南開大學
Nankai University

计算机学院和网络空间安全学院

机器学习期末报告

基于 Lenet5 的 Mnist 手写体识别算法实现

姓名：魏伯繁

学号：2011395

专业：信息安全

2023 年 1 月 8 日

目录

1 实验要求

在这次大作业中，需要用 Python 实现 LeNet5 来完成对 MNIST 数据集中 0-9 10 个手写数字的分类。代码只能使用 python 实现，不能使用 PyTorch 或 TensorFlow 框架。

2 实验理论

本次实验的理论核心是对 Lenet-5 的正确理解，所以在本实验报告中我会首先介绍 lenet5 的详细内容，并在下个章节详细介绍其代码实现

2.1 lenet5 简介

LeNet-5 由 LeCun 等人提出于 1998 年提出，是一种用于手写体字符识别的非常高效的卷积神经网络。出自论文《Gradient-Based Learning Applied to Document Recognition》

2.2 lenet5 结构

LetNet-5 是一个较简单的卷积神经网络。下图清晰的显示了其结构：输入的二维图像（单通道），先经过两次卷积层到池化层，再经过全连接层，最后为输出层。整体上是：input layer->convulational layer->pooling layer->activation function->convulational layer->pooling layer->activation function->convulational layer->fully connect layer->fully connect layer->output layer.

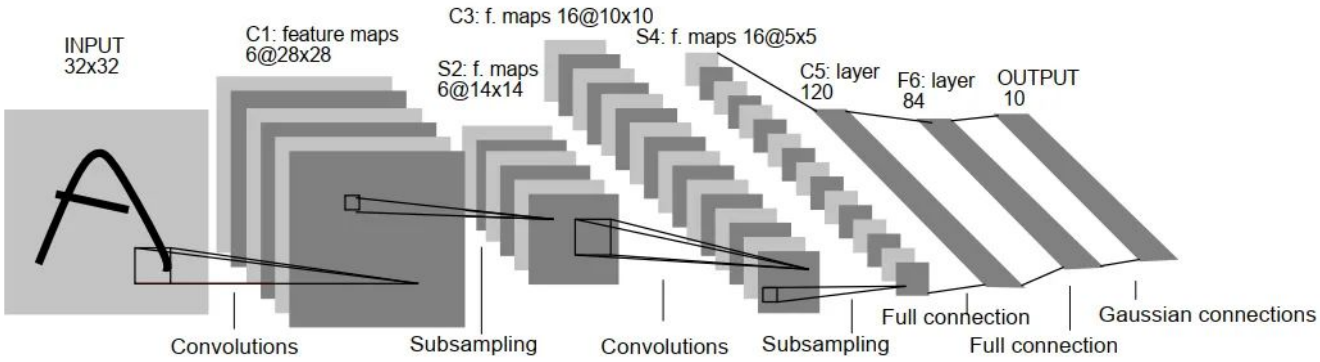

图 2.1: lenet5 结构图

整个 LeNet-5 网络不包含输入层总共包括 7 层，分别是：C1、S2、C3、S4、C5、F6、OUTPUT。

其中对参数的解释为：

层编号特点是英文字母 + 数字，英文字母代表以下一种：C→ 卷积层、S→ 下采样层（池化）、F→ 全连接层，数字代表当前是第几层，而非第几卷积层，参数包括权重 w 与偏置 b，连接数就是连接数连线数参数计算：每个卷积核对应于一个偏置 b，卷积核的大小对应于权重 w 的个数 (特别注意通道数)

2.3 输入层

输入层（INPUT）是 32x32 像素的图像，输入的通道数为 1，也就是说输入的图像是黑白色的图片，不是平常我们看到的三层的彩色图片，同样我们应该注意到，lenet5 默认的输入为 32*32 的大小，

而 mnist 数据集的读取是 28*28 的大小，所以我们在做数据读取后应该对其进行 padding

2.4 C1 层

C1 层是卷积层，使用 6 个 5×5 大小的卷积核，padding=0，stride=1 进行卷积，得到 6 个 28×28 大小的特征图 32-5+1=28

参数个数：(5*5+1)*6=156，其中 5*5 为卷积核的 25 个参数 w，1 为偏置项 b。

连接数：156*28*28=122304，其中 156 为单次卷积过程连线数，28*28 为输出特征层，每一个像素都由前面卷积得到，即总共经历 28*28 次卷积。

