南开大学
Nankai University

南　　开　　大　　学

计算机与网络空间安全学院

机器学习实验报告

实验三：LeNet5

2011763 黄天昊

年级：2020 级

指导教师：谢晋

2022 年 12 月 31 日

摘要

在本次练习中，使用 Python 手动搭建 LeNet5 来实现对 MNIST 数据集中 0-9 共 10 个手写数字的分类。

关键字：LeNet5

目录

一、 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。

二、 实验环境

- 操作系统：Windows 10 专业版
- 编程语言：Python 3.9
- IDE：PyCharm

三、 LeNet5 网络结构

LeNet5 是一个较简单的卷积神经网络。下图显示了其结构：输入的二维图像，先经过两次卷积层到池化层，再经过全连接层，最后使用 softmax 分类作为输出层。如图1

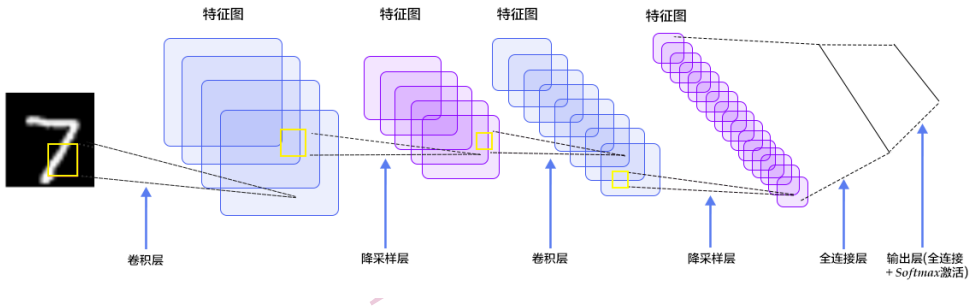

图 1: Lenet5 网络结构

下面我们主要介绍卷积层和池化层。

(一) 卷积层

图2给出了一个卷积计算过程的示例图，输入图像大小为 H=5,W=5,D=3，即 5×5 大小的 3 通道（RGB，也称作深度）彩色图像。这个示例图中包含两（用 K 表示）组卷积核，即图中滤波器 W0 和 W1。在卷积计算中，通常对不同的输入通道采用不同的卷积核，如图示例中每组卷积核包含（D=3）个 3×3（用 F×F 表示）大小的卷积核。另外，这个示例中卷积核在图像的水平方向（W 方向）和垂直方向（H 方向）的滑动步长为 2（用 S 表示）；对输入图像周围各填充 1（用 P 表示）个 0，即图中输入层原始数据为蓝色部分，灰色部分是进行了大小为 1 的扩展，用 0 来进行扩展。经过卷积操作得到输出为 3×3×2（用 Ho×Wo×K 表示）大小的特征图，即 3×3 大小的 2 通道特征图，其中 Ho 计算公式为：Ho=(H−F+2×P)/S+1，Wo 同理。

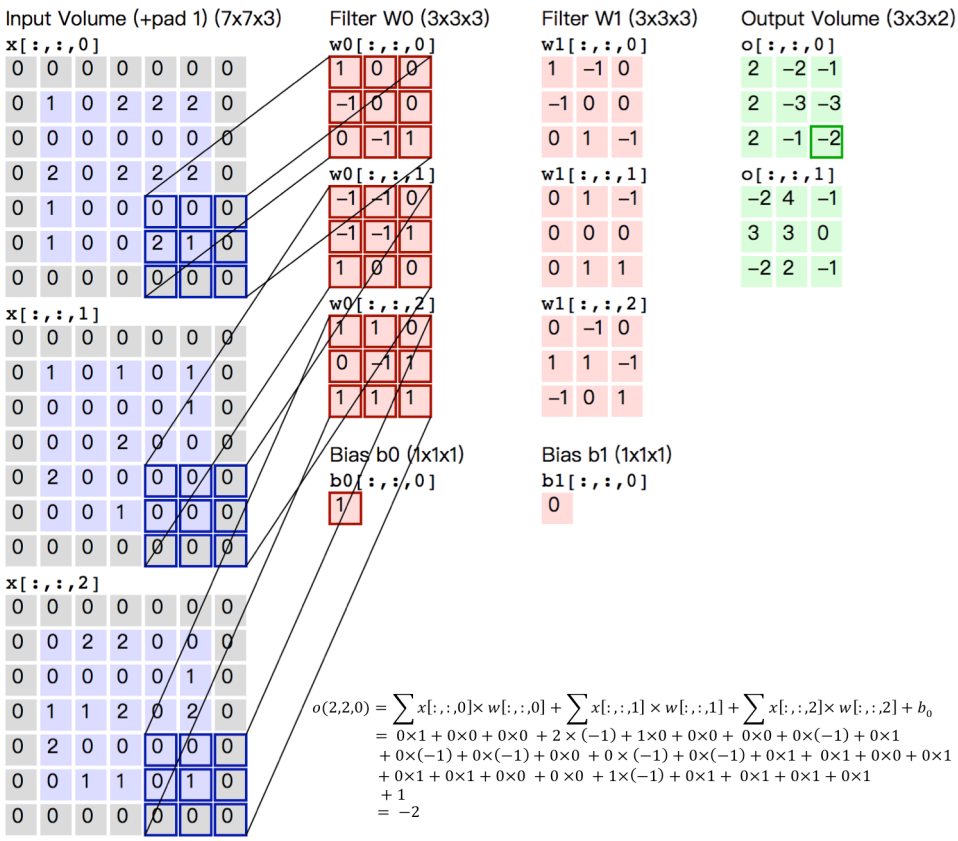

图 2: 卷积计算示例图

而输出特征图中的每个像素，是每组滤波器与输入图像每个特征图的内积再求和，再加上偏置 bo，偏置通常对于每个输出特征图是共享的。输出特征图 o[:,:,0] 中的最后一个 −2 计算如图2右下角公式所示。

