智能算法与应用 实验二

实验题目：神经网络和卷积神经网络（浅层模型&深度学习）解决数字识别问题

> 成员1：唐瑞怡 18340159，负责BP算法实现
>
> 成员2：张琛颐 18340205，负责CNN算法实现

> 日期：2021/5/29

摘要

本实验利用反向传播神经网络（$BPNN$）和卷积神经网络（CNN）两个方法，从浅层模型和深度学习模型两个角度利用MNIST数据集来解决数字识别问题，即对图片上0-9的数字进行识别判断。

在浅层模型的探索中，对相同规模的数字识别问题，搭建了BP神经网络，对网络中的各个参数进行调参分析（如隐藏层神经元个数、激活函数种类、学习率、随机初始化参数方法等），比较了各参数下的运行结果，最终得到了识别率达到的结果，并得出结论：...参数对$BPNN$框架的影响明显，而...参数影响不明显。

在深度学习模型的探索中，利用卷积神经网络对0-9数字进行识别，以Lenet网络为结构基础，对比了不同网络结构对实验的影响（比如卷积层数量，卷积核大小，卷积核数量，池化层等等因素），最终得到了识别率达到的结果。

一、导言

> 要解决的问题描述，问题背景介绍；
>
> 拟使用的方法，方法的背景介绍；

1.1 数字识别问题

本次实验中，需要对手写的0-9的图像利用神经网络进行识别，这是一个10分类的问题，也就是说神经网络输入了手写图像的数据，最后输出为一个10维的向量，对应当前输入属于哪一个图像的概率，然后取概率最高的类别来作为输入的预测结果。

这次实验中，利用了MINST数据集来训练和测试模型，这个数据集中，包含了0-9的图像，有60000个训练样例和10000个测试样例。对于数据集中的内容，每一个图像都是标准的28*28的大小图片，并且已经将手写的数字固定到图片中心。这个数据集数据规范，大小合适，非常适用用于本次实验来对神经网络进行训练。

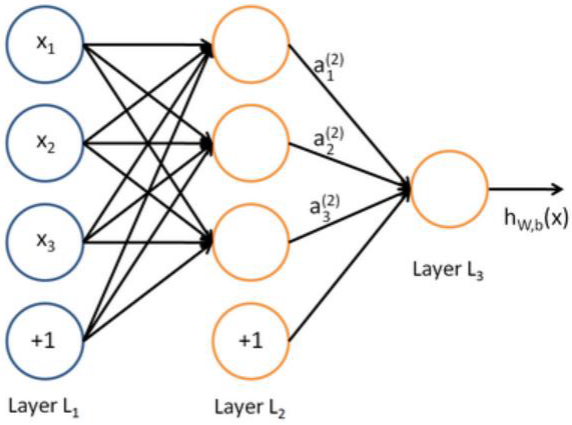
1.2 反向传播神经网络（$BPNN$）

$BPNN$ 全称为 Back Propagation Neural Network，意思为反向传播神经网络。BP方法是用来对人工神经网络进行优化的，即误差反向传播算法。它属于有教师指导的学习方式。包括两个过程：正向传播（输入信号从输入层经隐含层，传至输出层的过程）和反向误差传播（将误差从输出层反向传至输入层，并通过梯度下降算法来调节连接各层之间权值 w 与偏置值 b 的过程）。

深度学习的基本原理是基于**人工神经网络**的，而$BPNN$就是其中的典型模型。人工神经网络起源于上世纪40～50年代，它是在基于人脑的基本单元——神经元的建模与联结，模拟人脑神经系统，形成一种具有学习、联想、记忆和模式识别等智能信息处理的人工系统，称为人工神经网络，是一种连接模型。在人脑中，利用神经元来传递和处理信息，每个神经元通过树突来接受神经冲动，然后神经元产生兴奋，再将结果通过轴突将神经冲动传到下一个神经元。模仿这个过程，就产生了神经网络的结构，对

于每一个节点，它模仿了神经元的功能，每一个节点的输入对应了树突传播神经冲动的过程，节点内的激活函数对应了神经元中产生兴奋的过程，节点的计算结果传输到下一次节点的过程模拟了轴突传播神经冲动的过程。

这次实验中，BPNN有3层结构，即一层输入层，一层隐藏层和一层输出层：



1.3 卷积神经网络（CNN)

卷积神经网络是多层感知机的变种，是前馈神经网络的一种，被广泛应用于二维数据上（比如这次实验要用的图像数据）。在图像处理中，CNN网络结构之所以可以取得成功，得益于其中卷积核的两大特点：局部连接和参数共享。由于这两个特点的存在，CNN网络有下面的优势：

1. 更少的参数存储

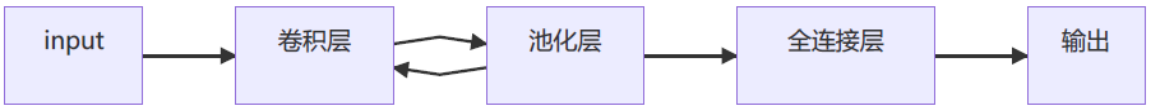
   与普通的神经网络不同，CNN中卷积结构仅进行局部连接，这可以大大减少存储的参数数量，并且降低训练模型的时间。

2. 提取出局部信息

   在传统的全连接方式里面，隐藏层的每一个节点都与输入的信息关联，每一个隐藏层都包含整体数据的信息特点，无法得到部分的信息。而在图像识别方面，通常需要局部特征来进行判断。由于局部连接的存储，CNN结构可以提取出图片中的局部特征，更合适图像识别。

   同时，CNN还可以提取出局部不变性的特征。比如说当图像进行了缩放、平移操作之后，从全局上看，整个数据改变了很多，但是如果只针对局部特征来说，变化却很少，可以有效减少由于数据不够规整对模型的影响。

在CNN网络结构中，需要3种最基本的结构：卷积层、池化层和全连接层，一般的CNN结果如下图：



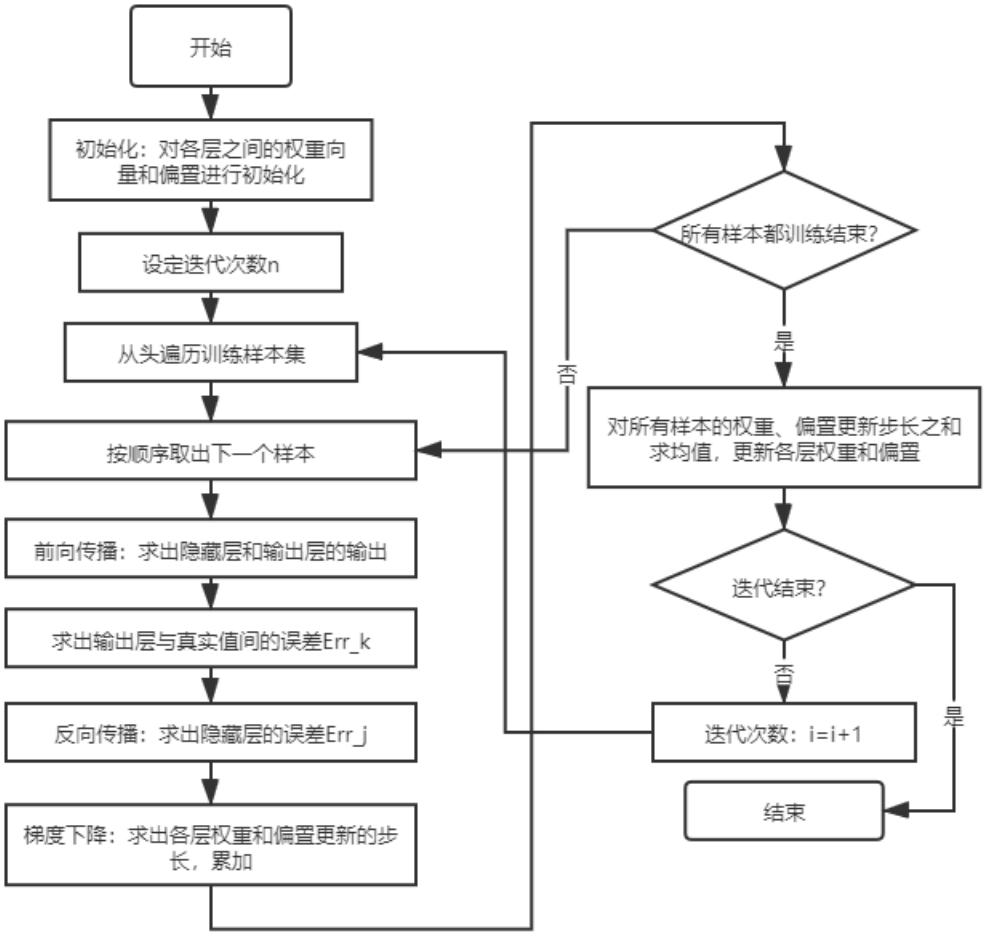
二、实验过程

> 所用的具体的算法思想流程；实现算法的程序主要流程，功能说明；

2.1 反向传播算法

2.1.1 算法流程

BPNN算法的流程图如下：

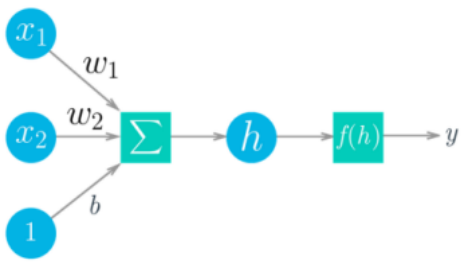


BPNN算法主要分为以下三大部分：

1. 数据处理

1. 读入数据，并将训练集，测试集数据用矩阵存储在变量train_images、train_labels、test_images、test_labels中
2. 给train_images、test_images矩阵中增加一列
3. 将train_labels矩阵进行拓展，形成one-hot矩阵，并且存储在新矩阵onehot_labels中

