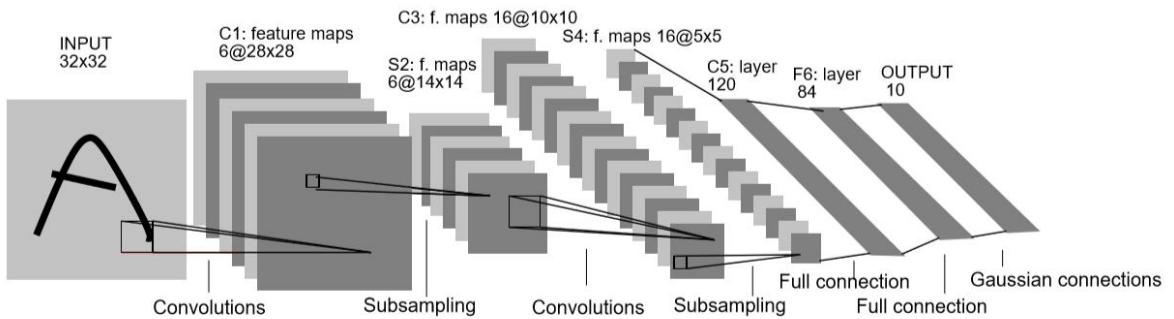


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

<https://blog.csdn.net/jlwz15307>

图 2.9 LeNet-5 网络结构

其中每一层的具体参数实现过程如下图所示。

```

# 第一层：卷积层的输入是 1 channel , 输出是 6 channel , kernel_size=(5, 5)
# 当 kernel_size=(5, 5) 时, 要使padding满足'same', 则 padding = (kernel_size - 1) / 2
self.conv1 = nn.Conv2d(in_dim, 6, (5, 5), 1, padding=2)

# 第二层：卷积层， 输入 6 channel , 输出 16 channel , kernel_size=(5, 5)
# padding=0, 默认, 此时进行 valid 操作
self.conv2 = nn.Conv2d(6, 16, (5, 5))

# 第三层：全连接层 (线性表示)
# 此时的全连接里面有 400(5*5*16)个节点, 其中每个节点中有 120 个神经元
self.fc1 = nn.Linear(5*5*16, 120)

# 第四层：全连接层
self.fc2 = nn.Linear(120, 84)

# 第五层：输出层
self.fc3 = nn.Linear(84, n_class)

```

图 2.10 pytorch 实现 LeNet-5 网络结构时的核心代码

Lenet-5 的网络结构其主线为，输入二维图像，先经过两次卷积层到池化层，再经过全连接层，最后使用 softmax 分类作为输出层。下面我们主要介绍卷积层和池化层。

1、卷积层

卷积层是卷积神经网络的核心基石。在图像识别里我们提到的卷积是二维卷积，即离散二维滤波器（也称作卷积核）与二维图像做卷积操作，简单的讲是二维滤波器滑动到二维图像上所有位置，并在每个位置上与该像素点及其领域像素点做内积。卷积操作被广泛应用与图像处理领域，不同卷积核可以提取不同的特征，例如边沿、线性、角等特征。在深层卷积神经网络中，通过卷积操作可以提取出图像低级到复杂的特征。

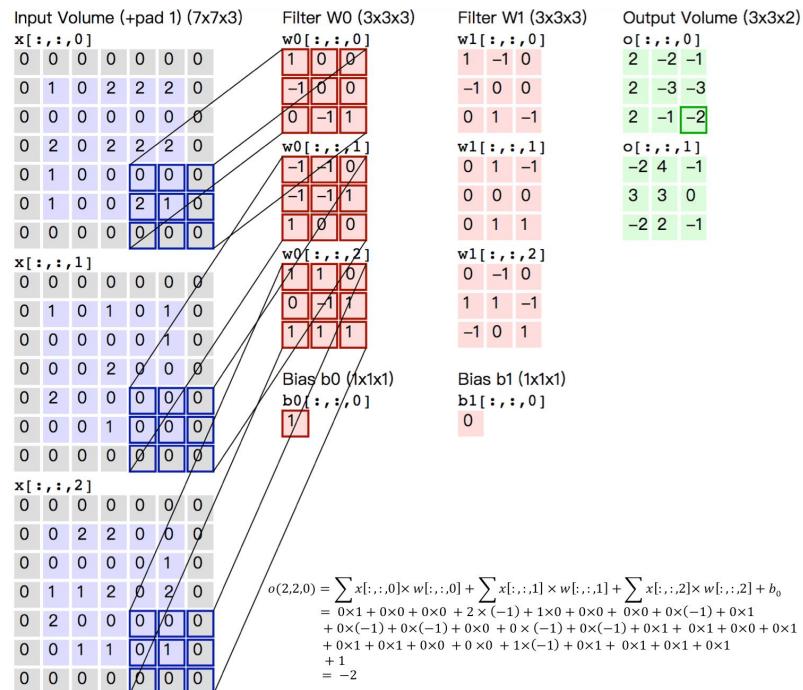


图 2.11 卷积过程示例

上图给出一个卷积计算过程的示例图，输入图像大小为 $H=5, W=5, D=3$ ，即 5×5 大小的 3 通道（RGB，也称作深度）彩色图像。这个示例图中包含两（用 K 表示）组卷积核，即图中滤波器 W_0 和 W_1 。在卷积计算中，通常对不同的输入通道采用不同的卷积核，如图示例中每组卷积核包含（ $D=3$ ）个 3×3 （用 $F \times F$ 表示）大小的卷积核。另外，这个示例中卷积核在图像的水平方向（ W 方向）和垂直方向（ H 方向）的滑动步长为 2（用 S 表示）；对输入图像周围各填充 1（用 P 表示）个 0，即图中输入层原始数据为蓝色部分，灰色部分是进行了大小为 1 的扩展，用 0 来进行扩展。经过卷积操作得到输出为 $3 \times 3 \times 2$ （用 $H_o \times W_o \times K$ 表示）大小的特征图，即 3×3 大小的 2 通道特征图，其中 H_o 计算公式为： $H_o = (H - F + 2 \times P) / S + 1$ ， W_o 同理。而输出特征图中的每个像素，是每组滤波器与输入图像每个特征图的内积再求和，再加上偏置 b_o ，偏置通常对于每个输出特征图是共享的。输出特征图 $o[:, :, 0]$ 中的最后一个 -2 计算如上图右下角公式所示。

在卷积操作中卷积核是可学习的参数，每层卷积的参数大小为 $D \times F \times F \times K$ 。卷积层的参数较少，这也是由卷积层的主要特性即局部连接和共享权重所决定。

通过介绍卷积计算过程，可以看出卷积是线性操作，并具有平移不变性（shift-invariant），平移不变性即在图像每个位置执行相同的操作。卷积层的局部连接和权重共享使得需要学习的参数大大减小，这样也有利于训练较大卷积神经网络。

2、池化层

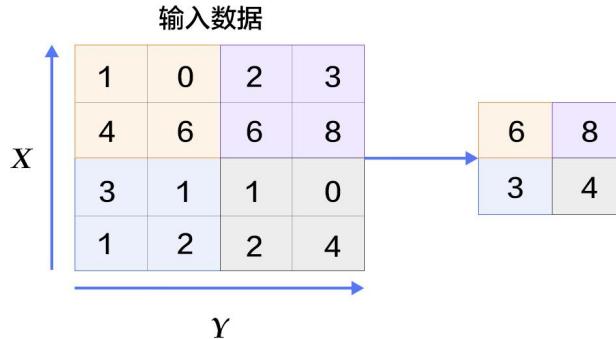


图 2.12 池化过程示例

池化是非线性下采样的一种形式，主要作用是通过减少网络的参数来减小计算量，并且能够在一定程度上控制过拟合。通常在卷积层的后面会加上一个池化层。池化包括最大池化、平均池化等。其中最大池化是用不重叠的矩形框将输入层分成不同的区域，对于每个矩形框的数取最大值作为输出层，如上图所示。

2.3.2 实验分析

为进行统一的对比实验，这里设置迭代次数 epoch 为 5 轮。

以下为 LeNet-5 模型分别在不同的学习率下的训练 loss 变化曲线和在测试集中的所呈现的混淆矩阵。从左到右，从上到下，学习率依次为 Learning rate=[0.0001、0.001、0.01、0.05、0.1]。

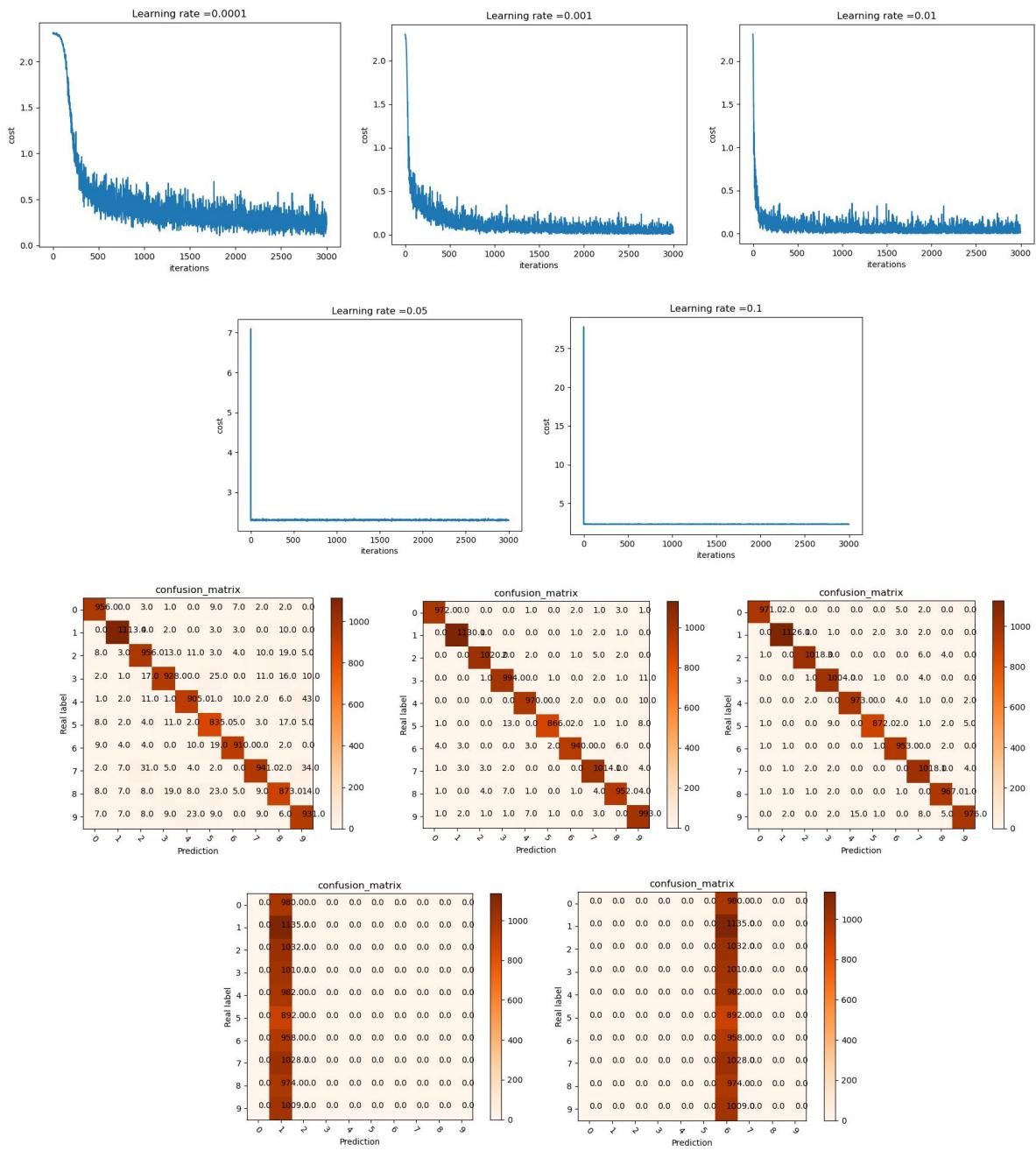


图 2.13 模型的训练误差曲线和测试的混淆矩阵

Accuracy of the model on the 10000 test images:

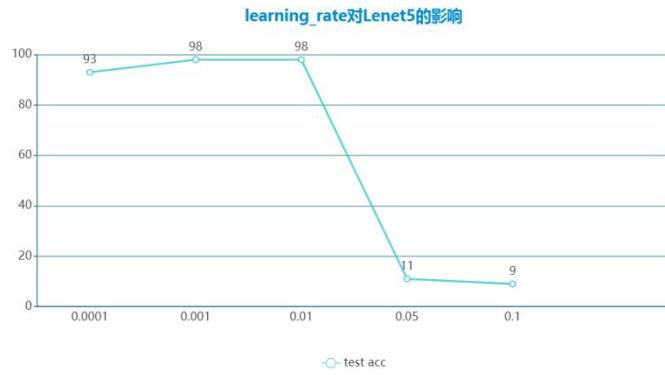


图 2.14 模型随学习率的变化曲线

【结果分析】结合 2.1 和 2.2 模型的结果精度值来看，lenet-5 网络在此数据集中表现出了最好的结果，在相同迭代次数和较低学习率的情况下，分类的精度确比 LR 和 MLP 高，其原因是由于，卷积这个操作能够很好得对图片进行表示学习，但我们从结果中也可以很明显的看到，学习率对 Lenet-5 网络的影响比 LR 和 MLP 更加的敏感，稍微改变都会使得模型从一个高精度变成过拟合的现象。

• 激活函数对模型分类精度的影响

由于卷积神经网络在图片数据集中的较好表现，这里对应用的激活函数也进行了一定的实验对比，其效果如下所示。从左到右分别为使用激活函数 Relu, Tanh 和 sigmoid。

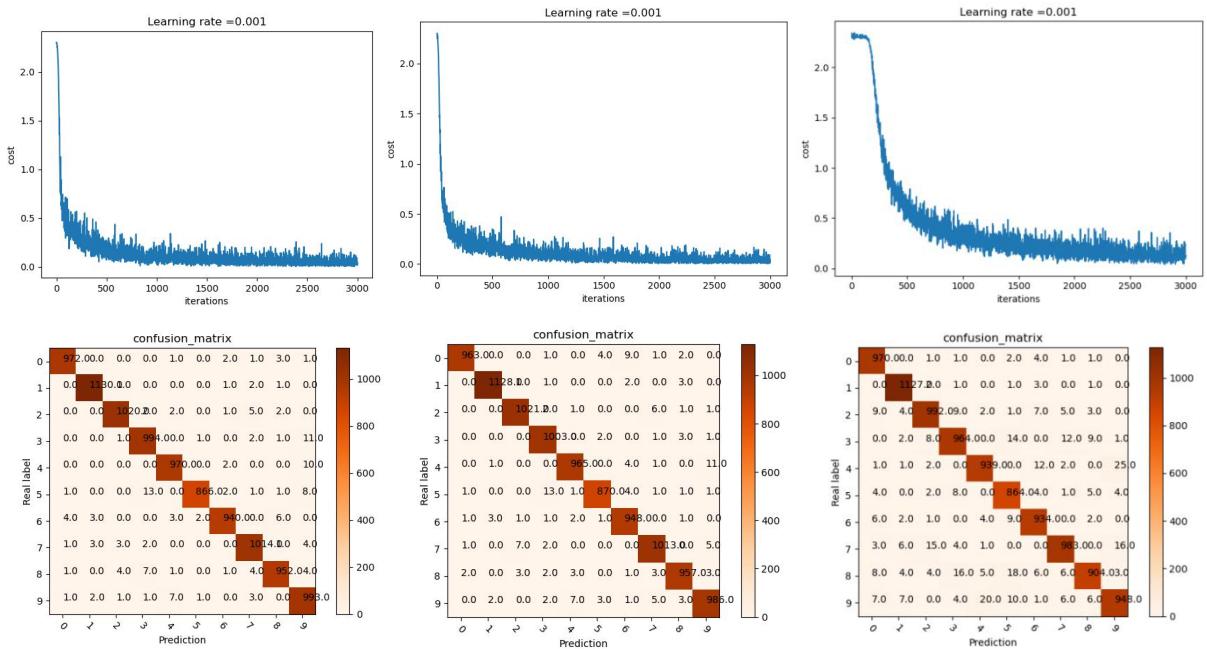


图 2.15 ReLU, Tanh 和 sigmoid 对模型训练误差和测试中混淆矩阵的影响

Accuracy of the model on the 10000 test images:

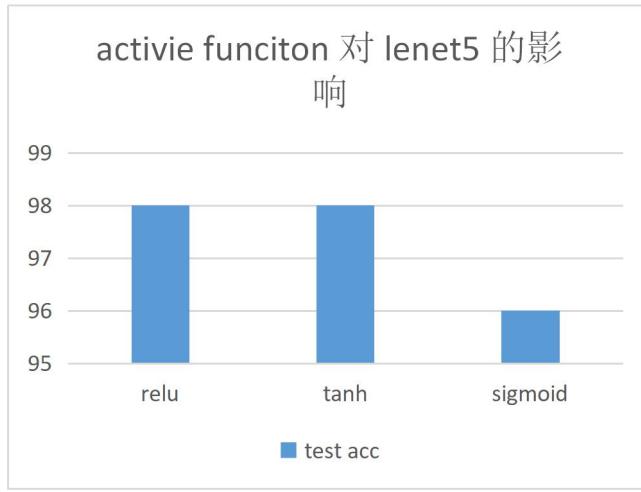


图 2.16 ReLU, Tanh 和 sigmoid 对模型精度的影响

【结果分析】可以看出在使用 relu 和 tanh 作为激活函数时，模型较为稳定且精度值较高。

- 网络结构对模型分类精度的影响

在尝试了对学习率和激活函数这种基本的参数进行选择后，这里为了进一步验证卷积与传统全连接网络对图片特征的学习性能，这里对网络结构也进行了一定的调整进行对比。首先，我们看一下这两个改变的网络结构与原始网络的差别，第一个的网络结构多加了一层全连接层，第二个的网络结构改变第一层卷积和第二层卷积的通道数，网络结构具体如下两图所示。

```

LeNet5(
    (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (fc1): Linear(in_features=400, out_features=120, bias=True)
    (fc2): Linear(in_features=120, out_features=84, bias=True)
    (fc3): Linear(in_features=84, out_features=32, bias=True)
    (fc4): Linear(in_features=32, out_features=10, bias=True)
)

```



```

LeNet5(
    (conv1): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1))
    (fc1): Linear(in_features=800, out_features=120, bias=True)
    (fc2): Linear(in_features=120, out_features=84, bias=True)
    (fc3): Linear(in_features=84, out_features=10, bias=True)
)

```

图 2.17 两种网络结构

两个网络的训练误差曲线和测试集中的分类的混淆矩阵如下所示，第一行为第一个网络结构的结果，第二行为第二个网络结构的结果。

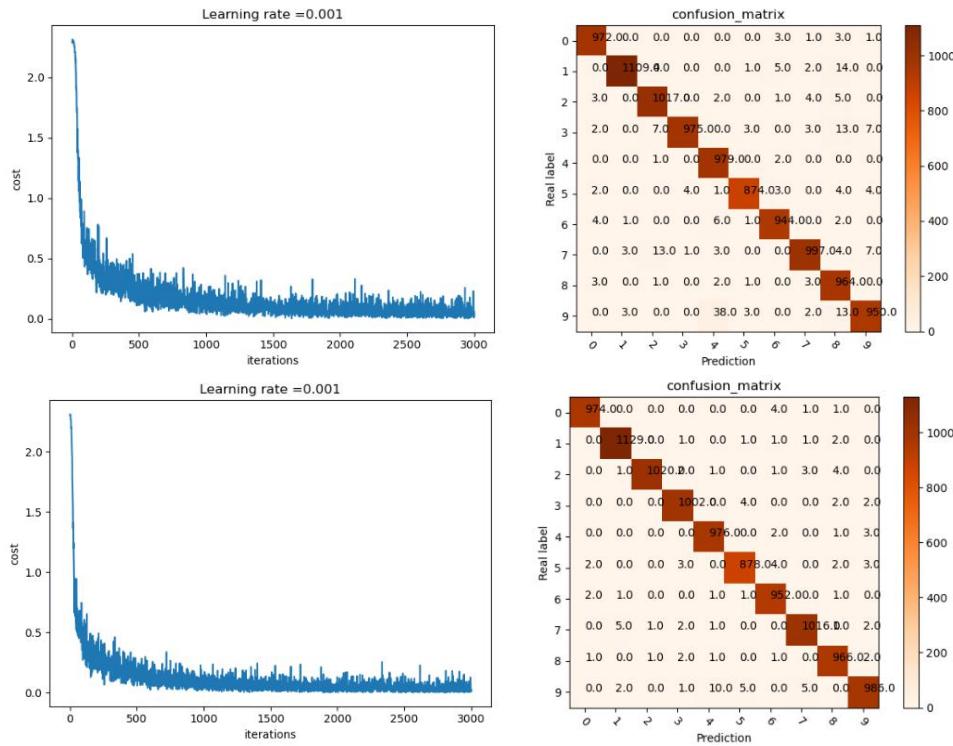


图 2.18 模型在两种网络结构下的训练误差曲线和测试的混淆矩阵

为了更加清楚的对比更改的网络结构与原始网络结构的结果对比，这里对精度进行了细化，同时通过对比表和折线图进行展示。

表 2.1 不同网络结构下的精度值

	原网络结构	增加全连接层	增加卷积通道数
Test acc	98.3%	97%	98.7%

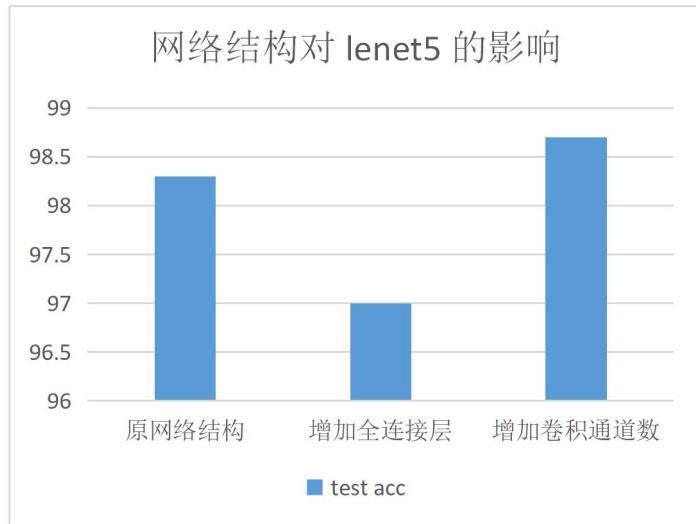


图 2.19 不同网络结构下的精度值柱状图

【结果分析】由此结果和上述结构的对比中均可以看出，在 lenet-5 网络中对模型有最大影响的应该为卷积操作。

2.4 长短期记忆网络 Long Short Term Memory networks

2.4.1 算法解析

LSTM 是循环神经网络 RNN 的其中一个变种，RNN 的关键点之一就是它可以用来连接先前的信息到当前的任务上，例如使用过去的视频段来推测对当前段的理解。但是当相关信息和当前预测位置之间的间隔变得非常大，RNN 会丧失学习到连接如此远的信息的能力。我们仅仅需要明白的是利用 BPTT 算法训练出来的普通循环神经网络很难学习长期依赖（例如，距离很远的两步之间的依赖），原因就在于梯度消失/发散问题。但是 RNN 绝对可以处理这样的长期依赖问题，人们可以仔细挑选参数来解决这类问题中的最初级形式，但在实践中，RNN 肯定不能够成功学习到这些知识。训练和参数设计十分复杂。LSTM 就是专门设计出来解决这个问题的。