关于卷积的符号表示如下：

- H：图片高度
- W：图片宽度；
- D：原始图片通道数，也是卷积核个数；
- F：卷积核高宽大小；
- P：图像边扩充大小；
- S：滑动步长。

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

- 局部连接:每个神经元仅与输入神经元的一块区域连接,这块局部区域称作感受野(receptive field)。在图像卷积操作中，即神经元在空间维度（spatial dimension，即上图示例 H 和 W 所在的平面）是局部连接，但在深度上是全部连接。对于二维图像本身而言，也是局部像

素关联较强。这种局部连接保证了学习后的过滤器能够对于局部的输入特征有最强的响应。局部连接的思想，也是受启发于生物学里面的视觉系统结构，视觉皮层的神经元就是局部接受信息的。

- 权重共享：计算同一个深度切片的神经元时采用的滤波器是共享的。例上图中计算 o[:,:,0] 的每个每个神经元的滤波器均相同，都为 W0，这样可以很大程度上减少参数。共享权重在一定程度上讲是有意义的，例如图片的底层边缘特征与特征在图中的具体位置无关。但是在一些场景中是无意的，比如输入的图片是人脸，眼睛和头发位于不同的位置，希望在不同的位置学到不同的特征。注意权重只是对于同一深度切片的神经元是共享的，在卷积层，通常采用多组卷积核提取不同特征，即对应不同深度切片的特征，不同深度切片的神经元权重是不共享。另外，偏重对同一深度切片的所有神经元都是共享的。

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

(二) 池化层

池化是非线性下采样的一种形式，主要作用是通过减少网络的参数来减小计算量，并且能够在一定程度上控制过拟合。通常在卷积层的后面会加上一个池化层。池化包括最大池化、平均池化等。其中最大池化是用不重叠的矩形框将输入层分成不同的区域，对于每个矩形框的数取最大值作为输出层，如下图3所示。

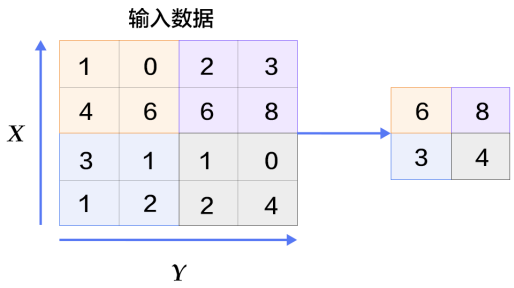

图 3: 池化计算示例图

(三) LeNet5

LeNet5 共有 7 层，不包含输入，每层都包含可训练参数；每个层有多个 Feature Map，每个 Feature Map 通过一种卷积滤波器提取输入的一种特征，然后每个 Feature Map 有多个神经元。

1. **Input 层-输入层**

   首先是数据 INPUT 层，输入图像的尺寸统一归一化为 32 * 32。

2. **C1 层-卷积层**

   输入图片：32 * 32
   卷积核大小：5 * 5
   卷积核种类：6
   输出 featuremap 大小：28 * 28 (32 - 5 + 1 = 28)

神经元数量：28 * 28 * 6
可训练参数：(5 * 5 + 1) * 6
(每个滤波器 5 * 5 = 25 个 unit 参数和一个 bias 参数，一共 6 个滤波器)
连接数：(5 * 5 + 1)* 6 * 28 * 28 = 122304
详细说明：对输入图像进行第一次卷积运算（使用 6 个大小为 5*5 的卷积核），得到 6 个 C1 特征图（6 个大小为 28*28 的 feature maps, 32-5+1=28)。我们再来看看需要多少个参数，卷积核的大小为 5*5，总共就有 6*(5*5+1)=156 个参数，其中 +1 是表示一个核有一个 bias。对于卷积层 C1，C1 内的每个像素都与输入图像中的 5*5 个像素和 1 个 bias 有连接，所以总共有 156*28*28=122304 个连接 (connection)。有 122304 个连接，但是我们只需要学习 156 个参数，主要是通过权值共享实现的。

3. **S2 层-池化层**

输入：28 * 28
采样区域：2 * 2
采样方式：4 个输入相加, 乘以一个可训练参数, 再加上一个可训练偏置。结果通过 sigmoid 激活函数
采样种类：6
输出 featureMap 大小：14 * 14 (28 / 2)
神经元数量：14 * 14 * 6
连接数：(2 * 2 + 1) * 6 * 14 * 14
S2 中每个特征图的大小是 C1 中特征图大小的 1/4。
详细说明：第一次卷积之后紧接着就是池化运算，使用 2*2 核进行池化，于是得到了 S2，6 个 14*14 的特征图（28/2=14)。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，然后将这个结果再做一次映射。同时有 5x14x14x6=5880 个连接。

