
# 基基于卷积神经网络图像分类

## 神经网络与深度学习课程实验报告

班级：105

姓名：耿翊中

学号：2021213382

2023 年 11 月

目录

## 1. 实验要求

Task: 基于卷积神经网络图像分类 (15 Points)

- Due Date:  Nov.9, 2023 (Tuesday)
- 数据集：MNIST data set，或其它数据集，如 CIFAR-10
- 本题目考察如何设计并实现一个基于卷积神经网络的图像分类器。设置本题目的目的如下：
  - 理解卷积神经网络的基本结构、代码实现及训练过程
  - 应用 dropout 和多种 normalization 方法，理解它们对模型泛化能力的影响
  - 理解如何通过交叉验证，为神经网络找到最好的 hyperparameters

- 附加题
  - 在训练网络的过程中，可根据需要自由尝试其它提升性能的方法，例如通过增加模型层数、使用不同的正则化方法、使用模型集成等 (+5 points)

# 2. 代码实现过程

## 2.1 代码结构

Main.py：主函数
model.py:卷积层，全连接层，池化层的实现
utils.py：加载数据等工具类和函数
Method.py：提升模型性能的方法类，Batch Normalization 和 dropout

mnist-original.mat：数据集
result1.csv,result2.csv：寻找最优超参数实验结果
result1.png,result2.png：寻找最优超参数实验结果
data2.npz：中间参数传递文件

## 2.2 数据集处理

使用 Minist 手写数字识别数据集，来源于网站：
https://www.kaggle.com/avnishnish/mnist-original/download
，下载到本地后为 mnist-original.mat 文件，使用 from scipy.io import loadmat 函数将其导入，将数据集分为训练集和测试集，根据构造函数中指定的 train_size 和 test_size 参数。
类中提供了两个方法来加载训练集和测试集：
load_train()方法：返回训练集的图像数据 X_train 和相应的标签数据 y_train。在这个方法内，图像数据被重新形状为 28x28 的图像，并且像素值被除以 255，以将其缩放到 0 到 1 的范围内。标签数据也经过了 one-hot 编码，其中 onehot()函数用于将标签转化为 one-hot 向量。
load_test()方法：返回测试集的图像数据 X_test 和相应的标签数据 y_test。同样，图像数据也被重新形状和缩放，标签数据也被 one-hot 编码。

```
class Load_Data:
    def __init__(self, train_size=60000, test_size=10000):
        '''''
        导入数据集，划分为 60,000 个训练样本，10,000 个测试样本
        '''
        # 加载数据文件
        data = loadmat('mnist-original.mat')
        # 提取数据的特征矩阵，并进行转置
        X = data['data']
        self.X = X.transpose()
        # 然后将特征除以 255，重新缩放到 [0,1] 的范围内，以避免在计算过程中溢出
        # 从数据中提取 labels
        y = data['label']
        self.y = y.flatten()
        # 将数据分割为 60,000 个训练集
        self.train_size = train_size
        # 和 10,000 个测试集
```

```
            self.test_size = test_size
            self.indices = np.arange(70000)
            np.random.shuffle(self.indices)
            self.X = self.X[self.indices]
            self.y = self.y[self.indices]

        def load_train(self):
            X_train = self.X[:self.train_size, :].reshape(self.train_size, 28, 28)
            y_train = self.y[:self.train_size]
            X_train = X_train.reshape(self.train_size, 28, 28, 1) / 255.  # 输入向量处理
            y_train = onehot(y_train, self.train_size)  # 标签 one-hot 处理 (60000, 10)
            return X_train, y_train

        def load_test(self):
            X_test = self.X[self.train_size:self.train_size + self.test_size, :].reshape(self.test_size, 28, 28)
            y_test = self.y[self.train_size:self.train_size + self.test_size]
            X_test = X_test.reshape(self.test_size, 28, 28, 1) / 255.
            return X_test, y_test

    def onehot(targets, num):
        result = np.zeros((num, 10))
        for i in range(num):
            result[i][int(targets[i])] = 1
        return result
```

## 2.3 卷积，池化，全连接层实现

### 2.3.1 卷积层

卷积层构造 Conv 类通过几个方法来实现卷积层的功能，包括初始化层、执行前向传播和执行反向传播（用于学习）。

__init__

当一个 Conv 类的对象被创建时，__init__ 方法首先被调用。这个方法负责初始化卷积层的权重和偏置。

kernel_shape 是一个四元组，包含了卷积核的宽度、高度、输入通道数和输出通道数。Xavier 初始化方法被用来初始化权重，这是一种常见的权重初始化方法，目的是为了保持输入和输出的方差一致，防止梯度消失或爆炸。

self.k 是卷积核的权重，self.b 是偏置。

self.k_gradient 和 self.b_gradient 用于存储权重和偏置的梯度，这些梯度在反向传播时更新权重和偏置。

```
    def __init__(self, kernel_shape, path=1):
        width, height, in_channel, out_channel = kernel_shape
        self.path = path
        scale = np.sqrt(3 * in_channel * width * height / out_channel)  # 使用 Xavier 初始化缩放因子
```

```
        self.k = np.random.standard_normal(kernel_shape) / scale  # 初始化 k，为一个四维向量
        self.b = np.random.standard_normal(out_channel) / scale  # 初始化 b，为一个一维向量
        self.k_gradient = np.zeros(kernel_shape)  # 相应存储 k 的梯度
        self.b_gradient = np.zeros(out_channel)  # 相应存储 b 的梯度
        pad = 0
        self.pad = pad
```

