



南開大學  
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

机器学习实验报告

---

## LeNet5 设计与实现

---

2014074 费泽锟

年级：2020 级

专业：信息安全

指导教师：谢晋

2023 年 1 月 4 日

## 摘要

在 LeNet5 的设计与实现之中，主要的问题其一就是对于 LeNet5 网络结构的理解，为了充分理解 LeNet5 网络结构，对 Lecun 教授的论文进行了进一步的学习，来充分了解 LeNet5 的具体网络结构。

其二就是在面对十分复杂的网络结构以及包括前向传播与后向传播，面对庞大的参数数目之时，如何能够成功地完成各种计算过程，保证模型的正确性和如何能够加快模型的训练速度，在本次设计之中，这些方面都经过了考虑与优化。

在 LeNet5 的设计与实现之中，从底层开始，使用 Python 语言和最基础的 numpy 第三方库实现了对 MNIST 数据集中 0-9 共 10 个手写数字的分类模型。

**关键字：**LeNet5; Numpy; MINST; 卷积层; 池化层;

## 目录

<b>一、 实验环境</b>	<b>1</b>
<b>二、 实验要求</b>	<b>1</b>
<b>三、 MINST 数据集介绍</b>	<b>1</b>
<b>四、 MINST 数据集导入</b>	<b>2</b>
<b>五、 LeNet5 网络结构</b>	<b>3</b>
(一) Input: 输入层 . . . . .	5
(二) C1: 卷积层 . . . . .	5
(三) S2: 池化层 . . . . .	11
(四) C3: 卷积层 . . . . .	13
(五) S4: 池化层 . . . . .	15
(六) C5: 卷积层 . . . . .	15
(七) F6: 全连接层 . . . . .	15
(八) Output: 输出层 . . . . .	16
(九) 一些参数说明 . . . . .	17
<b>六、 模型结果分析</b>	<b>18</b>
<b>七、 模型优化策略</b>	<b>20</b>
(一) batch 随机梯度下降 . . . . .	20
(二) L-M 方法学习率优化 . . . . .	21
(三) 不同的激活函数 . . . . .	23
(四) 性能优化尝试 . . . . .	24
<b>八、 总结</b>	<b>24</b>

## 一、 实验环境

- 操作系统: Windows 11 操作系统
  - Python 版本: Python 3.7.5
  - IDE: PyCharm Commemunity Edition 2021.2.2
- (仅使用 numpy、os、sys、math、time 等基础的 Python 第三方库)

## 二、 实验要求

在这个练习中, 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。

MNIST 数据集可在[数据集下载地址](#)中下载。

## 三、 MNIST 数据集介绍

在开始设计实现 LeNet5 网络结构之前, 我们需要先充分了解 MNIST 数据集是如何存储的以及存储的格式等各方面的内容, 在这一部分之中, 将会根据 Lecun 教授的官网内容, 来介绍 MNIST 数据集的存储以及使用方法。

首先, 我们需要认识经典的数据集 MNIST 的二进制内容存储格式。如下图所示:

### TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

图 1: MNIST 数据集格式

在 Lecun 教授的官网上详细介绍了 MNIST 数据集的存储格式, 首先会存储一个 32 位 int 型数据作为 magic 信息类似于 PE 头中的 5A4D。

接着的三个 32 位 int 型的数据存储的分别是图片的数量以及对应的数据集中的图片的二维信息, 该数据集中的图片为 28\*28 像素的大小, 也就是 784 维的特征, 所以在载入该数据集时, '>iiii' 大端法读取 4 个 unsigned int32, 后续的二进制 bit 存储的就是图片的像素信息了。

如果将二进制对应的灰度进行转换, 可以看到图像近似如下:

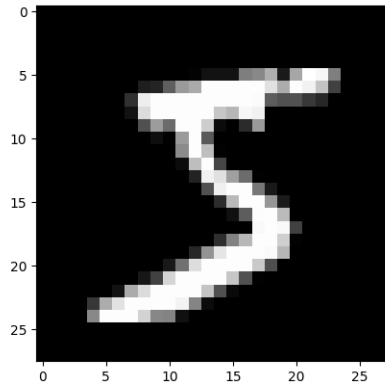


图 2: MNIST 数据集图像

在我们已经了解了 MNIST 数据集的存储格式之后,我们就可以灵活地运用 MNIST dataset 之中的数据了。

## 四、 MNIST 数据集导入

我们已经了解了 MNIST 数据集的存储格式,那么现在离我们能够使用这些数据就仅仅只差一步,那就是使用 Python 语言将这些数据成功导入为我们能够使用的数据类型,这显然不是轻松的一步,但是感谢于第一次作业的内容,也就是 softmax regression 的实现,在 softmax regression 实现的框架之中就已经给出了完整的数据集导入的方法。

具体代码如下:

### MNIST 数据集导入

```

1 # 加载MNIST数据集的数据
2 def load_mnist(file_dir, is_images='True'):
3     # 读取二进制文件
4     bin_file = open(file_dir, 'rb')
5     bin_data = bin_file.read()
6     bin_file.close()
7     # 判断每个数据的首部是图像数据还是标签
8     if is_images:
9         # 读取图像
10        fmt_header = '>iiii'
11        magic, num_images, num_rows, num_cols = struct.unpack_from(fmt_header,
12                               bin_data, 0)
13        data_size = num_images * num_rows * num_cols
14        mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_data,
15                                     struct.calcsize(fmt_header))
16        mat_data = np.reshape(mat_data, [num_images, num_rows, num_cols])
17    else:
18        # 读取标签
19        fmt_header = '>ii'
20        magic, num_images = struct.unpack_from(fmt_header, bin_data, 0)
21        mat_data = struct.unpack_from('>' + str(num_images) + 'B', bin_data,
22                                     struct.calcsize(fmt_header))

```

```

20     mat_data = np.reshape(mat_data, [num_images])
21     print('Load images from %s, number: %d, data shape: %s' % (file_dir,
22         num_images, str(mat_data.shape)))
23     return mat_data

```

因为我们知道在 MINST 数据集之中，有图像和标签两种类型的数据，并且每种类型的数据的首部信息是不同的，分别为 4 个 32 位大小与 2 个 32 位大小的区别，所以我们在判断首部之后会跳过首部，然后读取数据，最终将数据转化为 np 形式的数组，需要注意的是我们需要读取的是二进制形式的文件，所以需要使用 struct 第三方库来逐 bit 读取二进制文件的内容。

需要注意的是，在存入了训练集和测试集的图片像素矩阵之后，还需要对其中的灰度矩阵进行处理，当其中像素点的值大于等于 40 的时候，将其设置为 1，当其中像素点的值小于 40 时将其设置为 0，具体代码如下：

data\_convert 函数

```

1 def data_convert(x, y, m, k):
2     x[x <= 40] = 0
3     x[x > 40] = 1
4     ont_hot_y = np.zeros((m,k)) # 60000*10大小的特征矩阵
5     for t in np.arange(0,m):
6         ont_hot_y[t,y[t]]=1
7     return x, ont_hot_y

```

同时在 data\_convert 函数之中，还对每一张图片的类别进行了独热编码，也就是将本来 0-9 的特征，转化为 10 列，如果其为数字 x 则将 x 列上的矩阵值设置为 1，其余的设置 0，这样的话无论是训练集的图片矩阵还是标签矩阵，都已经转化为了 0、1 矩阵，就不需要进一步的标准化了。

至此我们已经能够成功地将二进制文件之中的数据转化为我们能够使用的 np 类型的数据，那么接下来将会详细地讲解 LeNet5 的网络结构。

## 五、 LeNet5 网络结构

LeNet5 的网络结构，如果算上输入层与输出层的情形，那么总计应当有 8 层的网络结构，具体网络结构组织如下：

Input -> C1 -> S2 -> C3 -> S4 -> C5 -> F6 -> Output

其中 C1 层和 C3 层均为卷积层，S2 层和 S4 层均为池化层，C5 层的主要作用是将原本二维的 feature map 通过卷积的操作展开变成 120 维的向量，这其实不仅包含了卷积层的特征也有一定全连接层的特征，在本文中将其称为卷积层。