4. **C3 层-卷积层**

输入：S2 中某几个特征 map 组合
卷积核大小：5 * 5
卷积核种类：16
输出 featureMap 大小：10 * 10 (14 - 5 + 1) = 10
C3 中的每个特征 map 是连接到 S2 中的所有 6 个或者几个特征 map 的，表示本层的特征 map 是上一层提取到的特征 map 的不同组合存在的一个方式是：C3 的前 6 个特征图以 S2 中 3 个相邻的特征图子集为输入。接下来 6 个特征图以 S2 中 4 个相邻特征图子集为输入。然后的 3 个以不相邻的 4 个特征图子集为输入。最后一个将 S2 中所有特征图为输入。则：可训练参数：6*(3*5*5+1)+6*(4*5*5+1)+3*(4*5*5+1)+1*(6*5*5+1)=1516
连接数：10*10*1516=151600
详细说明：第一次池化之后是第二次卷积，第二次卷积的输出是 C3，16 个 10x10 的特征图，卷积核大小是 5*5. 我们知道 S2 有 6 个 14*14 的特征图，怎么从 6 个特征图得到 16 个特征图? 这里是通过对 S2 的特征图特殊组合计算得到的 16 个特征图。具体如下：

	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 4: 计算方式

C3 的前 6 个 feature map（对应上图第一个红框的 6 列）与 S2 层相连的 3 个 feature map 相连接（上图第一个红框），后面 6 个 feature map 与 S2 层相连的 4 个 feature map 相连接（上图第二个红框），后面 3 个 feature map 与 S2 层部分不相连的 4 个 feature map 相连接，最后一个与 S2 层的所有 feature map 相连。卷积核大小依然为 5*5，所以总共有 6*(3*5*5+1)+6*(4*5*5+1)+3*(4*5*5+1)+1*(6*5*5+1)=1516 个参数。而图像大小为 10*10，所以共有 151600 个连接。

C3 与 S2 中前 3 个图相连的卷积结构如下图所示:

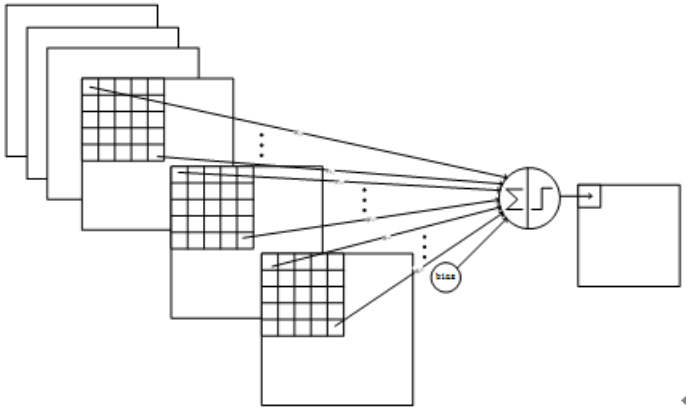

图 5: 具体计算方法

5. **S4 层-池化层**

输入：10 * 10
采样区域：2 * 2
采样方式：4 个输入相加，乘以一个可训练参数，再加上一个可训练偏置。结果通过 sigmoid 激活函数
采样种类：16
输出 featureMap 大小：5 * 5 (10 / 2)
神经元数量：5 * 5 * 16 = 400
连接数：16 *(2 * 2 + 1) * 5 * 5 = 2000
S4 中每个特征图的大小是 C3 中特征图大小的 1/4
详细说明：S4 是 pooling 层，窗口大小仍然是 2*2，共计 16 个 feature map，C3 层的 16 个 10x10 的图分别进行以 2x2 为单位的池化得到 16 个 5x5 的特征图。有 5x5x5x16=2000 个连接。连接的方式与 S2 层类似。

6. **C5 层-卷积层**

   输入：S4 层的全部 16 个单元特征 map（与 s4 全相连）
   卷积核大小：5*5
   卷积核种类：120
   输出 featureMap 大小：1 * 1 (5 - 5 + 1)
   可训练参数/连接：120 * (16 * 5 * 5 + 1)= 48120
   详细说明：C5 层是一个卷积层。由于 S4 层的 16 个图的大小为 5x5，与卷积核的大小相同，所以卷积后形成的图的大小为 1x1。这里形成 120 个卷积结果。每个都与上一层的 16 个图相连。所以共有 (5x5x16+1)x120 = 48120 个参数，同样有 48120 个连接。

7. **F6 层-全连接层**

   输入：c5 120 维向量
   计算方式：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。
   可训练参数:84 * (120 + 1) = 10164
   详细说明：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。

8. **Output 层-全连接层**

   Output 层也是全连接层，共有 10 个节点，分别代表数字 0 到 9，且如果节点 i 的值为 0，则网络识别的结果是数字 i。采用的是径向基函数（RBF）的网络连接方式。假设 x 是上一层的输入，y 是 RBF 的输出，则 RBF 输出的计算方式是：

$$y_i = \sum_j (x_j - w_{ij})^2$$

   上式 w_ij 的值由 i 的比特图编码确定，i 从 0 到 9，j 取值从 0 到 7*12-1。RBF 输出的值越接近于 0，则越接近于 i，即越接近于 i 的 ASCII 编码图，表示当前网络输入的识别结果是字符 i。该层有 84x10=840 个参数和连接。