我们都知道，在所有的 RNN 都具有一种重复神经网络模块的链式形式。在标准 RNN 中，这个重复的结构模块只有一个非常简单的结构，例如一个 tanh 层。

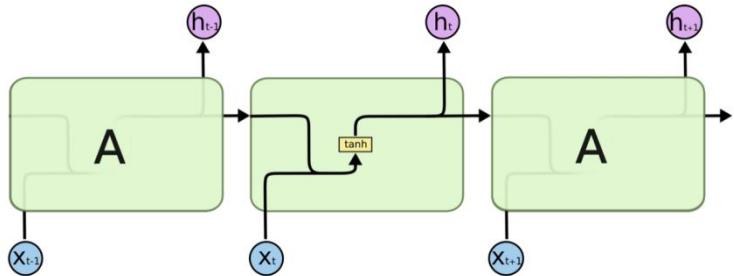


图 2.20 RNN 网络结构

LSTM 同样是这样的结构，但是重复的模块拥有一个不同的结构。不同于单一神经网络层，这里有四个，以一种非常特殊的方式进行交互。

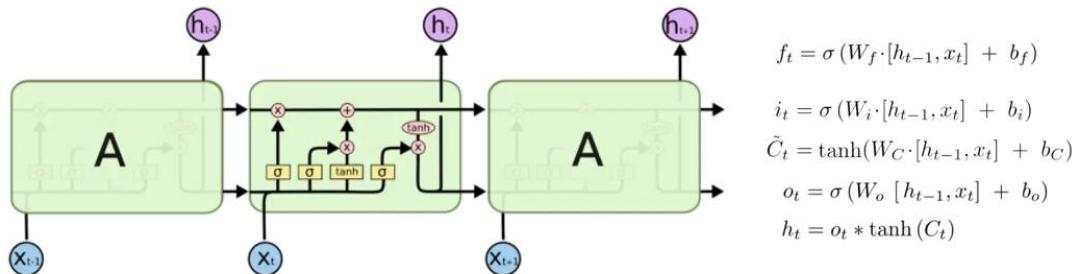


图 2.21 LSTM 网络结构

• 遗忘门

LSTM 中的第一步是决定从细胞状态中丢弃什么信息。这个决定通过一个称为忘记门层完成。该门会读取 h_{t-1} 和 x_t ，输出一个在 0 到 1 之间的数值给每个在细胞状态 C_{t-1} 中的数字。其中，1 表示“完全保留”，0 表示“完全舍弃”， h_{t-1} 表示上一个 cell 的输出， x_t 表示当前 cell 的输入。

• 输入门

决定让多少新的信息加入到 cell 状态 中来。实现这个需要包括两个 步骤：首先，一个叫做“input gate layer”的 sigmoid 层决定哪些信息需要更新；一个 tanh 层生成一个向量，也就是备选的用来更新的内容 \tilde{C}_t ，最后将这两部分联合起来，对 cell 的状态进行一个更新。

• 输出门

确定输出什么值。首先，我们运行一个 sigmoid 层来确定细胞状态的哪个部分将输出出去。接着，我们把细胞状态通过 tanh 进行处理（得到一个在 -1 到 1 之间的值）并将它和 sigmoid 门的输出相乘，最终我们仅仅会输出我们确定输出的那部分。

2.4.2 实验分析

同样的，为进行统一的对比实验，这里设置迭代次数 epoch 为 5 轮。以下为 LSTM 网络的代码实现。

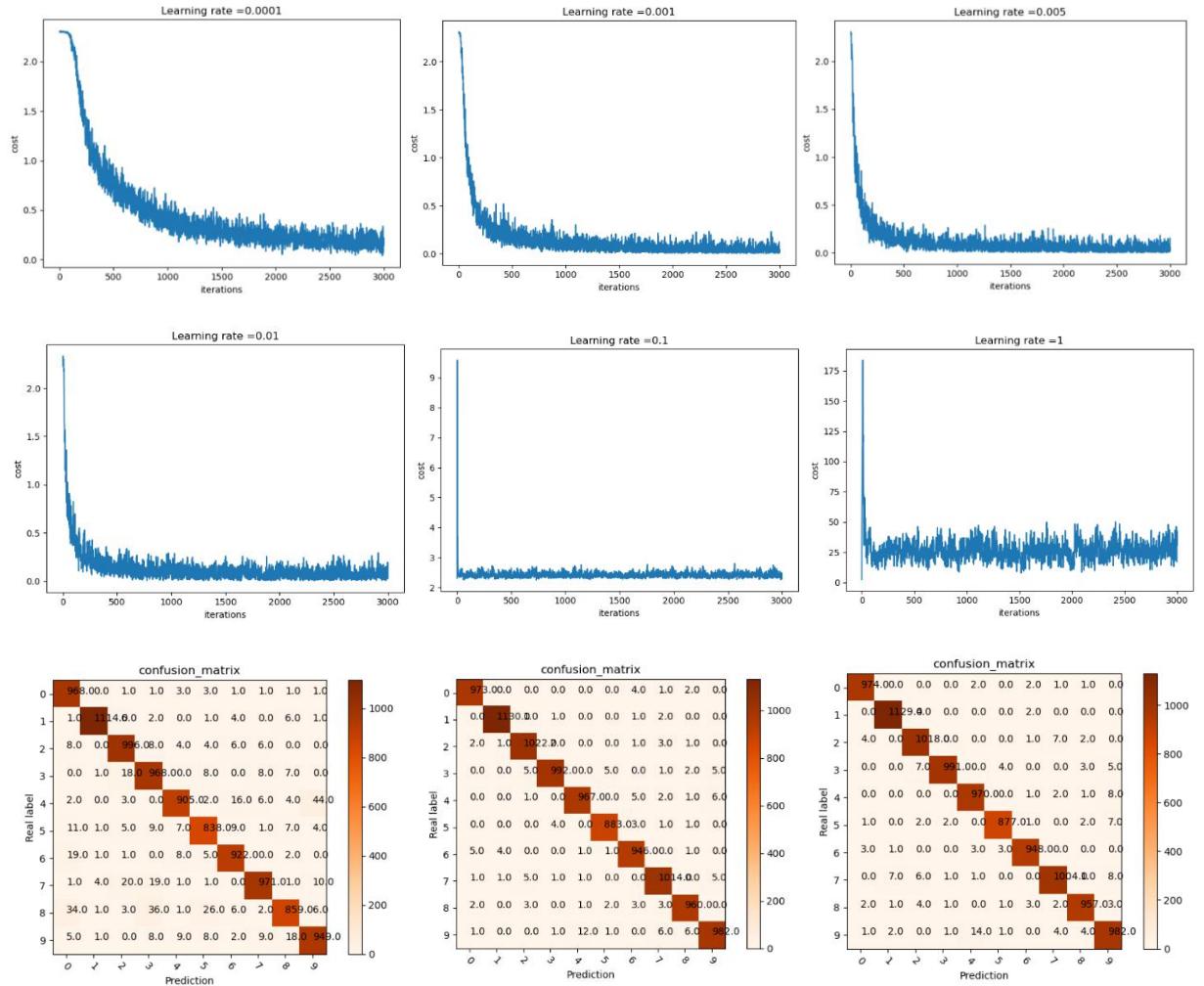
```

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

```

图 2.22 LSTM 在 Pytorch 中的实现

由于 2.3 节进行了相关的其他参数对比实验，这里仅对学习率对模型的影响进行了可视化对比展示，其他的相关对比，这里就不进行过多的赘述了。以下为 LSTM 模型分别在不同的学习率下的训练 loss 变化曲线和在测试集中的所呈现的混淆矩阵。从左到右，从上到下，学习率依次为 Learning rate=[0.0001、0.001、0.005、0.01、0.1、1]。



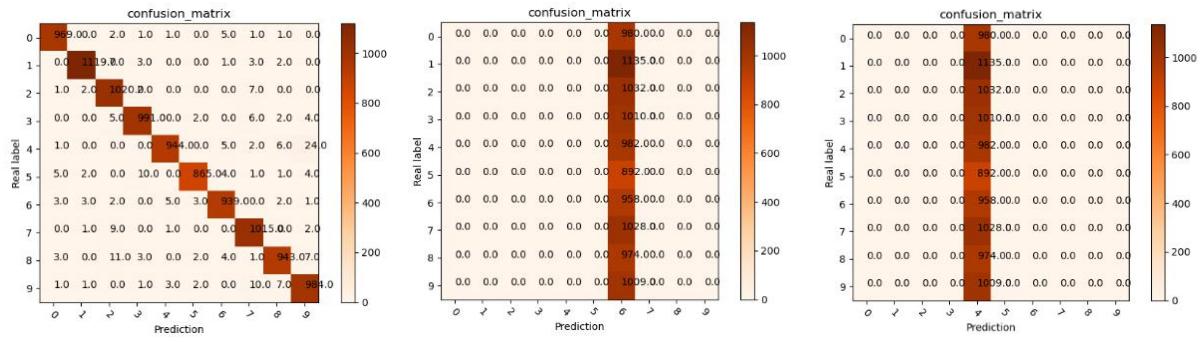


图 2.23 模型的训练误差曲线和测试的混淆矩阵

Accuracy of the model on the 10000 test images:



图 2.24 模型随学习率的变化曲线

【结果分析】可以看出，LSTM 对于学习率也是十分的敏感，学习率对模型精度变化情况与 CNN 类似。在 0.001 时达到最佳分类精度，随后降低，产生过拟合现象。

三 无监督学习 Unsupervised learning

3.1 自编码 Auto Encoders

3.1.1 算法解析

Auto Encoder（自动编码器）的最直观的作用就是利用网络中的隐藏层对原始输入进行降维，然后对输入进行重建，以如下网络示意图为例。编码器会创建一个隐藏层（或多个隐藏层），这个隐藏层包含了输入数据含义的低维向量。之后解码器，会通过隐藏层的低维向量来重建输入数据。它可以帮助数据分类、可视化、存储。AE 是一个自动编码器是一个非监督的学习模式，只需要输入数据，不需要 label 或者输入输出对的数据。

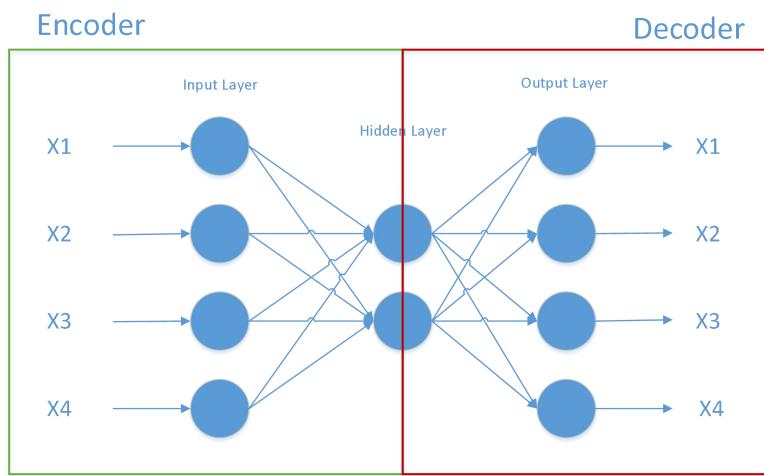


图 3.1 Auto Encoder 网络结构图

虽然 Auto Encoder 是一个非监督学习算法，但是如果它的解码器是线性重建数据，可以使用 MSE 来表示它的损失函数，公式如下：

$$L(x, y) = \sum (x - h_{W, b}(x))^2 \quad y = h_{W, b}(x)$$

在本实验中，由于是要对图片进行自编码的操作，所以在这里讲上图中的全连接层用卷积和池化代替。具体的层间参数如下所示。

```
self.encoder = nn.Sequential(
    nn.Conv2d(1, 16, 3, stride=3, padding=1), # b, 16, 10, 10
    nn.ReLU(True),
    nn.MaxPool2d(2, stride=2), # b, 16, 5, 5
    nn.Conv2d(16, 8, 3, stride=2, padding=1), # b, 8, 3, 3
    nn.ReLU(True),
    nn.MaxPool2d(2, stride=1) # b, 8, 2, 2
)
self.decoder = nn.Sequential(
    nn.ConvTranspose2d(8, 16, 3, stride=2), # b, 16, 5, 5
    nn.ReLU(True),
    nn.ConvTranspose2d(16, 8, 5, stride=3, padding=1), # b, 8, 15, 15
    nn.ReLU(True),
    nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1), # b, 1, 28, 28
    nn.Tanh()
)
```

