Exercise2-LeNet5

学院：网络空间安全学院 学号：2111673 专业：物联网工程 姓名：岳志鑫

一、问题分析

在这个练习中，需要使用 Python 实现 LeNet5 来完成对 MNIST 数据集中 0-9 共 10 个手写数字的分类。代码只能使用 Python 实现，其中数据读取可使用 PIL、opencv-python 等库，矩阵运算可使用 numpy 等计算库，网络前向传播和梯度反向传播需要手动实现，不能使用 PyTorch、TensorFlow、Jax 或 Caffe 等自动微分框架。

MNIST 数据集可在 http://yann.lecun.com/exdb/mnist/下载。

二、实验环境

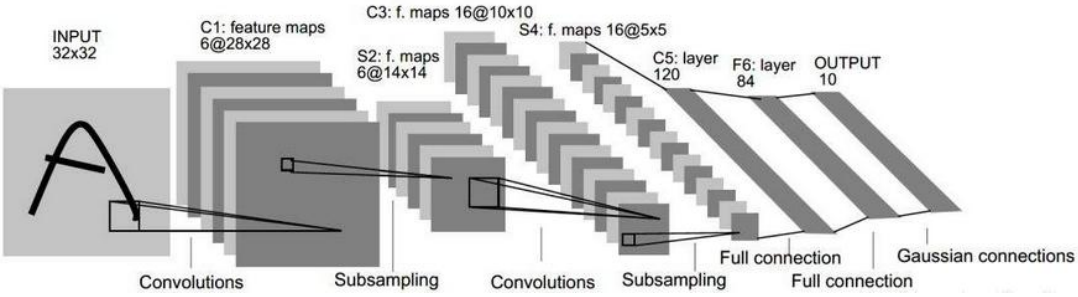
PyCharm Community Edition 2023.2.1

Python 3.8.6

三、实验原理

模型的输入数据是包含手写数字信息的二维图像，将其输入到网络模型中，经过模型的前向计算得到输出的识别结果，通过损失函数度量计算输出结果与输入图像标签的差异度，并通过反向传播算法根据这个差异来调整网络各层的参数值，经过反复迭代输入，最终使得输出的识别结果与标签一致，得到一个能准确识别输入图像的网络模型。

LeNet-5 模型总共有 7 层，下图展示了 LeNet-5 模型的结构。

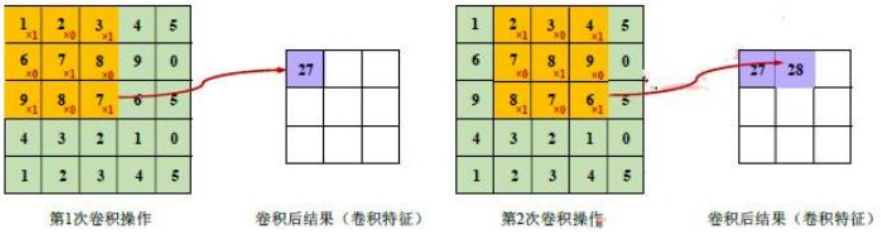


分别为卷积层、池化层、全连接层和输出层。

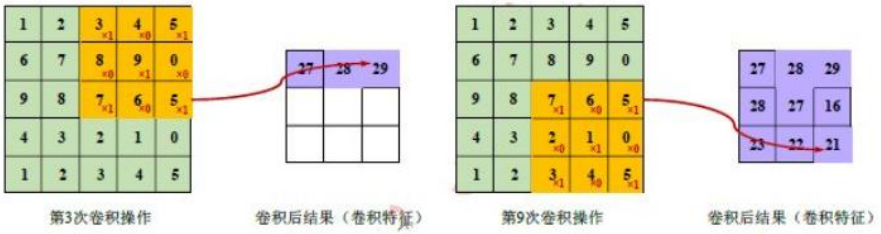
输入的二维图像经过两个卷积层、一个池化层，再经过三个全连接层将前层计算得到的特征空间映射样本标记空间，最后使用 softmax 分类作为输出层。

1.卷积层

卷积运算实际是分析数学中的一种运算方式，在卷积神经网络中通常是仅涉及离散卷积的情形。每个卷积层由若干卷积单元组成，每个卷积单元的参数都是通过反向传播算法最佳化得到的。卷积运算的目的是提取输入的不同特征，第一层卷积层可能只能提取一些低级的特征如边缘、线条和角等层级，更多层的网路能从低级特征中迭代提取更复杂的特征。

