实验报告

————BP 神经网络和卷积神经网络 CNN

姓名：张子玉　　学号：19335276　　日期：2021.11.27

摘要：

构造一个三层的 BP 神经网络和一个卷积神经网络，完成手写 0-9 数字的识别:

1. 设计网络的结构，比如层数，每层的神经元数，单个神经元的输入输出函数;
2. 根据数字识别的任务，设计网络的输入和输出;
3. 实现 BP 网络的错误反传算法，完成神经网络的训练和测试，最终识别率达到 70%以上;
4. 数字识别训练集可以自己手工制作，也可以网上下载，要求具有可视化图形界面，能够输入输出。
5. 进一步的，用卷积神经网络实现以上任务，对比深度学习与浅层模型。

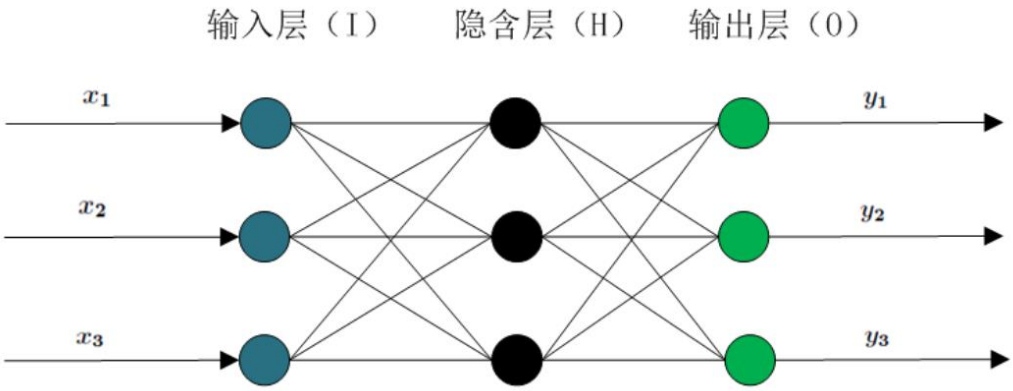
使用 python 编写，下载 mnist 数据集来完成模型的训练。

导言

BP 神经网络

BP（Back-propagation，反向传播）神经网络是最传统的神经网络，它是一种按误差反向传播 (简称误差反传)训练的多层前馈网络。其算法称为 BP 算法 ，它的基本思想是梯度下降法，利用梯度搜索技术，以期使网络的实际输出值和期望输出值的误差均方差为最小。 基本 BP 算法包括信号的前向传播和误差的反向传播两个过程。 即计算误差输出时按从输入到输出的方向进行，而调整权值和阈值则从输出到输入的方向进行。

BP 神经网络分为两个过程：（1）工作信号正向传递子过程；（2）误差信号反向传递子过程在 BP 神经网络中，单个样本有 m 个输入，有 n 个输出，在输入层和输出层之间通常还有若干个隐含层。实际上，1989 年 Robert Hecht-Nielsen 证明了对于任何闭区间内的一个连续函数都可以用一个隐含层的 BP 网络来逼近，这就是万能逼近定理。所以一个三层的 BP 网络就可以完成任意的维到维的映射。即这三层分别是输入层 (I)，隐含层 (H)，输出层 (O)。如下图示



卷积神经网络

卷积神经网络（Convolutional Neural Network, CNN）是一种前馈神经网络，它的人工神经元可以响应一部分覆盖范围内的周围单元，对于大型图像处理有出色表现。
CNN 的基本结构由输入层、卷积层（convolutional layer）、池化层（pooling layer，也称为取样层）、全连接层及输出层构成。卷积层和池化层一般会取若干个，采用卷积层和池化层交替设置，即一个卷积层连接一个池化层，池化层后再连接一个卷积层，依此类推。由于卷积层中输出特征图的每个神经元与其输入进行局部连接，并通过对应的连接权值与局部输入进行加权求和再加上偏置值，得到该神经元输入值，该过程等同于卷积过程，CNN 也由此而得名。

实验过程

mnist 数据集处理

mnist 数据集是机器学习领域中非常经典的一个数据集，由 60000 个训练样本和 10000 个测试样本组成，每个样本都是一张 28 * 28 像素的灰度手写数字图片。

在 MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges 上下载 mnist 数据集：

train-images-idx3-ubyte.gz: training set images (9912422 bytes)
train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

官网对该数据集的格式解释如下：

TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

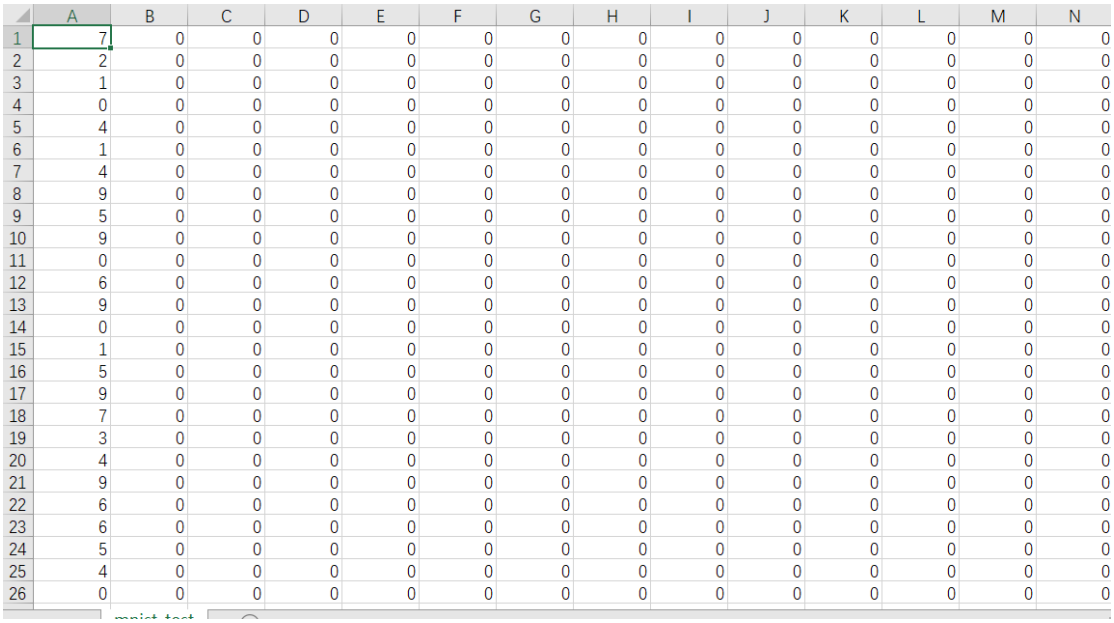
[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