图 3.2 pytorch 实现 Auto Encoder 的关键网络结构

3.1.2 实验分析

将训练过程中得到的重建图片保存下来，输出的图像如下，从左到右，从上往下，依次为 epoch 递增（0、10、20、30、40、50、60、80、90）的情况。

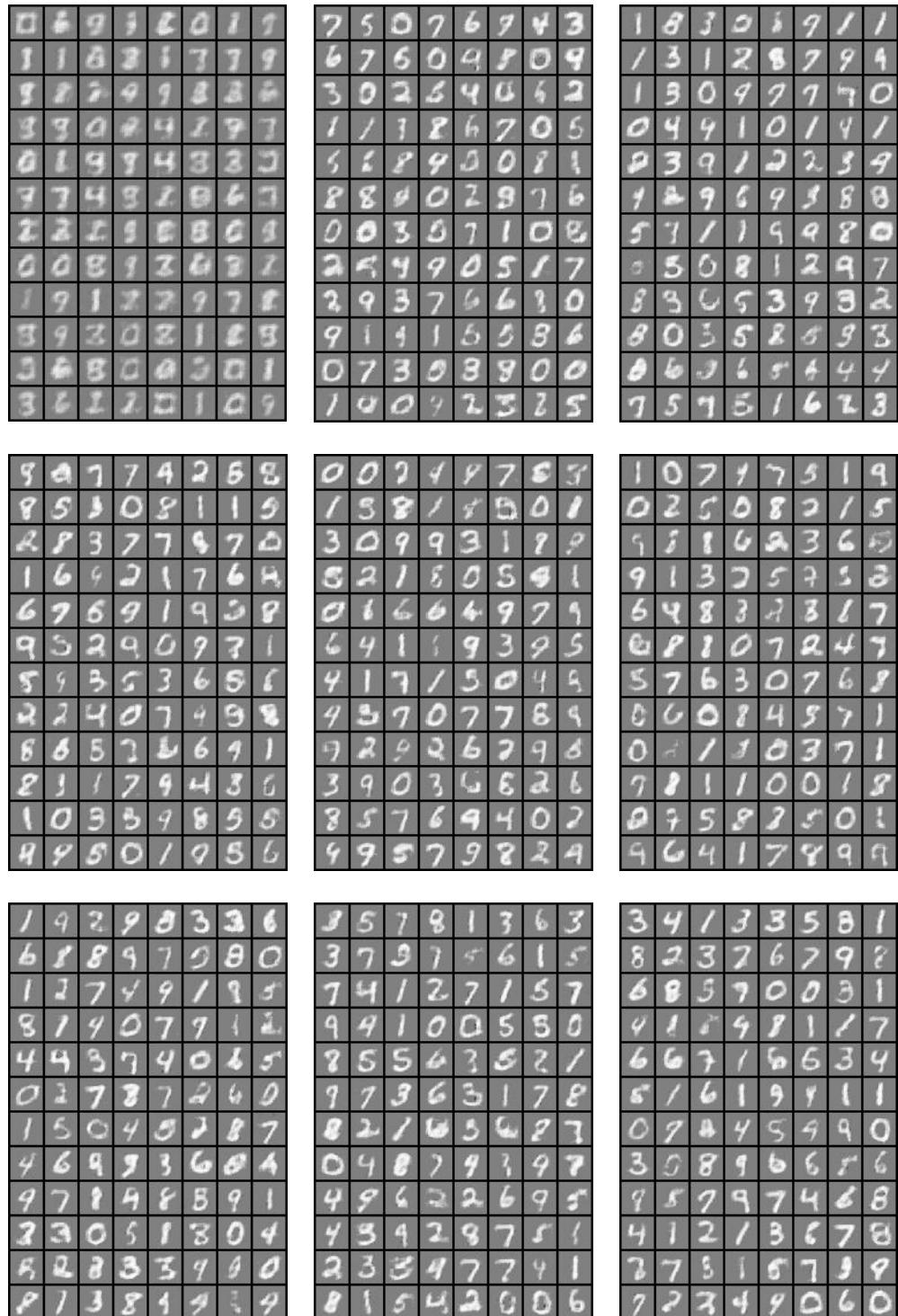


图 3.3 随 epoch 递增的重建图

【结果分析】可以很明显的看出，随着 epoch 的增加，经过 decoder 生成的图像越来越接近真实图片。根据网络结构可分析出，当收敛后，便可使用隐藏层特征作为输入的特征表示。

其中模型在训练过程中的 loss 曲线图，如下所示：

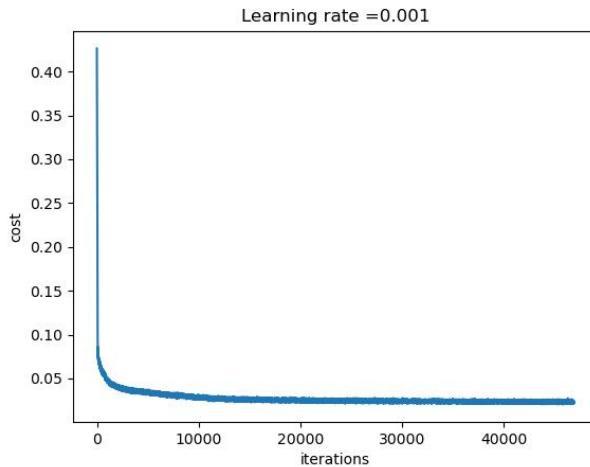


图 3.4 模型在训练过程中的 loss 曲线图

3.2 降噪自编码 Denoising Autoencoders

3.2.1 算法解析

为了提高模型的泛化能力，Denoising Autoencoder（降噪自动编码器）就是在 Autoencoder 的基础之上，对输入的数据（网络的输入层）加入噪音，使学习得到的编码器具有较强的鲁棒性，从而防止模型的过拟合。DAE 的一种方式是随机的删除数据集中的某些数据，然后用完整的数据去评判，在这个过程中 DAE 会尝试去预测恢复缺失的部分。Denoising Auto-encoder 是 Bengio 在 08 年提出的，具体内容可参考其论文：《Extracting and composing robust features with denoising autoencoders》。其中 Denoising Auto-encoder 的模型框架如下所示。

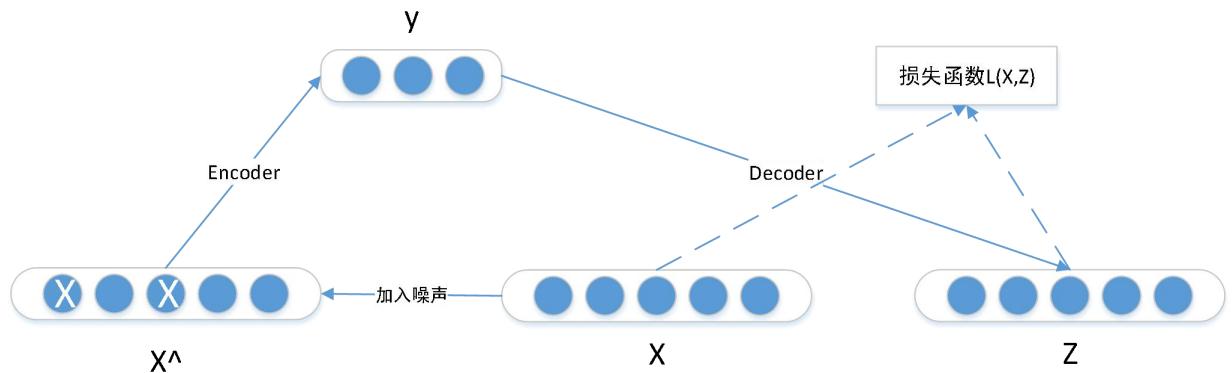


图 3.5 Denoising Autoencoder 网络结构图

从框架图中不难发现，DAE 与 AE 最大的不同就是在模型的输入层上，DAE 通过对

输入进行 Corrupted 处理后，再将处理后的输入作为模型的最终输入，来让网络学习 Corrupted 部分之外，可以减少模型训练时的复杂度，有些类似神经网络中的 Dropout 操作。

所以 DAE 的重点就体现在了如何对输入进行 Corrupted 处理，即原文中的 $\tilde{x} \sim q_D(\tilde{x} | x)$ 。本项目采用和原文中一致的做法，对输入进行随机映射覆盖，即随机选取像素值进行 0 的赋值，具体代码如下图所示。

```
def masking_noise(data, frac):
    """
    data: Tensor
    frac: fraction of unit to be masked out
    """
    data_noise = data.clone()
    rand = torch.rand(data.size())
    data_noise[rand < frac] = 0
    return data_noise
```

图 3.6 Corrupted 处理代码实现

3.2.2 实验分析

这里为了与 AE 进行结果的对比，选择了同样大小的迭代次数 100，学习率 1e-3 以及噪声比（corrupt=frac），下图为随 epoch 的增加而输出的重建图，从左到右，从上到下，epoch 依次为 1、11、21、31、41、51、61、71、81、91。

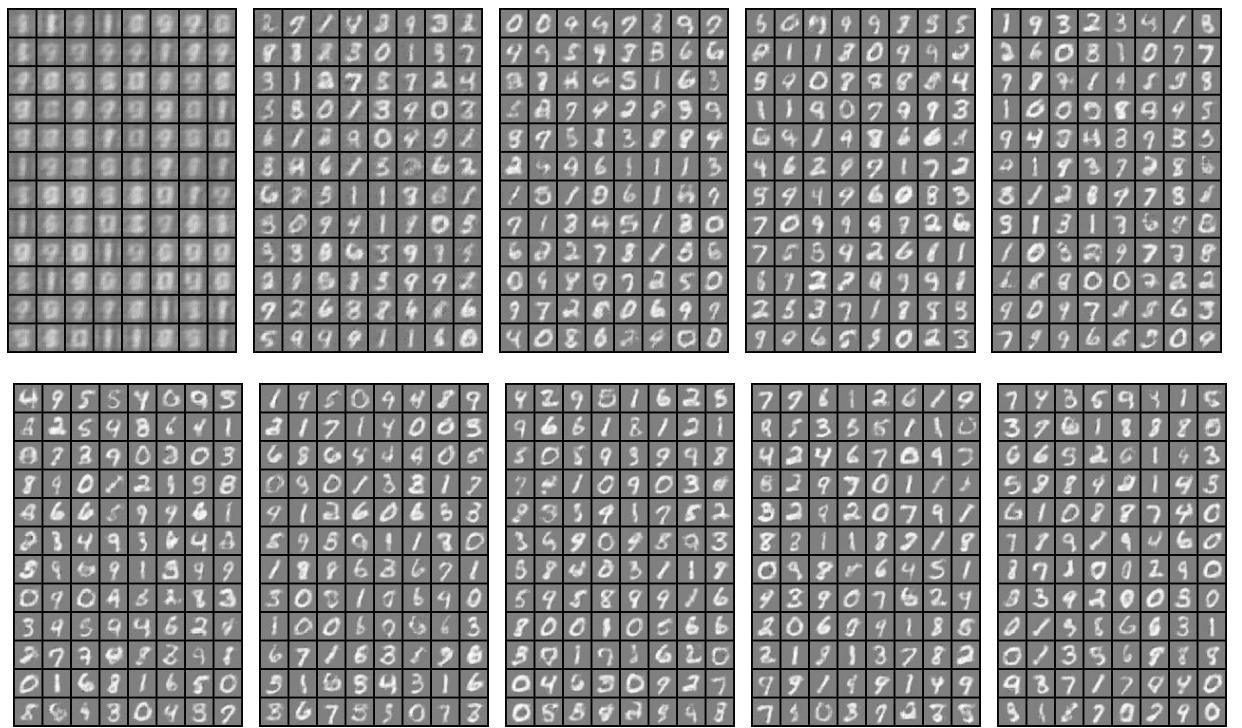


图 3.6 随 epoch 递增的重建图

【结果分析】结果和 AE 类似，随着 epoch 的增加，经过 decoder 生成的图像越来越接近真实图片。同样的，当收敛后，便可使用隐藏层特征作为输入的特征表示。