下图为具体的 LeNet5 网络结构的组织图：

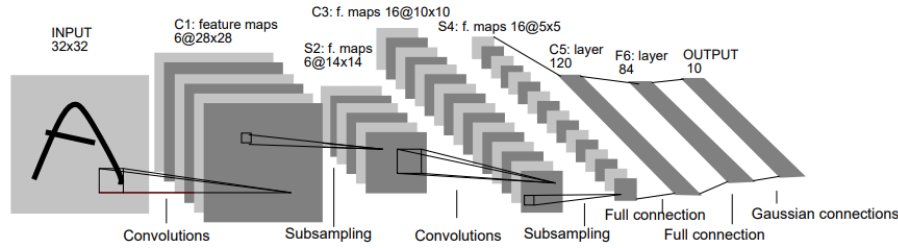


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

图 3: LeNet5 网络结构

为了能够适配整个的 LeNet5 的网络结构，在设计实现之中实现了 LeNet5 类，来对 LeNet5 中需要的变量与函数进行封装，具体类实现的代码如下：

#### LeNet5 类的实现

```

1 class LeNet5(object):
2     def __init__(self):
3         layer_shape = {"C1": (5, 5, 1, 6), "C3": (5, 5, 6, 16), "C5": (5, 5,
4             16, 120), "F6": (120, 84), "OUTPUT": (84, 10)}
5
6         # S2池化层需要选择不同的feature map来结合进行池化操作
7         C3_mapping = [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 0],
8             [5, 0, 1],
9             [0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 0], [4, 5, 0,
10                 1],
11             [5, 0, 1, 2], [0, 1, 3, 4], [1, 2, 4, 5], [0, 2, 3, 5], [0, 1, 2, 3,
12                 4, 5]]
13
14         # 设置步长
15         h_parameters_convolution = {"stride": 1, "pad": 0}
16         h_parameters_pooling = {"stride": 2, "f": 2}
17
18         self.C1 = ConvolutionLayer(layer_shape["C1"],
19             h_parameters_convolution)
20         self.a1 = Activation("sigmoid") # 激活函数 可以选择不同的激活函数
21         self.S2 = PoolingLayer(h_parameters_pooling, "average") # padding的
22             策略
23
24         self.C3 = ConvolutionLayer_maps(layer_shape["C3"],
25             h_parameters_convolution, C3_mapping)
26         self.a2 = Activation("LeNet5_squash")
27         self.S4 = PoolingLayer(h_parameters_pooling, "average")
28
29         self.C5 = ConvolutionLayer(layer_shape["C5"],
30             h_parameters_convolution)
31         self.a3 = Activation("LeNet5_squash")

```

```

25     self.F6 = FCLayer(layer_shape["F6"])
26     self.a4 = Activation("LeNet5_squash")
27
28     self.Output = RBFLayer(bitmap)

```

这其中对于每一层的卷积核的大小都进行了初始化的设置，其中的 C1、C3、C5 层均作为卷积层对卷积核的大小进行了设置，而最后的全连接层则是进行特征的抽取，输出层则是通过 84 维的特征来对 0-9 的数字进行分类，为什么是 84 维的特征将会在输出层进行进一步的讲解。

### (一) Input：输入层

对于 LeNet5 结构之中的 Input 输入层，该层主要是需要将 train 训练集和 test 测试集之中的图片进行标准化，这里有需要注意的一点，那就是本来从 MINST 数据集二进制格式的文件之中导出的图片数据只是 28\*28 大小的，而网络需要的输入数据是 32\*32 大小的，所以我们需要以 padding=2 的格式将图片进行填充。

这里选择的填充方式是以 0 进行填充，而在填充之后我们还选择标准化的策略，虽然我们已使用了 data\_convert 函数完成了灰度到 0-1 的转化，但是这里可以选择参数进行适当的标准化，单纯使用 Minmax 标准化的结果将不变，具体的代码如下：

输入层图片数据预处理

```

1 # 使用0进行图片大小的填充
2 def zero_pad(X, pad):
3     X_pad = np.pad(X, ((0, 0), (pad, pad), (pad, pad), (0, 0)), 'constant',
4                     constant_values=(0, 0))
5     return X_pad
6
7 # 标准化输入的图像
8 def normalize(image, mode='LeNet5'):
9     # Minmax标准化
10    image -= image.min()
11    image = image / image.max()
12    if mode == '0p1':
13        return image
14    elif mode == 'LeNet5':
15        image = image * 1.275 - 0.1
16    else:
17        pass;
18    return image

```

这样我们就能够在输入层获得合适的 32\*32 大小的输入了，main 函数中的标准化与填充代码这里就不再展示了，在获得了合适的输入数据后，我们就可以进入到了卷积层的运算。

### (二) C1：卷积层

进入到了卷积层的网络结构后，我们首先要知道卷积层的运算到底是如何进行的，接下来将会介绍卷积层的运算原理。

在使用卷积神经网络进行图像识别时，输入为进行过转换的图片数据，一张宽为  $w$ ，高为  $h$ ，深度为  $d$  的图片，表示为  $h*w*d$ 。这里，深度为图像存储每个像素所用的位数，比如彩色图像，

其一个像素有 RGB 三个分量，其深度为 3。

从数学的角度来看， $h*w*d$  的图片即为  $d$  个  $h*w$  的矩阵。例如  $6*16*3$  的图片，其对应 3 个  $6*16$  的矩阵。在大部分运用中，输入图片的大小  $h$  和  $w$ ，一般是相等的。

很庆幸的是，在我们使用到的 MINST 数据集之中，仅仅使用的数据为灰度图像，也就是说实际 Input 层输入的深度是为 1 的，但是接下来在我们的 C1 也就是第一层卷积层之中，使用到的卷积核的深度就不为 1 了。

在卷积运算时，会给定一个大小为  $F*F$  的方阵，称为过滤器，又叫做卷积核，该矩阵的大小又称为感受野。过滤器的深度  $d$  和输入层的深度  $d$  保持一致，因此可以得到大小为  $F*F*d$  的过滤器，从数学的角度出发，其为  $d$  个  $F*F$  的矩阵。在实际的操作中，不同的模型会确定不同数量的过滤器，其个数记为  $K$ ，每一个  $K$  包含  $d$  个  $F*F$  的矩阵，并且计算生成一个输出矩阵。

#### 1. Padding。

在进行卷积运算时，输入矩阵的边缘会比矩阵内部的元素计算次数少，且输出矩阵的大小会在卷积运算中相比较于输入变小。因此，可在输入矩阵的四周补零，称为 padding，其大小为  $P$ 。比如当  $P=1$  时，原  $5*5$  的矩阵如下，蓝色框中为原矩阵，周围使用 0 作为 padding。

0	0	0	0	0	0	0
0	1	4	4	1	0	0
0	2	4	3	0	1	0
0	1	7	2	2	1	0
0	2	0	0	2	7	0
0	6	2	4	3	3	0
0	0	0	0	0	0	0

图 4: padding

2. 进行卷积运算时，过滤器在输入矩阵上移动，进行点积运算。移动的步长 stride，记为  $S$ 。当  $S=2$  时，过滤器每次移动 2 个单元。如下图，红色框为第一步计算，蓝色框为  $S=2$  时的第二步运算。

0	0	0	0	0	0	0
0	1	4	4	1	0	0
0	2	4	3	0	1	0
0	1	7	2	2	1	0
0	2	0	0	2	7	0
0	6	2	4	3	3	0

图 5: stride

有了以上两个参数  $P$  和  $S$ ，再加上参数  $W$ （输入矩阵的大小），过滤器的大小  $F$ ，输出矩阵的大小为：

$$(W - F + 2P) / S + 1$$

我们给出一个具体的包含多个卷积核的实例，输入矩阵为  $5*5*3$ ，即  $W=5$ ，填充  $P$  为 1，过滤器有  $K=2$  个，每个过滤器的大小为  $3*3*3$ ，即  $F=3$ ，同时设定计算步长  $S=2$ 。这样可得到输出中单个矩阵的大小为  $(5-3+2*1)/2+1=3$ ，由于  $K=2$ ，所以输出的  $3*3$  矩阵有 2 个。