2. 前向传播



当输入数据特征只有1维的情况下，设输入数据为x，线性函数为：

$$f(x)=\sigma(wx+b)$$

其中，σ为激活函数

我们希望尽可能让预测值$f(x)$与真实的y接近。

当输入数据的特征多维的情况下，$\vec{x}=[x_1,x_2,\ldots,x_n]$，设系数矩阵$\vec{w}=[w_1,w_2,\ldots,w_n]$，线性函数可以写成：

$$f(\vec{x}) = \sigma(\vec{x}\vec{w}^T + b)$$

若将$\vec{x}, \vec{w}$改写成$[1, x_1, x_2, \ldots, x_n]$、$[w_0, w_1, \ldots, w_n]$,可以得到:

$$f(\vec{x}) = \sigma(\vec{x}\vec{w}^T)$$

激活函数有很多种，如

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

$$f(x) = Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f(x) = Leaky\_Relu(x) = max(0.01x, x) = \begin{cases} x, & x >= 0 \\ 0.01x, & x < 0 \end{cases}$$

它们的比较如下：

(1) $Sigmoid$函数

- 优点：Sigmoid的取值范围在(0, 1)，而且是单调递增，比较容易优化； 求导比较容易，可以直接推导得出。
- 缺点： Sigmoid函数**收敛比较缓慢**；Sigmoid函数并不是以（0,0）为中心点； 由于Sigmoid是软饱和，容易产生梯度消失，对于深度网络训练不太适合（多层隐藏层时）。

(2) $Tanh$函数

- 优点： 函数输出以（0,0）为中心，收敛速度相对于`Sigmoid`更快。
- 缺点：`tanh`并没有解决`sigmoid`梯度消失的问题

(3) $Relu$函数

- 优点： 相比`Sigmoid`/`Tanh`函数，使用梯度下降法时，收敛速度更快； 相比`sigmoid`/`tanh`函数，`Relu`只需要一个门限值，即可以得到激活值，计算速度更快
- 缺点： `Relu`的输入值为负的时候，输出始终为0，其一阶导数也始终为0，这样会导致神经元不能更新参数，也就是神经元不学习了，这种现象叫做“Dead Neuron死神经元”。

(4) $Leaky\ Relu$函数

- 优点： 收敛速度要比`Sigmoid`和`tanh`快很多，有效的缓解了梯度消失问题； 且**与`Relu`相比**，对于小于0的值，梯度也不会永远为0，使得负值的信息不回全部丢失。 解决了`Relu`函数进入负区间后，导致神经元不学习的问题。
- 缺点：在基于梯度的学习会比较慢。

3.后向传播

在反向传播阶段， 权重更新方程应用于相反的方向。也就是说， 第$(l+1)$层的权重在更新第l层的权重之前被更新，这允许我们使用第$(l+1)$层神经元的误差来估计第l层神经元的误差。并且，使用“**梯度下降法**”不断迭代更新权重向量w和偏置b。

下面以本实验要求的**三层神经网络**，且输出层只有十个节点（对应数字0-9）为例分析。

- **损失函数**：在优化模型中，我们可以定义一个损失函数，来判断模型预测值和真实值之间的差距，通过最小化损失函数，进一步优化模型。在这次实验中，利用了交叉熵函数作为损失函数：

交叉熵：$L(w_1, w_2, \ldots, w_k) = -\frac{1}{N}\sum_{l=1}^{N}\sum_{k=1}^{K} y_k^{(l)} log softmax_k(\vec{x}^{(l)} W^T)$

$$\frac{\partial L(W)}{\partial W} = \frac{1}{N}\sum_{l=1}^{N} X^{(l)T}(softmax(X^{(l)}W) - \vec{y}^{(l)})$$

其中，l表示第l条数据，下标k表示这个数据在第k个位置的取值

- 对于**输出层**中的单元`k`，误差Err_k 由下式计算：

  $$Err_k = (y - y')f'(x)$$

  其中h 为输出节点的输入，y'为输出层预测值，y为输出层真实值。

- 对于**隐藏层**单元`j`，误差Err_j为：

  $$Err_j = Err_k\ w_{jk}\ f'(x_j)$$

  其中，Err_k为输出层误差，w_{jk}为连接隐藏层和输出层之间的权重向量，$f'(h_j)$ 为隐藏层激活函数的导数，h_j为隐藏层输入。

- 每个数据点更新权重**步长（梯度）**：

  $$\triangle W_{jk} = \triangle W_{jk} + Err_k * O_j$$
  $$\triangle W_{ij} = \triangle W_{ij} + Err_j * O_i$$
  $$\triangle b_j = \triangle b_j + Err_k$$
  $$\triangle b_i = \triangle b_i + Err_j$$

  其中，`i`表示输入层，`j`表示隐藏层，`k`表示输出层，O_x表示对应层的输出。

- 当所有数据点都计算更新完之后，求均值，更新权重向量：

  $$W_{jk} = W_{jk} + \eta\frac{\triangle W_{jk}}{m}$$
  $$W_{ij} = W_{ij} + \eta\frac{\triangle W_{ij}}{m}$$
  $$b_i = b_i + \eta\frac{\triangle b_i}{m}$$
  $$b_j = b_j + \eta\frac{\triangle b_j}{m}$$

  其中，m为数据点的个数，η为学习率。

2.1.2 主要流程及功能说明

数据预处理

在分batch之前，对训练集进行shuffle操作。

```
# 随机打乱数据集：先通过zip操作绑定，shuffle，再解绑
def shuffle_data(train_images, train_labels):
    data_class_list = list(zip(train_images, train_labels))
    random.shuffle(data_class_list)
    train_images, train_labels = zip(*data_class_list)  # 解压
    return np.array(list(train_images)), np.array(list(train_labels))
```

对数据集（训练+测试集）的`x`进行扩展，对`label`进行one-hot操作。

- one-hot操作是为了后面的计算方便。
- 对`x`进行扩展是为了加一列，用于偏置b的计算。

```
def preprocess_data(class_num, train_images, train_labels, test_images,
test_labels):
    # 给images加一列：用于b
    train_images = train_images.T
    train_images = np.insert(train_images, 0, np.ones(train_images.shape[1]),
axis=0)
    train_images = train_images.T
    test_images = test_images.T
    test_images = np.insert(test_images, 0, np.ones(test_images.shape[1]),
axis=0)
    test_images = test_images.T
    # 拓展labels：转成one-hot形式
    onehot_train_label = np.eye(class_num)[train_labels]  # [样例数，类别数]
    onehot_test_label = np.eye(class_num)[test_labels]
    return train_images, onehot_train_label, test_images, onehot_test_label
```

参数初始化

对参数`w`和偏置`b`进行随机初始化。

```
# choice表示初始化的方式（0-全零初始化；1-随机初始化；2-xavier Glorot normal），row和col
表示矩阵的大小
def init_w(choice, row, col)
```

训练函数

BPNN训练函数：包括前向传播和后向传播两个过程。并且进行了$mini-batch$操作。

```
# BPNN训练函数
def train(train_images, train_labels, test_images, test_labels,
train_onehot_labels, test_onehot_labels, w_hidden, w_output, b_output,
activation_choice, lr, iteration_number, batch_size):
    train_acc_list = []
    test_acc_list = []
    train_loss_list = []
    test_loss_list = []
    data_num = train_images.shape[0]
    for i in range(iteration_number):
        train_acc=0
        train_loss=0
        for j in range(data_num//batch_size):
            # 得到训练集的当前batch
            train_images_batch = train_images[j*batch_size: (j+1)*batch_size]
            train_labels_batch = train_labels[j*batch_size: (j+1)*batch_size]
            train_onehot_labels_batch = train_onehot_labels[j*batch_size:
(j+1)*batch_size]
            # 前向传播
            train_hidden_output, train_output_output = forward_pass(w_hidden,
w_output, b_output, train_images_batch, activation_choice)
            # 计算train和test的正确率
            train_acc += cal_accuracy(train_output_output, train_labels_batch)
            # 计算误差
            train_loss += cross_entropy_loss(train_output_output,
train_onehot_labels_batch)
             # 后向传播
```

```
            w_hidden, w_output, b_output = backward_pass(train_hidden_output,
train_output_output, w_hidden,
w_output,b_output,activation_choice,train_images_batch,
train_onehot_labels_batch, lr)
        # 运行测试集结果
        _, test_output_output = forward_pass(w_hidden, w_output, b_output,
test_images, activation_choice)
        test_acc = cal_accuracy(test_output_output, test_labels)
        test_loss = cross_entropy_loss(test_output_output, test_onehot_labels)
        # 计算训练集、测试集误差误差（分batch后需算平均值）
        train_acc_list.append(train_acc/(data_num//batch_size))
        test_acc_list.append(test_acc)
        train_loss_list.append(train_loss/(data_num//batch_size))
        test_loss_list.append(test_loss)
        print("第%s次迭代：train_acc=%s, test_acc=%s" % (i,
train_acc/(data_num//batch_size), test_acc))
        print("第%s次迭代：train_loss=%s, test_loss=%s" % (i,
train_loss/(data_num//batch_size), test_loss))
    return train_acc_list, test_acc_list, train_loss_list, test_loss_list
```