其中模型在训练过程中的 loss 曲线图，如下所示：

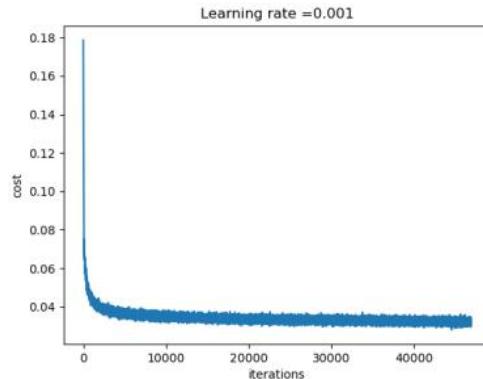


图 3.7 模型在训练过程中的 loss 曲线图

单看以上 10 个图我们只能看出随着 epoch 的增加，decoder 能够更好对输入的图片进行重建。为了对比 DAE 和 AE，这里将 AE(每个子图中的左图) 重建出的 1 和 91 和 DAE(每个子图中的右图) 重建出的 1 和 91 进行一个粗略的比较。

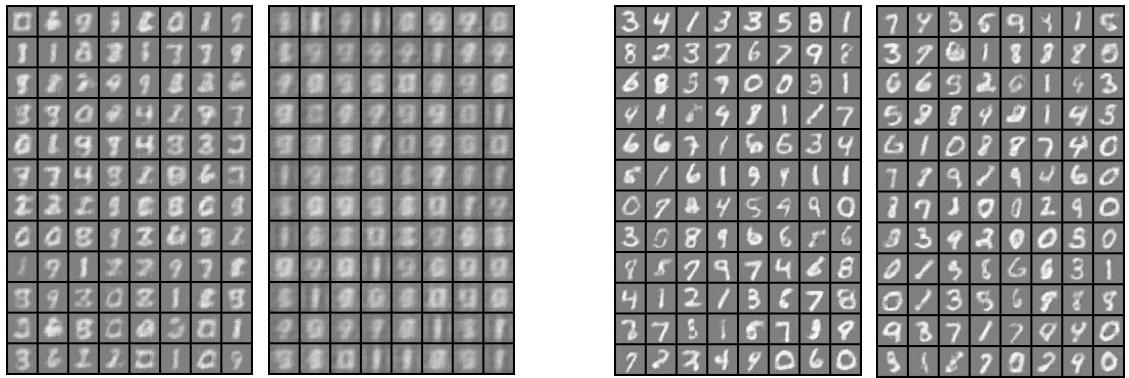


图 3.8 epoch=1 (左) epoch=91 (右)

【结果分析】由于在 DAE 中加入了 corrupt，所以在之经过第一次迭代后，结果较 AE 模糊，但随着 epoch 的增加，DAE 模型的性能也逐渐增加，在 91 次时，两个模型结果相差不大。

3.3 堆叠降噪自编码 Stacked Denoising Auto-Encoders

3.3.1 算法解析

上述两种自编码器从算法解析中，可以看出，都为单个自编码器，即模型的输入只被编码过一次，通过构造一个 3 层的神经网络，来拟合数据特征。如果进一步简化 Auto Encoder 模型，则可由下图表示。



图 3.9 简易 AE 网络结构图

当模型收敛后，我们可以近似的认为 h 为输入 X 的特征表示，SAE 便是利用了这思想，把 3 层的自编码器扩展成了一个深层的结构，也就是将降噪自编码器进行逐层堆叠。但在训练时并不是一蹴而就，而是不断通过上一层的输入预训练出当前输入的 h ，即对每一次的输入都做一次的自编码过程，取收敛后的 h 作为下一层的输入，逐层训练。具体内容可参见 2007 年 Bengio 等人在 NIPS 上发表的《Greedy Layer-Wise Training of Deep Networks》。一旦 SAE 训练完成，其高层的特征就可以用做传统的监督算法的输入。当然，也可以在最顶层添加一层 logistic regression layer (softmax 层)，然后使用带 label 的数据来进一步对网络进行微调 (fine-tuning)，即用样本进行有监督训练。

本项目中使用的是 SDAE，其原理与 SAE 类似，只是在预训练时选择的是 DAE (即 3.2 介绍的降噪自动编码器)。下面将对其原因进行更详细的介绍，下图所示，先对每个单隐层的降噪自动编码器单元进行无监督预训练，然后再进行堆叠，最后进行整体的反向调优训练，就得到了一个两层隐藏层结构的堆叠式降噪自动编码器。

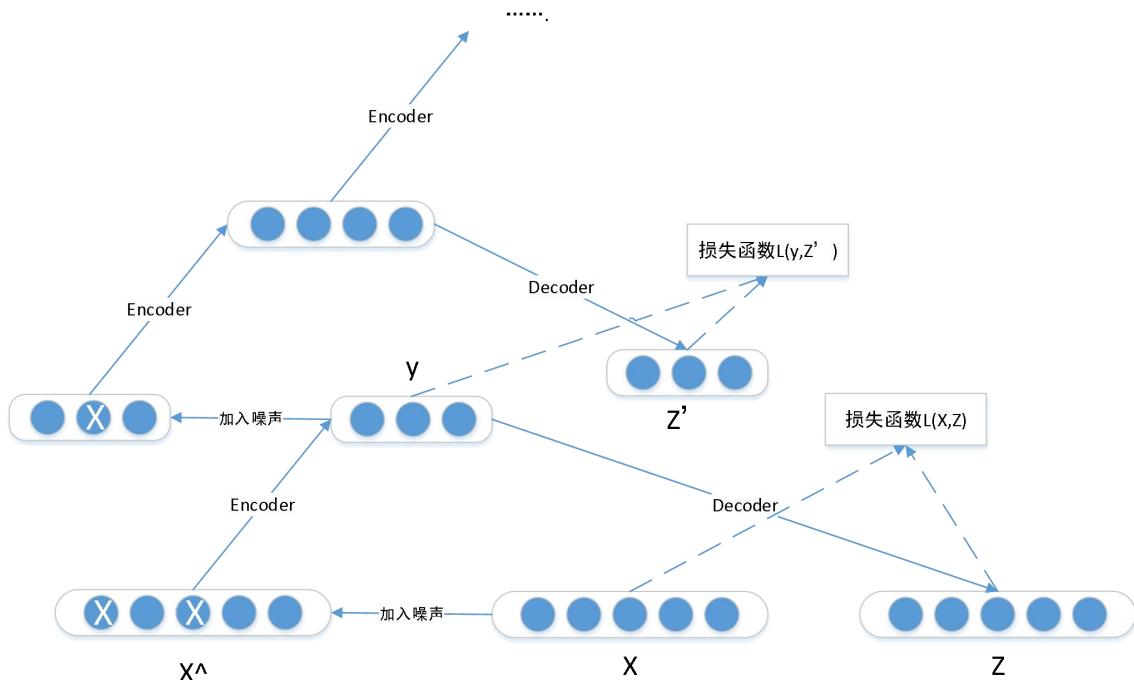


图 3.10 SDAE 网络结构图

3.3.2 实验分析

在本项目中，使用的 SDAE 的网络层结构如下如所示，使用了 3 个 DAE 进行堆叠。

```
class StackedDAutoEncoder(nn.Module):

    def __init__(self):
        super(StackedDAutoEncoder, self).__init__()

        self.ae1 = DAE(1, 16, 2)
        self.ae2 = DAE(16, 32, 2)
        self.ae3 = DAE(32, 64, 2, 0)
```

图 3.11 pytorch 中 SDAE 的网络结构实现

由于 SDAE 能够比 AE 和 DAE 有较好的性能，所以在进行 SDAE 实验时，为了进一步探究模型的性能，这里不仅对 MNIST 数据集中进行了重构的实验，同时对 CIFAR10 这个彩色数据集也进行了重构的实验，其中迭代的次数均为 1000 次，实验结果如下所示。