那么具体的运算流程就会如同下图所示，要考虑到每个卷积核的偏执 bias 的计算过程：

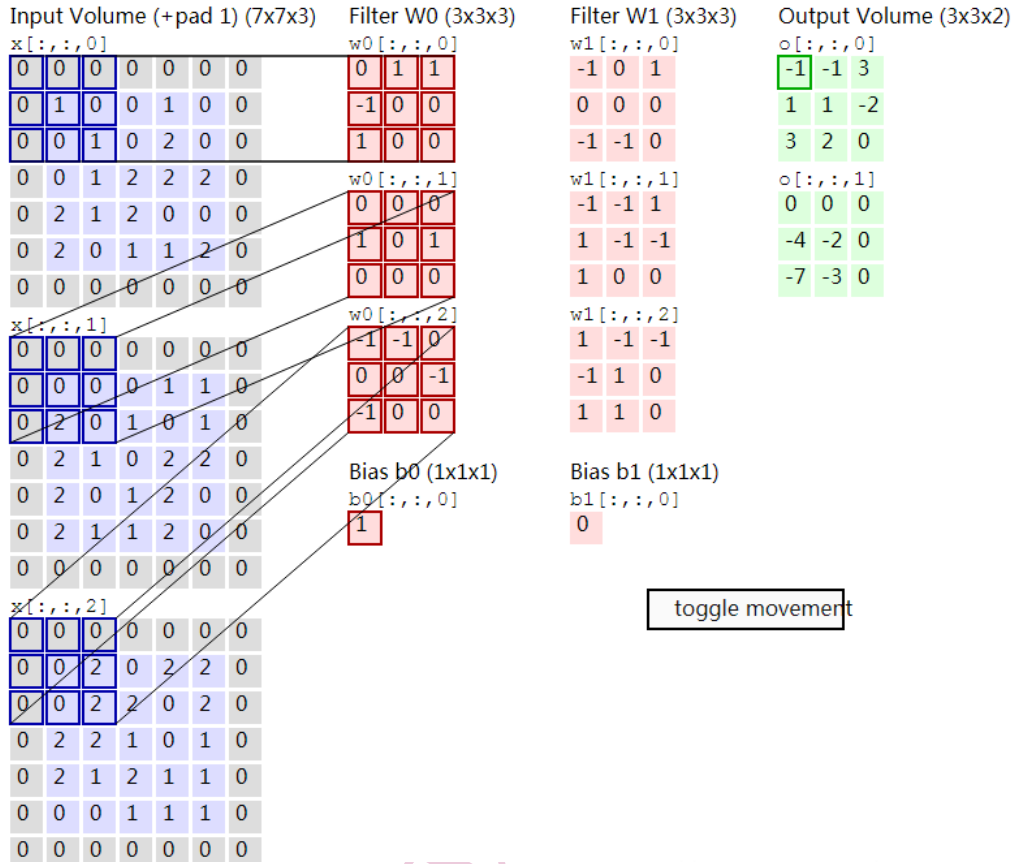


图 6: 卷积层计算流程

具体的卷积核运算的代码如下所示，但是需要注意的是，下述卷积层运算代码仅仅适用于 C1 层的卷积层操作，因为 C3 层的卷积核需要结合多个 feature map 进行运算，需要对原来的卷积层运算进行修改。

C1 卷积层

```

1 class ConvolutionLayer(object):
2     def __init__(self, kernel_shape, parameters, init_mode='Gaussian_dist'):
3         """
4         :param kernel_shape: 卷积核(n_f, n_f, n_C_prev, n_C)
5         :param parameters: 用于表征指明进行的步骤{"stride": s, "pad": p}
6         """
7         self.weight, self.bias = initialize(kernel_shape, init_mode)
8         self.v_w, self.vb_u = np.zeros(kernel_shape), np.zeros((1, 1, 1,
9                             kernel_shape[-1]))
10        self.parameters = parameters
11
12    def forward_propagation(self, input_map):
13        output_map, self.cache = conv_forward(input_map, self.weight, self.
14            bias, self.parameters)
15        return output_map

```

```

15 def back_propagation(self, dK, momentum, weight_decay):
16     dA_prev, dW, db = convb_uackward(dK, self.cache)
17     self.weight, self.bias, self.v_w, self.vb_u = update(self.weight,
18                                                         self.bias,
19                                                         dW, db, self.v_w, self.vb_u, self.lr, momentum,
20                                                         weight_decay)
21     return dA_prev

```

我们可以看到在 C1 卷积层类之中，需要对输入的 weights 和 bias，也就是权重和偏置进行包括前向传播和反向传播的操作，那么我們也需要对前向传播和反向传播进行一个了解，其中反向传播的部分我们已经在 softmax regression 的 project 之中进行了了解和实现，接下来主要介绍前向传播的部分。

前向传播：就是通过上一层的各结点以及对应的连接权值进行加权和运算，最终结果再加上一个偏置项，最后在通过一个非线性函数（即激活函数），如 ReLu, sigmoid 等函数，最后得到的结果就是本层结点 w 的输出。最终不断的通过这种方法一层层的运算，得到输出层结果。

具体可见下图展示的前向传播的过程，其实也就是顺序的网络结构计算：

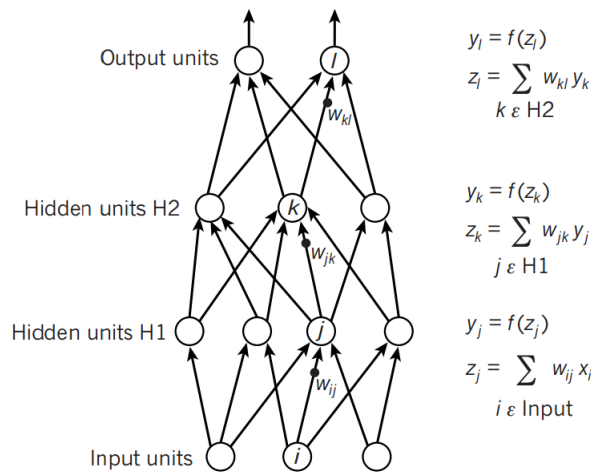


图 7: 前向传播

那么对于前向传播的运算过程，我们就能够得到一个可以通用的函数了，反向传播的算法就类似于前向传播的流程，这里的矩阵运算的基本流程沿用 softmax regression 部分的逻辑，前向传播和反向传播可能用到的一些激活函数以及激活函数的导数都进行了封装：

#### 激活函数与其导数

```

1 # sigmoid 函数，这里需要注意的是，这里均保留了 sigmoid 函数的一阶导数和二阶导数
2 def sigmoid(x):
3     return 1 / (1 + np.exp(-x))
4
5 def d_sigmoid(x):
6     return np.exp(-x) / np.power((1 + np.exp(-x)), 2)
7
8 def d2_sigmoid(x):
9     return 2 * np.exp(-2 * x) / np.power(np.exp(-x) + 1, 3) - np.exp(-x) / np
10     .power((1 + np.exp(-x)), 2)
11 # tanh 函数

```

```

12 def tanh(x):
13     return np.tanh(x)
14
15 def d_tanh(x):
16     return 1 / np.power(np.cosh(x), 2)
17
18 def d2_tanh(x):
19     return -2 * (tanh(x)) / np.power(np.cosh(x), 2)
20
21 # ReLU函数
22 def ReLU(x):
23     return np.where(x > 0, x, 0)
24
25 def d_ReLU(x):
26     return np.where(x > 0, 1, 0)
27
28 def d2_ReLU(x):
29     return np.zeros(d_ReLU(x).shape)