激活函数及其倒数

在输出层和隐藏层中，都需要用到激活函数。而激活函数的计算容易溢出，因此采用了不少tricks来进行防溢出操作。

```
# sigmoid函数
def sigmoid(x):
    result = 0.5*(1+np.tanh(0.5*x))
    return result
# 激活函数
def activation_func(func_type, x):
    if func_type == "sigmoid":
        return sigmoid(x)
    elif func_type == "tanh":
        return 2 * sigmoid(2*x) - 1
    elif func_type == "relu":
        return np.maximum(0.01*x, x)
# 激活函数的求导
def derivation(func_type, hidden_output):
    if func_type == "sigmoid":
        return hidden_output * (1 - hidden_output)
    elif func_type == "tanh":
        return 1 - np.square(hidden_output)
    elif func_type == "relu":
        temp = hidden_output.copy()      # 深拷贝numpy
        temp[temp < 0] = 0.01
        temp[temp >= 0] = 1
        return temp
# 计算交叉熵，返回输出层loss
def cross_entropy_loss(output_output, onehot_labels):
    data_num = output_output.shape[0]
    log_result = np.log(output_output+1e-8)
    loss = -1 / data_num * np.sum(onehot_labels * log_result)
    return loss
# 交叉熵的求导：返回输出层error和cross-entropy导数
def derivative_CE(output_output, onehot_labels, hidden_output):
    data_num = output_output.shape[0]
```

```
    softmax_result = output_output - onehot_labels
    result = np.dot(hidden_output.T, softmax_result) / data_num
    return softmax_result, result
```

前向传播

前向传播过程：通过当前的参数`w`和`b`，计算并返回隐藏层输出和输出层输出。

```
# activate_choice为激活函数类型: "sigmoid", "tanh", "relu"
def forward_pass(w_hidden, w_output, b_output, dataset, activate_choice):
    # 隐藏层计算
    hidden_input = np.dot(dataset, w_hidden)
    hidden_output = activation_func(activate_choice, hidden_input)
    # 输出层计算
    data_num = dataset.shape[0]
    b_output1 = np.repeat(b_output, data_num,axis=0)     # 沿着行扩展成[N,10]
    output_input = np.dot(hidden_output, w_output) + b_output1
    output_output = softmax(output_input)
    return hidden_output, output_output
```

后向传播

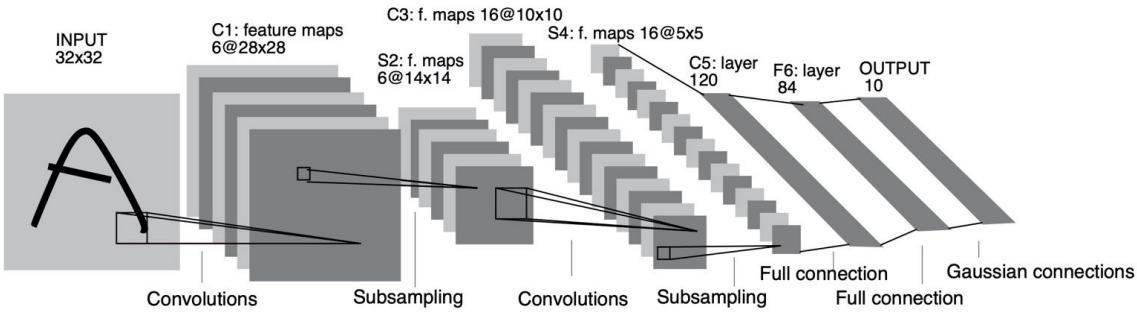
后向传播过程：计算输出层、隐藏层误差（包括梯度），并更新两个参数`w_output`和`w_hidden`。

```
def backward_pass(hidden_output, output_output, w_hidden, w_output, b_output,
activation_choice, dataset, onehot_labels, lr):
    # 计算输出层误差，输出层梯度
    output_error, output_grad = derivative_CE(output_output, onehot_labels,
hidden_output)    # [N,10] [hidden_num,10]
    # 计算隐藏层误差
    hidden_error = derivation(activation_choice, hidden_output) *
np.dot(output_error, w_output.T)
    # 更新隐藏层-输出层参数：w_output
    w_output = w_output - lr * output_grad
    output_error_mean = np.mean(output_error, axis=0)   # 算出output_error均值：从
[N,10]变成[1,10]
    b_output = b_output - lr * output_error_mean
    # 更新输入层-隐藏层参数：w_hidden
    w_hidden = w_hidden - lr * np.dot(dataset.T, hidden_error) /
dataset.shape[0]
    return w_hidden, w_output, b_output
```

2.2 卷积神经网络

2.2.1 算法流程

这次实验中，以LeNet网络作为基础，来探究卷积网络中不同结构对效果的影响。LeNet结构如下：



整个实验伪代码如下：

```
procedure: CNN_identify_number
input: train_data, train_label, test_data, test_label
output: train_acc_list, train_loss_list, test_acc_list, test_loss_list

train_data_batches <- 将train_data根据batch_num进行分批
train_label_batches <- 将train_label根据batch_nu进行分批
test_data_batches <- 将test_data根据batch_nu进行分批
test_label_batches <- 将test_label根据batch_nu进行分批

for i:=1 to max_epoch:
    total_loss_train:=0
    total_loss_test:=0
    total_acc_train:=0
    total_acc_test:=0
    for j:=0 to batch_num:
            train_predict_label=CNN(train_data_batches[j])
            test_predict_label=CNN(test_data_batches[j])
            loss_train=cross_entropy(train_predict_label,train_label_batches[j])
            loss_test=cross_entropy(test_predict_label,test_label_batches[j])
            loss_train.backward()
            total_loss_train += loss_train
            total_loss_test += loss_test
            total_acc_train +=
cal_acc(train_predict_label,train_label_batches[j])
            total_acc_test += cal_acc(test_predict_label,test_label_batches[j])
    train_acc_list.append(total_acc_train/batch_num)
    test_acc_list.append(total_acc_test/batch_num)
    train_loss_list.append(total_loss_train/batch_num)
    test_loss_list.append(total_loss_test/batch_num)
```

神经网络结构分析

- 卷积层

  卷积层通过局部连接的方式来提取图像的特征，利用了生物学上感知野的原理，也就是一个神经元只能被特定的感知野激活。

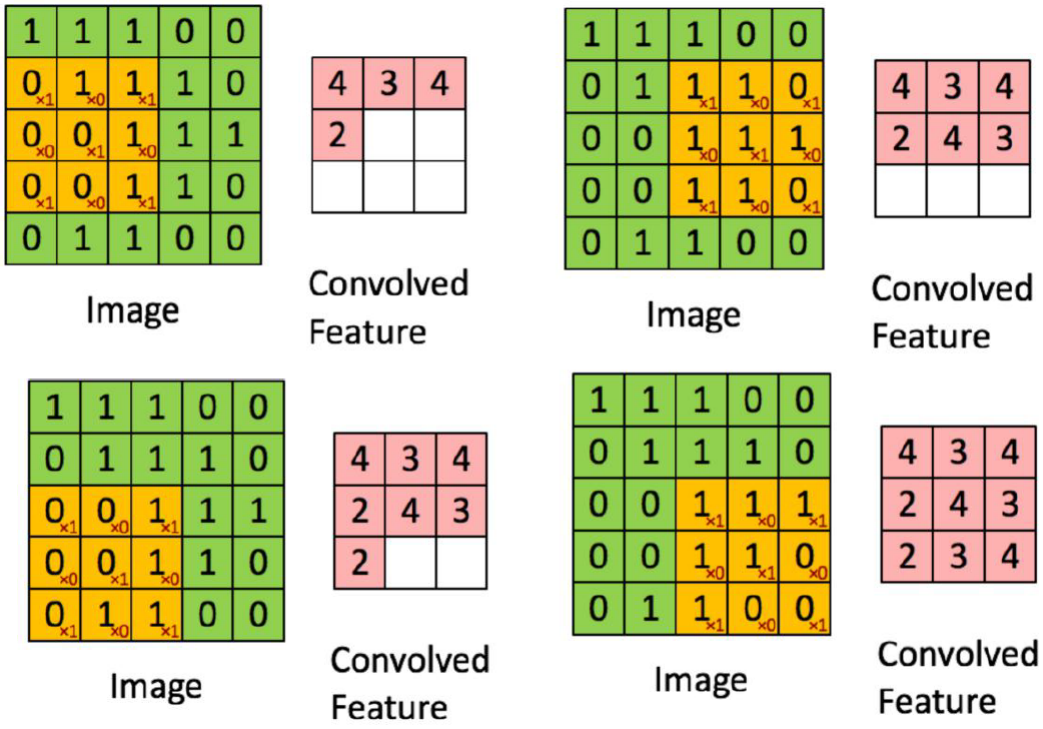
  在二维卷积核中，公式如下：

$$设X为输入，W为卷积核，它们都是2维矩阵：$$

$$s(i,j)=(X*W)(i,j)=\sum_m\sum_n x(i+m,j+n)w(m,n)$$

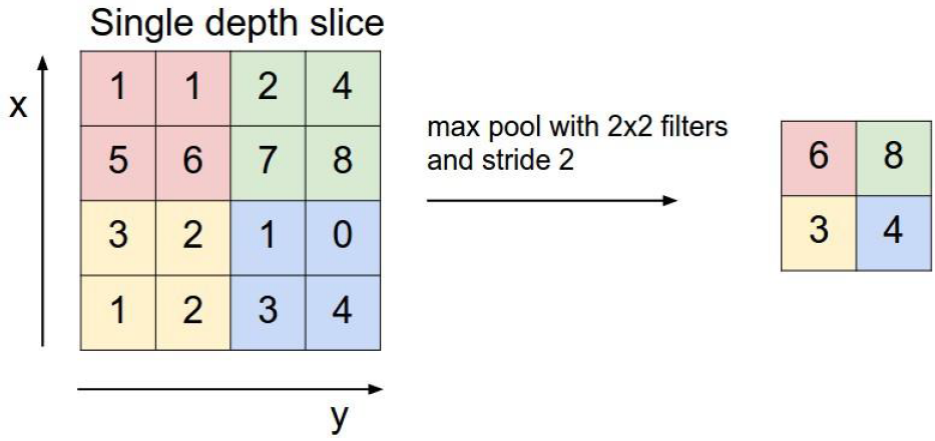
  当需要进行整个图像的特征提取时，可以让卷积核在整个图片上进行滑动，对重叠的地方进行上式的计算。

