一、问题引入

手写数字识别是图像识别任务中常见的任务，计算机通过手写体图片来识别出图片中的字，与印刷字体不同的是，不同人的手写体风格迥异，大小不一， 造成了计算机对手写识别任务的一些困难。本文将手写数值识别描述成优化问题，采用梯度下降法[1]求出识别手写数字的局部最优参数。为了更好的描述如何在手写数字识别任务中使用梯度下降进行参数优化，这里我对问题进行简化和定义。

手写数字识别任务的识别对象是手写体的数字，为了专注于对字体的判读，这里采用单通道灰度图像（如图 1）进行手写数字识别。

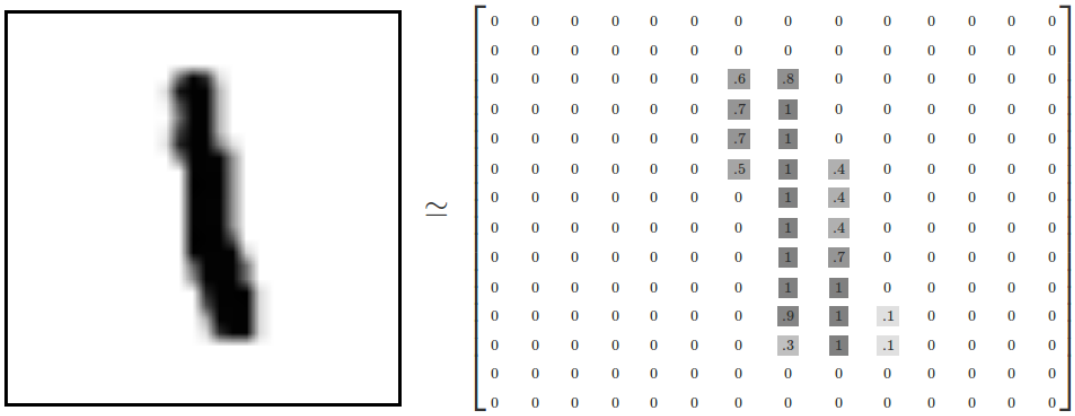

图 1 单通道图片

手写数字识别的目标就是给定一张写有数字的 28*28 的单通道的图片，判断图片上的数字。

二、数学模型

1. 多层感知机模型

本文介绍使用多层感知机作为手写数字的识别模型。首先，我们将单通道图片可以看成一个二维矩阵，矩阵中每个元素表示一个像素点，范围在[0,255]之间。然后，我们将这个二维矩阵平展列成一维向量$\mathrm{v} \in \mathrm{R}^{n}$。为了模型能更好的优化，我们将数值除以 255 来规约到[0,1]之间。之后，将 v 最为多层感知机[2]的输入。多层感知机的功能可以使用以下公式表述：

$$\mathrm{O}_1 = a(\mathrm{W}_1 v + b_1)$$
$$O_2 = a(W_2 O_1 + b_2)$$
$$\cdots$$
$$O_{\mathrm{n}} = W_{\mathrm{n}} O_{n-1} + b_n$$
$$\mathrm{y} = \mathrm{softmax}(\mathrm{O}_{\mathrm{n}})$$

其 中 ， $\mathrm{W}_1 \in \mathrm{R}^{n_1 \times n}, W_{\mathrm{n}-1} \in R^{n_2 \times n_1}, \ldots, W_{\mathrm{n}-1} \in R^{n_{n-1} \times \mathrm{n}_{n-2}}, W_n \in \mathrm{R}^{10 \times n_{n-1}}, \mathrm{b_i} \in \mathrm{R}^{\mathrm{n_i}}, a$表示激活函数，本文使用 Relu 作为激活函数。

令$\mathrm{O_n} = \begin{bmatrix} o_1 \\ o_2 \\ \ldots \\ o_k \end{bmatrix}$，则 $\mathrm{softmax}(\mathrm{O_n}) = \begin{bmatrix} y_1 \\ y_2 \\ \ldots \\ y_k \end{bmatrix}$，其中$\mathrm{y_i} = \frac{e^{o_i}}{\sum_j e^{o_j}}$ 。