forward

forward 方法实现了前向传播，这是神经网络计算其输出的过程。其中关键步骤是利用矩阵乘法实现了卷积的操作。

输入数据 x 可能会根据 pad 参数被填充，以确保输出尺寸的正确性。

方法中使用了 img2col 函数，这个函数会将输入图像展开为列，以便能够使用矩阵乘法来实现卷积操作。首先，函数接收一个三维的输入数据 x，这个数据的形状是 [width, height, channel]，代表图像的宽度、高度和通道数。函数计算特征图的尺寸 feature_w，这是经过卷积后的输出宽度和高度，根据卷积核的大小 ksize 和步长 stride 来确定。然后创建一个用于存放“展开”后图像块的矩阵 image_col。这个矩阵的每一行都将对应输入数据中的一个局部区域，该区域的大小和卷积核相匹配。矩阵的列数是卷积核大小乘以输入通道数，即 ksize * ksize * cx。接下来，通过两个嵌套循环遍历特征图的宽度和高度，对于特征图中的每一个位置，都会从输入数据 x 中提取出对应的局部区域。这些局部区域会被“展开”成一维数组，并被赋值到 image_col 矩阵的相应行。展开是通过 reshape(-1) 实现的，这个操作会保持数据的顺序，但是把它变成一维数组。num 变量用于跟踪 image_col 矩阵的行索引，每处理一个局部区域后，num 会增加，以便下一个局部区域可以放在下一行。最终，输出的维度是 (batch_size, bkimg_size, bkimg_size, kernel_num)。

```
    def img2col(x, ksize, stride):
        wx, hx, cx = x.shape  # [width,height,channel]
        feature_w = (wx - ksize) // stride + 1  # 返回的特征图尺寸
        image_col = np.zeros((feature_w * feature_w, ksize * ksize * cx))
        num = 0
        for i in range(feature_w):
            for j in range(feature_w):
                image_col[num] = x[i * stride:i * stride + ksize, j * stride:j * stride + ksize, :].re
shape(-1)
                num += 1
        return image_col

```

```
    def forward(self, x):
        self.x = x
        if self.pad != 0:
            self.x = np.pad(self.x, ((0, 0), (self.pad, self.pad), (self.pad, self.pad), (0, 0)), 'con
stant')
        # 批量，宽度，高度，输入通道数
        batch_size, in_width, in_height, in_channel = self.x.shape
        # 卷积核宽度，高度，卷积核输入通道数,卷积核个数
        kernel_width, kernel_height, kin_channel, kernel_num = self.k.shape
        # 返回图像的参数，批量，高，宽，通道
```

```
        bkimg_size = (in_width - kernel_width) // self.path + 1
        bk_img = np.zeros((batch_size, bkimg_size, bkimg_size, kernel_num))

        # 进行卷积运算
        self.image_col = []
        kernel = self.k.reshape(-1, kernel_num)
        for i in range(batch_size):
            image_col = img2col(self.x[i], kernel_width, self.path)
            bk_img[i] = (np.dot(image_col, kernel) + self.b).reshape(bkimg_size, bkimg_size, kernel_nu
m)
            self.image_col.append(image_col)
        return bk_img
```

backward 方法

backward 方法执行反向传播，这是训练神经网络时用于计算梯度并更新权重的过程。方法计算从下一层回传的梯度 delta，并根据这些梯度来更新卷积核的权重 self.k 和偏置 self.b。

在权重更新之前，权重梯度 self.k_gradient 和偏置梯度 self.b_gradient 会被累积和平均。

最后，计算了传播到前一层的梯度 delta_backward，这是通过对卷积核进行 180 度旋转并执行卷积操作实现的。

```
    def backward(self, delta, learning_rate):
        # 批量，宽度，高度，输入通道数
        batch_size, in_width, in_height, in_channel = self.x.shape
        # 卷积核宽度，高度，卷积核输入通道数,卷积核个数
        kernel_width, kernel_height, kin_channel, kernel_num = self.k.shape
        # 从上一层传回来的梯度,批量,宽度,高度,卷积核个数
        delta_batch, delta_width, delta_height, delta_channel = delta.shape

        delta_col = delta.reshape(delta_batch, -1, delta_channel)
        for i in range(batch_size):
            self.k_gradient += np.dot(self.image_col[i].T, delta_col[i]).reshape(self.k.shape)
        self.k_gradient /= batch_size
        self.b_gradient += np.sum(delta_col, axis=(0, 1))
        self.b_gradient /= batch_size

        # 计算 delta_backward
        delta_backward = np.zeros(self.x.shape)
        k_180 = np.rot90(self.k, 2, (0, 1))  # numpy 矩阵旋转 180 度
        k_180 = k_180.swapaxes(2, 3)
        k_180_col = k_180.reshape(-1, kin_channel)

        if delta_height - kernel_height + 1 != in_height:
            pad = (in_height - delta_height + kernel_height - 1) // 2
            pad_delta = np.pad(delta, ((0, 0), (pad, pad), (pad, pad), (0, 0)), 'constant')
```

```
        else:
            pad_delta = delta

        for i in range(batch_size):
            pad_delta_col = img2col(pad_delta[i], kernel_width, self.path)
            delta_backward[i] = np.dot(pad_delta_col, k_180_col).reshape(in_width, in_height, in_chann
    el)

        # 反向传播
        self.k -= self.k_gradient * learning_rate
        self.b -= self.b_gradient * learning_rate

        return delta_backward
```

### 2.3.2 池化层

Pool 类实现了池化层的操作，用来减少特征图的空间尺寸，从而减少参数的数量和计算的复杂度，并且帮助提取特征。在这个类中，实现了最大池化（max pooling）操作。

forward

forward 方法实现了前向传播过程中的池化操作。

方法首先接收输入数据 x，其形状为 (batch_size, in_width, in_height, in_channel)，代表批次大小、输入宽度、输入高度和通道数。

创建输出特征图 bk_img，其尺寸是输入的一半，因为这里实现的是步长为 2 的 2x2 最大池化。

self.max_address 是一个与输入 x 形状相同的数组，用于在池化过程中记录每个通道的最大值的位置，这在反向传播中非常有用。

在四个嵌套循环中，对于输入数据的每个批次、每个通道，以及在宽度和高度上的每个 2x2 的区域：找到当前 2x2 区域中的最大值，并将其赋值给输出特征图 bk_img 相应的位置。使用 np.argmax 找到这个最大值在 2x2 区域中的索引，并将该位置在 self.max_address 中标记为 1。最后，返回池化后的输出特征图 bk_img。

```
    def forward(self, x):
        batch_size, in_width, in_height, in_channel = x.shape
        bkimg_size = in_width // 2
        bk_img = np.zeros((batch_size, bkimg_size, bkimg_size, in_channel))
        # 记录池化位置,在反向传播中使用
        self.max_address = np.zeros((batch_size, in_width, in_height, in_channel))
        for b in range(batch_size):
            for c in range(in_channel):
```

```
            for w in range(bkimg_size):
                for h in range(bkimg_size):
                    bk_img[b, w, h, c] = np.max(x[b, w * 2:w * 2 + 2, h * 2: h * 2 + 2, c])
                    index = np.argmax(x[b, w * 2:w * 2 + 2, h * 2: h * 2 + 2, c])
                    self.max_address[b, w * 2 + index // 2, h * 2 + index // 2, c] = 1
    return bk_img
```

backward

backward 方法实现了池化层的反向传播。

方法接收来自后一层的梯度 delta，其形状与池化后的输出特征图相同。由于最大池化操作不是一个有参数的操作，我们不需要计算参数的梯度，只需要将错误梯度从池化层传播回卷积层。使用 np.repeat 方法将 delta 在每个轴上扩展两倍，这样就恢复到了池化前的尺寸。