下面图展示了一个3*3的卷积核计算的部分过程：



- 池化层

  池化层通常用于卷积层后面，可以有效减少数据大小和进一步提取特征，提供模型的鲁棒性。一般来说有最大池化、均值池化和高斯池化等等方法，在这个实验中， 我使用了最大池化方法



- 全连接层

  网络最后的结构就是全连接层，这个部分起到分类器的作用。它可以将前面卷积层和池化层映射到特征隐空间的数据，映射到样本标记空间。在这次实验中，由于需要为0-9进行分类，网络最后的输出是一个10个节点，分别表示当前数据对应每一个标签的概率。

- 激活函数

  在这个网络结构中，使用了RULE这个激活函数，可以给网络提供非线性建模的能力，通常应用在卷积层之后，可以将卷积层线性产生的数据增加一些非线性因素。

损失函数

这里使用了交叉熵这个损失函数，和BPNN情况一样，这里就不赘述了

优化器

优化器用于对模型进行梯度下降，一个好的优化器可以朝着梯度下降的方向不断前进，让模型快速收敛，loss降低得非常迅速，常见优化器有SGD算法、SGD Momentum算法和Adam算法

- SGD算法

  SGD算法每次权重的更新都只利用数据集上的一个样本来进行，具有收敛速度快的特点，同时也会出现陷入局部最优解的问题，也会因为病态曲率的问题造成慢收敛。

  公式如下：

$$\mathbf{w}_{t+1} = \mathbf{w} - lr * \bigtriangledown f(\mathbf{w}_t)$$

$$其中 \bigtriangledown f(\mathbf{w}_t)是部分训练数据的损失，lr是学习率$$

- SGD Momentum算法

  这个算法是为了克服SGD算法中陷入局部最优的问题，引入了动量的概念，利用物理中惯性的概率，如果当前收敛效果好，就可以加速收敛；否则就减慢它的步伐。当进入局部最优的时候，由于有了动量的存在，可以借助动量跳出局部最优点。

  计算公式如下：

$$\mathbf{v}_t = \rho \mathbf{v}_{t-1} + \bigtriangledown f(\mathbf{w}_t)$$

$$\mathbf{w}_{t+1} = \mathbf{w} - lr * \mathbf{v}_t$$

$$其中\rho是延迟率$$

  这个方法虽然可以有效解决陷入局部最优的问题，但是病态曲率问题仍然存在。

- Adam算法

  Adam算法是RMSProp算法的修改版，这个方法可以缓解病态曲率的问题。在RMSProg中，计算了微分平方加权平均数来缓解病态曲率的问题。Adam中，结合了RMSProg算法和Momentum算法，引入了m_t来表示之前梯度的移动平均值。

  计算公式如下：

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) \bigtriangledown f(\mathbf{w}_t)$$

$$s_t = \beta_2 * s_{t-1} + (1 - \beta_2)(\bigtriangledown f(\mathbf{w}_t))^2$$

$$w_{t+1} = w_t - lr * m_t \oslash \sqrt{s_t}$$

2.2.2 主要流程及功能说明

数据预处理

由于读到的图像数据是将原来2维图像展开成一维了，而CNN中可以提取出2维图像的局部特征，因此需要将数据重新构建成2维图像

```
#对1维图像还原成2维图像
train_images=train_images.reshape(-1,1,28,28)
test_images=test_images.reshape(-1,1,28,28)
```

在实验中，使用mini-batch的方法，因此需要对数据进行分批操作

同时，为了避免原来数据分别不均匀，导致分批梯度下降造成误差较大的问题，需要对原来的训练集做洗牌操作

```
train_dataset=TensorDataset(torch.tensor(train_images),torch.tensor(train_labels
))
test_dataset=TensorDataset(torch.tensor(test_images),torch.tensor(test_labels))
train_loader=DataLoader(dataset=train_dataset,
batch_size=batch_size,shuffle=True)
test_loader=DataLoader(dataset=test_dataset, batch_size=batch_size,shuffle=True)
```

网络结构

```
def __init__(self):
    super(Net,self).__init__()
    #输入[batchsize，1,28,28]
    self.conv1=torch.nn.Conv2d(in_channels=1,out_channels=6,kernel_size=5)
    self.pooling1=torch.nn.MaxPool2d(kernel_size=2,stride=2)
    self.relu1=torch.nn.ReLU()
    self.conv2=torch.nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5)
    self.pooling2=torch.nn.MaxPool2d(kernel_size=2,stride=2)
    self.relu2=torch.nn.ReLU()

    self.linear1=torch.nn.Linear(256,120)
    self.relu3=torch.nn.ReLU()
    self.linear2=torch.nn.Linear(120,84)
    self.relu4=torch.nn.ReLU()
    self.linear3=torch.nn.Linear(84,10)
```

```
def forward(self,x):
        # x batchsize,1,28,28
    out=self.conv1(x) # out batchsize, 6,24,24
    out=self.relu1(out)
    out=self.pooling1(out) # batchsize,6,12,12

    out2=self.conv2(out) # batchsize,16,8,8
    out2=self.relu2(out2)
    out2=self.pooling2(out2) #batchsize,16,4,4

    out3=out2.view(out2.shape[0],-1)
    out4=self.linear1(out3)
    out4=self.relu3(out4)
    out4=self.linear2(out4)
    out4=self.relu4(out4)
    result=self.linear3(out4)

    return result
```

训练过程

```
def train():
    train_loss = 0
    train_acc = 0

    for item in (train_loader):
        data, label = item[0].float().to(device), item[1].to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output,label.long()).sum()
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
        train_acc += (torch.argmax(output, dim=1) == label).sum().item()
    return train_loss / train_images.shape[0], train_acc / train_images.shape[0]
```

三、结果分析

> 交代实验环境，算法设计设计的参数说明；
>
> 结果（图或表格），比如在若干次运行后所得的最好解，最差解，平均值，标准差。
>
> 分析算法的性能，包括解的精度，算法的速度，或者与其他算法的对比分析。
>
> 算法的优缺点；本实验的不足之处，进一步改进的设想。

3.1 实验环境

- 操作系统：Windows 10
- 编程语言：Python 3.8
- 本地IDE： pycharm / Jupyter notebook

3.2 BPNN实验结果

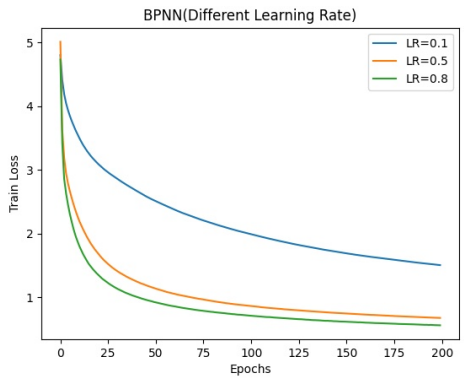
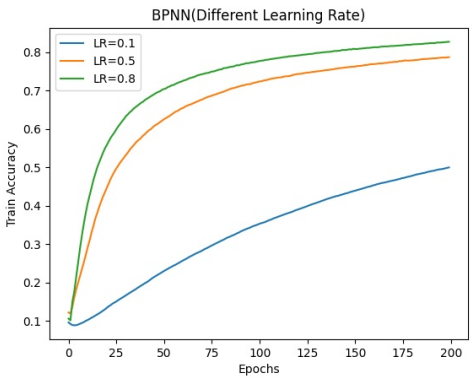
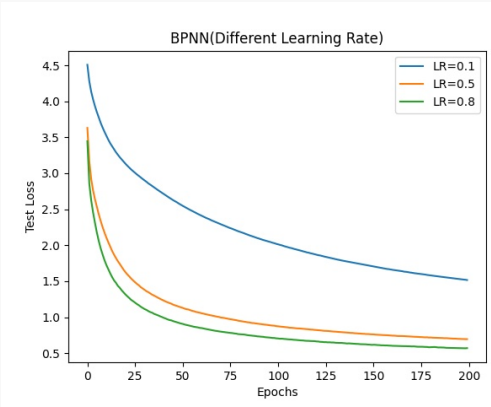
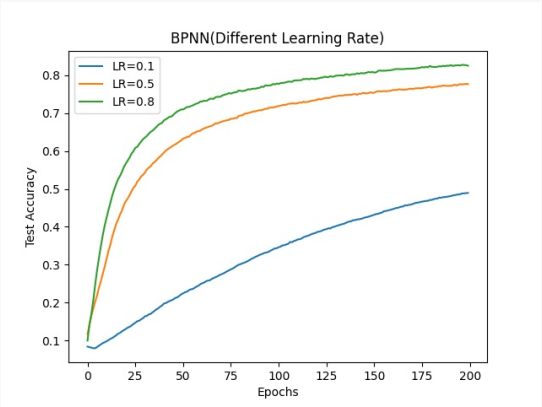
以下对学习率、激活函数、隐藏层节点数、batch_size大小、参数初始化方法进行调参。

3.2.1 学习率

参数

LR=`0.1/0,5/0.8`，激活函数为`sigmoid`，隐藏层节点数为`50`，batch_size大小为`60000`（不进行mini-batch操作），参数初始化方法为random（choice=1）。