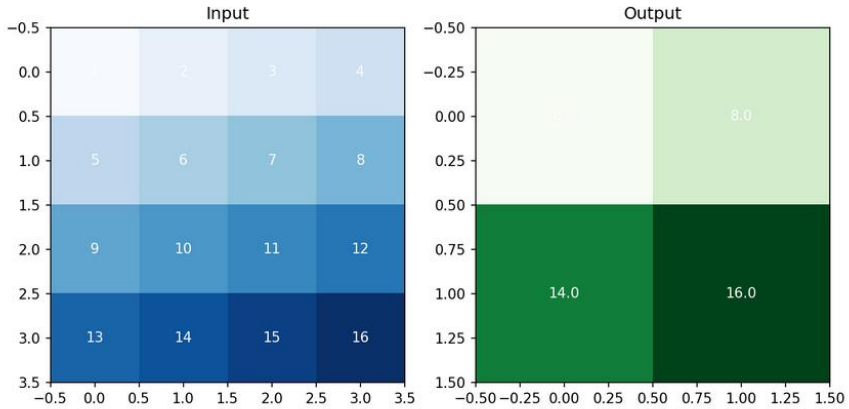


(a) 第一次卷积操作及得到的卷积特征。　(b) 第二次卷积操作及得到的卷积特征。



2.池化层

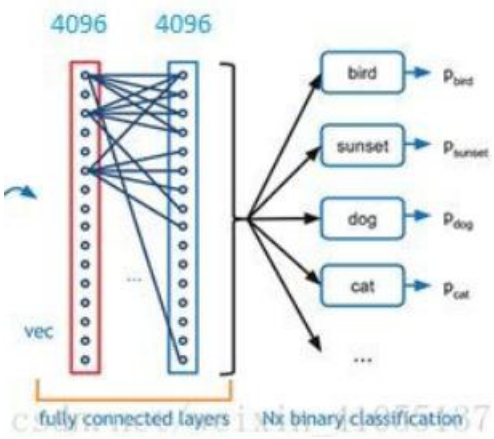
池化层是深度学习中常用的一种层级结构，它可以对输入数据进行降采样，减少数据量，同时保留重要的特征信息。池化层通常紧跟在卷积层之后，可以有效地减少数据量和计算复杂度，提高模型的训练速度和泛化能力。



池化层的结构与卷积层类似，它也由多个滤波器组成，每个滤波器对输入数据进行卷积操作，得到一个输出特征图。不同的是，池化层的卷积操作通常不使用权重参数，而是使用一种固定的池化函数，例如最大池化、平均池化等。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

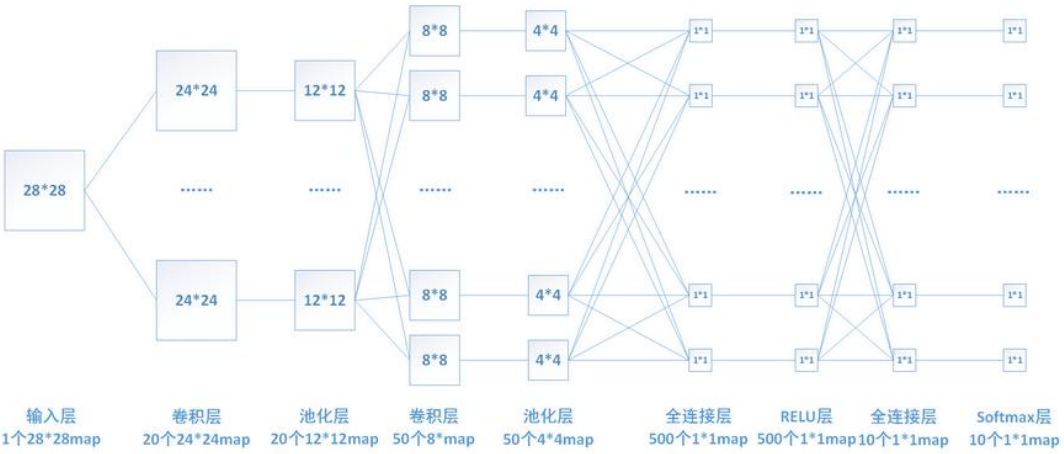
3.全连接层

全连接层，是每一个结点都与上一层的所有结点相连，用来把前边提取到的特征综合起来。由于其全相连的特性，一般全连接层的参数也是最多的，位于整个神经网络的最后，负责将卷积输出的二维特征图转换成一维向量，由此实现了端到端的学习过程