最后，将扩展后的梯度与 self.max_address 相乘，以确保梯度只回传到每个 2x2 区域中最大值的位置，其他位置的梯度为 0。

```
def backward(self, delta):
    return np.repeat(np.repeat(delta, 2, axis=1), 2, axis=2) * self.max_address
```

### 2.3.3 全连接

位于神经网络的末端，对特征进行最后的决策分类处理。

当 Linear 类的一个实例被创建时，__init__ 方法首先被调用。这个方法负责初始化全连接层的权重 W 和偏置 b。

inChannel 表示输入特征的数量，outChannel 表示输出特征的数量。

权重和偏置使用标准正态分布随机初始化，并且使用 Xavier 初始化的变种（使用输入通道数量的平方根除以 2 作为分母）来进行缩放，以保持激活值在训练开始时的合理范围。

self.W_gradient 和 self.b_gradient 用于存储权重和偏置的梯度。

forward 方法实现了全连接层的前向传播。

backward 方法实现了全连接层的反向传播。

这里的处理和普通神经网络区别不大，不再多加赘述。

## 2.4 搭建神经网络

将神经网络的测试和训练封装在了一个 Run 类中，这样可以方便我来调整神经网络的数据集和超参数，方便进行交叉验证。

```
class Run:

```

```
3.        def __init__(self, train_size, test_size, learning_rate=0.01, dropout_rate=0.3, epoch_num=2, b
    atch_size=3):
4.            self.train_size = train_size
5.            self.test_size = test_size
6.            self.learning_rate = learning_rate  # 学习率
7.            self.dropout_rate = dropout_rate  # drop_out 概率
8.            self.epoch_num = epoch_num  # 迭代次数
9.            self.batch_size = batch_size  # 批量大小
10.           self.data = Load_Data(train_size=self.train_size, test_size=self.test_size)
```

__init__ 方法是类的构造函数，用于初始化一些参数和数据。这里的参数包括：

train_size 和 test_size 是训练集和测试集的大小，用于划分数据集。

learning_rate 是学习率，控制梯度下降的速度。

dropout_rate 是 dropout 概率，用于防止过拟合。

epoch_num 是迭代次数，表示训练过程中重复使用训练集的次数。

batch_size 是批量大小，表示每次训练使用的样本数量。

self.data 是一个 Load_Data 类的实例，用于加载和处理图像数据。

### 2.4.1 训练

train 方法是类的训练函数，用于构建和训练卷积神经网络。这里的主要步骤包括：

从 self.data 中加载训练集的特征和标签，分别赋值给 X_train 和 y_train。定义卷积神经网络的结构，包括两个卷积层、两个池化层、一个全连接层、一个 softmax 层、一个 dropout 层和一个批归一化层。这些层的作用如下：

卷积层（Conv）是用于提取图像特征的层，它使用一个滤波器（kernel_shape）在输入图像上滑动，并计算滤波器和图像局部区域的点积，得到一个特征图（feature map）。

池化层（Pool）是用于降低特征图维度的层，它使用一个固定大小的窗口（默认为 2x2）在特征图上滑动，并取窗口内的最大值作为输出。

全连接层（Linear）是用于将特征图展平为一维向量，并进行线性变换的层，它有两个参数：输入维度（in_features）和输出维度（out_features）。

softmax 层（Softmax）是用于将全连接层的输出转换为概率分布的层，它使用 softmax 函数对每个类别的得分进行归一化，使其和为 1。

dropout 层（Dropout）是用于随机丢弃一些神经元的层，它有一个参数：丢弃概率（rate）。它可以防止网络过度依赖某些神经元，从而提高泛化能力。

批归一化层（BatchNorm）是用于对每个批次的数据进行归一化的层，它有一个参数：特征图的通道数（num_features）。它可以加速收敛，提高稳定性，减少梯度消失或爆炸的风险。