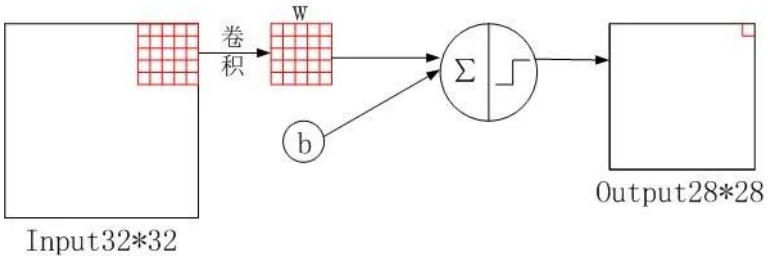

图 2.2: C1 示意图

2.5 S2 层

S2 层是降采样层，使用 6 个 2×2 大小的卷积核进行池化，padding=0，stride=2，得到 6 个 14×14 大小的特征图：28/2=14。

S2 层其实相当于降采样层 + 激活层。先是降采样，然后激活函数 sigmoid 非线性输出。先对 C1 层 2x2 的视野求和，然后进入激活函数，即：

$$sigmoid(w \cdot \sum_{i=1}^{4} x_i + b)$$

图 2.3: sigmoid 函数

但是在本次实验的实现中，为了解决谢晋老师在课堂中所提及到的梯度消失问题，我在本次实验中使用 Relu 函数进行非线性输出。

参数个数：(1+1)*6=12，其中第一个 1 为池化对应的 2*2 感受野中最大的那个数的权重 w，第二个 1 为偏置 b。

连接数：(2*2+1)*6*14*14= 5880，虽然只选取 2*2 感受野之和，但也存在 2*2 的连接数，1 为偏置项的连接，14*14 为输出特征层，每一个像素都由前面卷积得到，即总共经历 14*14 次卷积。

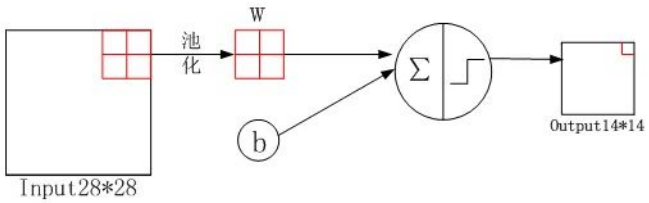

图 2.4: S2 示意图

2.6 C3 层

C3 层是卷积层，使用 16 个 5×5xn 大小的卷积核，padding=0，stride=1 进行卷积，得到 16 个 10×10 大小的特征图：14-5+1=10。

16 个卷积核并不是都与 S2 的 6 个通道层进行卷积操作，如下图所示，C3 的前六个特征图(0,1,2,3,4,5)由 S2 的相邻三个特征图作为输入，对应的卷积核尺寸为：5x5x3；接下来的 6 个特征图（6,7,8,9,10,11）由 S2 的相邻四个特征图作为输入对应的卷积核尺寸为：5x5x4；接下来的 3 个特征图（12,13,14）号特征图由 S2 间断的四个特征图作为输入对应的卷积核尺寸为：5x5x4；最后的 15 号特征图由 S2 全部(6 个) 特征图作为输入，对应的卷积核尺寸为：5x5x6。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

图 2.5: C3 特征图分布

卷积核是 5×5 且具有 3 个通道，每个通道各不相同，这也是下面计算时 5*5 后面还要乘以 3,4,6 的原因。这是多通道卷积的计算方法。

参数个数：(5*5*3+1)*6+(5*5*4+1)*6+(5*5*4+1)*3+(5*5*6+1)*1=1516。

连接数：1516*10*10 = 151600。10*10 为输出特征层，每一个像素都由前面卷积得到，即总共经历 10*10 次卷积。

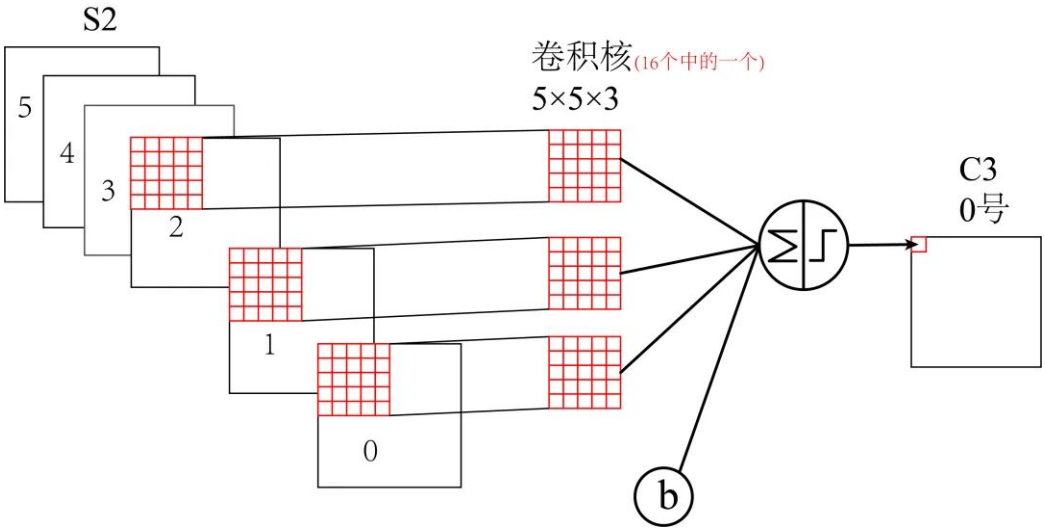

图 2.6: C3 示意图

2.7 S4 层

S4 层与 S2 一样也是降采样层，使用 16 个 2×2 大小的卷积核进行池化，padding=0，stride=2，得到 16 个 5×5 大小的特征图：10/2=5。