9. **总结**

   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。卷积层的参数较少，这也是由卷积层的主要特性即局部连接和共享权重所决定。

四、 代码细节

main 函数

```
# get the data
train_images, train_labels, test_images, test_labels = load_data()
print("Got data...\n")

# data processing, normalization&zero-padding
print("Normalization and zero-padding...")
```

```
train_images = normalize(zero_pad(train_images[:, :, :, np.newaxis], 2), '
    LeNet5')
test_images = normalize(zero_pad(test_images[:, :, :, np.newaxis], 2), '
    LeNet5')
print("The shape of training image with padding: ", train_images.shape)
print("The shape of testing image with padding: ", test_images.shape)
print("Finish data processing...\n")

# train LeNet5
LeNet5 = LeNet5()
print("Start training...")
start_time = time.time()
train(LeNet5, train_images, train_labels)
end_time = time.time()
print("Finished training, the total training time is {}s \n".format(end_time
    - start_time))

# read model
# with open('model_data_13.pkl', 'rb') as input_:
#     LeNet5 = pickle.load(input_)

# evaluate on test dataset
print("Start testing...")
error01, class_pred = LeNet5.Forward_Propagation(test_images, test_labels, '
    test')
print("error rate:", error01 / len(class_pred))
print("Finished testing, the accuracy is {} \n".format(1 - error01 / len(
    class_pred)))
```

mian 函数中，首先是对数据的读取和正则化，由于 mnist 数据集中的图片为 28 * 28，需要 padding 到 32 * 32。

然后，将处理之后的数据输入 train 函数，开始训练。

训练完成后，在测试集上测试训练网络的性能。

LeNet5 网络结构类

```
class LeNet5(object):
    """
    C1 -> S2 -> C3 -> S4 -> C5 -> F6 -> Output

    Reference: https://www.cnblogs.com/fengff/p/10173071.html
    """

    def __init__(self):
        layer_shape = {"C1": (5, 5, 1, 6),
                       "C3": (5, 5, 6, 16),
                       "C5": (5, 5, 16, 120),
                       "F6": (120, 84),
                       "OUTPUT": (84, 10)}
```

```
14
15          # Designate combination of kernels and feature maps of S2.
16          C3_mapping = [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 0],
                [5, 0, 1],
17                        [0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 0],
                              [4, 5, 0, 1],
18                        [5, 0, 1, 2], [0, 1, 3, 4], [1, 2, 4, 5], [0, 2, 3, 5],
                              [0, 1, 2, 3, 4, 5]]
19
20          hyper_parameters_convolution = {"stride": 1, "pad": 0}
21          hyper_parameters_pooling = {"stride": 2, "f": 2}
22
23          self.C1 = ConvolutionLayer(layer_shape["C1"],
                hyper_parameters_convolution)
24          self.a1 = Activation("LeNet5_squash")
25          self.S2 = PoolingLayer(hyper_parameters_pooling, "average")
26
27          self.C3 = ConvolutionLayer_maps(layer_shape["C3"],
                hyper_parameters_convolution, C3_mapping)
28          self.a2 = Activation("LeNet5_squash")
29          self.S4 = PoolingLayer(hyper_parameters_pooling, "average")
30
31          self.C5 = ConvolutionLayer(layer_shape["C5"],
                hyper_parameters_convolution)
32          self.a3 = Activation("LeNet5_squash")
33
34          self.F6 = FCLayer(layer_shape["F6"])
35          self.a4 = Activation("LeNet5_squash")
36
37          self.Output = RBFLayer(bitmap)
38
39      def Forward_Propagation(self, input_image, input_label, mode):
40          self.label = input_label
41          self.C1_FP = self.C1.forward_propagation(input_image)
42          self.a1_FP = self.a1.forward_propagation(self.C1_FP)
43          self.S2_FP = self.S2.forward_propagation(self.a1_FP)
44
45          self.C3_FP = self.C3.forward_propagation(self.S2_FP)
46          self.a2_FP = self.a2.forward_propagation(self.C3_FP)
47          self.S4_FP = self.S4.forward_propagation(self.a2_FP)
48
49          self.C5_FP = self.C5.forward_propagation(self.S4_FP)
50          self.a3_FP = self.a3.forward_propagation(self.C5_FP)
51
52          self.flatten = self.a3_FP[:, 0, 0, :]
53          self.F6_FP = self.F6.forward_propagation(self.flatten)
54          self.a4_FP = self.a4.forward_propagation(self.F6_FP)
55
```

```
        # output sum of the loss over mini-batch when mode = 'train'
        # output tuple of (0/1 error, class_predict) when mode = 'test'
        out = self.Output.forward_propagation(self.a4_FP, input_label, mode)

        return out

    def Back_Propagation(self, momentum, weight_decay):
        dy_pred = self.Output.back_propagation()

        dy_pred = self.a4.back_propagation(dy_pred)
        F6_BP = self.F6.back_propagation(dy_pred, momentum, weight_decay)
        reverse_flatten = F6_BP[:, np.newaxis, np.newaxis, :]

        reverse_flatten = self.a3.back_propagation(reverse_flatten)
        C5_BP = self.C5.back_propagation(reverse_flatten, momentum,
            weight_decay)

        S4_BP = self.S4.back_propagation(C5_BP)
        S4_BP = self.a2.back_propagation(S4_BP)
        C3_BP = self.C3.back_propagation(S4_BP, momentum, weight_decay)

        S2_BP = self.S2.back_propagation(C3_BP)
        S2_BP = self.a1.back_propagation(S2_BP)
        C1_BP = self.C1.back_propagation(S2_BP, momentum, weight_decay)

    def SDLM(self, mu, lr_global):
        """
        Stochastic Diagonal Levenberg-Marquardt method for determining the
            learning rate before the beginning of each epoch
        :param mu: mean value
        :param lr_global:
        :return: None
        """
        d2y_pred = self.Output.SDLM()
        d2y_pred = self.a4.SDLM(d2y_pred)

        F6_SDLM = self.F6.SDLM(d2y_pred, mu, lr_global)
        reverse_flatten = F6_SDLM[:, np.newaxis, np.newaxis, :]

        reverse_flatten = self.a3.SDLM(reverse_flatten)
        C5_SDLM = self.C5.SDLM(reverse_flatten, mu, lr_global)

        S4_SDLM = self.S4.SDLM(C5_SDLM)
        S4_SDLM = self.a2.SDLM(S4_SDLM)
        C3_SDLM = self.C3.SDLM(S4_SDLM, mu, lr_global)

        S2_SDLM = self.S2.SDLM(C3_SDLM)
        S2_SDLM = self.a1.SDLM(S2_SDLM)
```

```
102         C1_SDLM = self.C1.SDLM(S2_SDLM, mu, lr_global)
```