使用双层循环来遍历训练集，每次取一个批次的数据，进行前向传播和反向传播。具体步骤如下：

前向传播：将输入数据 X 依次通过卷积层、批归一化层、激活函数（Relu）、dropout层、池化层、全连接层和 softmax 层，得到输出 predict。

计算损失：使用 softmax 层的 cal_loss 方法，根据输出 predict 和真实标签 Y，计算交叉熵损失（loss）和误差（delta）。

反向传播：根据误差 delta 和学习率 learning_rate，依次调用各层的 backward 方法，更新网络的参数（权重和偏置）。

打印信息：打印当前的迭代次数（epoch）、批次编号（i）和损失值（loss）。

衰减学习率：每完成一次迭代，就将学习率乘以一个衰减因子（0.95 ** (epoch + 1)），使学习率随着训练的进行而逐渐减小，从而避免在最优解附近震荡。

保存模型：使用 np.savez 函数，将网络的参数（卷积层和全连接层的权重和偏置，批归一化层的 gamma、beta、running_mean 和 running_var）保存到一个文件（data2.npz）中，方便后续加载和使用。

```
    def train(self):
        data = self.data
        X_train, y_train = data.load_train()

        learning_rate = self.learning_rate
        dropout_rate = self.dropout_rate
        epoch_num = self.epoch_num
        batch_size = self.batch_size

        conv1 = Conv(kernel_shape=(5, 5, 1, 6))
        relu1 = Relu()
        pool1 = Pool()
        conv2 = Conv(kernel_shape=(5, 5, 6, 16))  # 8x8x16
        relu2 = Relu()
        pool2 = Pool()  # 4x4x16
        link1 = Linear(256, 10)
        softmax = Softmax()
        dropout1 = Dropout(rate=dropout_rate)
        batchnorm1 = BatchNorm(6)
```

```
20.
21.      for epoch in range(epoch_num):
22.          for i in range(0, self.train_size, batch_size):
23.              X = X_train[i:i + batch_size]
24.              Y = y_train[i:i + batch_size]
25.
26.              predict = conv1.forward(X)
27.              predict = batchnorm1.forward(predict)
28.              predict = relu1.forward(predict)
29.              predict = dropout1.forward(predict, training=True)
30.              predict = pool1.forward(predict)
31.              predict = conv2.forward(predict)
32.              predict = relu2.forward(predict)
33.              predict = pool2.forward(predict)
34.              predict = predict.reshape(batch_size, -1)
35.              predict = link1.forward(predict)
36.
37.              loss, delta = softmax.cal_loss(predict, Y)
38.
39.              delta = link1.backward(delta, learning_rate)
40.              delta = delta.reshape(batch_size, 4, 4, 16)
41.              delta = pool2.backward(delta)
42.              delta = relu2.backward(delta)
43.              delta = conv2.backward(delta, learning_rate)
44.              delta = pool1.backward(delta)
45.              delta = dropout1.backward(delta)
46.              delta = relu1.backward(delta)
47.              delta = batchnorm1.backward(delta, learning_rate)
48.              conv1.backward(delta, learning_rate)
49.
50.              print("Epoch-{}-{:05d}".format(str(epoch), i), ":", "loss:{:.4f}".format(loss))
51.
52.          learning_rate *= 0.95 ** (epoch + 1)
53.          np.savez("data2.npz", k1=conv1.k, b1=conv1.b, k2=conv2.k, b2=conv2.b, w3=link1.W, b3=link1.
     b,
54.                   g1=batchnorm1.gamma, beta1=batchnorm1.beta, run_mean1=batchnorm1.running_mean,
55.                   run_var1=batchnorm1.running_var)
```

### 2.4.2 测试

eval 方法是类的评估函数，用于测试卷积神经网络的性能。这里的主要步骤包括：加载模型：使用 np.load 函数，从文件（data2.npz）中读取网络的参数，并赋值给相应

的层。

从 self.data 中加载测试集的特征和标签，分别赋值给 X_test 和 y_test。使用单层循环来遍历测试集，每次取一个样本，进行前向传播和预测。具体步骤如下：

前向传播：将输入数据 X 依次通过卷积层、批归一化层、激活函数、池化层和全连接层，得到输出 predict。注意，这里不需要使用 dropout 层和 softmax 层，因为测试时不需要丢弃神经元，也不需要计算概率分布。

预测：根据输出 predict 和真实标签 Y，判断预测是否正确，如果正确，就将计数器 num 加一。

打印信息：打印测试集的大小（size）和正确率（num / size）。

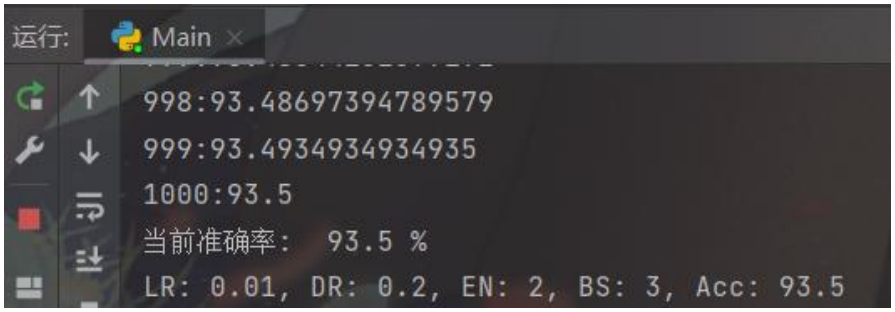
```
    def eval(self):
        r = np.load("data2.npz")
        data = self.data
        X_test, y_test = data.load_test()
        conv1 = Conv(kernel_shape=(5, 5, 1, 6))  # 24x24x6
        batchnorm1 = BatchNorm(6)
        relu1 = Relu()
        pool1 = Pool()  # 12x12x6
        conv2 = Conv(kernel_shape=(5, 5, 6, 16))  # 8x8x16
        relu2 = Relu()
        pool2 = Pool()  # 4x4x16
        link1 = Linear(256, 10)
        softmax = Softmax()

        conv1.k = r["k1"]
        conv1.b = r["b1"]
        conv2.k = r["k2"]
        conv2.b = r["b2"]
        link1.W = r["w3"]
        link1.n = r["b3"]
        batchnorm1.gamma = r["g1"]
        batchnorm1.beta = r["beta1"]
        batchnorm1.running_mean = r["run_mean1"]
        batchnorm1.running_var = r["run_var1"]

        num = 0
        size = len(y_test)
        print(size)
        for i in range(size):
            X = X_test[i]
            X = X[np.newaxis, :]
```

```
            Y = y_test[i]

            predict = conv1.forward(X)
            predict = batchnorm1.forward(predict, training=False)
            predict = relu1.forward(predict)
            predict = pool1.forward(predict)
            predict = conv2.forward(predict)
            predict = relu2.forward(predict)
            predict = pool2.forward(predict)
            predict = predict.reshape(1, -1)
            predict = link1.forward(predict)

            predict = softmax.predict(predict)

            if np.argmax(predict) == Y:
                num += 1
                print(f'{i + 1}:{num / (i + 1) * 100}')

        print("TEST-ACC: ", num / size * 100, "%")
        return num / size * 100
```