参数个数：(1+1)*16=32。

连接数：(2*2+1)*16*5*5= 2000。

2.8 C5 层

C5 层是卷积层，使用 120 个 5×5x16 大小的卷积核，padding=0，stride=1 进行卷积，得到 120 个 1×1 大小的特征图：5-5+1=1。即相当于 120 个神经元的全连接层。

与 C3 层不同，这里 120 个卷积核都与 S4 的 16 个通道层进行卷积操作。

参数个数：(5*5*16+1)*120=48120。

连接数：48120*1*1=48120。

2.9 F6 层

F6 是全连接层，共有 84 个神经元，与 C5 层进行全连接，即每个神经元都与 C5 层的 120 个特征图相连。计算输入向量和权重向量之间的点积，再加上一个偏置，结果通过 sigmoid 函数输出。

F6 层有 84 个节点，对应于一个 7x12 的比特图，-1 表示白色，1 表示黑色，这样每个符号的比特图的黑白色就对应于一个编码。该层的训练参数和连接数是 (120 + 1)x84=10164。ASCII 编码图如下：

图 2.7: ASC 码编码示意图

参数个数：(120+1)*84=10164。

连接数：(120+1)*84=10164。

2.10 OUTPUT 层

最后的 Output 层也是全连接层，是 Gaussian Connections，采用了 RBF 函数（即径向欧式距离函数），计算输入向量和参数向量之间的欧式距离（目前已经被 Softmax 取代）。

Output 层共有 10 个节点，分别代表数字 0 到 9。假设 x 是上一层的输入，y 是 RBF 的输出

参数个数：84*10=840。

连接数：84*10=840。

下图完整展示了手写数字 3 的识别情况：

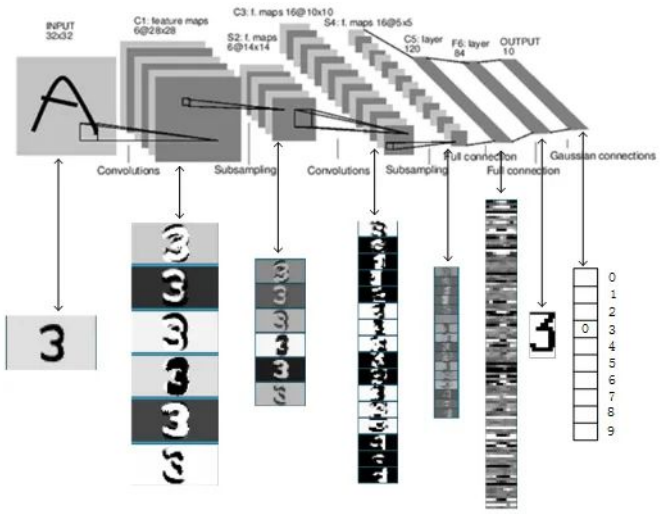

图 2.8: 手写数字 3 识别情况

3 实验实现

在本次实验实现中，我把我的代码分为四个模块，分别是：数据读取模块、前向传播反向传播模块、网络结构模块、运行调用模块。

3.1 数据读取模块

在数据读取模块,我的实现参考了前人的工作:https://blog.csdn.net/hxxjxw/article/details/113727973，大体的数据读入操作流程是一样的，只不过为了更好的完成实验流程，我在数据读入时进行了如下优化:

第一，在训练集中划分真正的训练集和验证集，这样可以在训练每一轮迭代时准确的得知模型效果。

第二，需要对读入的数据进行 padding，因为在前面介绍 lenet5 网络结构时已经介绍清楚，他的 input size 是 32*32 的，但是读入的 mnist 是 28*28 的，所以需要在两侧进行 padding=2 的填充工作第三，在读入时对数据进行归一化操作，优化实验效果