直接下载下来的数据是无法通过解压或者应用程序打开的，因为这些文件不是任何标准的图像格式而是以字节的形式进行存储的，所以这里，我们通过编写程序，将训练集和训练标签集成到一个 csv 文件里（测试文件同）

```
1. def convert(imgf, labelf, outf, n):
2.     f = open(imgf, "rb")
3.     o = open(outf, "w")
4.     l = open(labelf, "rb")
5.
6.     f.read(16)
7.     l.read(8)
8.     images = []
9.
10.    for i in range(n):
11.        image = [ord(l.read(1))]
12.        for j in range(28*28):
13.            image.append(ord(f.read(1)))
14.        images.append(image)
15.
16.    for image in images:
17.        o.write(",".join(str(pix) for pix in image)+"\n")
18.    f.close()
19.    o.close()
20.    l.close()
21.
22.convert("train-images.idx3-ubyte", "train-labels.idx1-
   ubyte","mnist_train.csv", 60000)
23.convert("t10k-images.idx3-ubyte", "t10k-labels.idx1-
   ubyte","mnist_test.csv", 10000)
```

经处理后，得到“mnist_train.csv”和“mnist_test.csv”两个文件，它们的格式如下：



每一行数据为一个样本的标签+其特征向量。标签即为该样本为数字几。

BP

定义类 BP 来实现 BP 神经网络模型:

```
class BP:
    def __init__(self,input_layer,hiden_layer,output_layer,learning_rate):
        self.ilayer = input_layer #输入层
        self.hlayer = hiden_layer #隐藏层
        self.olayer = output_layer #输出层
        self.step = learning_rate #学习步长
        self.input_hidden_weight = np.random.normal(0.0,pow(self.hlayer,-0.5),(self.hlayer,self.ilayer)) #连接输入层和隐藏层的权重矩阵
        self.hidden_output_weight = np.random.normal(0.0,pow(self.olayer,-0.5),(self.olayer,self.hlayer)) #连接隐藏层和输出层的权重矩阵

        self.activation_function = lambda x : scipy.special.expit(x) #激活函数
        def forward(self,input_vector)
        def backward(self,label_vector,output_outputs)
        def update(self,inputs,hidden_outputs,output_outputs,hidden_errors,output_errors)
        def train(self,input_vector,label_vector)
        def predict(self,input_vector)
        def save_model(self)
```

```
17.         def load_model(self)
```

main 函数中，定义网络模型参数，实例化对象：

```
1.      input_node_num = 784 #24 * 24 =784
2.      hidden_node_num = 35 #隐藏层为35 个神经元
3.      output_node_num = 10 #输出 0-9 所对应的概率分布
4.      learning_rate = 0.2 #学习的步长
5.
6.      #创建神经网络
7.      network = BP(input_node_num,hidden_node_num,output_node
   _num,learning_rate)
```

下面分别讲解各个部分的实现

隐含层的选取

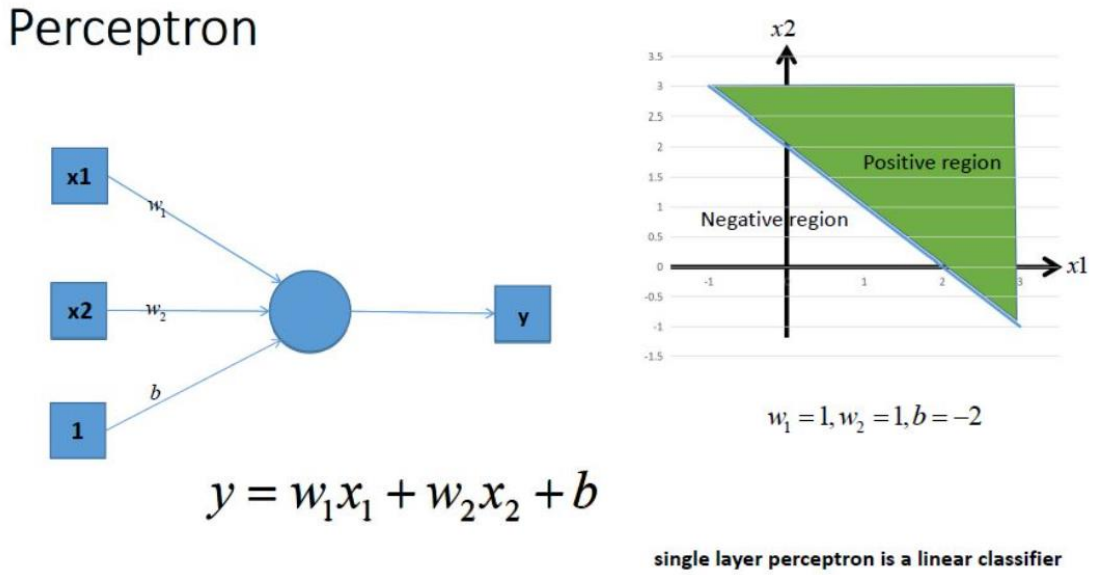
在 BP 神经网络中，输入层和输出层的节点个数都是确定的，而隐含层节点个数不确定。输入层和输出层的节点数是按照输入数据的维度以及输出 label 的种类来确定的。本次实验中，使用的数据集中的数据图像为 28*28 个像素点，即输入层为 784 个节点。输出 0-9，所以输出层为 10 个节点。
隐含层的节点个数对性能有较大影响，我们通过经验公式来确定隐含层的节点数目：