### 2.4.3 输出结果

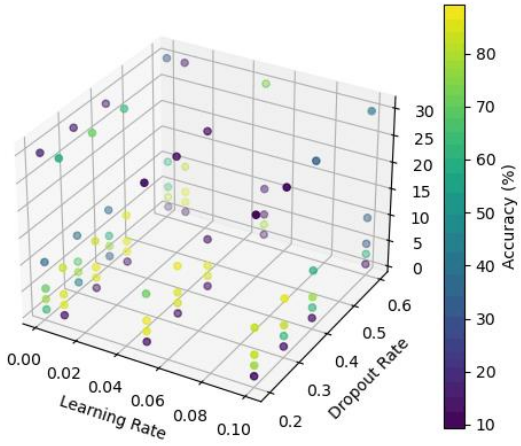
在训练时每一次迭代后，会输出当前损失值变化情况


在测试时会得到正确率情况，（下方为所输入的超参数组合）




多种超参数训练完成后，会将结果以 csv 形式存在本地，并生成一张三维图片

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | | Learning F | Dropout Ra | Batch Size | Accuracy | |
| 2 | 62 | 0.05 | 0.2 | 5 | 89.3 | |
| 3 | 31 | 0.01 | 0.4 | 3 | 88.6 | |
| 4 | 68 | 0.05 | 0.3 | 10 | 88.3 | |
| 5 | 37 | 0.01 | 0.6 | 5 | 88.2 | |
| 6 | 27 | 0.01 | 0.3 | 5 | 87.3 | |
| 7 | 36 | 0.01 | 0.6 | 3 | 87.1 | |
| 8 | 32 | 0.01 | 0.4 | 5 | 86.4 | |
| 9 | 48 | 0.1 | 0.3 | 10 | 85.9 | |
| 10 | 61 | 0.05 | 0.2 | 3 | 85.4 | |
| 11 | 26 | 0.01 | 0.3 | 3 | 85.2 | |
| 12 | 33 | 0.01 | 0.4 | 10 | 85.2 | |
| 13 | 67 | 0.05 | 0.3 | 5 | 85.2 | |
| 14 | 72 | 0.05 | 0.4 | 5 | 85.1 | |
| 15 | 21 | 0.01 | 0.2 | 3 | 85 | |
| 16 | 22 | 0.01 | 0.2 | 5 | 84.2 | |
| 17 | 23 | 0.01 | 0.2 | 10 | 82.9 | |
| 18 | 71 | 0.05 | 0.4 | 3 | 82.7 | |
| 19 | 66 | 0.05 | 0.3 | 3 | 82.7 | |
| 20 | 42 | 0.1 | 0.2 | 5 | 82.1 | |
| 21 | 47 | 0.1 | 0.3 | 5 | 82.1 | |
| 22 | 43 | 0.1 | 0.2 | 10 | 81.9 | |
| 23 | 28 | 0.01 | 0.3 | 10 | 81.2 | |
| 24 | 38 | 0.01 | 0.6 | 10 | 79.8 | |




# 3. 题目问题回答与深入研究

## 3.1Dropout 和 Normalization 方法

### 3.1.1Dropout

Dropout 是一种防止神经网络过拟合的技术，它的基本思想是在训练过程中随机丢弃一些神经元，使得网络结构变得更加稀疏，从而减少参数之间的共适应，提高网络的泛化能力。

Dropout 类有三个主要的方法：

__init__ 方法是类的构造函数,用于初始化一个参数 rate，表示丢弃神经元的概率。

forward 方法是类的前向传播函数，用于实现神经元的随机丢弃。它有两个参数：x 是输入的数据，training 是一个布尔值，表示是否处于训练模式。如果是训练模式，那么它会生成一个和 x 形状相同的随机二项分布 self.mask，其中每个元素的值为 0 或 1，且 1 的概率为 1 - self.rate。然后它会将 x 和 self.mask 相乘，相当于将 x 中的一些元素置为 0，同时将剩余的元素除以 1 - self.rate，以保持期望不变。如果是测试模式，那么它会直接返回 x，不做任何改变。

backward 方法是类的反向传播函数，用于传递误差梯度。它有一个参数：delta 是上一层传来的误差梯度。它会将 delta 和 self.mask 相乘，相当于只保留没有被丢弃的神经元的梯度，然后返回结果。

在代码中，Dropout 类被用在了第一个卷积层和第一个激活函数之间，即 predict = dropout1.forward(predict, training=True)。这样做的目的是为了增加输入层的鲁棒性，防止网络过度依赖某些特定的输入特征 4。在反向传播时，也调用 delta = dropout1.backward(delta)，以更新相应的参数。

```
class Dropout:
    def __init__(self, rate):
        self.rate = rate  # dropout 的概率

    def forward(self, x, training=True):
        if training:
            self.mask = np.random.binomial(1, 1 - self.rate, size=x.shape) / (1 - self.rate)
            return x * self.mask
        else:
            return x

    def backward(self, delta):
        return delta * self.mask
```

### 3.1.2Batch Normalization

Batch Normalization 是一种数据归一化的方法，它可以加速深度神经网络的训练，提高模型的泛化能力，减少梯度消失或爆炸的风险。它的基本思想是在每个层的输入进行归一化，使得每个特征的均值为 0，方差为 1，然后通过可学习的参数进行缩放和平移，以保持网络的表达能力。

BatchNorm 类是 Batch Normalization 的一个实现，它有以下几个主要的方法：

__init__ 方法是类的构造函数，用于初始化一些参数。这里的参数包括：

num_features 是特征的数量，也就是输入数据的通道数。

gamma 和 beta 是可学习的缩放和平移参数，初始值分别为 1 和 0，它们的形状与 num_features 相同，可以在通道维度上广播。

running_mean 和 running_var 是运行时的均值和方差，用于在测试时代替批次的均值和方差，初始值分别为 0 和 1，它们不参与训练，而是通过指数移动平均的方式更新。

x_norm 是归一化后的数据，用于在反向传播时计算梯度。

mu 和 var 是批次的均值和方差，用于在反向传播时计算梯度。

eps 是一个很小的正数，用于防止除以 0 的情况。

x_reshaped 是将输入数据转换为 (N*H*W, C) 形状的数据，用于计算每个通道的均值和方差。

```
class BatchNorm:
    def __init__(self, num_features):
        # Gamma: 缩放参数，Beta: 平移参数，都需要学习
        self.gamma = np.ones(num_features)
        self.beta = np.zeros(num_features)
        # 运行时的均值和方差
        self.running_mean = np.zeros(num_features)
        self.running_var = np.zeros(num_features)
        # 用于反向传播的保存参数
```

```
        self.x_norm = None
        self.mu = None
        self.var = None
        self.eps = 1e-5  # 防止除以0
        self.x_reshaped = None
```

forward 方法是类的前向传播函数，用于实现数据的归一化、缩放和平移。它有三个参数：x 是输入的数据，training 是一个布尔值，表示是否处于训练模式，momentum 是一个浮点数，表示运行时均值和方差的更新速率。它的主要步骤包括：
获取输入数据的形状，即 (N, H, W, C)，其中 N 是批次大小，H 和 W 是高度和宽度，C 是通道数。