读入 mnist 数据集

```
# 在本函数中完成了本地的数据加载
# 根据参数取出相应的验证集并做正则化
def load_minist_data():
    trainImages = load_mnist("data/train-images.idx3-ubyte",True)
    trainLabels = load_mnist("data/train-labels.idx1-ubyte",False)
    testImages = load_mnist("data/t10k-images.idx3-ubyte",True)
    testLabels = load_mnist("data/t10k-labels.idx1-ubyte",False)

    # 对矩阵进行进行填充
    # https://blog.csdn.net/qq_34650787/article/details/80500407
    trainImages = np.pad(trainImages, ((0, 0), (2, 2), (2, 2)))
    testImages = np.pad(testImages, ((0, 0), (2, 2), (2, 2)))

    # 获取矩阵维度
```

```
    real_num, real_rows, real_cols = trainImages.shape
    real_num2,real_rows2,real_cols2 = testImages.shape
    # print('real_rows=%d,real_cols=%d' % (real_rows,real_cols))

    # 这个类型转换我也搞不太懂
    # 进行类型转换 https://blog.csdn.net/u012267725/article/details/77489244
    # 我的理解是在这里把很多页书拼起来拼成一页，但是这一页很长
    # print(trainImages.shape) #:这里他的输出是（60000,32,32）
    trainImages = trainImages.astype(np.float32).reshape(real_num, 1, real_rows,
        real_cols)
    testImages = testImages.astype(np.float32).reshape(real_num2, 1, real_rows,
        real_cols)
    # print(trainImages.shape) #:在这里输出就变成了（60000,1,32,32）

    # 接下来在训练集中划分验证集
    # 这里的参数是可以调整的
    varProof = -500 # 取最后的500个
    proofImages = trainImages[varProof:]
    proofLabels = trainLabels[varProof:]
    trainImages = trainImages[:varProof]
    trainLabels = trainLabels[:varProof]

    # 对数据进行归一化，这里其实也可以做一个参数选项，可选可不选
    # https://blog.csdn.net/sdgfbhgfj/article/details/123780347
    if True:
        mean = np.mean(trainImages,axis=0)
    else:
        mean = np.zeros_like(trainImages)
    trainImages -= mean
    proofImages -= mean
    testImages -= mean
    print('加载完毕')
    # 依次返回训练数据&标签 验证数据&标签 测试数据&标签
    return trainImages,trainLabels,proofImages,proofLabels,testImages,testLabels
```

3.2 前向传播与反向传播模块

由于 lenet-5 使用了卷积层、relu、最大池化、全连接层四种具有不同特征的层次结构，所以我们需要对其每一个部分都编写前向传播与反向传播函数，

池化层前向传播

```
# 卷积层前向传播的过程
def convolution_spread_forward(train, convolution_entity, bias, para):
    pic_num = train.shape[0]
    channel = train.shape[1]
    height = train.shape[2]
    wide = train.shape[3]
    convolution_num = convolution_entity.shape[0]
```

```
    convolution_height = convolution_entity.shape[2]
    convolution_wide = convolution_entity.shape[3]
    step = para['step'] #这里代表着步长
    padding = para['pad'] # 填充部分的大小
    # 做矩阵填充
    new_train = np.pad(train,((0,0),(0,0),(padding,padding),(padding,padding)))
    # 根据步长算到底需要多少次遍历矩阵
    loop_outside  = height + 2 * padding - convolution_height
    loop_outside  = loop_outside / step
    loop_outside = math.ceil(loop_outside)
    loop_outside  = loop_outside + 1
    loop_inside = wide + 2 * padding - convolution_wide
    loop_inside = loop_inside / step
    loop_inside = math.ceil(loop_inside)
    loop_inside = loop_inside + 1
    # 把这个临时权重置零
    weight = np.zeros((pic_num, convolution_num, loop_outside, loop_inside))
    for i in range(loop_outside):
        for j in range(loop_inside):
            temp = []
            # 对矩阵根据输入进行赋值
            temp = new_train[:,:,step*i:step*i+convolution_height,
            step*j:step*j+convolution_wide]
            temp = temp.reshape(pic_num,1,channel,convolution_height,convolution_wide)
            new_convolution = convolution_entity.reshape(1,convolution_num,channel,
            convolution_height,convolution_wide)
            # 加上偏置
            weight[:,:,i,j] = np.sum(temp * new_convolution,axis=(-3,-2,-1))
            weight[:,:,i,j] += bias
    # 把输入值保存以便反向传播时使用
    package = (train,convolution_entity,bias,para)
    return weight,package
```