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

- **平均运行时间**：190.05s；
- **分析**：在上述参数的设置下，在迭代次数=200时，学习率的设置**对收敛速度和收敛效果影响非常大。**训练集和测试集的`loss`都是整体先下降后趋于平稳的，且$LR=0.5/0.8$的曲线下降速度明显高于$LR=0.1$的情况。其中，最优参数为$LR=0.8$，最终趋于收敛时`test_loss=0.571`，`test_acc=0.8197`；最差参数为$LR=0.1$，最终并没有趋于收敛，`test_loss=1.516`，`test_acc=0.489`，明显差于前者。

  这说明**学习率的设置**在$BPNN$中是十分重要的，只有设置合适的学习率才会有较好的结果。

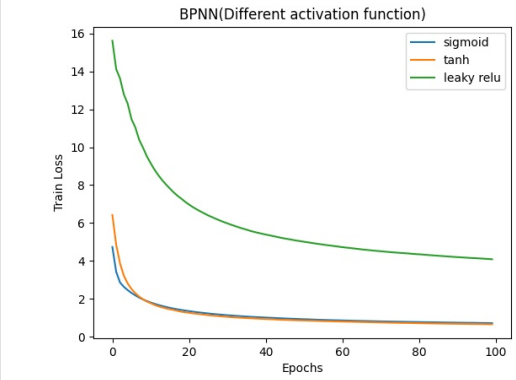
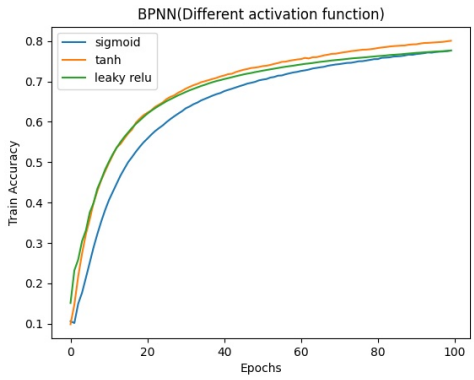
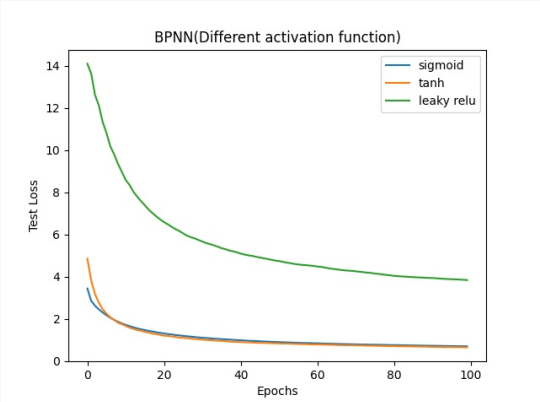
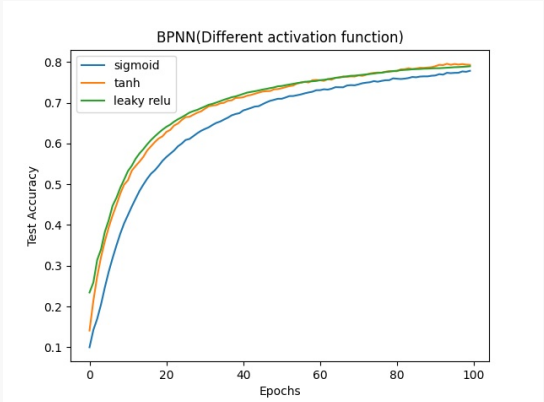
3.2.2 激活函数

参数

以下取各激活函数对应最好的学习率参数，以比较各激活函数的效果。其他参数相同，分别为：隐藏层节点数为`50`，batch_size大小为`60000`（不进行mini-batch操作），参数初始化方法为random（choice=1）。

- 激活函数为`sigmoid`，LR=`0.8`；
- 激活函数为`tanh`，LR=`0.8`；
- 激活函数为`Leaky Relu`，LR=`0.0005`。

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

- **平均运行时间**：99.86s；
- **分析**：在上述参数的设置下，在迭代次数=100时，激活函数种类**对收敛速度和收敛效果较大。**训练集和测试集的`loss`都是整体先下降后趋于平稳的，且对于$tanh$和$sigmoid$来说，两者的学习率相同，最终收敛速度相近，$tanh$的收敛效果要略微优于$sigmoid$（略高3%）。而由$leaky\ relu$的特性可知其`loss`的量级与前两者不同，故`loss`不具有可比性；而`test_accuracy`最终接近于$tanh$的收敛效果。

  因此，综上所述，在迭代次数为100时，最优参数为$tanh$，最终趋于收敛时`test_loss=0.661`，`test_acc=0.7927`；最差参数为$sigmoid$，最终趋于收敛时，`test_loss=0.706`，`test_acc=0.7782`。（但实际两者的差距十分小）

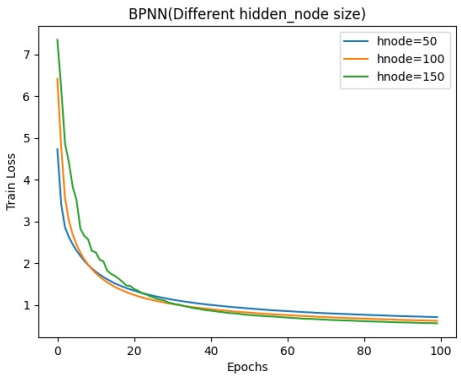
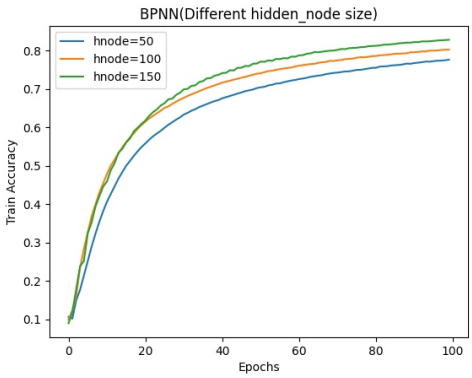
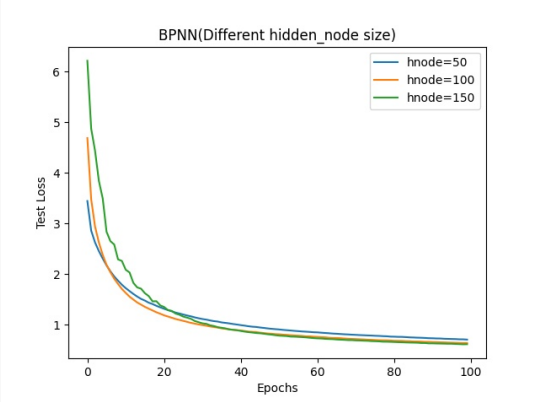
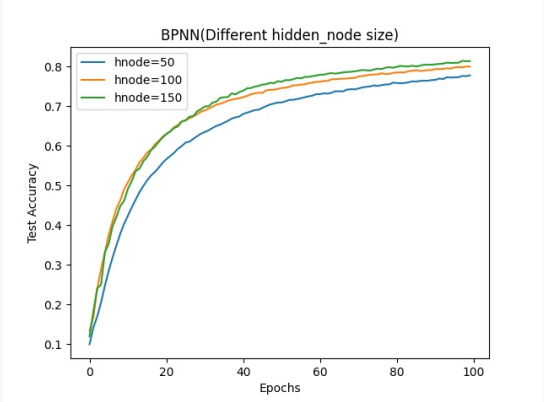
  这说明**激活函数的选择**在$BPNN$中影响较小，只要设置好对应合适的学习率，就会有较好的结果。

3.2.3 隐藏层节点数

参数

LR=`0.8`，激活函数为`sigmoid`，隐藏层节点数为`50/100/150`，batch_size大小为`60000`（不进行mini-batch操作），参数初始化方法为random（choice=1）。

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

- **平均运行时间**：131.68s（时间随着隐藏层节点数增大而明显增大）
- **分析**：在上述参数的设置下，在迭代次数=100时，隐藏层节点数的设置**对收敛速度和收敛效果影响较大。**训练集和测试集的`loss`都是整体先下降后趋于平稳的，且三个隐藏层节点数对应的收敛速度相同。而收敛效果是`hnode`= 150 > `hnode`=100 > `hnode`=50，即随着隐藏层节点的增加，`test_accuracy`增加。其中，最优参数为$hnode = 150$，最终趋于收敛时`test_loss=0.615`，`test_acc=0.8141`；最差参数为$hnode = 50$，最终趋于收敛时`test_loss=0.706`，`test_acc=0.7782`。