- MNIST

其中每一对中，左边为原始图像，右边为重构后的图像，epoch 从左到右，从上到下，依次为 epoch=0、10、50、100、500、990

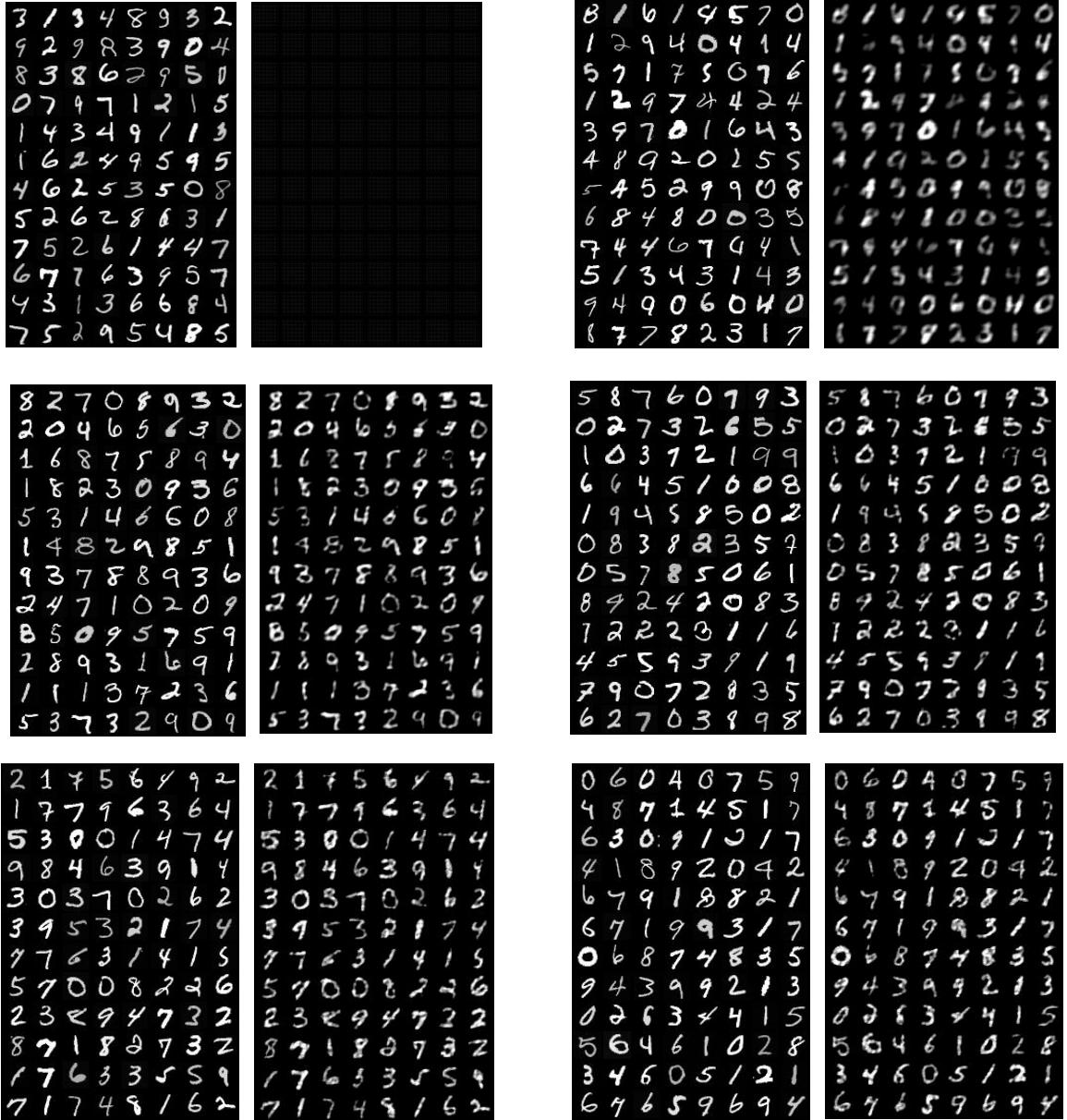


图 3.12 随 epoch 递增的重建图

- CIFAR10

其中每一对中，左边为原始图像，右边为重构后的图像，epoch 从左到右，从上到下，依次为 epoch=0、10、50、100、500、990

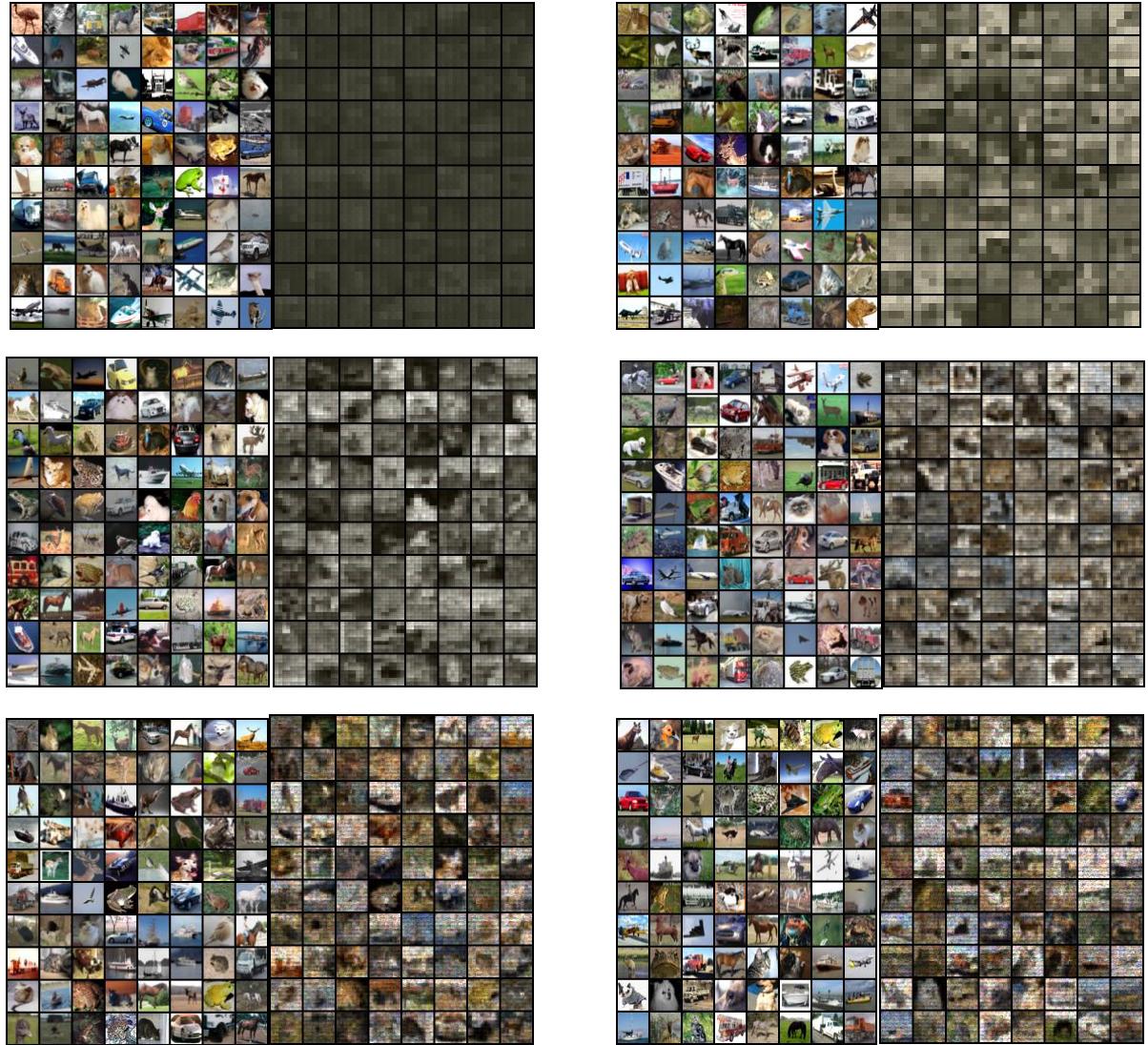


图 3.13 随 epoch 递增的重建图

【结果分析】从以上两个可视化结果中，可以很明显的看出，随着 epoch 的增加，模型的重建能力越来越强，在 MNIST 这种单通道的图像重构中，能够较好的重建出一些边缘信息，但是在 CIFAR10 数据集中，由于数据集中图片本身的复杂性，所以 SDAE 在 1000 次迭代后，也只是对原始图片进行了较为模糊的重建，但从 CIFAR10 数据集的学习中可以更清晰的看出 SDAE 的学习过程，相信经过更多次的迭代后，在 CIFAR10 数据集中也可以有较好的重建结果。

3.4 受限波尔兹曼 Restricted Boltzmann Machines

3.4.1 算法解析

波尔兹曼机是一种经典的无向图模型，也属于深度生成模型，波尔兹曼机最初作为一种广义的‘联结主义’引入，用来学习二值向量上的任意概率分布。在进入算法解析之前，我们先定义几个符号和公式。

一个无向模型是一个定义在无向模型 G 上的结构化概率模型。对于图中的每一个团（图中结点的一个子集，且其中的点是全连接的） C ，一个因子 $\phi(C)$ 衡量了团中变量每一种可能的联合状态所对应的密切程度。他们一起定义了未归一化概率函数：

$$\tilde{p}(\mathbf{x}) = \prod_{C \in G} \phi(C) \quad (3.1)$$

在无向模型中基本的理论结果都依赖于 $\forall \mathbf{x}, \tilde{p}(\mathbf{x}) > 0$ 这个假设，而使这个条件满足的一个简单的方式便是使用基于能量的模型（EBM），其中：

$$\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x})) \quad (3.2)$$

$E(\mathbf{x})$ 被称作是能量函数。又因为由 $\exp(x)$ 函数的性质可知，对所有的 x 而言， $\exp(x)$ 均为正，这保证了没有一个能量函数会使得某一个状态 \mathbf{x} 的概率为 0。服从式 (3.2) 形式的任意分布都是玻尔兹曼分布的一个实例，所以许多基于能量的模型被称为玻尔兹曼机，受限玻尔兹曼机便是其中的一种。

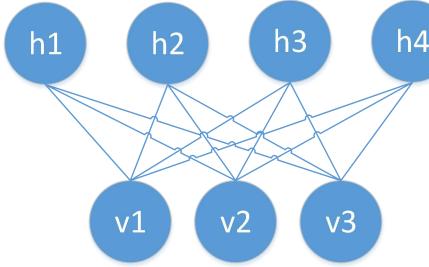


图 3.14 RBM 结构图

由上图可看出，RBM 本身是基于二分图的无向图模型，具体来说是包含一层可观察变量和单层潜变量的无向概率图模型，且可观察变量之间没有连接，潜变量之间也没有任何的连接。这里，令观察层由一组 n_v 个二值随机变量组成，统称为向量 v ；将 n_h 个二值随机变量的隐藏层记为 h ，则其联合概率分布由能量函数指定：

$$P(\mathbf{v} = v, \mathbf{h} = h) = \frac{1}{Z} \exp(-E(v, h)) \quad (3.3)$$

RBM 的能量函数为：

$$E(v, h) = -b^T v - c^T h - v^T \mathbf{W} h \quad (3.4)$$

其中 \mathbf{W} 为权重， b, c 为偏置，该函数为关于 w 和 b 的线性函数，易求得其偏导。其中 Z 是被称为配分函数的归一化常数：

$$Z = \sum_v \sum_h \exp\{-E(v, h)\} \quad (3.5)$$

同时，RBM 结构的限制产生了良好的属性，使得计算独立的条件分布时较为容易：

$$p(\mathbf{h} | \mathbf{v}) = \prod_i p(h_i | v) ; \quad p(\mathbf{v} | \mathbf{h}) = \prod_i p(v_i | h)$$

以下为推导独立条件分布的过程：

$$\begin{aligned}
p(h|v) &= \frac{p(h,v)}{p(v)} \quad v \text{ 为可观察变量} \\
&= \frac{1}{p(v)} \cdot \frac{1}{2} \exp \{ b^T v + c^T h + v^T w h \} \\
&= \frac{1}{2^l} \cdot \exp \{ c^T h + v^T w h \} \\
&= \frac{1}{2^l} \cdot \exp \left\{ \sum_{j=1}^{n_h} c_j^T h_j + \sum_{i=1}^{n_v} v_i^T w_{i,j} h_j \right\} \\
&= \frac{1}{2^l} \cdot \prod_{j=1}^{n_h} \exp \{ c_j^T h_j + v^T w_{i,j} h_j \}
\end{aligned}$$

又：向量 h 上的联合概率可写成单个元素 h_j 上 $(h_j \mid v)$ 分布的乘积。

原问题 \Leftrightarrow 对单个二值 h_j 上的分布进行归一化。 sigmoid 函数。

$$\begin{aligned}
p(h_j=1|v) &= \frac{\tilde{p}(h_j=1|v)}{\tilde{p}(h_j=0|v)} = \frac{\exp \{ c_j + v^T w_{i,j} \}}{\exp \{ c_j \} + \exp \{ c_j + v^T w_{i,j} \}} \quad \text{sigmoid 函数。} \\
\text{则 } p(h|v) &= \prod_{j=1}^{n_h} \sigma(c_j + v^T w_{i,j})
\end{aligned}$$

同理可得， $p(v|h) = \prod_{i=1}^{n_v} \sigma(b_i + w_i^T h)$

图 3.15 独立条件分布概率推导

3.4.2 实验分析

为了看到模型的学习过程，这里训练了两个模型，第一个模型对所有的数据集进行训练，第二种模型只训练训练集中的某一个特定的数字，来生成 10 类手写体中的任意一个数字，结果如下。