$$h = \sqrt{m+n} + a$$

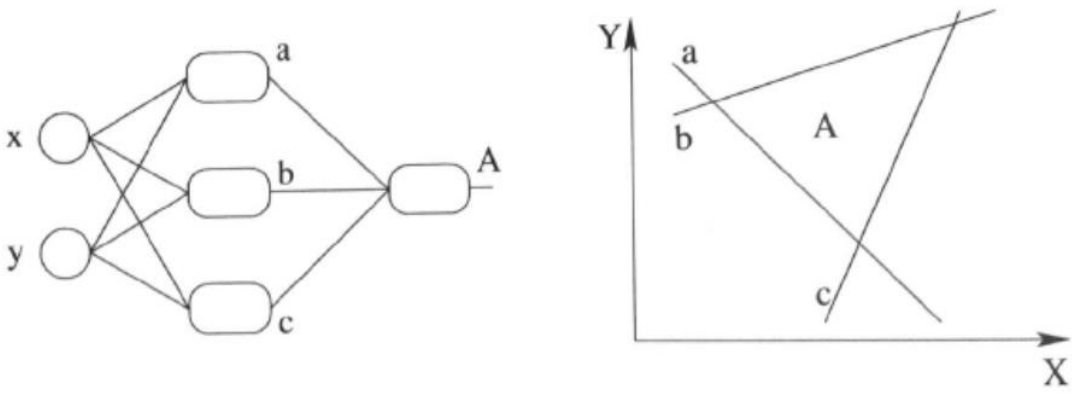
其中 h 为隐含层节点数目，m 为输入层节点数目，n 为输出层节点数目，a 为 1-10 之间的调节常数。
这里我们选取隐含层节点数目为 35。

激活函数

激活函数的主要作用是提供网络的非线性建模能力。如果没有激活函数，那么该网络仅能够表达线性映射，此时即便有再多的隐藏层，其整个网络跟单层神经网络也是等价的。，就像下图：

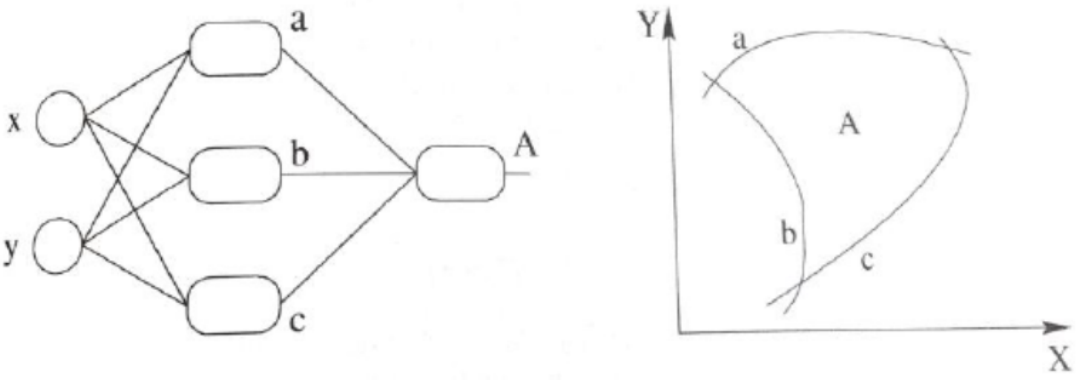


这样的简单神经网络的效果可能如下图：



with step activation function

所以我们引入非线性函数作为激活函数，来提高神经网络的表达能力。
加入激励函数后效果变为下图：



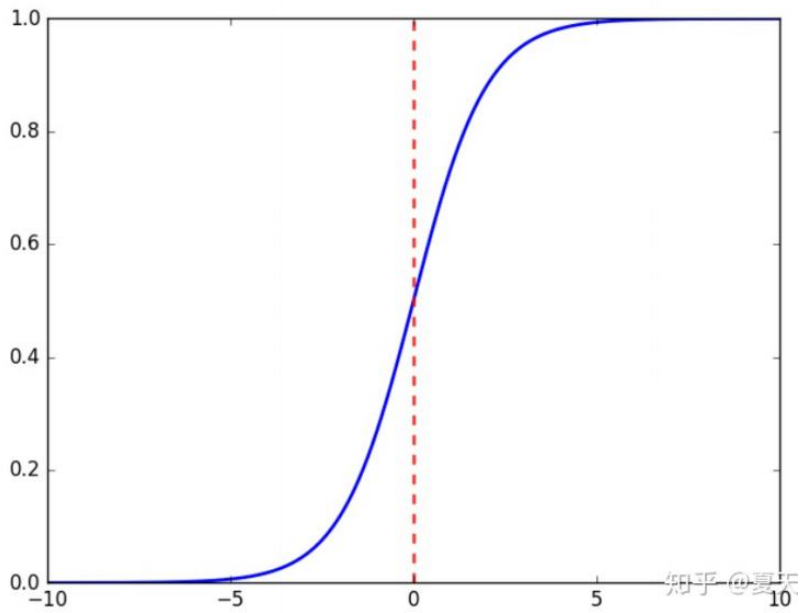
with sigmoid activation function

因此也可以认为，只有加入了激活函数之后，深度神经网络才具备了分层的非线性映射学习能力。

在这次实验中我们用到的激活函数是 sigmod 函数，其公式如下：

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

它的函数曲线如下：



我们直接用 scipy.special.expit(x) 来计算。

```
1. self.activation_function = lambda x : scipy.special.expit(x
   ) #激活函数
```