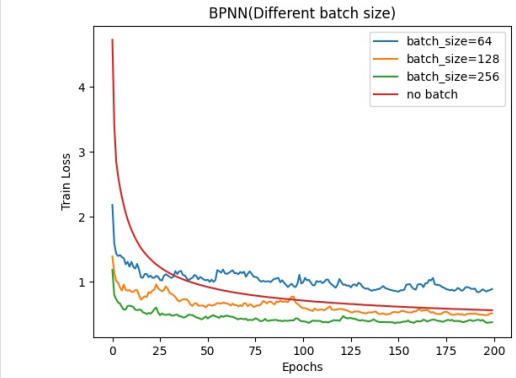
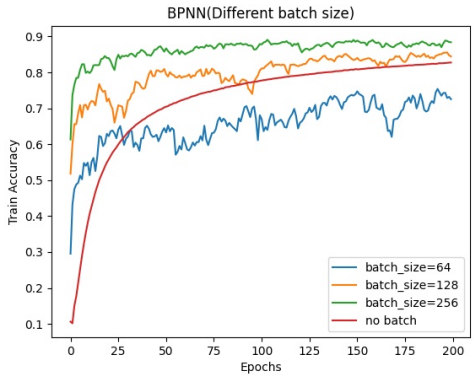
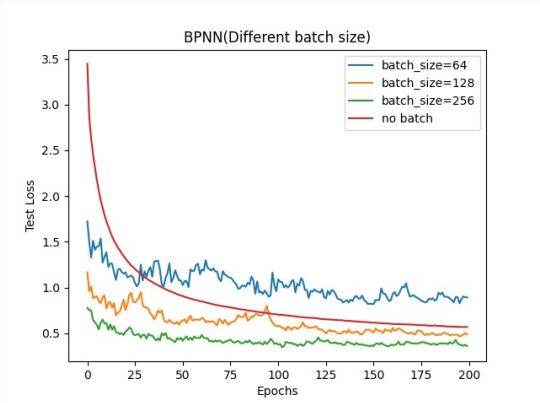
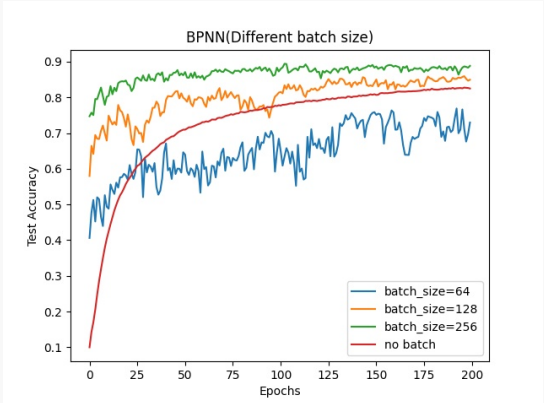
  这说明**隐藏层节点数的设置**在$BPNN$中影响较小，随着隐藏层节点数的增加，测试集准确率相应小幅度增加，但运行时间也相应增加。

3.2.4 batch_size大小

参数

LR=`0.8`，激活函数为`sigmoid`，隐藏层节点数为`50`，batch_size大小为`64/128/256/60000`（60000表示不进行$mini-batch$操作），参数初始化方法为random（choice=1）。

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

- **平均运行时间**：175.12s
- **分析**：在上述参数的设置下，在迭代次数=200时，batch_size的设置**对收敛速度和收敛效果影响非常大。**训练集和测试集的loss都是整体先抖动下降后趋于平稳的，且由于使用了梯度下降，收敛速度非常快；且`batch_size`越大，收敛速度越快（原因在于`batch_size`小的话抖动较大）。而收敛效果是`batch_size`=256 > `batch_size`=128 > `batch_size`=64，即随着batch_size节点的增

加，`test_accuracy`明显增加；且`batch_size`=128/256对应的`test_accuracy`要大于不采用$mini-batch$的`test_accuracy`。其中，最优参数为`batch_size`=256，最终趋于收敛时`test_loss=0.3612`，`test_acc=0.8884`；最差参数为`batch_size`=64，最终一直抖动并没有趋于收敛，`test_loss=0.892`，`test_acc=0.7298`。

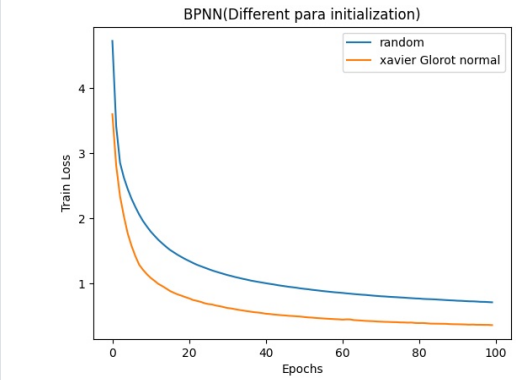
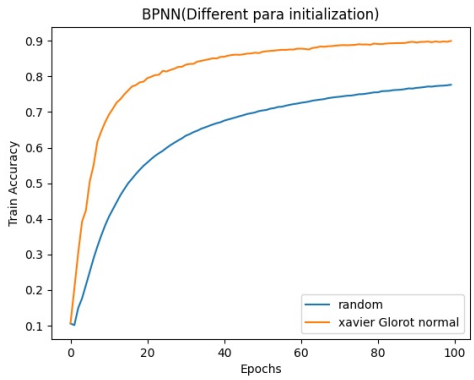
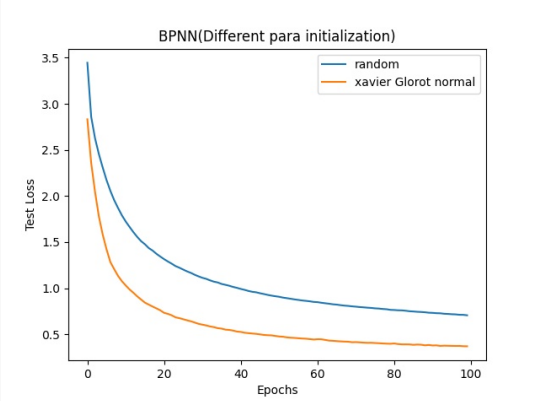
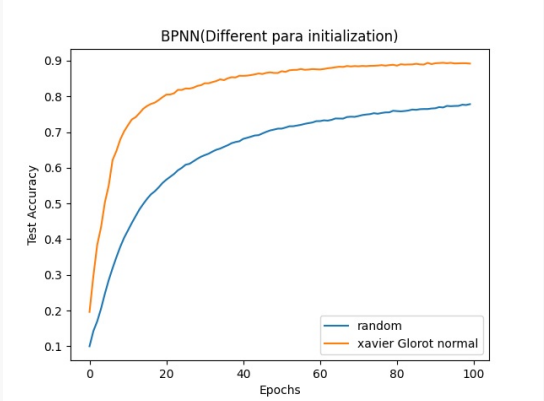
这说明使用$mini-batch$策略时，**batch_size的设置**在$BPNN$中影响较大，随着batch_size的增加，测试集准确率增加，且运行时间相应减小。

3.2.5 参数初始化方法

参数

LR=`0.8`，激活函数为`sigmoid`，隐藏层节点数为`50`，batch_size大小为`60000`（表示不进行$mini-batch$操作），参数初始化方法为`random`/`xavier Glorot normal`（choice=1/2）。

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

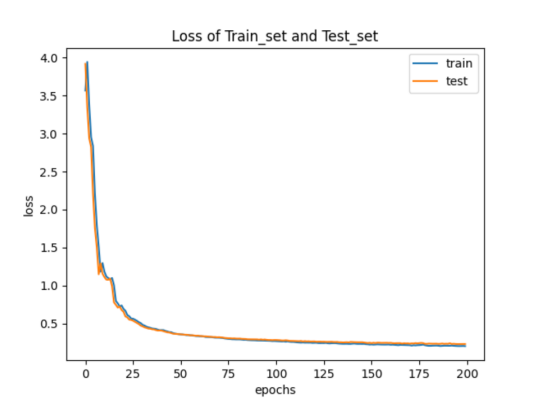
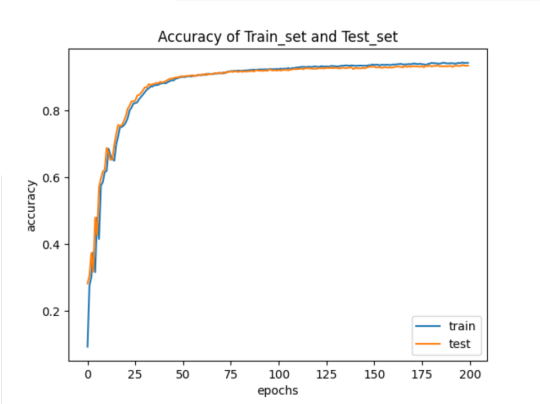
- **平均运行时间**：121.61s
- **分析**：在上述参数的设置下，在迭代次数=100时，参数初始化方法的设置**对收敛速度和收敛效果影响非常大。**训练集和测试集的`loss`都是整体先下降后趋于平稳的，且采用`xavier Glorot normal`的曲线下降速度明显快于`random`的情况。而收敛效果也是前者明显优于后者。因此，最优参数为`xavier Glorot normal`，最终趋于收敛时`test_loss=0.370`，`test_acc=0.8920`；最差参数为`random`，最终趋于收敛时`test_loss=0.706`，`test_acc=0.7782`，明显差于前者。

  这说明**参数初始化方法的设置**在$BPNN$中是十分重要的，只有设置合适的参数初始化方法才会有较好的结果。

3.2.6 比较

下面是多次实验后，迭代200次时，测试集准确率最高对应的参数取值：

激活函数	学习率	隐藏层节点数	batch_size大小	参数初始化方法
sigmoid	0.8	150	60000	xavier Glorot normal

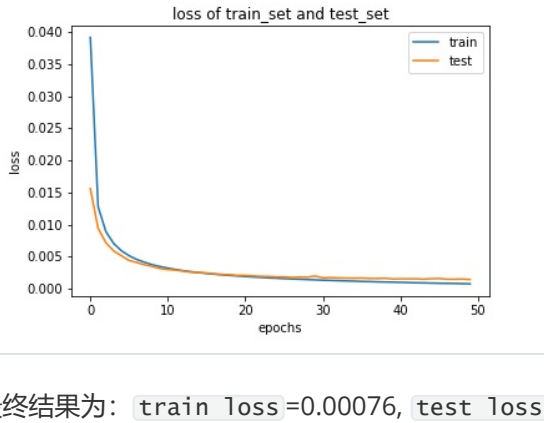
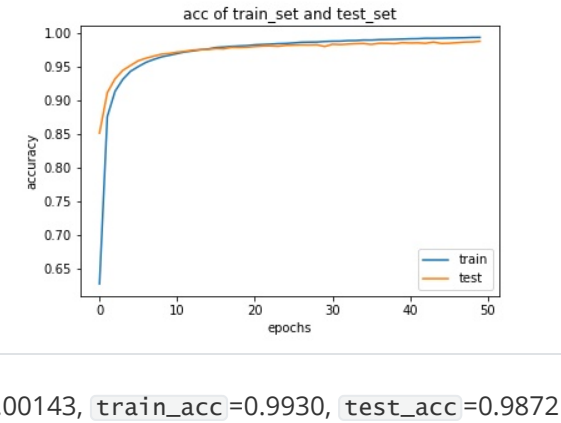
Loss	Accuracy
	