LeNet5 类的 Forward_Propagation 和 Back_Propagation 函数会调用每一层对应的 forward_propagation 和 back_propagation 函数，可以看到，层与层之间的参数会一层层的进行传递。这里应该注意的是，学习率 (learning rate) 这个超参数并不是给定的，而是通过 SDLM 算法进行计算得到的每一轮的学习率。

数据处理

```
# Load the MNIST data for this exercise
def load_mnist(file_dir, is_images='True'):
    # Read binary data
    bin_file = open(file_dir, 'rb')
    bin_data = bin_file.read()
    bin_file.close()
    # Analysis file header
    if is_images:
        # Read images
        fmt_header = '>iiii'
        magic, num_images, num_rows, num_cols = struct.unpack_from(fmt_header, bin_data, 0)
        data_size = num_images * num_rows * num_cols
        mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_data, struct.calcsize(fmt_header))
        mat_data = np.reshape(mat_data, [num_images, num_rows, num_cols])
    else:
        # Read labels
        fmt_header = '>ii'
        magic, num_images = struct.unpack_from(fmt_header, bin_data, 0)
        mat_data = struct.unpack_from('>' + str(num_images) + 'B', bin_data, struct.calcsize(fmt_header))
        mat_data = np.reshape(mat_data, [num_images])
    print('Load images from %s, number: %d, data shape: %s' % (file_dir, num_images, str(mat_data.shape)))
    return mat_data


# call the load_mnist function to get the images and labels of training set and testing set
def load_data():
    print('Loading MNIST data from files...')
    train_images = load_mnist(os.path.join(data_dir, train_data_dir), True)
    train_labels = load_mnist(os.path.join(data_dir, train_label_dir), False)
    test_images = load_mnist(os.path.join(data_dir, test_data_dir), True)
    test_labels = load_mnist(os.path.join(data_dir, test_label_dir), False)
    return train_images, train_labels, test_images, test_labels


# transfer the image from gray to binary and get the one-hot style labels
```

```
def data_convert(x, y, m, k):
    x[x <= 40] = 0
    x[x > 40] = 1
    ont_hot_y = np.zeros((m, k))
    for t in range(m):
        ont_hot_y[t, y[t]] = 1
    return x, ont_hot_y


# padding for the matrix of images
def zero_pad(X, pad):
    X_pad = np.pad(X, ((0, 0), (pad, pad), (pad, pad), (0, 0)), 'constant',
        constant_values=(0, 0))
    return X_pad


# normalization of the input images
def normalize(image, mode='LeNet5'):
    image -= image.min()
    image = image / image.max()
    if mode == '0p1':
        return image  # range = [0,1]
    elif mode == 'n1p1':
        image = image * 2 - 1  # range = [-1,1]
    elif mode == 'LeNet5':
        image = image * 1.275 - 0.1  # range = [-0.1,1.175]
    return image
```