具体的前向传播过程比较好理解，第一次进入时将权重初始化，然后通过反向传播调整权重和偏执，在前向传播时只需要利用这些权重和偏置进行矩阵计算就可以了。

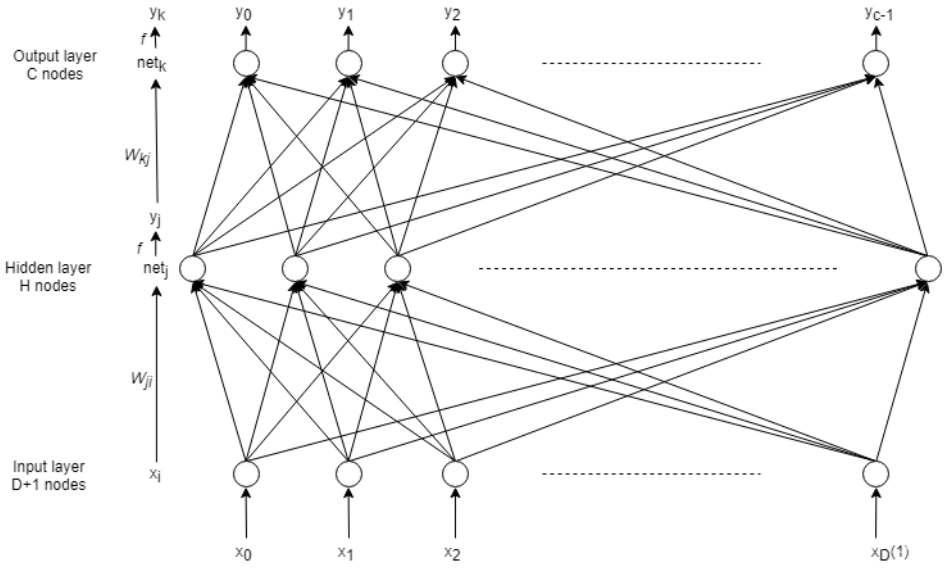

图 3.9: 前向传播

其正向传播具体计算公式可以参考如下公式：K 为卷积核个数，W 为卷积核参数 bias 为偏置

$$x_{i,j}^{lk} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} W_{a,b}^{lk} y_{i+a,j+b}^{l-1} + bias^{lk}, 0 \le k < K$$
$$y_{i,j}^{lk} = f^{l}(x_{i,j}^{lk}) = x_{i,j}^{lk}$$

图 3.10: 卷积层前向传播公式

再反向传播时，对偏重以及权重的公式为：

$$\frac{\partial E}{\partial W_{a,b}^{l}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{i,j}^{l}} \frac{\partial x_{i,j}^{l}}{\partial W_{a,b}^{l}}$$
$$= \sum_{i=0}^{N-m}\sum_{j=0}^{N-m} \delta_{i,j}^{l} y_{i+a,j+b}^{l-1}$$

$$\frac{\partial E}{\partial bias^{l}} = \sum_{i=0}^{N-m}\sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{i,j}^{l}} \frac{\partial x_{i,j}^{l}}{\partial bias^{l}}$$
$$= \sum_{i=0}^{N-m}\sum_{j=0}^{N-m} \delta_{i,j}^{l}$$

图 3.11: 卷积层反向传播公式

于是，也可以得到对下一层的传播公式：

$$\frac{\partial E}{\partial y_{i,j}^{l-1}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} \frac{\partial E}{\partial x_{i-a,j-b}^{l}} \frac{\partial x_{i-a,j-b}^{l}}{\partial y_{i,j}^{l-1}} = \sum_{a=0}^{m-1}\sum_{b=0}^{m-1} \delta_{i-a,j-b}^{l} W_{a,b}^{l}$$

图 3.12: 卷积层反向传播公式 2

卷积层的反向传播如下所示:

卷积层反向传播

```
def convolution_spread_backward(derivation,pack):
    #首先需要拿到前向传播的一些参数
    num = derivation.shape[0]
    convolution_num = derivation.shape[1]
    loop_outside = derivation.shape[2]
    loop_inside = derivation.shape[3]
    train = pack[0]
    height = train.shape[2]
    wide = train.shape[3]
    convolution = pack[1]
    derivation_convolution = np.zeros_like(convolution)
    channel =convolution.shape[1]
    convolution_height = convolution.shape[2]
    convolution_wide = convolution.shape[3]
    bias = pack[2]
    derivation_bias = np.zeros_like(bias)
    para = pack[3]
    padding = para['pad']
    step = para['step']
    # 为了矩阵运算参数 进行padding
    new_train = np.pad(train,((0,0),(0,0),(padding,padding),(padding,padding)))
    derivation_train = np.zeros_like(new_train)
    for i in range(loop_outside):
        for j in range(loop_inside):
            # 根据公式推导对weight矩阵进行调整
            temp = derivation[:,:,i,j]
            temp = temp.reshape((num,1,1,1,convolution_num))
            temp2 = convolution.transpose((1,2,3,0))
            temp2 =
                temp2.reshape((1,channel,convolution_height,convolution_wide,convolution_num))
            temp_sum = np.sum(temp*temp2,axis=-1)
            derivation_train[:,:,step*i:step*i+convolution_height,
            step*j:step*j+convolution_wide] +=temp_sum
            temp3 = derivation[:,:,i,j].T
            temp3 = temp3.reshape((convolution_num,1,1,1,num))
            temp4 = new_train[:,:,step*i:step*i+convolution_height,
```

```
            step*j:step*j+convolution_wide].transpose(1,2,3,0)
            对bias进行调整
            temp_sum = np.sum(temp3*temp4, axis=-1)
            derivation_convolution += temp_sum
            temp_sum = np.sum(derivation[:,:,i,j], axis=0)
            derivation_bias += temp_sum

    # 赋值真正最后的计算结果
    derivation_weight = []
    derivation_weight =
        derivation_train[:,:,padding:padding+height,padding:padding+wide]
    return derivation_weight,derivation_convolution,derivation_bias
```