前向传播

现在设节点 i 和节点 j 之间的权值为 wij，节点的阀值为 bj，每个节点的输出值为 xj，而每个节点的输出值是根据上层所有节点的输出值、当前节点与上一层所有节点的权值和当前节点的阀值还有激活函数来实现的。具体计算方法如下：

$$S_j = \sum_{i=0}^{m-1} w_{ij}x_i + b_j$$

$$x_j = f(S_j)$$

其中 f 为激活函数。
而在 BP 神经网络中，输入层节点没有阈值。

在类中定义下面的函数来实现前向传播的过程：

```
    def forward(self,input_vector):
        #将输入向量形成矩阵作为输入层的输出
        inputs = np.array(input_vector,ndmin=2).T

        #将输入层的结果通过权重矩阵输入到隐藏层并用激活函数计算隐藏层的输出
        hidden_inputs = np.dot(self.input_hidden_weight,inputs)
        hidden_outputs = self.activation_function(hidden_inputs)

        #将隐藏层的输出通过权重矩阵作为输出层的输出并用激活函数计算输出层的输出
        output_inputs = np.dot(self.hidden_output_weight,hidden_outputs)
        output_outputs = self.activation_function(output_inputs)
        return inputs,hidden_outputs,output_outputs
```

使用 np.array 构建输入向量矩阵 inputs
使用 np.dot 计算输入向量矩阵和权重的乘积，得到计算结果 hidden_inputs
将 hidden_inputs 经过激活函数，即得到了隐含层的输出 hidden_outputs
同样的，将隐含层的输出作为输出层的输入，乘上权重矩阵并通过激活函数，获得输出层的输出。

反向传播

反向传播的基本思想是对损失函数求导得到梯度，使用梯度下降法更新参数，使网络的实际输出值和期望输出值的误差均方差为最小。

本次实验中，我们定义一个比较简单的损失函数:

$$l(x, y_{true}) = f(x|w) - y$$

其中 $f(x|w)$ 为神经网络对于输入的输出,则我们的目标函数为:

$$f(w) = min_w \sum_{x \in S} l(x, y_{true})$$

因此整个反向传播的过程为:

```
    def backward(self,label_vector,output_outputs):
        labels = np.array(label_vector,ndmin=2).T
        #计算损失函数的值
        output_errors = labels - output_outputs

```

```
        #反向传播到隐藏层
        hidden_errors = np.dot(self.hidden_output_weight.T, output_errors)
        return output_errors,hidden_errors
```

output_errors 为损失函数的值，即输出层的误差，将其与隐含层的 weight 相乘，获得隐含层的误差 hidden_errors

更新权重

通过反向传播的过程获得了各层误差（损失函数）后，便可更新各层之间的 weight
根据数学推导我们可以得到各层的计算梯度的函数为：

$$\nabla W_{i,j} = E_j O_j (I - O_j) O_j^T$$

其中$W_{i,j}$为连接第 i 层和第 j 层的权重矩阵，E_j为第 j 层的损失，O_j为第 j 层神经元的输出，I 为单位矩阵

我们将这个值乘以步长η便可以拿来更新 weight 了：

$$W_{i,j} = W_{i,j} + \eta \nabla W_{i,j}$$

这里编写 update 函数来实现对 weight 的更新：

```
    def update(self,inputs,hidden_outputs,output_outputs,hidden_errors,output_errors):
        #更新权重矩阵
        self.hidden_output_weight += self.step * np.dot((output_errors * output_outputs * (1.0-output_outputs)),np.transpose(hidden_outputs))
        self.input_hidden_weight += self.step * np.dot((hidden_errors * hidden_outputs * (1.0-hidden_outputs)),np.transpose(inputs))
```

训练

训练即包括上面的 3 个步骤：前向传播、反向传播、更新权重

```
    def train(self,input_vector,label_vector):
        #训练过程：前向传播、反向传播、更新权重
        inputs,hidden_outputs,output_outputs = self.forward(input_vector)
        output_errors,hidden_errors = self.backward(label_vector,output_outputs)
        self.update(inputs,hidden_outputs,output_outputs,hidden_errors,output_errors)
```

预测

预测函数中，只需要做前向传播，最终输出层的输出为一个向量，表示样本 x 为每一类的概率，所以仅需要利用 np.argmax 函数求出概率值最大的下标即可，下标 0-9 正好对应数字的值。
这里编写 predict 函数来实现对 weight 的更新：

```
    def predict(self,input_vector):
        input = np.array(input_vector,ndmin=2).T
        hidden_inputs = np.dot(self.input_hidden_weight,input) #得到隐藏层的输入
        hidden_outputs = self.activation_function(hidden_inputs) #得到隐藏层的输出
        output_inputs = np.dot(self.hidden_output_weight,hidden_outputs) #得到输出层的输入
        output_outputs = self.activation_function(output_inputs) #得到输出层的输出

        return np.argmax(output_outputs)
```