```

在这些激活函数内容准备就绪后，我们就可以来继续实现前向传播与反向传播算法了，因为代码过长，这里仅给出前向传播算法的实现代码：

#### 前向传播算法

```

1 # Numpy第三方库：np.tensordot()的快速优化
2 def conv_forward(A_prev, W, b, h_parameters):
3     (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
4     (f, f, n_C_prev, n_C) = W.shape
5
6     pad = h_parameters["pad"]
7     stride = h_parameters["stride"]
8
9     n_H = int((n_H_prev + 2 * pad - f) / stride + 1)
10    n_W = int((n_W_prev + 2 * pad - f) / stride + 1)
11
12    # 初始化Z列 0矩阵
13    Z = np.zeros((m, n_H, n_W, n_C))
14    A_prev_pad = zero_pad(A_prev, pad)
15    for h in range(n_H):
16        for w in range(n_W):
17            # 使用定义的A_prev_pad进行切片
18            A_slice_prev = A_prev_pad[:, h * stride:h * stride + f, w *
19                                   stride:w * stride + f, :]
20            # 使用对应的权重和偏置进行计算
21            Z[:, h, w, :] = np.tensordot(A_slice_prev, W, axes=([1, 2, 3],
22                                                                [0, 1, 2])) + b
23
24    cache = (A_prev, W, b, h_parameters)
25    return Z, cache

```

具体代码的解释可看注释中的内容，在实现完所需要的前向传播和反向传播的算法之后，我们就已经成功的完成了卷积层所需要的代码了。那么接下来就是对 LeNet5 网络结构之中的 C1 卷积层来进一步阐述。

为了更好的理解 LeNet5 网络结构之中的各个数量之间的关系，我们首先需要知道两个指标，那就是：参数量和计算量。

### 1. 参数量

参数量是指的网络中可以被学习的变量的数量，包括卷积核的权重 weights，批归一化 (BatchNormalization) 的缩放参数，以及偏移系数，虽然有些没有 BatchNormalization 层的也可能有偏置项 bias，这些参数都是可以被学习的，也就是在训练模型开始被赋予初值，在训练过程当中通过链式法则不断的迭代更新，整个模型的参数量主要由卷积核的权重 weights 的数量决定，参数量越大，对计算机的运行内存也就越高，对硬件的设备也就要求越高，在同样准确率的情况下参数量的多少是一个网络结构重要的评价指标。

### 2. 计算量

神经网络的前向推断过程基本上都是乘累加计算，所以它的计算量也是指前向推断过程中的乘加运行的次数，通常使用 FLOPS (Floating Point Operations) 表示。计算量越大，在同样条件下运行延时有就越长，尤其是在移动端/嵌入式这种资源受限的平台上想要达到实时性的要求就必须要求模型的计算量尽可能的低，这个还跟算子的密集程度相关。

那么就可以来详细地量化 C1 层的各个参数与数据了：

- 采用六个 5\*5 卷积核进行卷积：5\*5\*6  $\rightarrow$  28\*28\*6
- 参数量：(5\*5\*1+1)\*6=156 (这里有 bias 偏置，一个卷积有一个 bias)
- 计算量：(5\*5\*1+1)\*28\*28\*6=122304

也就是说 C1 卷积层，通过 6 个 5\*5 地卷积核来获取得到了 6 个不同 feature map，通过对网络结构的理解，我们可以得知需要进行运算的参数量和参数格式，在运算之初，我们需要对所有的参数进行初始化的操作，这个初始化的操作可以采取采用随机数生成的形式，也可以全部设置为 0 或者是为 1/n 平均值，n 为参数总数。

这里采用的是使用随机数的参数矩阵初始化的实现，同时也给出每次经历过梯度下降之后，对各个参数进行更新的函数实现：

#### 参数初始化函数与更新函数

```

1 # 这个要加上
2 # 权重与偏置的初始化
3 def initialize(kernel_shape, mode='Fan-in'):
4     bias_shape = (1, 1, 1, kernel_shape[-1])
5     if len(kernel_shape) == 4 else (kernel_shape[-1],)
6     if mode == 'Gaussian_dist':
7         mean, sigma = 0, 0.1 # mean: 平均值
8         weight = np.random.normal(mean, sigma, kernel_shape)
9         bias = np.ones(bias_shape) * 0.01
10    else:
11        pass
12    return weight, bias
13
14 # 更新权重
15 def update(weight, bias, dW, db, vw, vb, lr, momentum=0, weight_decay=0):

```

```

16     vw_U = momentum * vw - weight_decay * lr * weight - lr * dW
17     weight_u = weight + vw_u
18     vb_u = momentum * vb - weight_decay * lr * bias - lr * db
19     bias_u = bias + vb_u
20     return weight_u, bias_u, vw_u, vb_u

```

第一层 C1 卷积层的讲解是最复杂的，因为要涉及到许多基本的函数的实现过程，当一些基本的函数实现之后，那么后续的池化层、全连接层和输出层之中的内容就容易理解了，值得一提的是，这里为了加速矩阵的运算，充分的运用了 numpy 提供的例如 np.dot 以及 np.tensordot 的使用。

至此第一层 C1 卷积层的讲解就告一段落。

### (三) S2：池化层

在完成了 LeNet5 网络结构的第一层也就是 C1 卷积层的部分之后，接着就是 S2 池化层的实现，S2 池化层的主要作用就是对 C1 卷积层得到的 6 个 feature map 进行子采样的操作，也就是将 28\*28 大小的 feature map 转换为 14\*14 大小的 feature map。

如下是 S2 层的主要特征：

- 这里使用 2\*2 池化单元核进行降采样：2\*2 -> 14\*14\*6
- 参数量：(1+1)\*6=12（这里有 bias 偏置，一个卷积有一个 bias）
- 计算量：(2\*2+1)\*14\*14\*6=5880

我们也可以看到这其中也需要对于 bias 偏置参数进行反向传播更新，所以我们必须对池化层进行进一步的了解和探究。

池化层：它实际上是一种形式的降采样。有多种不同形式的非线性池化函数，而其中“最大池化（Max pooling）”是最为常见的。它是将输入的图像划分为若干个矩形区域，对每个子区域输出最大值。直觉上，这种机制能够有效地原因在于，在发现一个特征之后，它的精确位置远不及它和其他特征的相对位置的关系重要。池化层会不断地减小数据的空间大小，因此参数的数量和计算量也会下降，这在一定程度上也控制了过拟合。通常来说，CNN 的卷积层之间都会周期性地插入池化层。

池化层的作用：

池化操作后的结果相比其输入缩小了。池化层的引入是仿照人的视觉系统对视觉输入对象进行降维和抽象。在卷积神经网络过去的工作中，目前普遍认为池化层有如下三个功效：

#### 1. 特征不变性

池化操作是模型更加关注是否存在某些特征而不是特征具体的位置。其中不变性包括，平移不变性、旋转不变性和尺度不变性。平移不变性是指输出结果对输入对小量平移基本保持不变。

#### 2. 特征降维（下采样）

池化相当于在空间范围内做了维度约减，从而使模型可以抽取更加广范围的特征。同时减小了下一层的输入大小，进而减少计算量和参数个数。

#### 3. 在一定程度上防止过拟合，更方便优化。

#### 4. 实现非线性（类似 ReLU）。

#### 5. 扩大感受野。

目前常见的池化层有：最大池化、平均池化、全局平均池化、全局最大池化。

平均池化（average pooling）：计算图像区域的平均值作为该区域池化后的值。

最大池化（max pooling）：选图像区域的最大值作为该区域池化后的值。

这样我们已经对池化层的内容有了一定的了解，在 LeNet5 网络结构的实现过程之中，不仅实现了 max 最大池化，也内置了 average 平均池化，使用字符串的特征来表征使用哪种池化的策略。

据具体池化层的参数的维度，参数的输入以及运算流程，给出具体的池化层部分的实现代码：

池化层部分代码

```

1 def pool_backward(dA, cache, mode):
2     A_prev, hyper_parameters = cache
3     stride = hyper_parameters["stride"]
4     f = hyper_parameters["f"]
5
6     m, n_H, n_W, n_C = dA.shape # 256,14,14,6
7     m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape # 256,28,28,6
8
9     dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev)) # 256,28,28,6
10    for h in range(n_H):
11        for w in range(n_W):
12            ver_start, horiz_start = h * stride, w * stride
13            ver_end, hor_end = ver_start + f, hor_start + f
14
15            if mode == "max":
16                A_prev_slice = A_prev[:, ver_start: ver_end, hor_start:
17                    hor_end, :]
18                A_prev_slice = np.transpose(A_prev_slice, (1, 2, 3, 0))
19                mask = A_prev_slice == A_prev_slice.max((0, 1))
20                # np.tranpose和np.T的作用是一样的
21                mask = np.transpose(mask, (3, 2, 0, 1))
22                dA_prev[:, ver_start: ver_end, hor_start: hor_end, :] \
23                    += np.transpose(np.multiply(dA[:, h, w, :][:, :, np.
24                        newaxis, np.newaxis], mask), (0, 2, 3, 1))
25
26            elif mode == "average":
27                da = dA[:, h, w, :][:, np.newaxis, np.newaxis, :] #
28                    256*1*1*6
29                dA_prev[:, ver_start: ver_end, horiz_start: hor_end, :]
30                    += np.repeat(np.repeat(da, 2, axis=1), 2, axis=2)
31
32    return dA_prev