2. 优化目标（损失函数定义）

经过多层感知机的识别，我们可以得到一个结果y，因为y经过 softmax 变换，所以y实质是一个 0~9 数字的概率分布（$\sum_j y_j$=1）。其中概率最大的数字作为模型识别出来的结果。

那么我们优化目标就可以表述成：

$$\text{argmax}_\theta \{y_t * p(y|\theta, v)\} \quad (1)$$

其中θ就是多层感知机中的参数矩阵，p(.)表示概率。y_t表示 v 对应的真实的数字的标签。当然，因为 softmax 中有一个指数变换，为了后面计算梯度的方便防止梯度消失或梯度爆炸，在目标函数上加上一个 log 来抵消指数变换的影响，所以目标函数可以表述成：

$$\text{argmax}_\theta \{-y_t * log\ p(y|\theta, v)\} \quad (2)$$

3. 优化

优化公式（2）可以使用梯度下降法[1]来求最优解，但是多层感知机的函数形式层层堆叠，直接计算会很困难，所以根据求导的链式法则，使用反向梯度传播由后向前一层一层的计算参数的梯度，然后进行梯度更新。在本文中，我们以多层感知机的其中一层展示反向梯度传播。

$$\frac{\partial O_i}{\partial W_i} = \frac{\partial a}{\partial (W_i O_{i-1} + b_i)} * \frac{\partial (W_i O_{i-1} + b_i)}{\partial W_i} = \frac{\partial a}{\partial (W_i O_{i-1} + b_i)} * O_{i-1} \quad (3)$$

$$\frac{\partial O_i}{\partial b_i} = \frac{\partial a}{\partial (W_i O_{i-1} + b_i)} * \frac{\partial (W_i O_{i-1} + b_i)}{\partial b_i} = \frac{\partial a}{\partial (W_i O_{i-1} + b_i)} \quad (4)$$

其中，O_{i-1}是多层感知机前向计算时第 i-1 层的结果。

三、实验与结果分析

1. 数据集与参数设置

实验采用手写数字识别数据集 MNIST 作为实验对象。MNIST 数据集（修改的国家标准与技术研究所——Modified National Institute of Standards and Technology），是一个大型的包含手写数字图片的数据集。该数据集由 0-9 手写数字图片组成，共 10 个类别。总共有 70000 张手写数字图片，取其中 60000 张作为训练数据，10000 张作为测试数据。

每张图片的大小为 28 * 28，所以n = 28 * 28 = 784。实验使用多层感知机层数为 3 层，学习率为 0.001。n_1=256, n_2=128, n_3=10。

2. 实验结果

经过 200 epoch 的迭代，3 层感知机达到 0.8957 的准确率，实验结果如图 2 所示。实验结果表明使用多层感知机模型对手写数字进行识别可以获得不错的准确率。事实上，增加多层感知机神经元个数，增加神经网络层数，可以进一步提高模型效率。

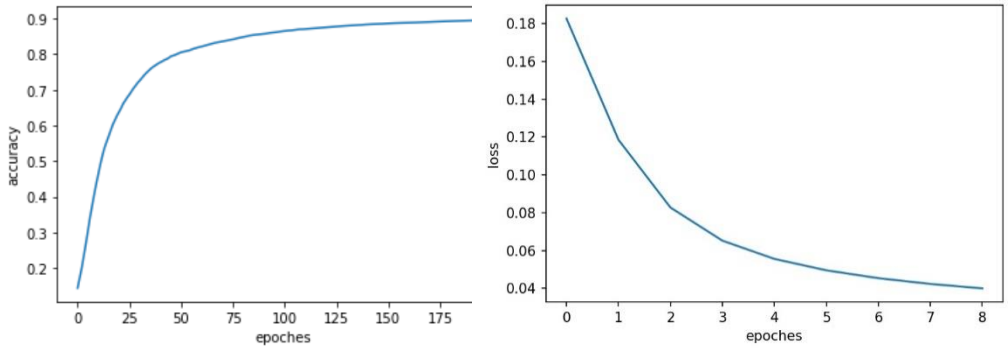

图 2 准确率和损失

四、参考文献

[1] Bottou L. Large-scale machine learning with stochastic gradient descent[M]//Proceedings of COMPSTAT'2010. Physica-Verlag HD, 2010: 177-186.
[2]《统计学习方法》,李航.