输入要预测的图像向量，输出其类别。

保存/加载模型

```
    def save_model(self):
        np.save("./BP_model/i_h_weight.npy",self.input_hidden_weight)
        np.save("./BP_model/h_o_weight.npy",self.hidden_output_weight)

    def load_model(self):
        self.input_hidden_weight = np.load("./BP_model/i_h_weight.npy")
        self.hidden_output_weight = np.load("./BP_model/h_o_weight.npy")
```

注，命名中，“i”代表 input，“h”代表 hidden，“o”代表 output，“i_h_weight.npy”即为输入层和隐含层之间的权重，“h_o_weight.npy”即为隐含层和输出层之间的权重

训练过程和预测

下面，我们通过调用类中的函数，来实现训练和预测。

编写 start_train 函数，将训练集文件内的向量读入，处理为矩阵后输入网络中的训练函数进行多轮训练（这里，epoch 设置为 5）；编写 start_test 函数，将测试集文件内的向量读入，处理为矩阵后输入网络中的预测函数进行预测：

```
def start_train(epochs,network):
```

```
    #读入训练集
    train_file = open("../dataset/mnist_train.csv","r")
    train_vectors = train_file.readlines()
    train_file.close()

    #进行训练
    print("start train...")
    for i in range(epochs):
        print("now is epoch %d" % (i))
        for sample in train_vectors:
            label_eigen = sample.split(',')
            inputs = (np.asfarray(label_eigen[1:])/255.0 * 0.99) + 0.01
            labels = np.zeros(output_node_num) + 0.01
            labels[int(label_eigen[0])] = 0.99
            network.train(inputs,labels)

    network.save_model()

def start_test(network):
    network.load_model()
    #读入测试集
    test_file = open("../dataset/mnist_test.csv","r")
    test_vectors = test_file.readlines()
    test_file.close()

    print("start test...")
    #测试准确率
    scoreboard = []
    for sample in test_vectors:
        label_eigen = sample.split(',')
        expected_value = int(label_eigen[0])
        image_array = np.asfarray(label_eigen[1:]).reshape((28,28))
        plt.imshow(image_array,cmap='Greys',interpolation='None')
        inputs = (np.asfarray(label_eigen[1:])/255.0 * 0.99) + 0.01
        label = network.predict(inputs)

        if (label == expected_value):
            scoreboard.append(1)
        else:
            scoreboard.append(0)
```

```
42.
43.     scoreboard_array = np.asarray(scoreboard)
44.     print("accuracy = ",scoreboard_array.sum() / scoreboard
    _array.size)
```

CNN

定义类 CNN 来实现 CNN 模型。
CNN 可以将复杂的问题简单化，把大量的参数降维成少量的参数再做处理，所以比起 BP，训练起来时间会少很多。
CNN 主要包括 5 层：输入层、卷积层、池化层、全连接层
其中，卷积层和池化层做的工作便是“将复杂的问题简单化”

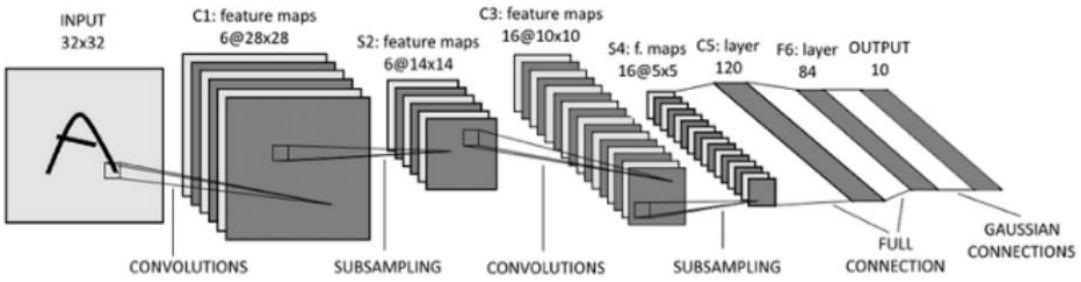
我们使用tensorflow 来搭建 cnn网络：

```
1.      def __init__(self):
2.          model = models.Sequential()
3.          model.add(layers.Conv2D(64, (3, 3), activation='rel
    u', input_shape=(28, 28, 1)))
4.          model.add(layers.MaxPooling2D((2, 2)))
5.          model.add(layers.Conv2D(64, (3, 3), activation='rel
    u'))
6.          model.add(layers.MaxPooling2D((2, 2)))
7.          model.add(layers.Conv2D(64, (3, 3), activation='rel
    u'))
8.          model.add(layers.Flatten())
9.          model.add(layers.Dense(64, activation='relu'))
10.         model.add(layers.Dense(10, activation='softmax'))
11.         model.summary()
12.         self.model = model
```