卷积层所用的前向传播和后向传播函数

```
# Numpy version: compute with np.tensordot()
def conv_forward(A_prev, W, b, hyper_parameters):
    """
    Implements the forward propagation for a convolution function

    :param A_prev: output activations of the previous layer, numpy array of
        shape (m, n_H_prev, n_W_prev, n_C_prev)
    :param W: Weights, numpy array of shape (f, f, n_C_prev, n_C)
    :param b: Biases, numpy array of shape (1, 1, 1, n_C)
    :param hyper_parameters: python dictionary containing "stride" and "pad"

    :return: Z -- conv output, numpy array of shape (m, n_H, n_W, n_C)
             cache -- cache of values needed for the conv_backward() function
    """
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = W.shape

    stride = hyper_parameters["stride"]
    pad = hyper_parameters["pad"]
```

```
19
20      n_H = int((n_H_prev + 2 * pad - f) / stride + 1)
21      n_W = int((n_W_prev + 2 * pad - f) / stride + 1)
22
23      # Initialize the output volume Z with zeros.
24      Z = np.zeros((m, n_H, n_W, n_C))
25      A_prev_pad = zero_pad(A_prev, pad)
26      for h in range(n_H):  # loop over vertical axis of the output volume
27          for w in range(n_W):  # loop over horizontal axis of the output
                volume
28              # Use the corners to define the (3D) slice of a_prev_pad.
29              A_slice_prev = A_prev_pad[:, h * stride:h * stride + f, w *
                    stride:w * stride + f, :]
30              # Convolve the (3D) slice with the correct filter W and bias b,
                    to get back one output neuron.
31              Z[:, h, w, :] = np.tensordot(A_slice_prev, W, axes=([1, 2, 3],
                    [0, 1, 2])) + b
32
33      assert (Z.shape == (m, n_H, n_W, n_C))
34      cache = (A_prev, W, b, hyper_parameters)
35      return Z, cache
36
37
38  # Numpy version: compute with np.dot
39  def conv_backward(dZ, cache):
40      """
41      Implement the backward propagation for a convolution function
42
43      :param dZ: gradient of the cost with respect to the output of the conv
            layer (Z), numpy array of shape (m, n_H, n_W, n_C)
44      :param cache: cache of values needed for the conv_backward(), output of
            conv_forward()
45
46      :return: dA_prev -- gradient of the cost with respect to the input of the
             conv layer (A_prev),
                    numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
47                dW -- gradient of the cost with respect to the weights of the
                     conv layer (W)
48                  numpy array of shape (f, f, n_C_prev, n_C)
49                db -- gradient of the cost with respect to the biases of the
                     conv layer (b)
50                  numpy array of shape (1, 1, 1, n_C)
51      """
52      (A_prev, W, b, hyper_parameters) = cache
53      (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
54      (f, f, n_C_prev, n_C) = W.shape
55      (m, n_H, n_W, n_C) = dZ.shape
56      stride = hyper_parameters["stride"]
57
```

```
    pad = hyper_parameters["pad"]

    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
    dW = np.zeros((f, f, n_C_prev, n_C))
    db = np.zeros((1, 1, 1, n_C))

    if pad != 0:
        A_prev_pad = zero_pad(A_prev, pad)
        dA_prev_pad = zero_pad(dA_prev, pad)
    else:
        A_prev_pad = A_prev
        dA_prev_pad = dA_prev

    for h in range(n_H):  # loop over vertical axis of the output volume
        for w in range(n_W):  # loop over horizontal axis of the output volume
            # Find the corners of the current "slice"
            vert_start, horiz_start = h * stride, w * stride
            vert_end, horiz_end = vert_start + f, horiz_start + f

            # Use the corners to define the slice from a_prev_pad
            A_slice = A_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :]

            # Update gradients for the window and the filter's parameters
            dA_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :] += np.transpose(np.dot(W, dZ[:, h, w, :].T), (3, 0, 1, 2))

            dW += np.dot(np.transpose(A_slice, (1, 2, 3, 0)), dZ[:, h, w, :])
            db += np.sum(dZ[:, h, w, :], axis=0)

    # Set dA_prev to the unpadded dA_prev_pad
    dA_prev = dA_prev_pad if pad == 0 else dA_prev_pad[:, pad:-pad, pad:-pad, :]

    # Making sure your output shape is correct
    assert (dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, dW, db


def conv_SDLM(dZ, cache):
    (A_prev, W, b, hparameters) = cache
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (f, f, n_C_prev, n_C) = W.shape
    stride = hparameters["stride"]
    pad = hparameters["pad"]
    (m, n_H, n_W, n_C) = dZ.shape
```

```
    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
    dW = np.zeros((f, f, n_C_prev, n_C))
    db = np.zeros((1, 1, 1, n_C))

    if pad != 0:
        A_prev_pad = zero_pad(A_prev, pad)
        dA_prev_pad = zero_pad(dA_prev, pad)
    else:
        A_prev_pad = A_prev
        dA_prev_pad = dA_prev

    for h in range(n_H):  # loop over vertical axis of the output volume
        for w in range(n_W):  # loop over horizontal axis of the output volume
            # Find the corners of the current "slice"
            vert_start, horiz_start = h * stride, w * stride
            vert_end, horiz_end = vert_start + f, horiz_start + f

            # Use the corners to define the slice from a_prev_pad
            A_slice = A_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :]

            # Update gradients for the window and the filter's parameters
            dA_prev_pad[:, vert_start:vert_end, horiz_start:horiz_end, :] += np.transpose(
                np.dot(np.power(W, 2), dZ[:, h, w, :].T), (3, 0, 1, 2))

            dW += np.dot(np.transpose(np.power(A_slice, 2), (1, 2, 3, 0)), dZ[:, h, w, :])
    # Set dA_prev to the unpaded dA_prev_pad
    dA_prev = dA_prev_pad if pad == 0 else dA_prev_pad[:, pad:-pad, pad:-pad, :]
    assert (dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))
    return dA_prev, dW
```