```

在 C1 卷积层运算后紧接着就是 S2 池化层运算，S2 池化层的运算使用 2\*2 核进行池化，6 个 14\*14 的特征图 (28/2=14)。根据上述代码实现，S2 这个 pooling 层是对 C1 中的 2\*2 区域内的像素求和乘以一个权值系数再加上一个偏置，将这个结果再做一次映射，这里既实现了 max 最大池化，也实现了 average 平均池化，在实验的过程之中使用的是平均池化进行的实验。

#### (四) C3: 卷积层

C3 卷积层的在前向传播算法和后向传播算法的基础是实现上基本一致，但是在整体的流程上是截然不同的，在输入层我们曾经提到过，输入层之中输入的图片的数据为深度为 1 的，但是经历了 C1 卷积层之后，深度就变为了 6，那么我们需要这几个 feature map 之间的影响，以及各自之间的相互关系，所以 C3 中的每个特征 map 是连接到 S2 中的几个 feature map 的，表示本层的 feature map 是上一层提取到的 feature map 的不同组合。

具体的连接组合可见 Lecun 教授论文中的描述：

	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 8: C3 中的 feature map 组合

具体的各个 feature map 连接计算方式可见下图：

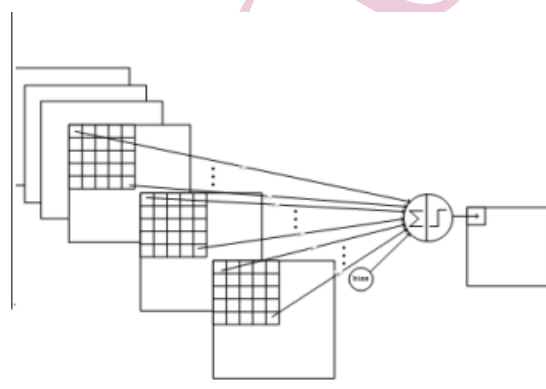


图 9: 各个 feature map 连接计算

通过上图可知，也就是将各个 feature map 与权值相乘再加上偏置，然后进行求和的操作就可以得到最后相应的在 C3 层结束之后的新的 feature map 了，从图 8 中的连接方式可以看到，连接的 map 数目从 3-6 均有，有 6 的原因是如果将所有的 map 都进行了累加，那么相当于对原图像进行了特征抽取后再一次的求取参数，并且中间还有一层池化的操作，所以 6 个连接是需要的。

根据 C3 卷积层的网络结构，我们来给出其参数量等指标：

- 采用 16 个 5\*5 卷积核进行卷积：5\*5\*16 → 10\*10\*16
- 参数量：(5\*5\*3+1)\*6 + (5\*5\*4+1)\*6 + (5\*5\*4+1)\*3 + (5\*5\*6+1)\*1 = 1516（这里有 bias 偏置，一个卷积有一个 bias）
- 计算量：1516\*10\*10=151600

根据上述的指标确定该层卷积核的维度、参数的数量以及维度，进行初始化，并且我们需要在 C1 的基础上使用字典和链表数据类型，来存储 C3 卷积层对应的 16 个卷积核，来获取 16 个新的特征图。



具体 C3 卷积层的算法代码修改如下：

#### 池化层部分代码

```

1 # C3卷积层的map连接list 链表
2 C3_map = [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 0], [5, 0, 1],
3           [0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 0], [4, 5, 0,
4             1],
5           [5, 0, 1, 2], [0, 1, 3, 4], [1, 2, 4, 5], [0, 2, 3, 5], [0, 1, 2, 3,
6             4, 5]]
7
8 // map初始化
9 for i in range(len(self.mapping)):
10     weight_shape = (kernel_shape[0], kernel_shape[1], len(self.mapping[i]
11       ), 1)
12     w, b = initialize(weight_shape, init_mode)
13     self.wb.append([w, b])
14     self.v_wb.append([np.zeros(w.shape), np.zeros(b.shape)])
15
16 def forward_propagation(self, input_map):
17     self.input_shape = input_map.shape # (n_m,14,14,6)
18     self.caches = []
19     output_maps = []
20     for i in range(len(self.mapping)):
21         output_map, cache = conv_forward(input_map[:, :, :, self.mapping[
22           i]], self.wb[i][0], self.wb[i][1], self.hyper_parameters)
23         output_maps.append(output_map)
24         self.caches.append(cache)
25     output_maps = np.swapaxes(np.array(output_maps), 0, 4)[0]
26     return output_maps
27
28 def back_propagation(self, dZ, momentum, weight_decay):
29     dA_prevs = np.zeros(self.input_shape)
30     for i in range(len(self.mapping)):
31         dA_prev, dW, db = conv_backward(dZ[:, :, :, i:i + 1], self.caches
32           [i])
33         self.wb[i][0], self.wb[i][1], self.v_wb[i][0], self.v_wb[i][1] =
34             \
35             update(self.wb[i][0], self.wb[i][1], dW, db, self.v_wb[i][0],
36               self.v_wb[i][1], self.lr, momentum, weight_decay)
37         dA_prevs[:, :, :, self.mapping[i]] += dA_prev
38     return dA_prevs

```

我们在代码实现之中添加了对于这 16 个不同链接模式的初始化已经 for 循环计算求和的内容，这样我们就能够实现 C3 的卷积层操作了，卷积核大小依然为 5\*5，第二次卷积的输出是 16 个 10x10 的特征图，卷积核大小是 5\*5。



## (五) S4: 池化层

终于在经历了 C1, S2 和 C3 这两个卷积层和一个池化层的不同连接方式之后, 迎来了一个容易实现的池化层网络结构, 在 S4 池化层的运算与连接之中, 与 S2 池化层除了参数的数目以及需要计算的特征图数目之外, 其余的部分是一模一样的, 我们只需要将 S2 复现在 S4 池化层即可。

在 S4 实现之前, 我们仍需要知道该层网络的规模:

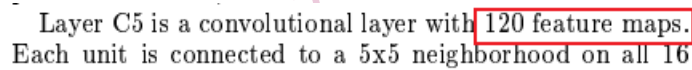
- 这里使用  $2 \times 2$  池化单元核进行降采样:  $2 \times 2 \rightarrow 5 \times 5 \times 16$
- 参数量:  $(1+1) \times 16 = 32$  (这里有 bias 偏置, 一个卷积有一个 bias)
- 计算量:  $(2 \times 2 + 1) \times 5 \times 5 \times 16 = 122304$

根据网络结构及其规模, 我们可以知道 S4 pooling 层窗口大小仍然是  $2 \times 2$ , 输入共计 16 个 feature map, 同样的输出 16 个 feature map, C3 卷积层的 16 个  $10 \times 10$  的图分别进行以  $2 \times 2$  为单位的池化得到 16 个  $5 \times 5$  的特图。

## (六) C5: 卷积层

对于这一层而言, 虽然我们还是能够将 C1 层的网络结构应用到 C5 层上, 这样就能够实现 C5 层的网络结构功能, 但是这一层又同时和全连接层很像, 因为 C5 层之中, 是 16 个  $5 \times 5$  的 feature map 中的每一个数据都会与 C5 层的输出的 120 个 feature map 特征图进行连接, 所以具有一定的全连接层的特点。

我们可以从 LeNet5 的原论文之中得到相关信息:



Layer C5 is a convolutional layer with 120 feature maps.  
Each unit is connected to a  $5 \times 5$  neighborhood on all 16

图 10: C5 层网络结构

这样我们就能够得到 C5 层网络结构的规模了:

- 每个  $5 \times 5$  特征图均与 120 个输出的特征相连接:  $5 \times 5 \times 16 \rightarrow 1 \times 120$
- 参数量:  $(5 \times 5 \times 16 + 1) \times 120 = 48120$
- 计算量:  $(5 \times 5 \times 16 + 1) \times 120 = 48120$

我们在实际实现的过程之中, 这一层的实现是基于 C1 层卷积层的实现, 与 C1 层中的内容没有什么变化, 只需要将卷积核的维度和输出的维度修改为 (5, 5, 16, 120) 即可, 在反向传播等算法的计算上, C5 层与 C1 层的实现是一样的。

## (七) F6: 全连接层

全连接层相对于卷积层的运算实现而言, 简便了不少, 我们还是对全连接层先做一个简单的介绍:

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” (下面会讲到这个分布式特征) 映射到样本标记空间的作用。在实际使用中, 全连接层可由卷积操作实现:

对前层是全连接的全连接层可以转化为卷积核为  $1 \times 1$  的卷积; 而前层是卷积层的全连接层可以转化为卷积核为  $h \times w$  的全局卷积,  $h \times w$  分别为前层卷积结果的高和宽。

如果把其当作卷积核为  $1 \times 1$  的卷积, 那么计算过程就不会那么复杂了, 我们先给出 F6 层的网络结构:

- 120 维输入向量与 84 维输出向量全连接：120 -> 84
- 参数量： $(1*1*120+1)*84=10164$
- 计算量： $(1*1*120+1)*84=10164$

F6 层通过计算输入向量和权重向量之间的点积，加上一个偏置 bias，结果通过 sigmoid 函数输出。C5 层和 F6 层的计算量明显小于 C1 和 C3 卷积层，我们对 C1 层的网络结构进行简化就能够得到 F6 层的实现代码了：

全连接层的各个算法代码

```

1  def forward_propagation(self, input_array):
2      self.input_array = input_array # (n_m, 120)
3      return np.matmul(self.input_array, self.weight) # (n_m, 84)
4
5  def back_propagation(self, dZ, momentum, weight_decay):
6      # (256, 84) * (84, 120) = (256, 120) (n_m, 84) * (84, 120) = (n_m, 120)
7      dA = np.matmul(dZ, self.weight.T)
8      db = np.sum(dZ.T, axis=1) # (84,)
9      dW = np.matmul(self.input_array.T, dZ)
10     self.weight, self.bias, self.v_w, self.v_b = update(self.weight, self.
        bias, dW, db, self.v_w, self.v_b, self.lr, momentum, weight_decay)
11     return dA

```

## (八) Output: 输出层

其实 Output 层也是全连接层的运算模式，相当于从 84 维特征向量计算出 10 维的输出数据，根据输出的结果判断是 0-9 的哪一个数字，只不过这一层采取的是径向基函数的连接方式，这里就解释了为什么最终 F6 层得到的输出为 84 维的特征向量，因为我们将数字 0-9 都采取了  $12*7$  大小的矩阵来进行表示，分别使用 1 和 -1 来表示灰度，也就是使用 bitmap 的形式来表示 0-9 这 10 个数字，具体的表示形式如下：

bitmap 表示 0-9 数字

```

1  # 初始化bitmap，为了能够更加快速的计算最后一层与实际数字的误差
2  bitmap[0] = np.array([
3      [-1, +1, +1, +1, +1, +1, -1],
4      [-1, -1, -1, -1, -1, -1, -1],
5      [-1, -1, +1, +1, +1, -1, -1],
6      [-1, +1, +1, -1, +1, +1, -1],
7      [+1, +1, -1, -1, -1, +1, +1],
8      [+1, +1, -1, -1, -1, +1, +1],
9      [+1, +1, -1, -1, -1, +1, +1],
10     [+1, +1, -1, -1, -1, +1, +1],
11     [-1, +1, +1, -1, +1, +1, -1],
12     [-1, -1, +1, +1, +1, -1, -1],
13     [-1, -1, -1, -1, -1, -1, -1],
14     [-1, -1, -1, -1, -1, -1, -1]
15 ])

```

接下来仍然给出最后一层网络结构输出层的网络结构：

- 84 维输入向量与 10 维输出向量全连接：84 -> 10
- 参数量：84\*10=840
- 计算量：84\*10=840

bitmap 的下标 index 就表示着对应的数字，例如 bitmap[0]，我们就能够从 1 和-1 之中看到模糊的 0 的形状，这也是 F6 层输出需要 84 维特征向量的原因，通过计算与 0-9 的 bitmap 之间的相似度，来对图像进行判别分类。

而 RBF 输出的计算方式，具体如下：

$$y_i = \sum_j (x_j - w_{i,j})^2 \quad (1)$$

输出层计算得到的输出的值越接近于 0，则越接近于 i，即越接近于 i 的编码图，表示当前网络输入的识别结果是字符 i。我们的代码实现就是根据上式的计算过程实现的，具体的输出层计算代码如下：

输出层判别 0-9 分类

```

1  def forward_propagation(self, input_array, label, mode):
2      if mode == 'train':
3          self.input_array = input_array
4          # (n_m, 84) 标签权重
5          self.weight_label = self.weight[label, :]
6          loss = 0.5 * np.sum(np.power(input_array - self.weight_label, 2),
7                                axis=1, keepdims=True)
8          return np.sum(np.squeeze(loss))
9      if mode == 'test':
10         subtract_weight = (
11             input_array[:, np.newaxis, :] - np.array([self.weight] *
12                                                         input_array.shape[0])) # (n_m,10,84)
13         # 分成10类
14         rbf_class = np.sum(np.power(subtract_weight, 2), axis=2)
15         class_pred = np.argmin(rbf_class, axis=1)
16         error01 = np.sum(label != class_pred)
17         return error01, class_pred

```

至此我们已经成功地实现了 LeNet5 的整体的网络结构，我们已经能够完成具体的数据集训练以及测试等各个步骤过程。

## (九) 一些参数说明

在模型的训练过程之中采用了 Lecun 教授原文中的训练轮数以及学习率：

epochs : 16

learning rate : 0.0005

同时我们还可以考虑深度学习的一些策略，例如深度学习中的 momentum 动量，momentum 动量能够在一定程度上解决局部最优的问题。momentum 动量是依据物理学的势能与动能之间能量转换原理提出来的。当 momentum 动量越大时，其转换为势能的能量也就越大，就越

有可能摆脱局部凹域的束缚，进入全局凹域。momentum 动量主要用在权重更新的时候。在我们的训练之中，将 momentum 设置为了 0.9。

同时我们还可以考虑权重衰减和学习率衰减等策略，但是在我们的训练之中，这两部分均设置为了 0，也就是并未设置这方面的策略。权重衰减（weight decay）是在损失函数中放在正则项前的一个系数，正则项一般指示模型的复杂度，所以 weight decay 的作用是调节模型复杂度对损失函数的影响，若 weight decay 很大，则复杂的模型损失函数的值也就大。

由于并没有对模型的训练过程进行最有效的并行化效率提升，所以我们想要训练一次测试一次的结果或者说调试是十分困难的，一般来说 10 轮的训练都会消耗接近 1 小时多的时间，所以我们可以使用 pickle 格式存储模型之中的参数也就是来存储模型，pickle 部分代码如下：

#### pkl 存储模型

```
1 # 使用 pickle 格式存储模型，也就是存储模型中的各个参数数据
2     if(epoch == epochs - 1):
3         with open('model_data.pkl', 'wb') as output:
4             pickle.dump(LeNet5, output, pickle.HIGHEST_PROTOCOL)
5     else:
6         pass
```