五、附录

```
import    tensorflow as tf
from       tensorflow import keras
from       tensorflow.keras import datasets
import    os
# 利用 Tensorflow2 中的接口加载 mnist 数据集
(x, y), (x_test, y_test) = datasets.mnist.load_data()
# 对数据进行预处理
def preprocess(x, y):
       x = tf.cast(x, dtype=tf.float32) / 255.
       y = tf.cast(y, dtype=tf.int32)
        return x,y
# 构建 dataset 对象，方便对数据的打乱，批处理等超操作
train_db = tf.data.Dataset.from_tensor_slices((x,y)).shuffle(1000).batch(128)
train_db = train_db.map(preprocess)
test_db = tf.data.Dataset.from_tensor_slices((x_test,y_test)).batch(128)
test_db = test_db.map(preprocess)
# 构建模型中会用到的权重
w1 = tf.Variable(tf.random.truncated_normal([784, 256], stddev=0.1))
b1 = tf.Variable(tf.zeros([256]))
w2 = tf.Variable(tf.random.truncated_normal([256, 128], stddev=0.1))
b2 = tf.Variable(tf.zeros([128]))
w3 = tf.Variable(tf.random.truncated_normal([128, 10], stddev=0.1))
b3 = tf.Variable(tf.zeros([10]))
# 学习率
lr = 1e-3
```

```
ac = []
losses = []
ls = 0
# epoch 表示整个训练集循环的次数 这里循环 100 次
for epoch in range(200):
    # step 表示当前训练到了第几个 Batch
    for step, (x, y) in enumerate(train_db):
        # 把训练集进行打平操作
        x = tf.reshape(x, [-1, 28*28])
        # 构建模型并计算梯度
        with tf.GradientTape() as tape: # tf.Variable
            # 三层非线性模型搭建
            h1 = x@w1 + tf.broadcast_to(b1, [x.shape[0], 256])
            h1 = tf.nn.relu(h1)
            h2 = h1@w2 + b2
            h2 = tf.nn.relu(h2)
            out = h2@w3 + b3
            out = tf.nn.softmax(out)
            # 把标签转化成 one_hot 编码
            y_onehot = tf.one_hot(y, depth=10)

            # 计算交叉熵
            loss = -y_onehot*tf.math.log(out+1e-6)
            #loss = tf.square(y_onehot - out)
            loss = tf.reduce_mean(loss)
            ls += loss

        # 计算梯度
        grads = tape.gradient(loss, [w1, b1, w2, b2, w3, b3])

        # w = w - lr * w_grad
        # 利用上述公式进行权重的更新
        w1.assign_sub(lr * grads[0])
        b1.assign_sub(lr * grads[1])
        w2.assign_sub(lr * grads[2])
        b2.assign_sub(lr * grads[3])
        w3.assign_sub(lr * grads[4])
        b3.assign_sub(lr * grads[5])

        # 每训练 100 个 Batch 打印一下当前的 loss
        if step % 100 == 0:
            ls /= 100
            losses.append(ls)
            #print(epoch, step, 'loss:', float(ls))
```

```
# 每训练完一次数据集 测试一下啊准确率
total_correct, total_num = 0, 0
for step, (x,y) in enumerate(test_db):

    x = tf.reshape(x, [-1, 28*28])

    h1 = tf.nn.relu(x@w1 + b1)
    h2 = tf.nn.relu(h1@w2 + b2)
    out = h2@w3 +b3
    # 把输出值映射到[0~1]之间
    prob = tf.nn.softmax(out, axis=1)
    # 获取概率最大值得索引位置
    pred = tf.argmax(prob, axis=1)
    pred = tf.cast(pred, dtype=tf.int32)

    correct = tf.cast(tf.equal(pred, y), dtype=tf.int32)
    correct = tf.reduce_sum(correct)
    # 获取每一个 batch 中的正确率和 batch 大小
    total_correct += int(correct)
    total_num += x.shape[0]
# 计算总的正确率
acc = total_correct / total_num
print('\r test acc:', acc,end='')
ac.append(acc)
```