- 学习训练集中的所有数据，并对学习到的 weight 进行输出。

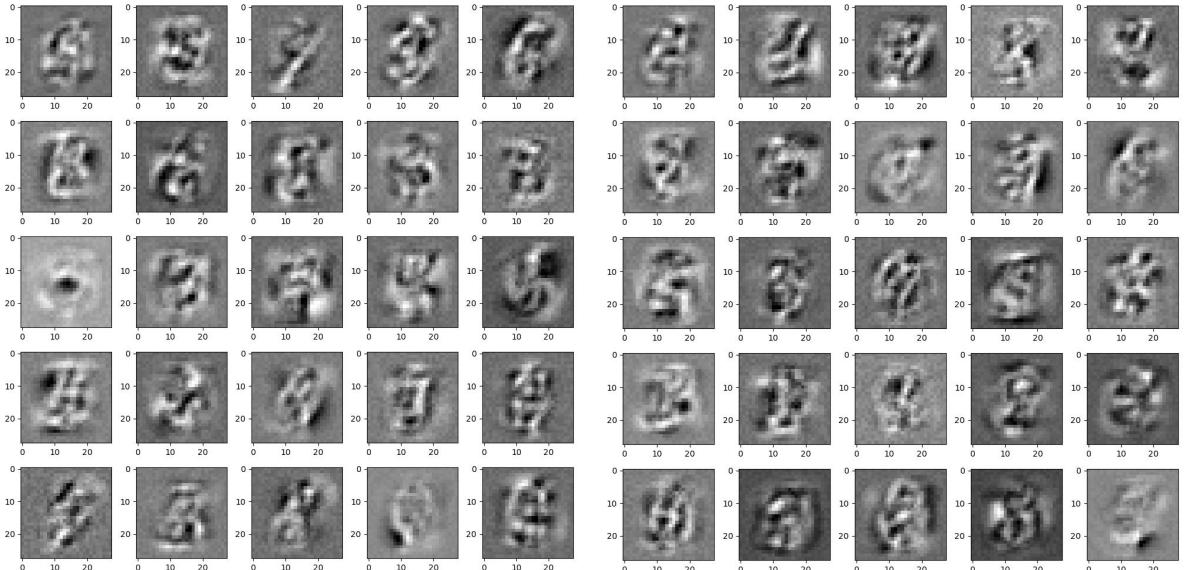


图 3.16 学习到的 weight 可视化

- 学习训练集中的特定数据（这里为 5），并对手写数字 3 进行学习。第一行为原始数据集中的 3，第二行为分别经过 epoch=10、15、20、25、30、35 后对数字 3 的重建。剩下的 6 附图分别为经过 epoch=10、15、20、25、30、35 时相应的 weight 输出。

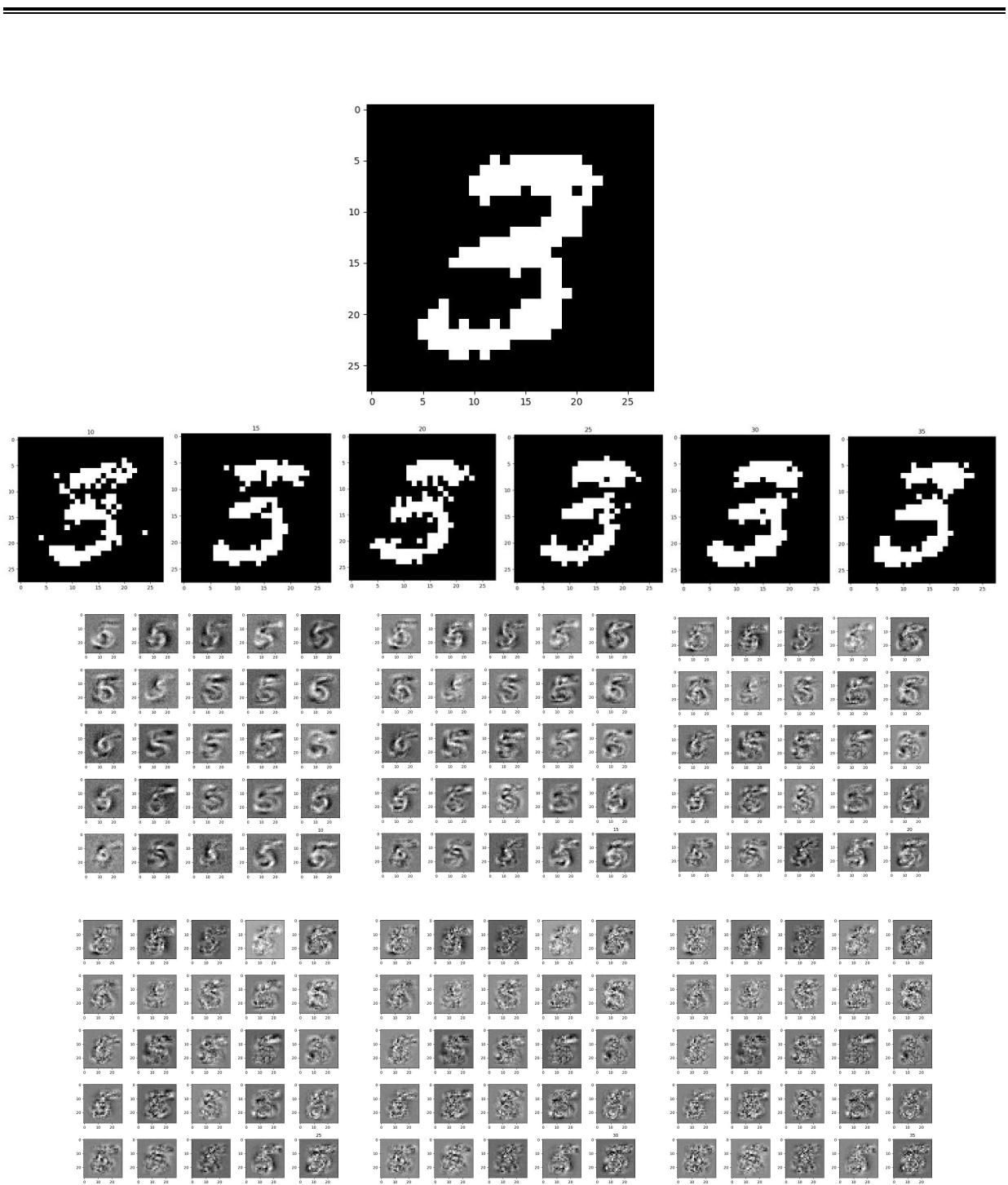


图 3.17 原始数字 3，随 epoch 增加生成的 3 以及学习到的 weight 可视化

【结果分析】从第二种训练的结果中，可以发现，在较少的 epoch 下生成的数字 3 中或多或少都有数字 5 的痕迹，但随着 epoch 的增加，5 的痕迹会越来越少，同时从 weight 的可视化图中可以发现，随着 epoch 的增加，模型能够学习到更细致的特征，从而能够更好的生成一个除训练数字之外的数字。

3.5 深度置信网络 Deep Belief Networks

3.5.1 算法解析

在有了对受限玻尔兹曼机模型的大概了解后，介绍 DBN 就从其网络结构开始说起。由下图，我们可以看出，深度信念网络时涉及有向和无向连接的混合图模型。与 RBM 一样，它同样没有层内连接，但不同点在于，DBN 具有两个隐藏层，因此隐藏单元之间的连接主要体现在分开的层中。值得一提的是，DBN 中所需的所有局部条件概率分布均可以复制 RBM 的局部条件概率分布。

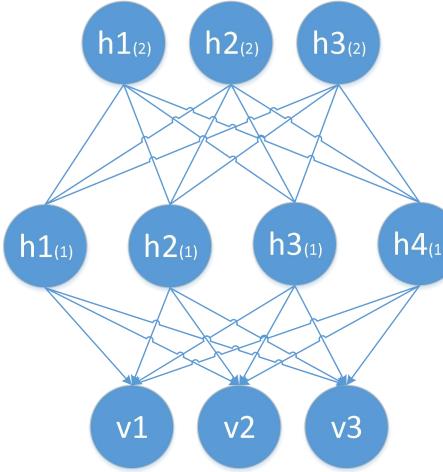


图 3.18 DBN 网络结构

除上图的画法之外，DBN 同样也可以使用完全无向图来进行表示，只不过在使用完全无向图进行表示时，需要层内连接，以此来捕获父节点间的依赖关系。

具有 1 个隐藏层的 DBN 包含 1 个权重矩阵： $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(l)}$ ，同时也包含 $l+1$ 个偏置向量： $b^{(0)}, b^{(1)}, \dots, b^{(l)}$ ，其中 $b^{(0)}$ 是可见层的偏置。结合受限玻尔兹曼机模型的推导和 DBN 的概率图模型，DBN 表示的概率分布计算如下：

$$P(h^{(l)}, h^{(l-1)}) \propto \exp(b^{(l)\top} h^{(l)} + b^{(l-1)\top} h^{(l-1)} + h^{(l-1)\top} \mathbf{W}^{(l)} h^{(l)}) \quad (3.6)$$

$$P(h_i^{(k)} = 1 | h^{(k+1)}) = \sigma(b_i^{(k)} + \mathbf{W}_{:,i}^{(k+1)\top} h^{(k+1)}) \quad \forall i, \forall k \in 1, \dots, l-2 \quad (3.7)$$

$$P(v_i = 1 | h^{(l)}) = \sigma(b_i^{(0)} + \mathbf{W}_{:,i}^{(l)\top} h^{(l)}) \quad \forall i \quad (3.8)$$

由上分析，可以看出一个 DBN 模型由若干个 RBM 堆叠而成，训练过程由低到高逐层进行训练。这个过程和想法可以类比之前的 DAE 和 SDAE（DAE 到 SDAE 的扩展其实借鉴了 RBM 到 DBN 的扩展而来），其训练过程如下图所示。

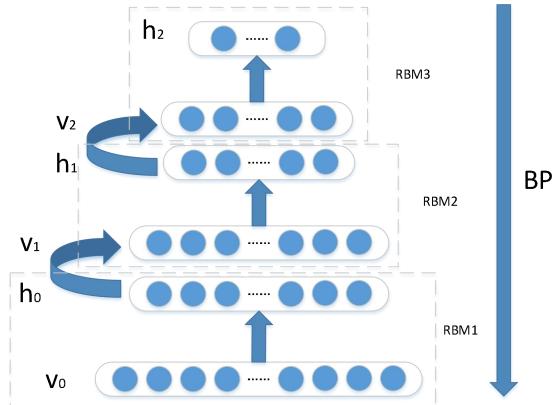


图 3.19 DBN 堆叠 RBM

通过隐层提取特征使后面层次的训练数据更加有代表性，通过可生成新数据能解决样本量不足的问题。逐层的训练过程如下：（1）最底部 RBM 以原始输入数据进行训练；（2）将底部 RBM 抽取的特征作为顶部 RBM 的输入继续训练；（3）重复这个过程训练以尽可能多的 RBM 层。

```
def forward(self, input_data):
    """
    running the forward pass
    do not confuse with training this just runs a forward pass
    ...
    v = input_data
    for i in range(len(self.rbm_layers)):
        v = v.view((v.shape[0], -1)).type(torch.FloatTensor)#flatten
        p_v, v = self.rbm_layers[i].to_hidden(v)
    return p_v, v
```

图 3.20 pytorch 关键代码

从以上关键代码中也可以看出，DBN 是在 RBM 基础上进行堆叠而形成的一种网络结构。

3.5.2 实验分析

为了与 RBM 进行对比，这里使用和 RBM 中一样的两种训练方式，第一个模型对所有的数据集进行训练，第二种模型只训练训练集中的某一个特定的数字，来生成 10 类手写体中的任意一个数字，结果如下。

- 学习训练集中的所有数据，并对学习到的 weight 进行输出。

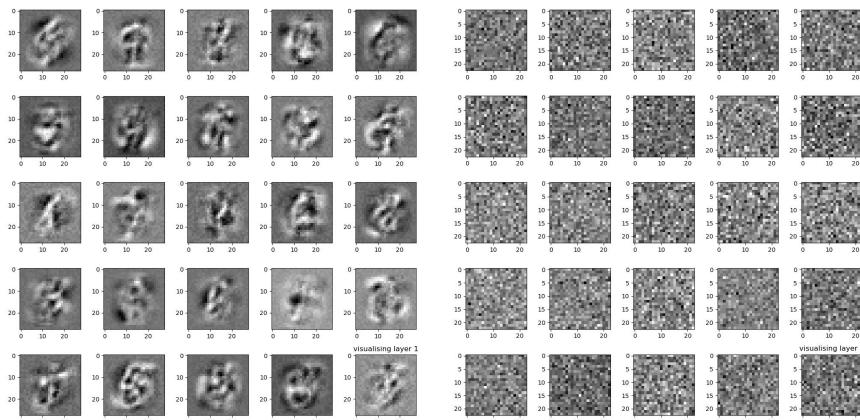


图 3.21 学习到的 weight 可视化

• 学习训练集中的特定数据（这里为 5），并对手写数字 1 进行学习。第一行为原始数据集中的 1，第二行为分别经过 epoch=10、15、20、25、30、35 后对数字 1 的重建。剩下的 6 附图分别为经过 epoch=10、15、20、25、30、35 时相应的 weight 输出。这里与 RBM 有所不同的是，为了验证其堆叠的效果，DBN 在此部分选择了对数字 1 进行重建，因为数字 5 和 1 之间的差别远远大于 RBN 实验中 5 和 3 的区别，结果如下所示。