在这里，卷积使用的 np.tensordot 函数大大加速了一个原本需要三个循环完成的卷积计算。

池化层前向传播和后向传播函数

```
def pool_forward(A_prev, hyper_parameters, mode):
    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    f = hyper_parameters["f"]
    stride = hyper_parameters["stride"]

    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    A = np.zeros((m, n_H, n_W, n_C))
    for h in range(n_H):  # loop on the vertical axis of the output volume
```

```
        for w in range(n_W):  # loop on the horizontal axis of the output volume
            # Use the corners to define the current slice on the ith training example of A_prev, channel c
            A_prev_slice = A_prev[:, h * stride:h * stride + f, w * stride:w * stride + f, :]
            # Compute the pooling operation on the slice. Use an if statement to differentiate the modes.
            if mode == "max":
                A[:, h, w, :] = np.max(A_prev_slice, axis=(1, 2))
            elif mode == "average":
                A[:, h, w, :] = np.average(A_prev_slice, axis=(1, 2))

    cache = (A_prev, hyper_parameters)
    assert (A.shape == (m, n_H, n_W, n_C))
    return A, cache


def pool_backward(dA, cache, mode):
    """
    Implements the backward pass of the pooling layer

    :param dA: gradient of cost with respect to the output of the pooling layer, same shape as A
    :param cache: cache output from the forward pass of the pooling layer, contains the layer's input and hyper-parameters
    :param mode: the pooling mode you would like to use, defined as a string ("max" or "average")
    :return: dA_prev -- gradient of cost with respect to the input of the pooling layer, same shape as A_prev
    """
    A_prev, hyper_parameters = cache

    stride = hyper_parameters["stride"]
    f = hyper_parameters["f"]

    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape  # 256,28,28,6
    m, n_H, n_W, n_C = dA.shape  # 256,14,14,6

    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))  # 256,28,28,6

    for h in range(n_H):  # loop on the vertical axis
        for w in range(n_W):  # loop on the horizontal axis
            # Find the corners of the current "slice"
            vert_start, horiz_start = h * stride, w * stride
            vert_end, horiz_end = vert_start + f, horiz_start + f

            # Compute the backward propagation in both modes.
```

```
            if mode == "max":
                A_prev_slice = A_prev[:, vert_start: vert_end, horiz_start:
                    horiz_end, :]
                A_prev_slice = np.transpose(A_prev_slice, (1, 2, 3, 0))
                mask = A_prev_slice == A_prev_slice.max((0, 1))
                mask = np.transpose(mask, (3, 2, 0, 1))
                dA_prev[:, vert_start: vert_end, horiz_start: horiz_end, :] \
                    += np.transpose(np.multiply(dA[:, h, w, :][:, :, np.
                        newaxis, np.newaxis], mask), (0, 2, 3, 1))

            elif mode == "average":
                da = dA[:, h, w, :][:, np.newaxis, np.newaxis, :]  #
                    256*1*1*6
                dA_prev[:, vert_start: vert_end, horiz_start: horiz_end, :]
                    += np.repeat(np.repeat(da, 2, axis=1), 2, axis=2) / f / f
    assert (dA_prev.shape == A_prev.shape)
    return dA_prev


def subsampling_forward(A_prev, weight, b, hparameters):
    A_, cache = pool_forward(A_prev, hparameters, 'average')
    A = A_ * weight + b
    cache_A = (cache, A_)
    return A, cache_A


def subsampling_backward(dA, weight, b, cache_A_):
    (cache, A_) = cache_A_
    db = dA
    dW = np.sum(np.multiply(dA, A_))
    dA_ = dA * weight
    dA_prev = pool_backward(dA_, cache, 'average')
    return dA_prev, dW, db
```

train 函数

```
# Number of epochs & learning rate in the original paper
epochs_original, lr_global_original = 10, np.array([5e-4] * 2 + [2e-4] * 3 +
    [1e-4] * 3 + [5e-5] * 4 + [1e-5] * 8)
# Number of epochs & learning rate I used
epochs, lr_global_list = epochs_original, lr_global_original * 100


def train(LeNet5, train_images, train_labels):
    momentum = 0.9
    weight_decay = 0
    batch_size = 256

    # Training loops
```

```
    cost_last, count = np.Inf, 0
    err_rate_list = []
    for epoch in range(0, epochs):
        print("---------------------------------- epoch{} begin
            ----------------------------------".format(epoch + 1))

        # Stochastic Diagonal Levenberg–Marquardt method for determining the
            learning rate
        batch_image, batch_label = random_mini_batches(train_images,
            train_labels, mini_batch_size=500, one_batch=True)
        LeNet5.Forward_Propagation(batch_image, batch_label, 'train')
        lr_global = lr_global_list[epoch]
        LeNet5.SDLM(0.02, lr_global)

        # print info
        print("global learning rate:", lr_global)
        print("learning rates in trainable layers:", np.array([LeNet5.C1.lr,
            LeNet5.C3.lr, LeNet5.C5.lr, LeNet5.F6.lr]))
        print("batch size:", batch_size)
        print("momentum:", momentum, ", weight decay:", weight_decay)

        # loop over each batch
        ste = time.time()
        cost = 0
        mini_batches = random_mini_batches(train_images, train_labels,
            batch_size)
        for i in range(len(mini_batches)):
            batch_image, batch_label = mini_batches[i]

            loss = LeNet5.Forward_Propagation(batch_image, batch_label, '
                train')
            cost += loss

            LeNet5.Back_Propagation(momentum, weight_decay)

            # print progress
            if i % (int(len(mini_batches) / 100)) == 0:
                sys.stdout.write("\033[F")  # CURSOR_UP_ONE
                sys.stdout.write("\033[K")  # ERASE_LINE
                print("progress:", int(100 * (i + 1) / len(mini_batches)), "
                    %, ", "cost =", cost, end='\r')
        sys.stdout.write("\033[F")  # CURSOR_UP_ONE
        sys.stdout.write("\033[K")  # ERASE_LINE

        print("Done, cost of epoch", epoch + 1, ":", cost, "
                                        ")

        error01_train, _ = LeNet5.Forward_Propagation(train_images,
```

```
            train_labels, 'test')
        err_rate_list.append(error01_train / 60000)
        # error01_test, _ = LeNet5.Forward_Propagation(test_images,
            test_labels, 'test')
        # err_rate_list.append([error01_train / 60000, error01_test / 10000])
        print("0/1 error of training set:", error01_train, "/", len(
            train_labels))
        # print("0/1 error of testing set: ", error01_test, "/", len(
            test_labels))
        print("Time used: ", time.time() - ste, "sec")
        print(
            "------------------------------------ epoch{} end
                ------------------------------------\n".format(epoch + 1))

        # conserve the model
        # with open('model_data_' + str(epoch) + '.pkl', 'wb') as output:
        #     pickle.dump(LeNet5, output, pickle.HIGHEST_PROTOCOL)

    err_rate_list = np.array(err_rate_list).T


# return random-shuffled mini-batches
def random_mini_batches(image, label, mini_batch_size=256, one_batch=False):
    m = image.shape[0]  # number of training examples
    mini_batches = []

    # Shuffle (image, label)
    permutation = list(np.random.permutation(m))
    shuffled_image = image[permutation, :, :, :]
    shuffled_label = label[permutation]

    # extract only one batch
    if one_batch:
        mini_batch_image = shuffled_image[0: mini_batch_size, :, :, :]
        mini_batch_label = shuffled_label[0: mini_batch_size]
        return mini_batch_image, mini_batch_label

    # Partition (shuffled_image, shuffled_Y). Minus the end case.
    num_complete_minibatches = math.floor(m / mini_batch_size)
    for k in range(0, num_complete_minibatches):
        mini_batch_image = shuffled_image[k * mini_batch_size: k *
            mini_batch_size + mini_batch_size, :, :, :]
        mini_batch_label = shuffled_label[k * mini_batch_size: k *
            mini_batch_size + mini_batch_size]
        mini_batch = (mini_batch_image, mini_batch_label)
        mini_batches.append(mini_batch)
    # Handling the end case (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
```

```
94          mini_batch_image = shuffled_image[num_complete_minibatches *
                mini_batch_size: m, :, :, :]
95          mini_batch_label = shuffled_label[num_complete_minibatches *
                mini_batch_size: m]
96          mini_batch = (mini_batch_image, mini_batch_label)
97          mini_batches.append(mini_batch)
98
99      return mini_batches
```

这里每轮训练都是在整个数据集上进行训练，即全梯度下降，用时较长。

五、 实验结果及分析

```
---------------------------------- epoch9 begin ----------------------------------
global learning rate: 0.005
learning rates in trainable layers: [1.38172684e-03 6.03415333e-06 4.19360083e-06 3.20932939e-07]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 9 : 83653.55444446256
0/1 error of training set: 1089 / 60000
Time used:  494.02822518348694 sec
---------------------------------- epoch9 end ----------------------------------

---------------------------------- epoch10 begin ----------------------------------
global learning rate: 0.005
learning rates in trainable layers: [1.33428029e-03 5.77749574e-06 4.20234613e-06 3.19935225e-07]
batch size: 256
momentum: 0.9 , weight decay: 0
Done, cost of epoch 10 : 79912.7043062596
0/1 error of training set: 992 / 60000
Time used:  497.2688364982605 sec
---------------------------------- epoch10 end ----------------------------------

Total time used:  5624.467736959457 sec
Finished training, the total training time is 5624.487748384476s

Start testing...
error rate: 0.0185
Finished testing, the accuracy is 0.9815


Process finished with exit code 0
```

图 6: 实验结果

我设置的训练轮次为 10 轮，可能存在训练不充分导致的欠拟合，因此在测试集上的准确率仅为 98.15%，如果加大训练轮次，效果可能更好。

可以看到，由于每一轮都是在整个训练集上的训练，因此一轮的训练时长需要 10 分钟左右。10 轮总计用时 2 小时左右。

六、 总结

通过本次实验，我从零开始搭建了一个 LeNet5 的网络，进一步理解了一个神经网络的内部结构以及各种计算（forward propagation，back propagation），收获颇丰。

个人仓库地址：https://github.com/Skyyyy0920/Building-LeNet5-from-Scratch

参考文献

[1] https://www.cnblogs.com/fengff/p/10173071.html