4.前向传播

在前向计算过程，也就是一个线性的加权求和的过程，全连接层的每一个输出都可以看成前一层的每一个结点乘以一个权重系数 W，最后加上一个偏置值 b 得到。如下图中第一个全连接层，输入有 50*4*4 个神经元结点，输出有 500 个结点，则一共需要 50*4*4*500=400000 个权值参数 W 和 500 个偏置参数 b。



5.反向传播

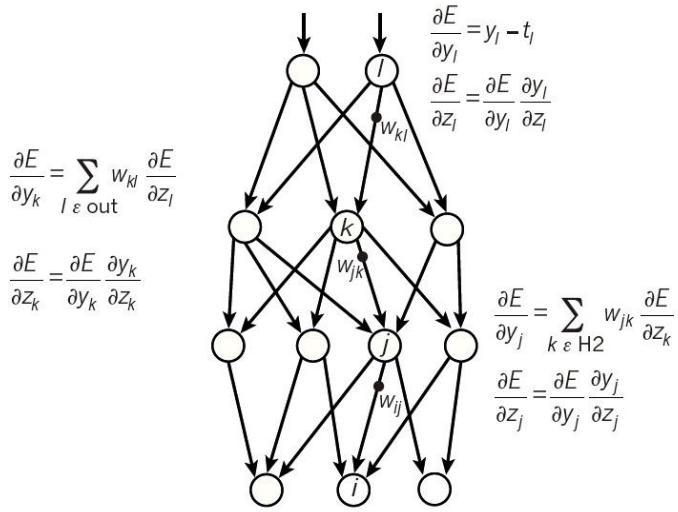
在反向传播过程中，若第 x 层的 a 节点通过权值 W 对 x+1 层的 b 节点有贡献，则在反向传播过程中，梯度通过权值 W 从 b 节点传播回 a 节点。

由于需要对 W 和 b 进行更新，还要向前传递梯度，所以我们需要计算如下三个偏导数。

（1）对上一层的输出（即当前层的输入）求导

（2）对权重系数 W 求导

（3）对偏置系数 b 求导



四、实现方式

1.读取数据

由于 LeNet5 所需的是 32x32 的图像，而 Mnist 数据集提供的图像都为 28x28 的大小，所以要对图像进行周围的填充使其满足 32x32 的大小。

```
def zero_pad(X, pad):
    X_pad = np.pad(X, ((0,), (pad,), (pad,), (0,)), "constant", constant_values=(0, 0))
    return X_pad
```

同时要对数据进行归一化，并进行零均值化

```
def normalise(image):
    image -= image.min()
    image = image / image.max()
    image = (image - np.mean(image)) / np.std(image)
    return image
```

由于数量级过大，因此选择将数量级分散成 8 个小的数量级以供每次训练时进行读取，减少训练整个训练集的耗时

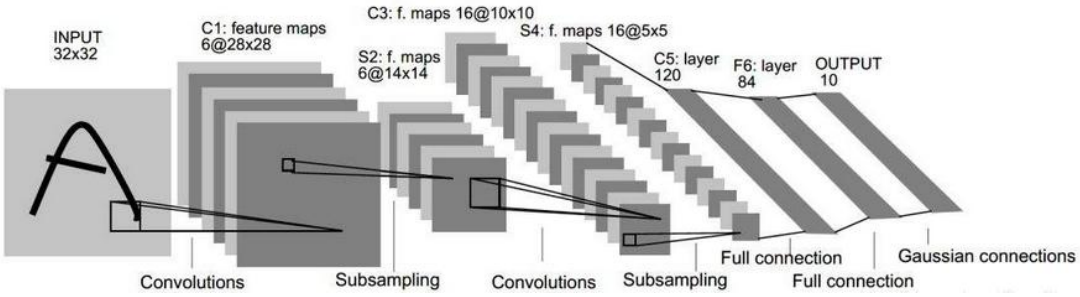
```
def random_mini_batches(image, label, mini_batch_size=256, one_batch=False):
    dataset_size = image.shape[0]_# 训练样本数
    mini_batches = []
    #  打乱 (image, label)
    permutation = list(np.random.permutation(dataset_size))
    shuffled_image = image[permutation, :, :, :]
    shuffled_label = label[permutation]
    # 提取一个批量
    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)
    # 分割 (shuffled_image, shuffled_label)。减去尾部情况。
    complete_minibatches_number = math.floor(
        dataset_size / mini_batch_size)  # 在分区中每个大小为mini_batch_size的小批次的数量
    for k in range(0, complete_minibatches_number):
        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)
```