存储模型后，发现每一轮训练得到的模型都有 12 个 G 左右的大小，所以只会存储最后一轮训练得到的模型结果，但是由于模型的文件太大，所以很难进行传输。

## 六、 模型结果分析

我们已经成功完成了网络结构的组织以及参数的设置，那么此时我们就可以对模型进行训练来看看模型的结果如何。经历了 16 轮的训练，每轮的训练时长接近 5-6 分钟，可以得到模型的结果如下：

```
----- epo 16 begin -----
Global learning rate: 0.001
Learning rates in layers: [2.45004419e-04 2.08575514e-06 9.99020653e-07 6.38026014e-08]
Batch size: 256
Cost of epo 16 : 61679.83564108807
error number sum of training set: 757 / 60000
Time used: 425.18139576911926 second
----- epo 16 end -----

Finished training, the total training time is 6809.933736562729 second

Start testing...
Error rate: 0.0138
Finished testing, the accuracy is 0.9862
```

图 11: 模型测试结果

我们可以发现经过 16 轮的训练之后，对于 train 训练集而言，总计 60000 个训练图像，训练集上分类错误数为 757 个，也就是说在训练集上的准确率达到了  $(1-757/60000)=0.9874$ ，达到了 98.74% 的训练集准确率，在测试集上的准确率达到了 98.62% 的测试集准确率，模型的训练结果是不错的。

这里就不再统计准确率、F1 score 和混淆矩阵等评价指标了（想用第三方库函数计算...）。

对于每轮训练的 Error rate 也进行了统计，我们可以使用 matplotlib 来画图进行展示，也可以将每轮的 Error rate 进行存储统计，然后使用 Word 文档进行画图。matplotlib 部分的代码如下：

matplotlib 绘图

```

1 # 可以保留图像展示，错误率图片
2 x = np.arange(epochs)
3 plt.xlabel('epochs')
4 plt.ylabel('Error rate')
5 plt.plot(x, error_rate_list[0])
6 plt.plot(x, error_rate_list[1])
7 plt.legend(['training data', 'testing data'], loc='upper right')
8 plt.show()

```

最终通过绘图得到的错误率图形化的结果如下：

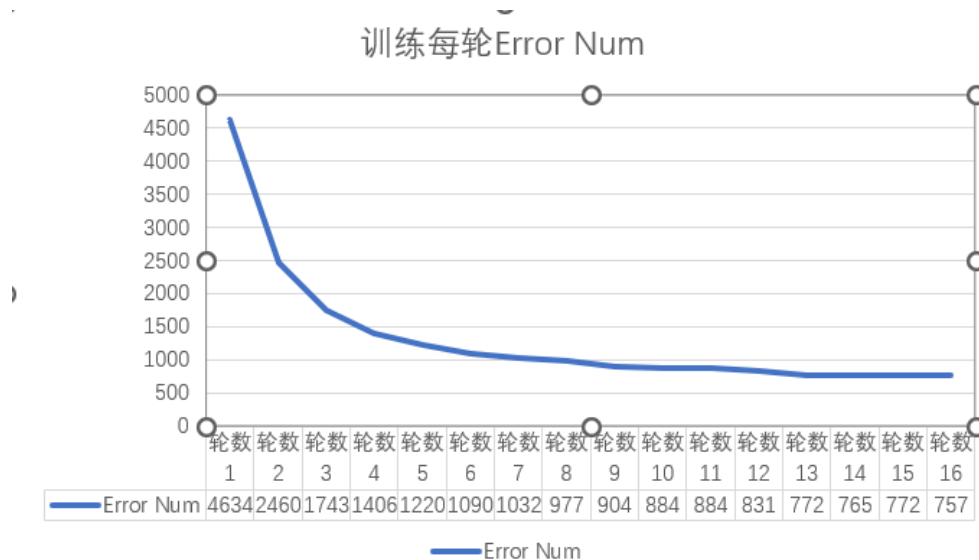


图 12: Error 数目变化曲线

从图中的数据表格之中我们也不难发现，其实在训练到 13 轮左右的时候，Error rate 还出现了回升的现象，所以这里推测在训练到 13 轮左右时，其实模型的训练就已经足够了，再多轮训练可能会造成过拟合的现象，下图的 Log 信息也能显示 Error rate 回升的现象：

```

Learning rates in layers: [3.00769353e-04 2.09552487e-06 1.03535703e-06 6.44458228e-08]
Batch size: 256
Cost of epo 13 : 64059.97482415088
error number sum of training set: 772 / 60000
Time used: 415.31020855903625 second
----- epo 13 end -----

----- epo 14 begin -----
Global learning rate: 0.001
Learning rates in layers: [2.70766566e-04 2.22522742e-06 1.04231387e-06 6.45055950e-08]
Batch size: 256
Cost of epo 14 : 62539.41752816466
error number sum of training set: 765 / 60000
Time used: 420.30223536491394 second
----- epo 14 end -----

----- epo 15 begin -----
Global learning rate: 0.001
Learning rates in layers: [2.54643518e-04 2.08769753e-06 1.00059760e-06 6.38487122e-08]
Batch size: 256
Cost of epo 15 : 62056.35985518823
error number sum of training set: 772 / 60000
Time used: 418.2652246952057 second
----- epo 15 end -----

```

图 13: Error 数目变化曲线

模型的结果分析就到此结束，接下来将会介绍一些实际实现时使用或者尝试模型优化策略。

## 七、 模型优化策略

因为模型的训练性能的提升和模型训练效果的提升都是很重要的一部分，同样也是十分困难的，接下来这一部分之中将会主要介绍实际实现过程之中使用到的或者进行尝试的优化策略。

### (一) batch 随机梯度下降

因为我们在实际的训练过程之中每轮的训练都需要 5-6 分钟，训练 10 轮就需要一个小时，这样显然是很耗费时间的，这时我们想到其实我们在训练之中是使用了全部的 60000 个数据来进行的训练过程，这样虽然能够保持较快的梯度下降速度，但是十分消耗时间，所以我们想到可以使用批量随机梯度下降来进行优化。

实现的具体的批量随机梯度下降的代码如下：

#### batch 随机梯度下降

```

1 def random_batches(image, label, random_batch_size=256, one_batch=False):
2     m = image.shape[0] # 60000 个训练数据
3     random_batches = []
4
5     # 获取随机的1-60000的随机排序
6     permutation = list(np.random.permutation(m))
7     random_label = label[permutation]
8     random_image = image[permutation, :, :, :]
9
10    # 可以选择只取一个batch进行训练
11    if one_batch:
12        random_batch_image = random_image[0: random_batch_size, :, :, :]

```

```

13     random_batch_label = random_label[0: random_batch_size]
14     return random_batch_image, random_batch_label
15
16     # 可以看看能够分出多少个分组，可以选择多个分组来进行训练
17     # TODO
18     num_complete_randombatches = math.floor(m / random_batch_size)
19     for k in range(0, num_complete_randombatches):
20         random_batch_image = random_image[k * random_batch_size: k *
21             random_batch_size + random_batch_size, :, :, :]
22         random_batch_label = random_label[k * random_batch_size: k *
23             random_batch_size + random_batch_size]
24         random_batch = (random_batch_image, random_batch_label)
25         random_batches.append(random_batch)
26
27     # 数据分batch的时候，要注意最后一个batch中的数目
28     if m % random_batch_size != 0:
29         random_batch_image = random_image[num_complete_randombatches *
30             random_batch_size: m, :, :, :]
31         random_batch_label = random_label[num_complete_randombatches *
32             random_batch_size: m]
33         random_batch = (random_batch_image, random_batch_label)
34         random_batches.append(random_batch)
35
36     return random_batches

```

我们可以选择只使用一个 batch 来进行训练，我们也能够设置每一个 batch 的大小，同样的我们能够选择多组 batch 来进行训练，或者我们可以直接选全部的数据进行训练，这里使用多组 batch 进行训练得到的模型的训练结果如下：

```

----- epo 16 begin -----
Global learning rate: 0.0001
Learning rates in layers: [4.91369636e-07 2.51805188e-08 2.04810505e-08 8.93716601e-09]
Batch size: 512
Cost of epo 16 : 286488.97705043684
error number sum of training set: 3130 / 60000
Time used: 381.4684932231903 second
----- epo 16 end -----

```

图 14: 多组 batch 训练模型结果

可以看到同样进行 16 轮的训练，虽然每轮的训练时间能够减少 1/4 左右，但是 16 轮训练之后的在训练集上的准确率仅仅有 94.78%，也可以看出随机梯度下降对模型的效果还是有影响的。