将 gamma 和 beta 的形状转换为 (1, 1, 1, C)，以便在通道维度上广播。

将输入数据的形状转换为 (N*H*W, C)，以便计算每个通道的均值和方差。

如果处于训练模式，那么：

计算批次的均值和方差，分别为 mu 和 var。

将输入数据减去均值，除以方差的平方根加上 eps，得到归一化后的数据，记为 x_norm。

将归一化后的数据的形状恢复为 (N, H, W, C)。

将运行时的均值和方差分别更新为 momentum 乘以原值加上 (1 - momentum) 乘以批次的均值和方差。

如果处于测试模式，那么：

将输入数据减去运行时的均值，除以运行时的方差的平方根加上 eps，得到归一化后的数据，记为 x_norm。

将归一化后的数据的形状恢复为 (N, H, W, C)。

将 gamma 乘以归一化后的数据，再加上 beta，得到输出数据，记为 out。

返回输出数据 out。

```
    def forward(self, x, training=True, momentum=0.9):
        N, H, W, C = x.shape
        # 确保 gamma 和 beta 的形状可以在通道维度上广播
        self.gamma = self.gamma.reshape(1, 1, 1, C)
        self.beta = self.beta.reshape(1, 1, 1, C)
        # 转换为(N*H*W, C)以计算每个通道的均值和方差
        x_reshaped = x.reshape((N * H * W, C))
        self.x_reshaped = x_reshaped

        if training:
            mu = x_reshaped.mean(axis=0)
            var = x_reshaped.var(axis=0)

            self.x_norm = (x_reshaped - mu) / np.sqrt(var + self.eps)
            self.x_norm = self.x_norm.reshape(N, H, W, C)

            self.running_mean = momentum * self.running_mean + (1 - momentum) * mu
            self.running_var = momentum * self.running_var + (1 - momentum) * var

            self.mu = mu
```

```
            self.var = var
        else:
            x_norm = (x_reshaped - self.running_mean) / np.sqrt(self.running_var + self.eps)
            self.x_norm = x_norm.reshape(N, H, W, C)
        out = self.gamma * self.x_norm + self.beta
        return out
```

backward 方法是类的反向传播函数，用于计算梯度并更新参数。它有两个参数：delta 是上一层传来的误差梯度，learning_rate 是学习率。它的主要步骤包括：

获取误差梯度的形状，即 (N, H, W, C)。

将误差梯度的形状转换为 (N*H*W, C)，以匹配归一化数据的形状。

将归一化数据的形状也转换为 (N*H*W, C)，以便计算梯度。

计算 beta 和 gamma 的梯度，分别为 delta 在第 0 维度上的求和，和 x_norm 与 delta 的逐元素乘积在第 0 维度上的求和。

计算归一化数据的梯度，为 delta 乘以 gamma。

计算方差的梯度，为归一化数据的梯度乘以输入数据减去均值，再乘以 -0.5 和方差加上 eps 的 -1.5 次方，在第 0 维度上的求和。

计算均值的梯度，为归一化数据的梯度乘以 -1 除以方差的平方根加上 eps，在第 0 维度上的求和，再加上方差的梯度乘以输入数据减去均值的 -2 倍，在第 0 维度上的平均值。

计算输入数据的梯度，为归一化数据的梯度除以方差的平方根加上 eps，再加上方差的梯度乘以输入数据减去均值的 2 倍除以 N，再加上均值的梯度除以 N。

将输入数据的梯度的形状恢复为 (N, H, W, C)，记为 dx。

将 gamma 和 beta 分别减去学习率乘以它们的梯度，更新参数。

返回输入数据的梯度 dx。

```
    def backward(self, delta, learning_rate):
        N, H, W, C = delta.shape
        # 首先，转换 delta 的形状以匹配归一化数据的形状
        delta_reshaped = delta.reshape((N * H * W, C))
        x_norm_reshaped = self.x_norm.reshape((N * H * W, C))

        # 计算 beta 和 gamma 的梯度
        dbeta = delta_reshaped.sum(axis=0)
        dgamma = np.sum(x_norm_reshaped * delta_reshaped, axis=0)

        # 计算归一化数据的梯度
        dx_norm = delta_reshaped * self.gamma

        # 归一化 x 的梯度
        dvar = np.sum(dx_norm * (self.x_reshaped - self.mu), axis=0) * -0.5 * np.power(self.var + self.eps, -1.5)
        dmu = np.sum(dx_norm * -1 / np.sqrt(self.var + self.eps), axis=0) + dvar * np.mean(
            -2 * (self.x_reshaped - self.mu), axis=0)

        # 归一化梯度
```

```
            dx_reshaped = (dx_norm / np.sqrt(self.var + self.eps)) + (dvar * 2 * (self.x_reshaped - self.mu) / N) + (
                    dmu / N)
            dx = dx_reshaped.reshape(N, H, W, C)
            self.gamma = self.gamma - learning_rate * dgamma
            self.beta = self.beta - learning_rate * dbeta
            return dx
```

在代码中，BatchNorm 类被用在了第一个卷积层和第一个激活函数之间，即 predict = batchnorm1.forward(predict)。这样做的目的是为了对卷积层的输出进行归一化，使得每个通道的特征分布更加稳定，从而加速梯度下降的收敛，提高模型的泛化能力。在反向传播时，也调用 delta = batchnorm1.backward(delta, learning_rate)，以更新相应的参数

## 3. 2 通过交叉验证，寻找最佳超参数

主函数采用通过交叉验证，寻找神经网络最佳超参数。交叉验证是一种通过估计模型的泛化误差，从而进行模型选择的方法。超参数是一些可以影响模型训练效果的参数，比如学习率、丢弃率、迭代次数和批次大小等。