指定数据集存放的路径并进行读取，将图像数据集转为独热码

```
test_image_path = "dataset/MNIST/t10k-images-idx3-ubyte"
test_label_path = "dataset/MNIST/t10k-labels-idx1-ubyte"
train_image_path = "dataset/MNIST/train-images-idx3-ubyte"
train_label_path = "dataset/MNIST/train-labels-idx1-ubyte"
```

```
def load_dataset(test_image_path, test_label_path, train_image_path, train_label_path):
    train_dataset = (train_image_path, train_label_path)
    test_dataset = (test_image_path, test_label_path)
    # 读取数据
    train_image, train_label = readDataset(train_dataset)
    test_image, test_label = readDataset(test_dataset)
    # 数据预处理
    train_image_normalised_pad = normalise(zero_pad(train_image[:, :, :, np.newaxis], pad: 2))
    test_image_normalised_pad = normalise(zero_pad(test_image[:, :, :, np.newaxis], pad: 2))
    return (train_image_normalised_pad, train_label), (test_image_normalised_pad, test_label)
```

2.LeNet5 组成



该模型拥有的卷积层、池化层、全连接层、输出层以及 RELU 激活函数等都拥有各自的前向传播和后向传播方法，使得他们联系起来成为一个完整的数据模型，使用 softmax 用于多类别分类问题的输出。

```
class LeNet5(object):
    def __init__(self):
        # kernel_shape 字典包含各层的卷积核形状信息。
        kernel_shape = {"C1": (5, 5, 1, 6),
                        "C3": (5, 5, 6, 16),
                        "C5": (5, 5, 16, 120),
                        "F6": (120, 84),
                        "F7": (84, 10)}
        self.C1 = Conv_Layer(kernel_shape["C1"], sigma = 0.1, bias_factor = 0.01)
        self.ReLU1 = ReLU_Layer()
        self.S2 = MaxPool_Layer()
        self.C3 = Conv_Layer(kernel_shape["C3"], sigma = 0.1, bias_factor = 0.01)
        self.ReLU2 = ReLU_Layer()
        self.S4 = MaxPool_Layer()
        self.C5 = Conv_Layer(kernel_shape["C5"], sigma = 0.1, bias_factor = 0.01)
        self.ReLU3 = ReLU_Layer()
        self.F6 = FC_Layer(kernel_shape["F6"], sigma = 0.1, bias_factor = 0.01)
        self.ReLU4 = ReLU_Layer()
        self.F7 = FC_Output_Layer(kernel_shape["F7"], sigma = 0.1, bias_factor = 0.01)
```

3.模型训练

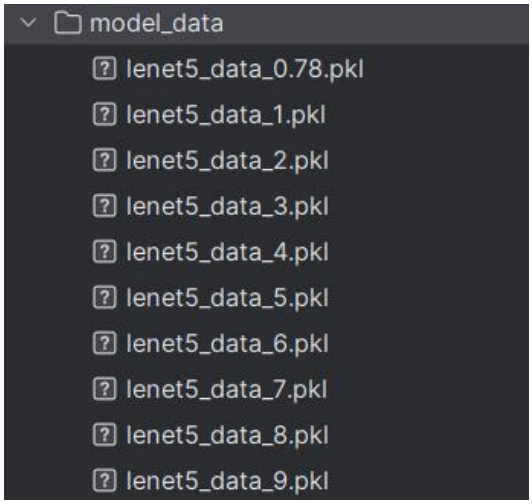
使用数据集对模型进行训练，设定了 20 次迭代，每次迭代选择一个小数据集 7500 个图像进行训练

```
# 训练循环
start_time = time.time()
error_rate_list = []
for epoch in range(0, epoches):
    print("迭代", epoch + 1, "开始")
    learning_rate = learning_rate_list[epoch]
    # 打印信息
    print("学习率: {}".format(learning_rate))
    print("分组大小: {}".format(batch_size))
    # 循环每个小批次
    start_time_epoch = time.time()
    cost = 0
    mini_batches = random_mini_batches(train_data[0], train_data[1], batch_size)
    print("训练中:")
    for i in range(len(mini_batches)):
        print('训练次数: %d /总数 %d' % (i, len(mini_batches)))
        batch_image, batch_label = mini_batches[i]
        loss = model.forward_propagation(batch_image, batch_label, 'train')
        print(' 损失: %f' % loss)
        cost += loss
        model.back_propagation(learning_rate)
```