最终，得到`train_loss`=0.2018, `test_loss`=0.2298， `train_acc`=0.9419， `test_acc`=0.934。

3.3 CNN实验结果

3.3.1使用LeNet网络的结果

下面结果是使用标准LeNet网络结构，batch size为32，使用Adam优化器，学习率为0.00001时，结果如下图：

Loss	Accuracy
	

最终结果为：`train loss`=0.00076, `test loss`=0.00143, `train_acc`=0.9930, `test_acc`=0.9872

从上面的结果可以看到，使用LeNet网络结构可以非常有效解决数字图片识别问题。从图上可以看到，在训练初期（epoch=1到5），训练集和测试集在正确率和loss上变化非常快，而且正确率很快就达到95%以上，这个效果是非常惊人的。而随着训练的增加，loss和正确率的变化变得缓慢，很多就达到收敛的位置，最高正确率在测试集上甚至可以达到0.98，这说明这个网络结构非常理想。LeNet网络结构，非常适合本次实验中的问题，具有收敛快，正确率高的优点。

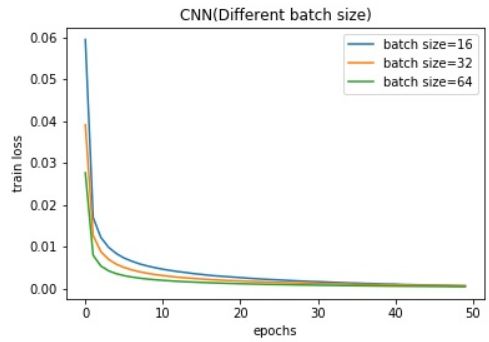
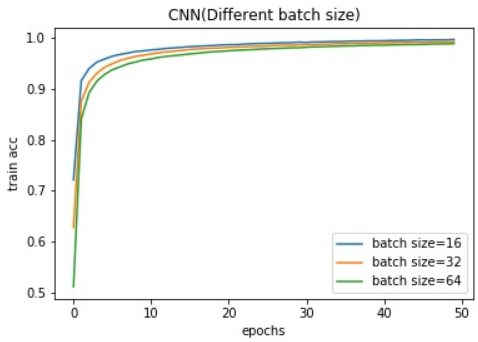
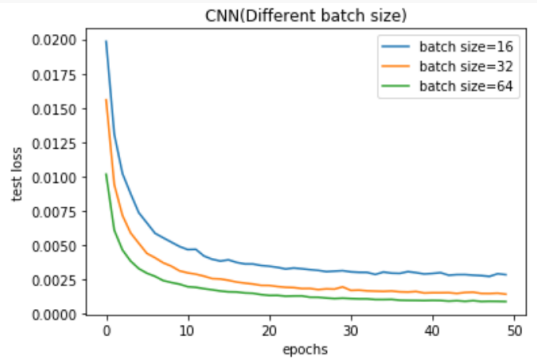
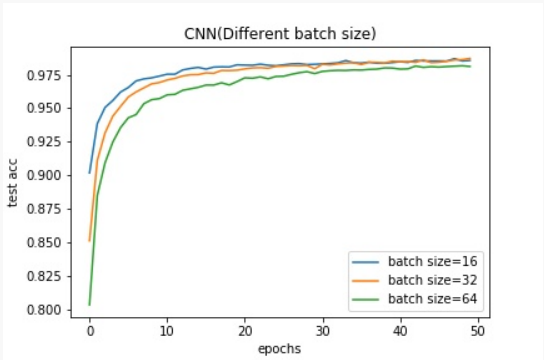
为了比较实验中各种参数对结果的影响，下面以标准LeNet网络为基准，对训练参数或者网络结构进行调整，来分析出不同参数对效果的影响。

3.3.2 batch size大小

参数

这个部分对比了batch size=16、32、64时网络的效果。其他参数与基准设置相同。

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

从上面的结果可以看到，batch size对**收敛速度和收敛效果影响不大。**从上面的图看到，总体来说，batch size=16的效果比较好，batch size=32次之，batch size=64再次之，表现为batch size=16时，收敛速度比较快，最后的正确率也是最高的。但实际上，不同batch size的取值，对应效果的不同是非常小的，它们在训练次数比较多时，最终的正确率几乎是相同的，因此batch size对模型的影响不大。

3.3.3 不同优化器

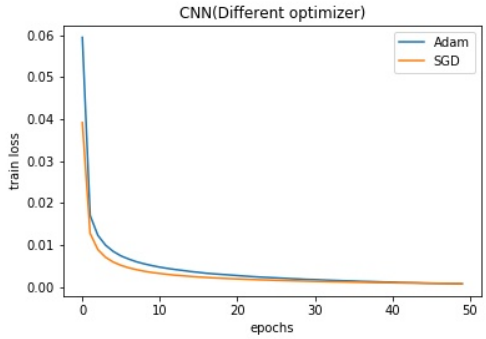
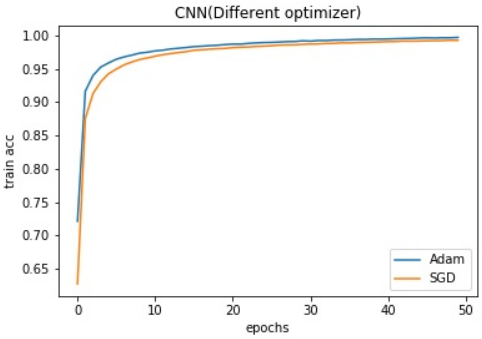
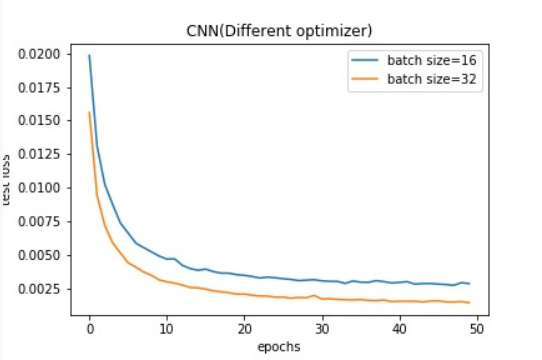
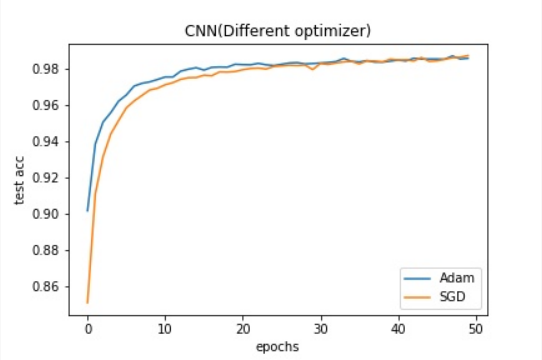
参数

这个部分对比了Adam优化器和带动量的SGD优化器对模型效果的影响。除了优化器不同之外，其他参数与基准设置相同。

Adam优化器的参数：学习率为0.00001

带动量的SGD优化器的参数：学习率为0.0001，weight_decay=1e-5，momentum=0.9

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

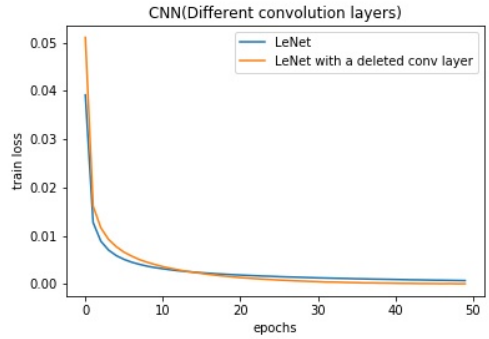
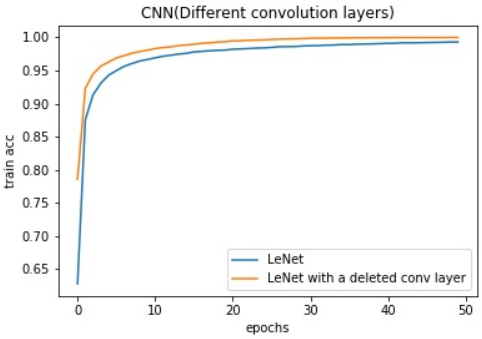
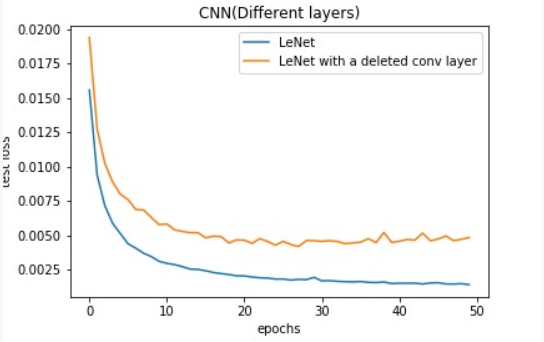
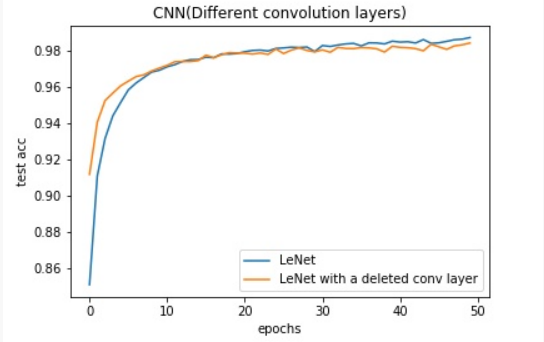
从上面的结果可以看到，优化器对**收敛速度和收敛效果影响不大。**从结果上说，使用Adam优化器整体效果都是要优于使用SGD优化器的，表现收敛速度更快，最后的正确率也更高上。但实际上，虽然两种优化器的效果不同，但是相差非常小，并且随着迭代次数的增加，在训练集和测试集上的正确率几乎是相同的，而且都是效果很好的。这里Adam算法更好，是因为Adam算法引入了微分平方加权平均数和动量来使得迭代朝着loss更小的方向进行。