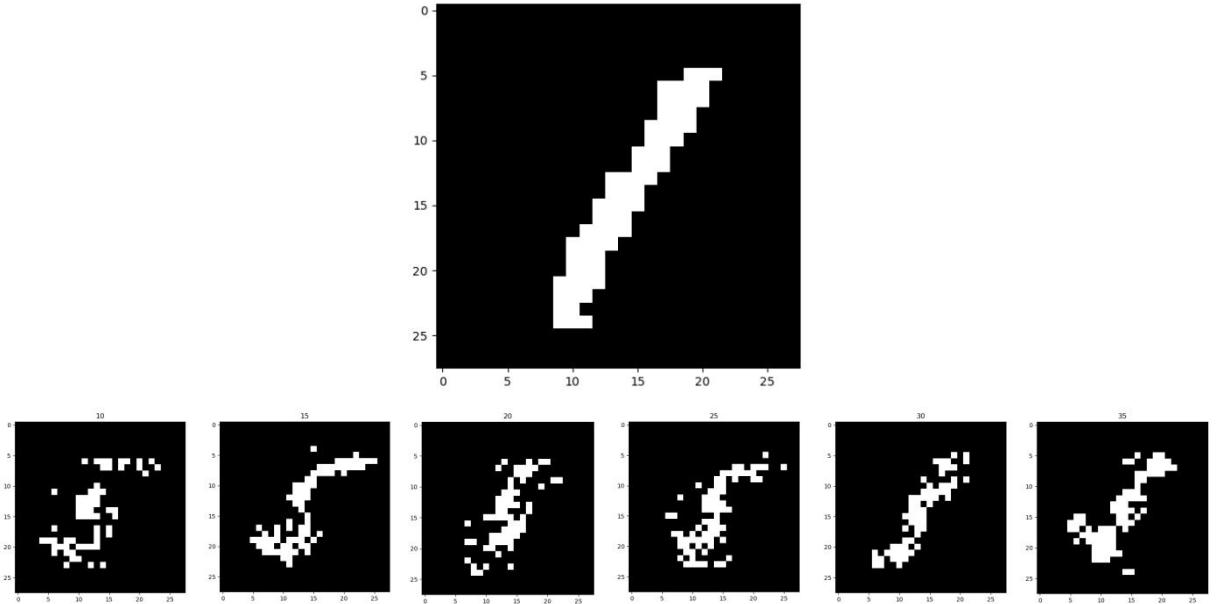


图 3.22 原始数字 1，随 epoch 增加生成的 1

【结果分析】在第二种训练结果中，发现其与 RBM 类似，在较少的 epoch 下生成的数字 1 中或多或少都有数字 5 的痕迹，但随着 epoch 的增加，5 的痕迹会越来越少，但在 epoch=30 的生成效果比 epoch=35 时的生成效果好。

回想一下 RBM，由可见层、隐层组成，显元用于接受输入，隐元用于提取特征，因此隐元也有个别名，叫特征检测器。也就是说，通过 RBM 训练之后，可以得到输入数据的特征。另外，RBM 还通过学习将数据表示成概率模型，一旦模型通过无监督学习被训练或收敛到一个稳定的状态，它还可以被用于生成新数据。由此可见，DBN 便是利用了这思想，将 RBM 进行堆叠，从而使得模型的性能更加。

四 项目总结

4.1 监督学习算法综合分析

监督学习 (supervised learning) 是指从标注数据中学习预测模型的机器学习问题。预测模型对给定的输入产生相应的输出，与在标注数据集中标注数据进行对比，并采用一定的代价函数和误差反向传递算法来更改预测模型中的参数，来达到模型学习的过程，使得预测模型可以具有更好的泛化能力。以下为监督学习的流程。

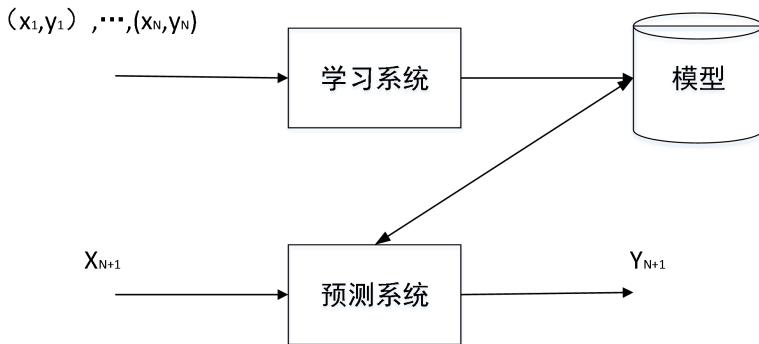


图 4.1 监督学习的流程

其学习过程大概可以概括为以下 6 个部分：

- (1) 得到一个有限个的训练数据集合；
- (2) 确定模型的假设空间，也就是所有的备选模型；
- (3) 确定模型选择的准则，即学习策略；
- (4) 实现求解最优模型的算法；
- (5) 通过学习方法选择最优模型；
- (6) 利用学习的最优模型对新数据进行预测或分析。

在本项目中所用到的四个监督学习的算法：Logistic Regression、Multilayer perceptron、Deep Convolutional Network 和 Long Short Term Memory network 这四个比较基础和经典的监督学习的算法，每个算法的流程也均为以上的 6 步。在 3.1、3.2 和 3.3 的实验分析中可以看出，在 mnist 数据集中表现较好的为深度卷积神经网络，其原因是卷积操作能够更深的提取到图片中的特征，包括其边缘特征，使得学习出来的预测模型能够由更好的泛化能力。下表为四个算法在本实验中相对较好的结果对比。

表 4.1 四个算法在本实验中相对较好的结果

	LR	MLP	Deep CNN	LSTM
Test acc	92%	97%	98.73%	98.69%

由于本项目使用的数据集是图片数据集，所以其在深度卷积神经网络中的性能最好，分类的精度最高。但 CNN 和 LSTM 的精度十分接近。

4.2 无监督学习算法综合分析

无监督学习（unsupervised learning）是指从无标注数据中学习预测模型地机器学习问题。无标注数据是自然得到地数据，预测模型表示数据地类别、转换或概率，无监督学习地本质是学习数据中地统计规律或潜在结构。

无监督学习可以用于对已有数据进行分析，同样也可以对未知的数据进行预测。如下图所示，在学习过程中，学习系统从训练数据及学习，得到一个最优模型，表示为函数 $z = \hat{g}(x)$ ，条件概率分布 $\hat{P}(z | x)$ 或者条件概率分布 $\hat{P}(x | z)$ 。在预测过程中，预测系统对于给定的输入 x_{N+1} ，由模型 $z_{N+1} = \hat{g}(x_{N+1})$ 或 $z_{N+1} = \arg \max_z \hat{P}(z | x_{N+1})$ 来给出相应的输出 z_{N+1} 进行聚类或降维，或者由模型 $\hat{P}(x | z)$ 给出输入的概率 $\hat{P}(x_{N+1} | z_{N+1})$ 进行概率估计。

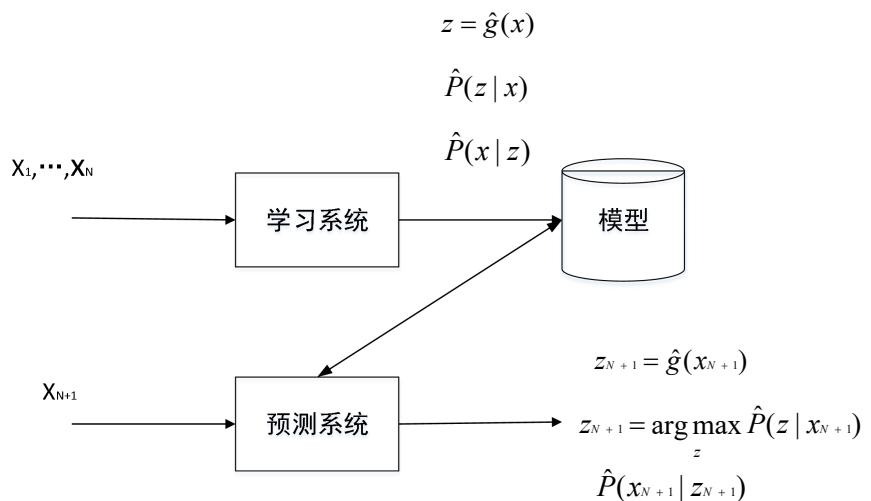


图 4.2 无监督学习的流程

在本次项目中，使用了 Auto Encoders、Denoising Autoencoders、Stacked Denoising Auto-Encoders、Restricted Boltzmann Machines、Deep Belief Networks 这五种无监督学习的算法。其中前三种算法依次递进，第三种是第二种的堆叠式，当然也可以是 SAE，同样的第五种是第四种的‘堆叠式’。

在这几种算法的结果中，均采用重构的方式来对预测模型进行验证，前三种算法分别在同一个测试集中的重建结果如下所示，从左到右，依次为 SDAE、DAE、AE。可以看出 SDAE 相较于 DAE 和 AE 有较好的重建结果。

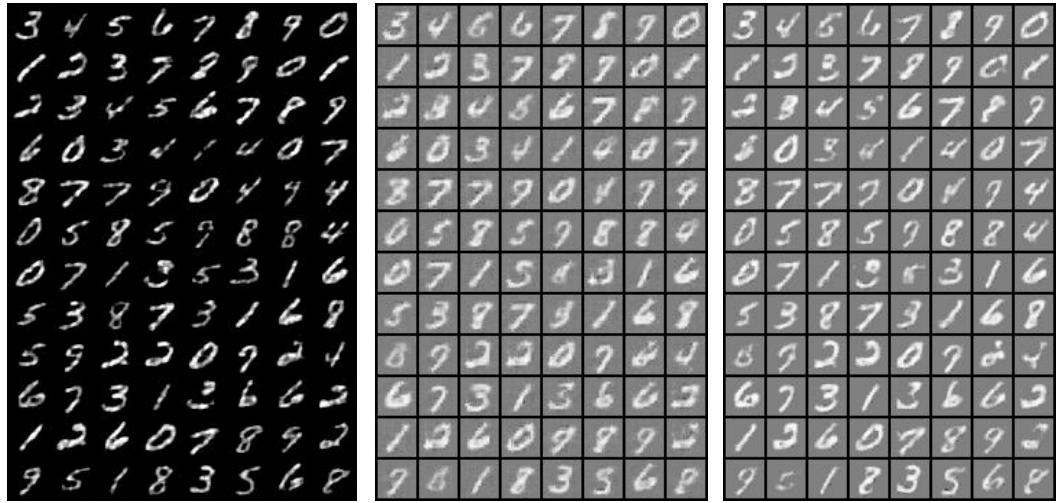


图 4.3 前三中算法分别在同一个测试集中的重建结果

同时对五种无监督学习最后一次重构出的 loss 值进行了对比，结果如下表所示。

表 4.2 五种无监督学习算法重构出的 loss 值

	AE	DAE	SDAE	RBM	DBN
loss	0.0255	0.0325	0.0157	0.0443	0.0350

由此结果可以看出，经过堆叠后，模型均在相应的网络结构中表现出了较优的性能。这里 DAE 比 AE 性能较差的原因，可能是因为在加入了 corrupt 之后，需要更多的数据集和迭代次数才能学习出 mask 的一部分，但本项目中使用了相同的迭代次数，所以 DAE 的效果相较于 AE 要差一些。

4.3 总结

为了便于以后更好的学习，本项目的所有代码和实验结果均以发布在 github 上，
https://github.com/Cathy-t/basic_AI_algorithm。