每次迭代后都会进行一次实际检测，并将该训练的模型保存下来，记录对应的准确率等信息，同时调整自己的学习率进行下一次训练。

```
error_train, _ = model.forward_propagation(train_data[0], train_data[1], 'test')
error_test, _ = model.forward_propagation(test_data[0], test_data[1], 'test')
error_rate_list.append([error_train / 60000, error_test / 10000])
print("训练正确量/总数量", len(train_data[1]) - error_train, "/", len(train_data[1]))
print("训练正确率:%.2f%%" % ((len(train_data[1]) - error_train) / len(train_data[1]) * 100))
print("测试正确量/总数量:", len(test_data[1]) - error_test, "/", len(test_data[1]))
print("测试正确率:%.2f%%" % ((len(test_data[1]) - error_test) / len(test_data[1]) * 100))
print("迭代", epoch + 1, "结束")
```

最终迭代结束后，会在所有已保存的模型中选择准确率最高的一个模型进行最终的测试



```python
def test(model_path, test_data):
    # read model
    with open(model_path, "rb") as model_file:
        model = pickle.load(model_file)
    print("测试 {}:".format(model_path))
    errors, predictions = model.forward_propagation(test_data[0], test_data[1], "test")
    print("正确率:", len(predictions) - errors / len(predictions))
```

```python
#模型的参数
batch_size = 8 #分成8个小数据集
epoches = 10 #训练10轮
learning_rate_list = np.array([5e-2] * 2 + [2e-2] * 3 + [1e-2] * 3 + [5e-3] * 4 + [1e-3] * 4 + [5e-4] * 4)
#选择学习率

#选择模型进行最终的测试
train_data, test_data = load_dataset(test_image_path, test_label_path, train_image_path, train_label_path)
model = LeNet5()
error_rate_list = train(model, train_data, test_data, epoches, learning_rate_list, batch_size)
test("model_data/lenet5_data_" + str(error_rate_list[1].argmin() + 1) + ".pkl", test_data)
```

四、实验结果

加载数据集

```
正在加载数据集...
加载数据 dataset/MNIST/train-labels-idx1-ubyte, 数据量: 60000
加载数据 dataset/MNIST/train-images-idx3-ubyte, 数据量: 60000
加载数据 dataset/MNIST/t10k-labels-idx1-ubyte, 数据量: 10000
加载数据 dataset/MNIST/t10k-images-idx3-ubyte, 数据量: 10000
```

前几次小数据集的训练结果发现损失在逐渐降低，准确率逐渐增加

```
训练次数: 7498 /总数 7500
 损失: 9.766084
训练次数: 7499 /总数 7500
 损失: 0.183430
训练正确量/总数量 58053 / 60000
训练正确率:96.75%
测试正确量/总数量: 9653 / 10000
测试正确率:96.53%
```

```
训练次数: 7498 /总数 7500
 损失: 0.810605
训练次数: 7499 /总数 7500
 损失: 0.003487
训练正确量/总数量 58960 / 60000
训练正确率:98.27%
测试正确量/总数量: 9791 / 10000
测试正确率:97.91%
```

```
训练次数: 7498 /总数 7500
 损失: 0.002924
训练次数: 7499 /总数 7500
 损失: 0.000509
训练正确量/总数量 59806 / 60000
训练正确率:99.68%
测试正确量/总数量: 9888 / 10000
测试正确率:98.88%
```

```
训练次数: 7498 /总数 7500
 损失: 0.000794
训练次数: 7499 /总数 7500
 损失: 0.010912
训练正确量/总数量 59885 / 60000
训练正确率:99.81%
测试正确量/总数量: 9906 / 10000
测试正确率:99.06%
```

最终迭代结束后，会在所有已保存的模型中选择准确率最高的一个模型进行最终的测试发现 data_1.pkl 的准确率最高选择了这个模型进行检测，准确率达到 99.999586%

五、心得体会

学习了 LeNet 的相关知识，对于卷积层、池化层、连接层有了更深的理解，对于规定的许多微分框架不能使用还是有些麻烦，需要从底层写起，造成了一定的困扰。

如果允许更高的迭代次数也许会取得更高的准确率，但是发现 20 次迭代已经需要超过三个小时的运行时间，且不能出现任何差错，较为耗时，因此只进行了这么多的训练。

六、附录

GitHub 链接：

https://github.com/Q-qiuqiu/machine-learning/tree/main/lab 2