搭建出来的网络结果如下图所示：

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 64)                36928
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 112,074
Trainable params: 112,074
Non-trainable params: 0
```

该网络结构参考的是 LeNet-5 的结构，如下图所示：



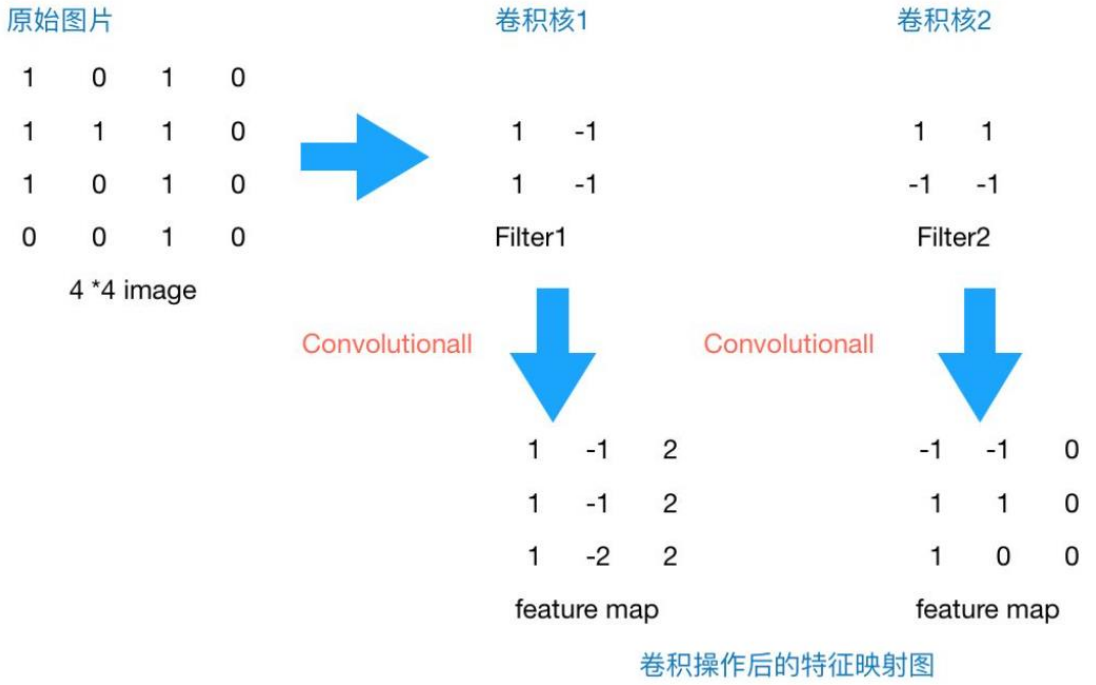
这个网络一共有 3 个卷积层，2 个池化层，1 个扁平层和 2 个全连接层。

下面我们就每一个层进行介绍。

卷积层

用传统的三层神经网络需要大量的参数，原因在于每个神经元都和相邻层的神经元相连接，显然，这种全连接层的方式对于图像数据不太友好，因为图像本身具有“二维空间特征”，通俗点说就是局部特性。一般来说，我们只需要看到图片的某些细节就能识别出来它。而卷积层，就是用于提取特征。因为一次卷积可能提取的特征比较粗糙，所以这里我们设置了 3 层卷积层，层层提取特征。
卷积操作是这样进行的：
卷积核为一个 nxn 的矩阵，用它去扫描输入二维矩阵（因为这里是图片），每次扫描时，将覆盖到的位置跟卷积核对应位置的元素做乘积，最后求和，获得新的元素，这样便将这块区域的特征提取出来了。

下面是 4*4 图像和 2*2 的两个卷积核做卷积操作的例子：



卷积操作后的特征映射图

由于同一层的神经元可以共享卷积核，所以对于高位数据的处理将会变得非常简单。使用卷积核后图片的尺寸变小，方便后续计算，并且我们不需要手动去选取特征，只用设计好卷积核的尺寸，数量和滑动的步长就可以让它自己去训练了。

这里设置卷积层的 filters=64,卷积核大小为（3，3），使用激活函数 ReLU。

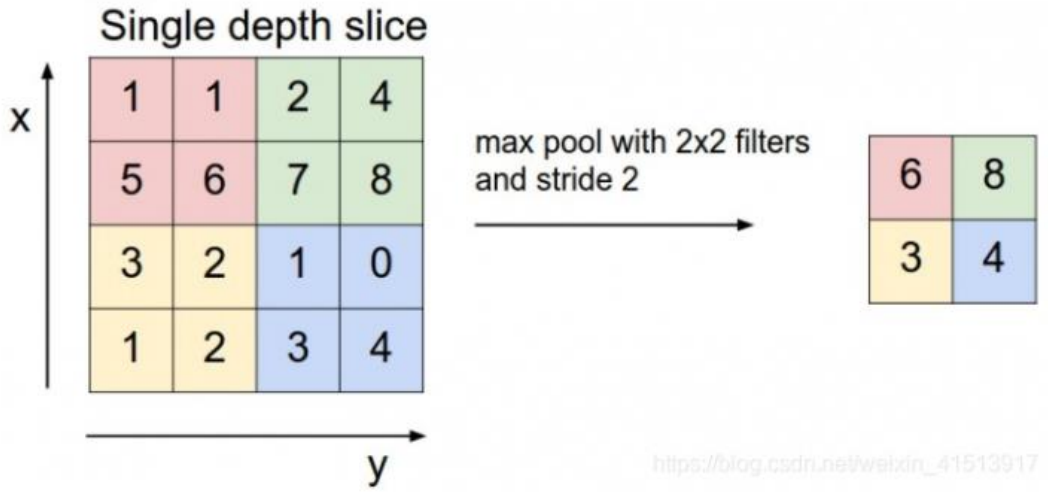
```
1. model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shap
   e=(28, 28, 1)))
```

池化层

池化层的主要目的是通过降采样的方式，在不影响图像质量的情况下，压缩图片，减少参数。

通常来说，池化方法一般有两种：
MaxPooling：取滑动窗口里最大的值
AveragePooling：取滑动窗口内所有值的平均值

这里我们选用 MaxPooling 的方法，因为最大池化可以提取特征纹理，提取边缘等“最重要”的特征，符合我们本次任务的特性，我们只要识别出值数大的地方（黑像素点）部分即可，不关心全局是怎么样的。
MaxPooling 的操作如下图所示：整个图片被不重叠的分割成若干个同样大小的小块（pooling size）。每个小块内只取最大的数字，再舍弃其他节点后，保持原有的平面结构得出 output。



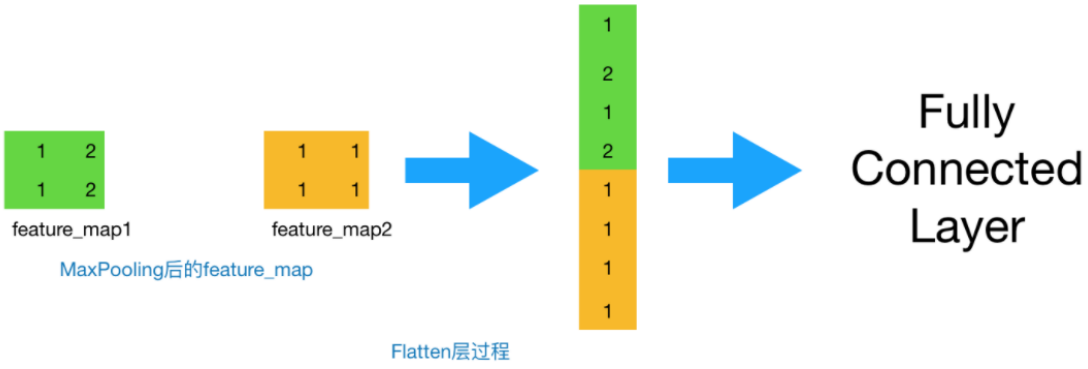
注意，池化作用于图像中不重合的区域（这与卷积操作不同，卷积操作作用于的区域会有重合）

这里，我们设置池化大小为 2*2：