3.3.4 网络层数

参数

这个部分对比网络层数的不同对数据的影响，分别是标准的LeNet结构和删去第二层卷积层结构进行对比。其他参数与基准设置相同。

实验结果

Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

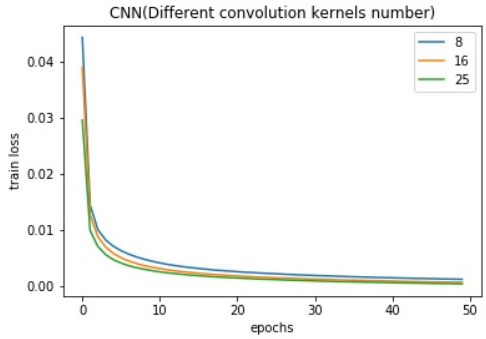
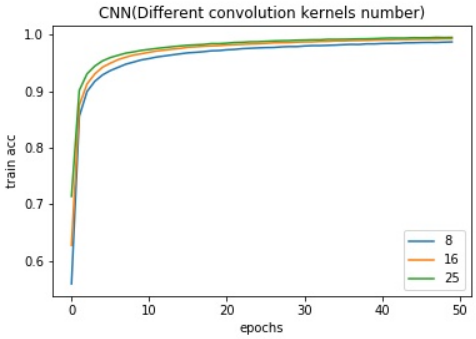
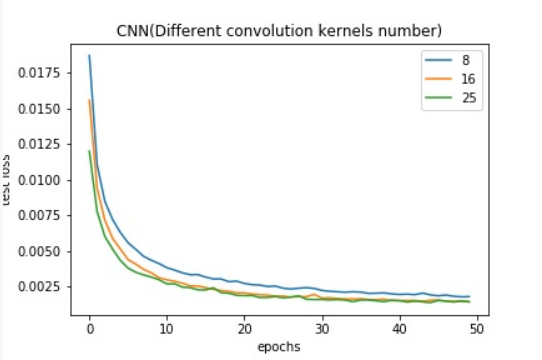
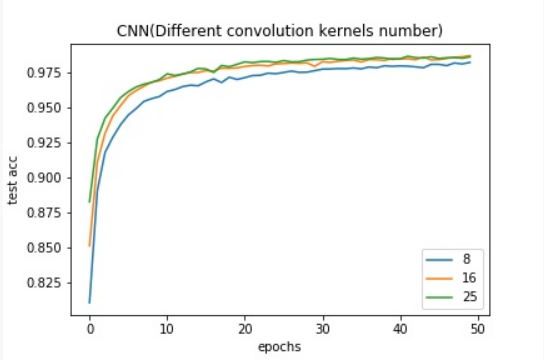
从上面的图上可以看到，虽然一层卷积层和两层卷积层在正确率的指标上相差不大，甚至一层卷积层比两层效果更好，但综合观察loss和acc的指标，可以看到用一层卷积层出现了过拟合的现象。在测试集上，一层卷积层在loss的变化上，出现了的loss变大的趋势。这个现象是很不好的，因此卷积层数对**收敛速度和收敛效果影响比较大。**这个主要是因为，多了一层卷积层，可以提取出更多有效的特征，不容易过分拟合训练集，同时由于2层卷积层需要训练的参数更多，使得模型在收敛速度上比较慢。

3.3.5 卷积核数量

参数

这个部分为了对比不同卷积核数量对模型效果的影响，在第二层卷积上分别设置了卷积核数量为8、16、25。除了这个参数之外，网络的结构和训练的参数和基准模型一样。

实验结果

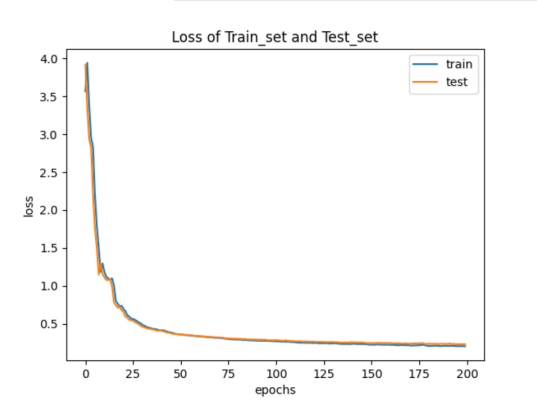
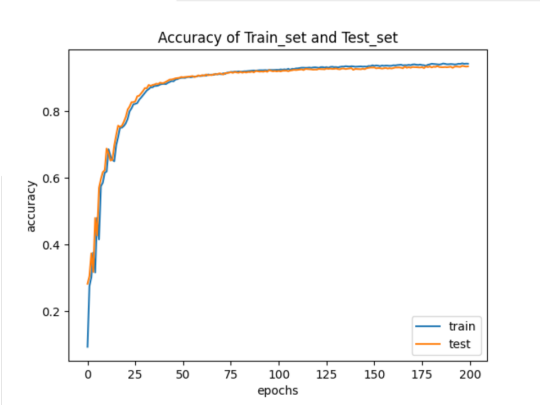
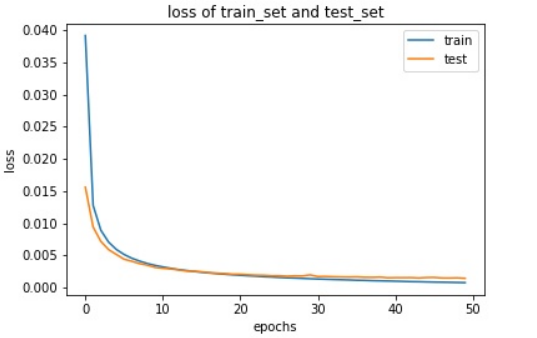
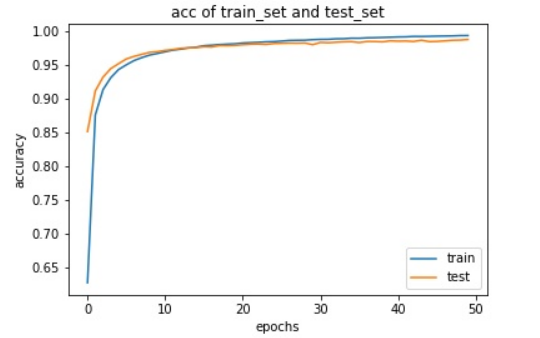
Loss (Train&Test)	Accuracy (Train&Test)
	
	

算法性能

从上面的结果可以看到，卷积核数量**对收敛速度和收敛效果影响不大。**对于不同的卷积核数量，卷积核数量为25时效果最好，次之为卷积核数为16的时候，在次之是卷积核数为8的时候。总的来说，卷积核数量越多，效果越好，特别是卷积核数为8的时候和另外两个情况差别比较大。这个是因为卷积核数量比较多，可以提取出更多的图片特征，更有利于分类。而对于卷积核数为16和25的情况，它们之前差距不明显，可能是因为16个滤波器数量已经可以提取出足够特征了，增加再多的滤波器对性能提升作用不大。

3.4 BPNN和CNN的比较

下面对比BPNN和CNN的效果：

BPNN Loss	BPNN Accuracy
	
CNN loss	**CNN Accuracy**
	

正确率对比：

	BPNN	CNN
训练集	0.9419	0.9930
测试集	0.934	0.9872

通过上面的对比，可以看出，使用CNN进行数字识别的效果明显要更好。在正确率上，CNN高出大概5个百分点；从模型训练过程来看，使用CNN收敛速度更快，BPNN在epoch=50开始变化放缓，到epoch=150的时候效果才比较优，而CNN在epoch=8的时候就开始收敛，在epoch=50的时候就达到比较优的正确率。造成CNN与BPNN效果不同主要在于卷积核的使用：在BPNN中，只有3层结构，依赖线性连接和激活函数对图片特征的提取有限，只能提取出全局的特征。同时，将2维图片展成1维来训练，提取不出2维特征。而CNN中，利用卷积核局部连接的特点，可以有效提取出局部特征，加快收敛速度，并且最终取得更好的效果。

四、结论

这次实验通过对比BPNN网络和CNN网络在手写数据集上识别的正确率，可以知道，由于CNN可以从二维的角度、从图像结构的角度提取出图像的特征，因此CNN网络在识别上可以得到更大的正确率，而且迭代次数很小的时候就可以达到比较优的效果。

五、主要参考文献

[1] 前向型神经网络之BPNN

[2] 神经网络背景知识

[3] 经典CNN之：LeNet介绍*daydayup_668819的博客-CSDN博客*lenet