首先，定义超参数的不同组合，分别为 learning_rates, dropout_rates, epoch_nums 和 batch_sizes。这些超参数的含义分别是：

learning_rate：学习率，控制模型在每次更新参数时的步长，太大可能导致不收敛，太小可能导致收敛速度慢 3。

dropout_rate：丢弃率，控制模型在每层神经元中随机丢弃一定比例的神经元，以防止过拟合 4。

epoch_num：迭代次数，控制模型在整个训练集上重复训练的次数，太少可能导致欠拟合，太多可能导致过拟合。

batch_size：批次大小，控制模型在每次训练时使用的数据量，太小可能导致梯度不稳定，太大可能导致内存不足。

```
if __name__ == '__main__':
    # 定义超参数的不同组合
    learning_rates = [0.001, 0.01, 0.1, 0.05]
    dropout_rates = [0.2, 0.3, 0.4, 0.6]
    epoch_nums = [2, 5]
    batch_sizes = [1, 3, 5, 10, 30]
```

然后，为了绘制三维图表，初始化坐标列表，分别为 lr_list, dr_list, bs_list 和 best_acc_list。这些列表的含义分别是：

lr_list：学习率列表，存储每组超参数组合中的学习率。

dr_list：丢弃率列表，存储每组超参数组合中的丢弃率。

bs_list：批次大小列表，存储每组超参数组合中的批次大小。

best_acc_list：最佳准确率列表，存储每组超参数组合中的最佳准确率。

接着，存储结果的字典，命名为 results。这个字典的键是超参数的四元组，即 (lr, dr, en, bs)，表示学习率、丢弃率、迭代次数和批次大小的组合。这个字典的值是对应的准确率，表示模型在测试集上的预测准确率。

```
    # 为了绘制三维图表，我们需要把数据整理成三维坐标形式
        # 初始化坐标列表
        lr_list = []
        dr_list = []
        bs_list = []
        best_acc_list = []
        # 存储结果的字典
        results = {}
```

然后，进行交叉验证，遍历所有的超参数组合，对于每一种组合，执行以下操作：初始化 Run 类的实例，传入训练集大小、测试集大小、学习率、丢弃率、迭代次数和批次大小等参数。Run 类是一个自定义的类，用于封装神经网络模型的训练和评估过程。

调用 run.train() 方法，用训练集训练神经网络模型。

调用 run.eval() 方法，用测试集评估神经网络模型的准确率。

将超参数组合和对应的准确率存储到 results 字典中。

打印当前的超参数组合和准确率。

接下来，选取每组 learning_rate, dropout_rate 和 batch_size 组合下，epoch_num 最好的结果。这是因为 epoch_num 的不同可能导致模型的收敛程度不同，所以我们只关注每组超参数组合中最优的一种。对于每一种组合，执行以下操作：

找出当前组合下，epoch_num 的最大准确率，记为 best_acc。

将当前组合和对应的最佳准确率添加到坐标列表中，即 lr_list, dr_list, bs_list 和 best_acc_list。

```
    # 进行交叉验证
        for lr in learning_rates:
            for dr in dropout_rates:
                for en in epoch_nums:
                    for bs in batch_sizes:
                        # 初始化 Run 类的实例
                        run = Run(train_size=600, test_size=1000, learning_rate=lr, dropout_rate=dr, e
poch_num=en,
                                  batch_size=bs)
                        # 训练模型
                        run.train()
                        # 评估模型
                        accuracy = run.eval()
                        # 存储结果
                        results[(lr, dr, en, bs)] = accuracy
                        print(f"LR: {lr}, DR: {dr}, EN: {en}, BS: {bs}, Acc: {accuracy}")

    # 选取每组 learning_rate, dropout_rate 和 batch_size 组合下，epoch_num 最好的结果
    for lr in learning_rates:
        for dr in dropout_rates:
            for bs in batch_sizes:
                # 找出当前组合下，epoch_num 的最大准确率
                best_acc = max(results[(lr, dr, en, bs)] for en in epoch_nums)
```

```
            # 将当前组合和对应的最佳准确率添加到列表
            lr_list.append(lr)
            dr_list.append(dr)
            bs_list.append(bs)
```

然后，绘制三维图表，展示不同超参数组合下的最佳准确率。

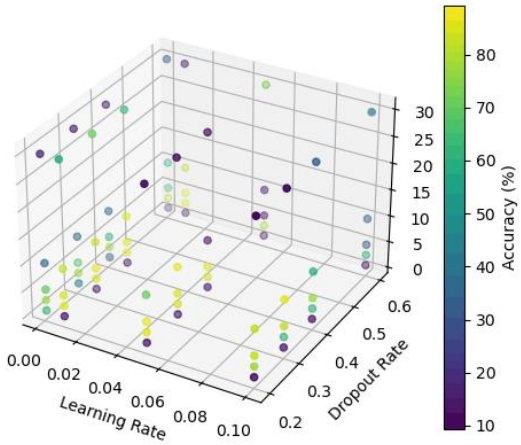
最后，创建数据字典，用于存储坐标列表中的数据，分别为 Learning Rate, Dropout Rate, Batch Size 和 Accuracy。然后，创建一个数据框，命名为 df，用于将数据字典转换为表格形式。接着，使用数据框的 sort_values 方法按 Accuracy 列降序排序，得到最优的超参数组合。然后，将数据框保存为 results.csv 文件。最后，绘制表格，但不包含索引。具体操作如下：

创建一个图形对象，命名为 fig。

result2.csv

result2.png

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | | Learning F | Dropout Ra | Batch Size | Accuracy | |
| 2 | 62 | 0.05 | 0.2 | 5 | 89.3 | |
| 3 | 31 | 0.01 | 0.4 | 3 | 88.6 | |
| 4 | 68 | 0.05 | 0.3 | 10 | 88.3 | |
| 5 | 37 | 0.01 | 0.6 | 5 | 88.2 | |
| 6 | 27 | 0.01 | 0.3 | 5 | 87.3 | |
| 7 | 36 | 0.01 | 0.6 | 3 | 87.1 | |
| 8 | 32 | 0.01 | 0.4 | 5 | 86.4 | |
| 9 | 48 | 0.1 | 0.3 | 10 | 85.9 | |
| 10 | 61 | 0.05 | 0.2 | 3 | 85.4 | |
| 11 | 26 | 0.01 | 0.3 | 3 | 85.2 | |
| 12 | 33 | 0.01 | 0.4 | 10 | 85.2 | |
| 13 | 67 | 0.05 | 0.3 | 5 | 85.2 | |
| 14 | 72 | 0.05 | 0.4 | 5 | 85.1 | |
| 15 | 21 | 0.01 | 0.2 | 3 | 85 | |
| 16 | 22 | 0.01 | 0.2 | 5 | 84.2 | |
| 17 | 23 | 0.01 | 0.2 | 10 | 82.9 | |
| 18 | 71 | 0.05 | 0.4 | 3 | 82.7 | |
| 19 | 66 | 0.05 | 0.3 | 3 | 82.7 | |
| 20 | 42 | 0.1 | 0.2 | 5 | 82.1 | |
| 21 | 47 | 0.1 | 0.3 | 5 | 82.1 | |
| 22 | 43 | 0.1 | 0.2 | 10 | 81.9 | |
| 23 | 28 | 0.01 | 0.3 | 10 | 81.2 | |
| 24 | 38 | 0.01 | 0.6 | 10 | 79.8 | |