池化层的正向传播比较简单，在这里不做过多介绍，而是介绍他的反向传播。

$$
\begin{aligned}
\delta_{i,j}^{l} &= \frac{\partial E}{\partial x_{i,j}^{l}} \\
&= \frac{\partial E}{\partial y_{i,j}^{l}} \frac{\partial y_{i,j}^{l}}{\partial x_{i,j}^{l}} \\
&= \frac{\partial E}{\partial y_{i,j}^{l}} sigmoid'(x_{i,j}^{l}) \\
&= \frac{\partial E}{\partial y_{i,j}^{l}} sigmoid(x_{i,j}^{l})(1 - sigmoid(x_{i,j}^{l}))
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial W^{l}} &= \sum_{i=0}^{N} \sum_{j=0}^{N} \frac{\partial E}{\partial x_{i,j}^{l}} \frac{\partial x_{i,j}^{l}}{\partial W^{l}} \\
&= \sum_{i=0}^{N} \sum_{j=0}^{N} \delta_{i,j}^{l} \left(\sum_{a=0}^{1} \sum_{b=0}^{1} y_{2i+a,2j+b}^{l-1} \right)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial bias^{l}} &= \sum_{i=0}^{N} \sum_{j=0}^{N} \frac{\partial E}{\partial x_{i,j}^{l}} \frac{\partial x_{i,j}^{l}}{\partial bias^{l}} \\
&= \sum_{i=0}^{N} \sum_{j=0}^{N} \delta_{i,j}^{l}
\end{aligned}
$$

图 3.13: 池化反向传播公式

下面是池化部分的反向传播代码实现：

卷积层反向传播

```python
def pool_spread_backward(derivation,pack):
    train = pack[0]
    derivation_train = np.zeros_like(train)
    train_num = train.shape[0]
    channel = train.shape[1]
    height = train.shape[2]
    wide = train.shape[3]
    para = pack[1]
    pool_height = para['pool_height']
    pool_wide = para['pool_wide']
    step = para['step']
    loop_outside = height - pool_height
    loop_outside = loop_outside // step
    loop_outside = loop_outside + 1
    loop_inside = wide - pool_wide
    loop_inside = loop_inside // step
    loop_inside = loop_inside + 1
    for i in range(loop_outside):
        for j in range(loop_inside):
            temp = train[:,:,i*step:i*step+pool_height,j*step:j*step+pool_wide]
            temp = temp.reshape(train_num,channel,-1)
            tempMax = np.argmax(temp,axis=-1)
            temp_max = np.unravel_index(tempMax,(pool_height,pool_wide))
            which = np.array(temp_max)
            for k in range(train_num):
                for l in range(channel):
                    temp2 = derivation[k][l][i][j]
                    index_3 = i*step+which[0][k][l]
                    index_4 = j*step+which[1][k][l]
                    derivation_train[k][l][index_3][index_4] += temp2

    return derivation_train
```

全连接层其实和第二次作业 CNN 部分很像，这一部分已经在 CNN 进行过十分详实的推导，随意也不做过多介绍。

3.3 网络结构模块

其实，构建网络结构的过程就是对几个权重矩阵、偏置矩阵进行初始化的过程，并且将他们分装成函数，这样只需要在训练时调用函数就可以自动完成正向传播和反向传播。在这里我们就需要充分利用在第一部分介绍过的一些参数，比如输入的通道数、卷积核的大小等等完成我们对网络结构的搭建:

网络结构搭建

```python
def __init__(self):
        # 输入的图像通道为1
        # 输入图像大小为32*32
```

```
        self.input_channel = 1
        self.input_height = 32
        self.input_wide = 32
        # 第一次卷积核有6个
        # 第二次卷积核有16个
        # 卷积核大小都是5*5
        self.convolution_num1 = 6
        self.convolution_num2 = 16
        self.convolution_height = 5
        self.convolution_wide =5
        # 全连接层的大小
        self.full_connect1 = 400
        self.full_connect2 = 120
        self.full_connect3 = 84
        # 输出类型个数
        self.output_size = 10
        # lenet5的参数
        # 卷积池化层的参数和偏置项
        # 全连接层的参数和偏置项
        self.para = {}
        # 正则项
        self.regularItem = 0.001
        self.datatype = np.float32
        # 生成时必要的标准差
        self.standard_deviation = 0.001

        #首先初始化第一个卷积层
        convolution1_para_weight = np.random.normal(
            loc=0.0,
            scale=self.standard_deviation,
            size=(self.convolution_num1,self.input_channel,self.convolution_height
            ,self.convolution_wide)
        )
        convolution1_para_bias = np.zeros(self.convolution_num1)

        #接下来初始化第二个卷积层
        convolution2_para_weight = np.random.normal(
            loc = 0.0,
            scale = self.standard_deviation,
            size =
                (self.convolution_num2,self.convolution_num1,self.convolution_height
            ,self.convolution_wide)
        )
        convolution2_para_bias = np.zeros(self.convolution_num2)

        #接下来初始化第一个全连接层
        full_connect1_para_weight = np.random.normal(
            loc = 0.0,
            scale = self.standard_deviation,
```

```
            size = (self.full_connect1,self.full_connect2)
        )
        full_connect1_para_bias = np.zeros(self.full_connect2)

        #接下来初始化第二个全连接层
        full_connect2_para_weight = np.random.normal(
            loc=0.0,
            scale = self.standard_deviation,
            size = (self.full_connect2,self.full_connect3)
        )
        full_connect2_para_bias = np.zeros(self.full_connect3)

        #接下来初始化第三个全连接层
        full_connect3_para_weight = np.random.normal(
            loc=0.0,
            scale = self.standard_deviation,
            size = (self.full_connect3,self.output_size)
        )
        full_connect3_para_bias = np.zeros(self.output_size)
```

3.4 接口调用

这个部分其实就比较简单了，就是将上述内容联系在一次并调用即可：

卷积层反向传播

```
TrainModel = MyLenet5()
myParameter = {}
myParameter['learn_rate']=1e-3
myTrain = Train(Lenet5=TrainModel,data=data,epoch=20,func='sgd',parameter=myParameter
,decrease=0.75,batchSize=200,exp=1)
myTrain.myTrain()
```

4 实验部分

4.1 实验环境

本次实验使用的编译器版本为 3.10，使用的集成开发环境为 Pycharm2022 Pro 以及 jupyter

4.2 实验效果

下图为数据加载过程的结果：

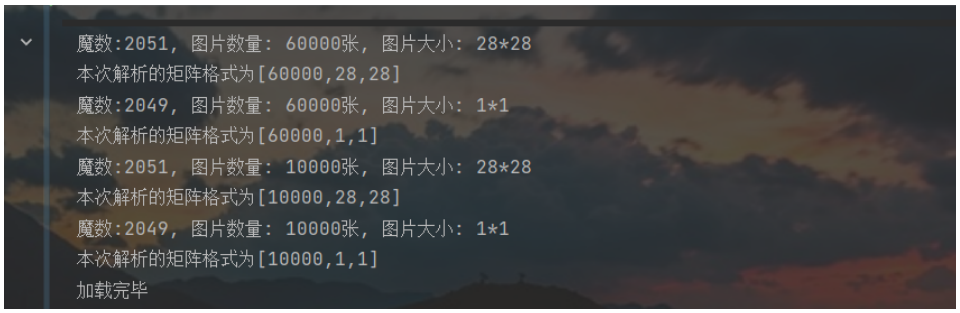

图 4.14: 数据加载结果

下图为训练时的结果:

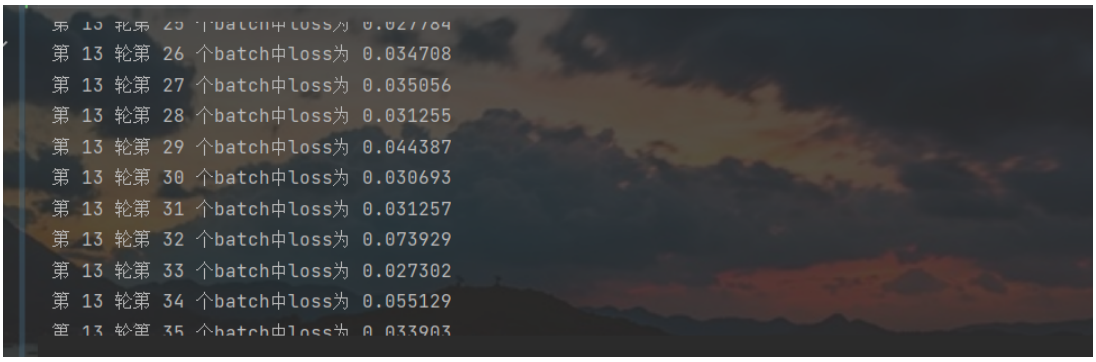

图 4.15: 训练时效果

最终训练效果:

图 4.16: 最终训练结果

训练过程中轮数-损失图:

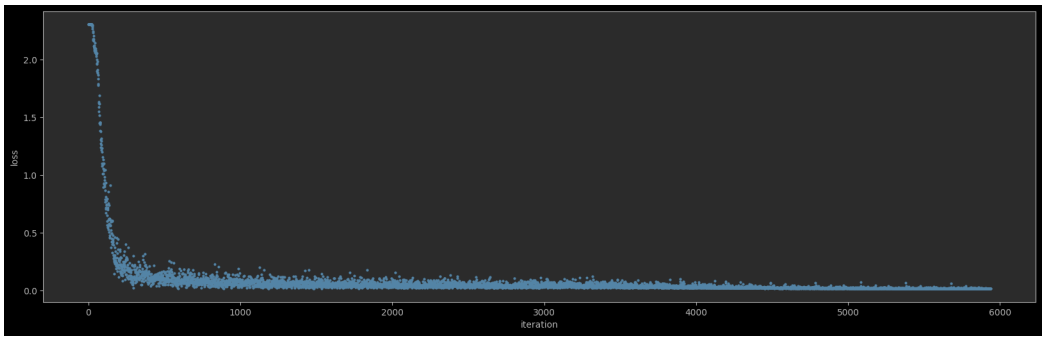

图 4.17: 轮数-损失函数图

最终测试集效果展示：

```
In 18  1  print(
       2      "Test accuracy:",
       3      myTrain.calAccuracy2(data['test_image'], data['test_label'],100)
       4  )

       Test accuracy: 0.9921
```

图 4.18: 最终测试集正确率

最终从训练集分割的验证集以及训练集的训练效果展示：

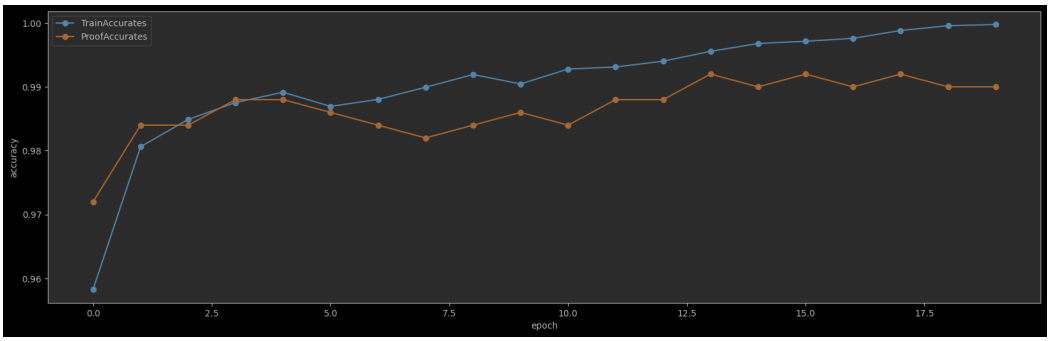

图 4.19: 训练集-验证集正确率函数图

4.3 实验分析

在本次实验中，为了不断加强实验效果，我也对训练过程中不同的参数设置、网络结构进行了动态调整，最终调整后的实验结果如上面的图片所展示，在 20 轮迭代后在测试集的正确率可以达到 99.21%，可以说非常完美的完成了 mnist 手写体识别的任务。

在本次实验中一开始我没有加入正则项，在没有加入正则项的情况下在 20 轮迭代后得到的模型训练后正确率可以达到 88%，该结果表明在不使用正则项的情况下模型也可以较为争取的完成手写体识别的任务，但是因为过拟合的情况导致最后的正确率不高

其实是加入了 batch_size 参数，不在一次训练过程中使用全部的样本，在不适用 batch_size 的情况下，完成 20 轮迭代训练需要接近 11 个小时，但是加入 batch_size 后训练时间减少至 3.5 小时左右

最后，我们不难发现，使用简单的 Lenet-5 框架可以非常好的完成手写体识别的任务，并且在训练精确度上完美胜于第一次实验所做的基于回归的手写体识别任务，这样证明了深度学习基本框架的极高的应用价值

5 总结

通过本次实验，我自己动手实现了哟一个 Lenet-5 网络，并完成了手写体识别任务，让我进一步加深了对于 lenet-5 的理解，并且不适用任何包也锻炼了我的代码能力，加深了我对前向传播、反向传播以及网络结构细节的理解，让我受益匪浅。

6 参考博客

由于本人在实验初期对 lenet-5 的了解有限，所以在网络上找到了一些博客进行学习，这些博客内容主要包括：mnist 数据集加载、反向传播公式推导、lenet5 网络结构简介等。

https://blog.csdn.net/red_stone1/article/details/121804658

https://blog.csdn.net/hzoi_ztx/article/details/84780441