## (二) L-M 方法学习率优化

这是一个关注于学习率和步长来进行的优化，我们都知道在进行参数学习的过程之中需要学习率来参与梯度下降的计算过程，有可能在梯度下降的图像之中某一步不需要很长最合适，这时如果有一个较大的学习率的话就会下降过度，而如果步长一直很小有可能导致训练多轮效果不显著，L-M 就是基于这方面的优化。

L-M 算法属于信赖域法，将变量行走的长度控制在一定的信赖域之内，保证泰勒展开有很好



的近似效果。L-M 算法使用了一种带阻尼的高斯-牛顿方法。L-M 全称 Levenberg-Marquardt, 相比于高斯-牛顿法增加了控制步长的逻辑, 是高斯牛顿法和梯度下降法的结合。牛顿法可以求得可以求得更新的步长, 而高斯-牛顿法在计算之中使用了近似的方法, 计算简单, 但是高斯-牛顿法没有控制步长的逻辑, 对数据较为敏感。L-M 方法就是在高斯-牛顿法之上添加了控制步长的逻辑。

具体实现的代码如下:

matplotlib 绘图

```

1 def conv_SDLM(dZ, cache):
2     (A_prev, W, b, hparameters) = cache
3     (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
4     (f, f, n_C_prev, n_C) = W.shape
5     stride = hparameters["stride"]
6     pad = hparameters["pad"]
7     (m, n_H, n_W, n_C) = dZ.shape
8     dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
9     dW = np.zeros((f, f, n_C_prev, n_C))
10    db = np.zeros((1, 1, 1, n_C))
11
12    if pad != 0:
13        A_prev_pad = zero_pad(A_prev, pad)
14        dA_prev_pad = zero_pad(dA_prev, pad)
15    else:
16        A_prev_pad = A_prev
17        dA_prev_pad = dA_prev
18
19    for h in range(n_H): # loop over vertical axis of the output volume
20        for w in range(n_W): # loop over horizontal axis of the output
21            # 使用最近的切片来进行
22            vert_start, horiz_start = h * stride, w * stride
23            vert_end, horiz_end = vert_start + f, horiz_start + f
24
25            A_slice = A_prev_pad[:, vert_start:vert_end, horiz_start:
26                                horiz_end, :]
27
28            # 更新窗口和过滤器的参数
29            dA_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :] +=
30                np.transpose(
31                    np.dot(np.power(W, 2), dZ[:, h, w, :].T), (3, 0, 1, 2))
32            dW += np.dot(np.transpose(np.power(A_slice, 2), (1, 2, 3, 0)), dZ
33                       [:, h, w, :])
34
35    # 设置dA_prev变量去更新dA_prev_pad
36    dA_prev = dA_prev_pad if pad == 0 else dA_prev_pad[:, pad:-pad, pad:-pad,
37        :]
38    assert (dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))
39    return dA_prev, dW

```



在实现了 L-M 方法之后，我们就能够发现在训练之中每轮的学习率步长也是会进行更新的，来通过该方法计算出较为适合下一轮迭代的学习率：

```
----- epo 2 begin -----
Global learning rate: 0.05
Learning rates in layers: [1.17360006e-04 3.41528652e-05 1.66148146e-05 5.43838522e-06]
Batch size: 256
Cost of epo 2 : 247200.80481367328
error number sum of training set: 2460 / 60000
Time used: 418.8085300922394 second
----- epo 2 end -----

----- epo 3 begin -----
Global learning rate: 0.02
Learning rates in layers: [2.65798873e-04 1.85970176e-05 8.98026918e-06 1.51350885e-06]
Batch size: 256
Cost of epo 3 : 160291.20272993896
error number sum of training set: 1743 / 60000
Time used: 414.7109627723694 second
----- epo 3 end -----

----- epo 4 begin -----
Global learning rate: 0.02
Learning rates in layers: [6.56485707e-04 2.09346241e-05 1.10642325e-05 1.43770807e-06]
Batch size: 256
Cost of epo 4 : 131248.6746929756
```

图 15: L-M 方法更新步长

### (三) 不同的激活函数

在实现之中，给出了 sigmoid、tanh、ReLU、Leaky ReLU 和 ELU 等激活函数的内容，包括其导数内容，具体代码如下：

matplotlib 绘图

```
1 # Leaky ReLU函数，对负值的输入有很小的坡度
2 def LReLU(x, a=alpha["lrelu"]):
3     return np.where(x > 0, x, a * x)
4
5 def d_LReLU(x, a=alpha["lrelu"]):
6     return np.where(x > 0, 1, a)
7
8 def d2_LReLU(x, a=alpha["lrelu"]):
9     return np.zeros(d_PReLU(x, a).shape)
10
11 # ELU函数，输入负数是有一定的输出的
12 def ELU(x, a=alpha["elu"]):
13     return np.where(x > 0, x, a * (np.exp(x) - 1))
14
15 def d_ELU(x, a=alpha["elu"]):
16     return np.where(x > 0, 1, ELU(x, a) + a)
17
18 def d2_ELU(x, a=alpha["elu"]):
19     return np.where(x > 0, 0, ELU(x, a) + a)
```

其中 Leaky ReLU 函数和 ELU 都是对原本的 ReLU 函数在小于 0 来进行导数部分优化的，但是因为每次训练测试一次都需要 60-90 分钟，测试时间过长，所以并没有对所有的激活函数进行测试，这里使用的是字符串与函数对应的字典关系，来对网络结构之中的激活函数进行设置。

#### (四) 性能优化尝试

在对不同的策略进行了尝试与优化后，我们需要考虑的部分就是训练的性能优化问题，训练部分的性能问题主要问题就是多轮的 for 循环计算，我们已经使用了 numpy 库函数中的如矩阵相乘 dot 和 tensordot 等函数保证了矩阵运算的性能，主要的问题就是训练过程之中的 for 循环的优化。

在针对训练过程之中的 for 循环计算基于全体样本的梯度下降值，以下给出三种方案。

第一种方案就是将 numpy 中的 ndarray 结构转化为 dataframe 结构，看能否使用 iterrows 等迭代器来加速处理，但是 dataframe 中的迭代器多是对自身中的元素进行操作，而对于两个矩阵的相乘操作较难处理。

第二种方案就是使用 numba 第三方库，来对 python 语句进行中间代码级别的优化，但是在使用 numba 库的时候遇到了间接调用 numpy 库函数报错的情形，这一点很难处理只能等待该第三方库的更新去解决这个问题，这个第三方库能够提供基于中间代码层面的性能优化。

第三种方案就是实现矩阵的分块并行化计算，想要真正实现 for 循环的大幅度优化，需要手动编写相适应的并行算法，这个需要坚实的多线程与并行计算的功底，这一点仍需要继续学习探究。

## 八、 总结

在本次实验之中，深入地了解 LeNet5 的网络结构，在经典数据集 MNIST 上手动实现了 LeNet5 卷积神经网络模型，LeNet5 是一种用于手写体字符识别的非常高效的卷积神经网络。卷积神经网络能够很好的利用图像的结构信息。在实现中尝试了不同的策略来优化模型的准确率，也探索学习到了不同的性能优化方案。

个人 Github 链接 [Machine Learning](#)

## 参考文献

- [1] <http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>
- [2] [https://blog.csdn.net/Keep\\_Trying\\_Go/article/details/123976153](https://blog.csdn.net/Keep_Trying_Go/article/details/123976153)
- [3] <https://github.com/mattwang44/LeNet-from-Scratch>
- [4] <https://zhuanlan.zhihu.com/p/179293801>
- [5] <https://arxiv.org/abs/1807.02176>
- [6] <https://blog.csdn.net/liu14lang/article/details/53991897>
- [7] <https://blog.csdn.net/bitcarmanlee/article/details/78819025>
- [8] [https://blog.csdn.net/qq\\_31820761/article/details/102588339](https://blog.csdn.net/qq_31820761/article/details/102588339)
- [9] <https://blog.csdn.net/happyorg/article/details/78274066>

NIJU