如图即为代码中寻找超参数的一组数据结果

## 3.3 使用其它不同方法优化神经网络

### 3.3.1 增加卷积层层数

原有一层卷积，其神经网络正确率只能在 85%左右，额外加入一层卷积后，在相同数据集和迭代次数下，神经网络的正确率可达 95%。

两层卷积的神经网络框架：

使用了 conv2 和 conv1 两个卷积层

```
    conv1 = Conv(kernel_shape=(5, 5, 1, 6))
            relu1 = Relu()
            pool1 = Pool()
            conv2 = Conv(kernel_shape=(5, 5, 6, 16))  # 8x8x16
            relu2 = Relu()
```

```
        pool2 = Pool()  # 4x4x16
        link1 = Linear(256, 10)
        softmax = Softmax()
        dropout1 = Dropout(rate=dropout_rate)
        batchnorm1 = BatchNorm(6)

        for epoch in range(epoch_num):
            for i in range(0, self.train_size, batch_size):
                X = X_train[i:i + batch_size]
                Y = y_train[i:i + batch_size]

                predict = conv1.forward(X)
                predict = batchnorm1.forward(predict)
                predict = relu1.forward(predict)
                predict = dropout1.forward(predict, training=True)
                predict = pool1.forward(predict)
                predict = conv2.forward(predict)
                predict = relu2.forward(predict)
                predict = pool2.forward(predict)
                predict = predict.reshape(batch_size, -1)
                predict = link1.forward(predict)
```

### 3. 3. 2 动态学习率

动态学习率是指在训练过程中根据一定的规则或策略调整学习率的大小，以达到更好的训练效果和收敛速度。学习率是控制参数更新步长的超参数，它影响着模型的学习速度和最终的性能。

代码中，动态学习率的作用是在每个 epoch 结束后，将学习率乘以一个衰减因子，即 learning_rate *= 0.95 ** (epoch + 1)。这样可以使学习率随着训练轮数的增加而逐渐减小，从而避免在接近最优解时出现过大的波动或震荡 3。这种动态学习率的策略属于指数衰减（exponential decay），它是一种常用的学习率调整方法。

```
    conv1.backward(delta, learning_rate)

                print("迭代-{}-{:05d}".format(str(epoch), i), ":", "损失值:{:.4f}".format(loss))

            learning_rate *= 0.95 ** (epoch + 1)
            np.savez("data2.npz", k1=conv1.k, b1=conv1.b, k2=conv2.k, b2=conv2.b, w3=link1.W, b3=lin
k1.b,
                     g1=batchnorm1.gamma, beta1=batchnorm1.beta, run_mean1=batchnorm1.running_mean,

                     run_var1=batchnorm1.running_var)
```

### 3.3.3Xavier 初始化

CNN 卷积层使用的初始化方式是 Xavier 初始化，它是一种根据输入和输出神经元的数量自适应调整权重分布的方差的方法。Xavier 初始化的目的是使每一层的输入和输出的方差保持一致，从而避免梯度消失或爆炸的问题。

代码中，卷积层的权重 k 和偏置 b 都是从一个均值为 0，方差为 scale 的正态分布中随机采样的，其中 scale 是一个根据卷积核的形状计算出来的缩放因子，它等于输入通道数、卷积核宽度和高度的乘积除以输出通道数的平方根。这样可以保证卷积层的输出的方差接近于输入的方差，从而提高网络的训练效果和收敛速度。

```
1.  class Conv:
2.      def __init__(self, kernel_shape, path=1):
3.          width, height, in_channel, out_channel = kernel_shape
4.          self.path = path
5.           scale = np.sqrt(3 * in_channel * width * height / out_channel)  # 使用 Xavier 初始化缩放因
    子
6.          self.k = np.random.standard_normal(kernel_shape) / scale  # 初始化 k，为一个四维向量
7.          self.b = np.random.standard_normal(out_channel) / scale  # 初始化 b，为一个一维向量
```

## 4. 经验总结

首先，我选择了 MNIST 数据集作为我的训练和测试数据，它包含了 10 个类别的手写数字图像，每个类别有 7000 张图像，其中 6000 张用于训练，1000 张用于测试 1。我使用了 Python 语言和 numpy 库来实现我的 CNN 模型。

其次，我设计了一个基于卷积层、池化层、全连接层和 Softmax 层的 CNN 结构，它可以从输入的 28x28 的灰度图像中提取特征，并输出 10 个类别的概率分布。我选择了 ReLU 作为我的激活函数，选择了多分类交叉熵损失函数，选择了随机梯度下降法作为我的优化算法。

然后，我实现了 CNN 的前向传播和反向传播的过程，以及卷积运算、池化运算、激活函数、损失函数和梯度更新的细节。我参考了一些网上的教程和博客，并且自己编写了一些辅助函数，来简化代码的逻辑和可读性。

接着，我训练了我的 CNN 模型，设置了一些超参数，如学习率、迭代次数、批次大小等，并且使用了动态学习率的策略，即在每个 epoch 结束后，将学习率乘以一个衰减因子，以达到更好的收敛效果。我还记录了每个 epoch 的损失值，并且绘制了误差发展的图表，来观察模型的学习情况。

最后，我测试了我的 CNN 模型，使用了测试集的数据，，并且打印了结果。我还尝试了一些提升性能的方法，如增加模型的层数，使用不同的初始化方法等，来比较不同方法的效果。

通过这个项目，我学习了 CNN 的基本原理和实现方法，加深了对 CNN 的理解和掌握，也提高了我的编程和调试能力。