```
1.          model.add(layers.MaxPooling2D((2, 2)))
```

扁平层

做完 Max Pooling 后，我们就会把这些数据“拍平”，丢到 flatten 层。
这是因为传递给全连接层的卷积层的输出必须在全连接层接受输入之前进行 flatten。
flatten 操作是一种特殊类型的 reshape 操作，其中所有的轴都被平滑或压扁在一起。



添加 flatten 层：

```
1.          model.add(layers.Flatten())
```

全连接层

全连接层（fully connected layers，FC）在整个卷积神经网络中起到“分类器”的作用。如果说卷积层、池化层和激活函数层等操作是将原始数据映射到隐层特征空间的话，全连接层则起到将学到的“分布式特征表示”映射到样本标记空间的作用。

这里构建 2 层全连接层，分别使用 relu 激活函数和 softmax 激活函数：

```
1.    model.add(layers.Dense(64, activation='relu'))
2.    model.add(layers.Dense(10, activation='softmax'))
```

relu 激活函数对传入的每一个值大于零则返回本身，小于零则返回零，相当于 y = max(0, x)，可以减轻梯度消失的问题。
因为是多分类问题，所以最后一层使用激活函数 softmax，其输出值是在 0-1 之间且合为 1 的十个数，相当于一个概率分布，分别对应十个输出神经元。

可视化

使用Python 的 PyQT5 库来实现我们的可视化手写数字输入，PyQT5 库可以提供画板等，让我们实现能在线写数字实时进行识别。
GUI 效果如下：

我们可以使用橡皮擦、改变画笔的颜色、改变画笔的粗细，测试效果表明如果画笔更粗的话识别成功率会更高。

结果分析

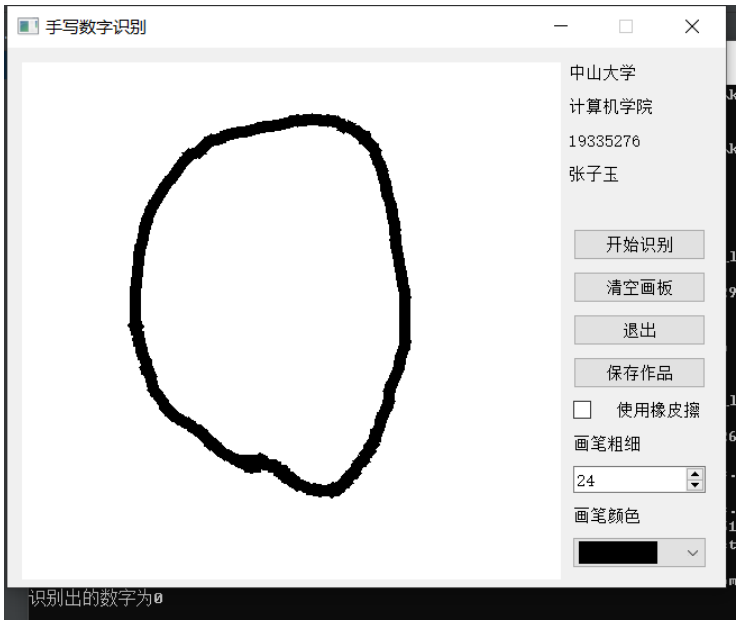
BP

训练并测试:

```
C:\Users\zzy\Desktop\大三上\人工智能\手写数字识别\src>python BP_neural_network.py
start train...
now is epoch 0
now is epoch 1
now is epoch 2
now is epoch 3
now is epoch 4
start test...
Warning: QT_DEVICE_PIXEL_RATIO is deprecated. Instead use:
   QT_AUTO_SCREEN_SCALE_FACTOR to enable platform plugin controlled per-screen factors.
   QT_SCREEN_SCALE_FACTORS to set per-screen DPI.
   QT_SCALE_FACTOR to set the application global scale factor.
accuracy =  0.9391
```

进行 5 轮训练，测试集上准确率有 93.91%。

使用 GUI 手写 0-9 测试:
（这里展示 0-4）




手写数字识别
中山大学
计算机学院
19335276
张子玉
开始识别
清空画板
退出
保存作品
使用橡皮擦
画笔粗细
24
画笔颜色
识别出的数字为1


手写数字识别
中山大学
计算机学院
19335276
张子玉
开始识别
清空画板
退出
保存作品
使用橡皮擦
画笔粗细
24
画笔颜色
识别出的数字为2


手写数字识别
中山大学
计算机学院
19335276
张子玉
开始识别
清空画板
退出
保存作品
使用橡皮擦
画笔粗细
24
画笔颜色
识别出的数字为3

CNN

训练并测试：

```
Epoch 1/5
1875/1875 [==============================] - 51s 27ms/step - loss: 0.1410 - accuracy: 0.9560
Epoch 2/5
1875/1875 [==============================] - 47s 25ms/step - loss: 0.0433 - accuracy: 0.9864
Epoch 3/5
1875/1875 [==============================] - 47s 25ms/step - loss: 0.0321 - accuracy: 0.9900
Epoch 4/5
1875/1875 [==============================] - 47s 25ms/step - loss: 0.0246 - accuracy: 0.9918
Epoch 5/5
1874/1875 [==============================>.] - ETA: 0s - loss: 0.0195 - accuracy: 0.9938
Epoch 00005: saving model to ./cnn_ckpt\cp-0005.ckpt
1875/1875 [==============================] - 47s 25ms/step - loss: 0.0195 - accuracy: 0.9938
start test...
313/313 [==============================] - 3s 9ms/step - loss: 0.0308 - accuracy: 0.9915
accuracy =  0.9915000200271606
```

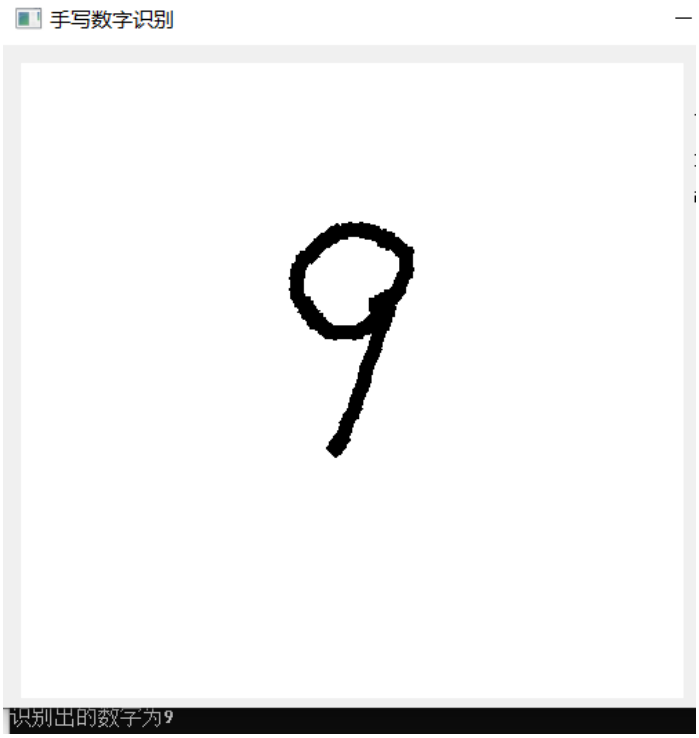
进行 5 轮训练，测试集上准确率有 99.15%。

使用 GUI 手写 0-9 测试：
（这里展示 5-9）


中山大学
计算机学院
19335276
张子玉
开始识别
清空画板
退出
保存作品
使用橡皮擦
画笔粗细
24
画笔颜色
识别出的数字为6


手写数字识别
中山大学
计算机学院
19335276
张子玉
开始识别
清空画板
退出
保存作品
使用橡皮擦
画笔粗细
24
画笔颜色
识别出的数字为7


手写数字识别
中山大学
计算机学院
19335276
张子玉
开始识别
清空画板
退出
保存作品
使用橡皮擦
画笔粗细
10
画笔颜色
识别出的数字为8



注，手写数字识别 GUI，要调高粗细程度至 20 来使用，太细的话，很难识别准确。

两者对比

（两者都进行 5 个 epoch 的训练）

时间上：

BP：花费 15 分钟左右

CNN：花费 5 分钟左右

准确度上：

BP：测试集准确率 93.91%

CNN：测试集准确率 99.15%

可以看出来，CNN 不仅训练耗时少，而且准确率也更高。

下面总结一下两种网络的优缺点。

BP：

优点：

网络结构简单，易于实现

具有神经网络的基本优势，如非线性映射能力、自学习能力、泛化能力

缺点：

由于处理的数据维度大，所以训练耗时长

局部极小化问题，传统的 BP 神经网络的权值是通过沿局部改善的方向逐渐进行调整的，这样会使算法陷入局部极值

容易出现过拟合现象，泛化能力一般

CNN：

优点：

对数据进行了降维，大大缩短了训练时间

泛化能力较强，因为卷积层对图像特征的提取使得全连接层可以学习到更多的信息

缺点：

网络结构较复杂

网络规模更大，对设备有要求

结论

通过本次实验，我了解了 BP 和 CNN 的网络结构，以及各层的作用和如何配置。本次实验中将课上所学的运用到了实际，自己动手构建了网络的各层，考虑了如何去配置各个参数。步骤都是参考着经典的网络结构去做的，做的过程中也学习领悟到了很多，也感受到了神经网络模型的自主学习的强大。同时，了解到了 BP 和 CNN 各自不同的优缺点，浅层网络和深层网络的不同，能够更好的选择适合实验的网络模型。

主要参考文献

(20 条消息) Python 手写数字识别+GUI 界面+手写板设计_鲁棒最小二乘支持向量机-CSDN 博客_python 手写板

MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges