AI 计算系统
实验操作手册

实验序号：一

实验名称：基于三层感知机实现手写数字识别

教　　师：朱光辉

学　　校：南京大学

时　　间：2022 年 8 月 23 日

一 基于三层感知机实现手写数字识别

1.1 实验目的

本实验的目的是掌握神经网络的设计原理，掌握神经网络的训练和推理方法。能够使用`python`语言实现一个可以进行手写数字分类的三层全连接神经网络的训练和推理，主要包括：

1、实现三层神经网络模型进行手写数字分类，建立完整的神经网络工程，通过本实验理解神经网络中基本模块的作用和模块间的联系，为后续建立更复杂的神经网络奠定基础。

2、利用`python`语言实现神经网络的基本单元的前向传播和反向传播，加深对神经网络中基本单元（全连接层、激活函数、损失函数等）的理解。

3、利用`python`语言实现神经网络训练所使用的梯度下降算法，加深对神经网络训练过程的理解

1.2 背景介绍

1.2.1 MLP网络结构

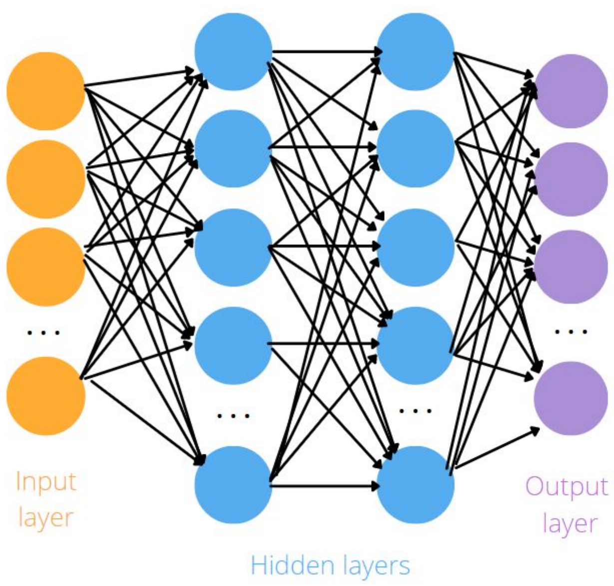
典型的`MLP`包括：**输入层、隐含层和输出层**，`MLP`神经网络不同层之间是全连接的（即**全连接层**）。`MLP`网络主要有三个基本要素：权重、偏置和激活函数。

权重：神经元之间的连接强度由权重表示，权重的大小表示可能性的大小。

偏置：即神经元的激活阈值，控制激活感知器的难易度。

激活函数：起非线性映射的作用，加强神经网络的拟合能力。

下图为三层`MLP`网络的示意图，左侧为输入层，中间两层为隐含层，右侧为输出层。



1.2.2 基本算子

1.2.2.1 全连接层

全连接层中每一个结点都与上一层的所有结点相连，其本质为由一个特征空间线性变换到另一个特征空间，核心操作为矩阵向量相乘。全连接层以向量作为输入，输入与权重相乘后再与偏置相加得到输出向量。即对输入数据进行线性变换，公式如下：

$$Y = XW^T + b$$

其中：输入为X，输出为Y，权重为矩阵W，偏置为b。

计算反向传播时，给定神经网络损失函数L对当前全连接层的输出Y的偏导$\nabla_Y L = \frac{\partial L}{\partial Y}$，其维度与全连接层的输出$Y$相同。根据链式法则，全连接层的权重的梯度$\nabla_W L = X^T \nabla_Y L$，偏置的梯度$\nabla_b L = \nabla_Y L$，以及损失函数对输入的偏导$\nabla_X L = \nabla_Y LW$，

算子初始化参数：

`in_features`：输入样本尺寸

`out_features`：输出样本尺寸

`has_bias`：是否使用偏置，默认为`True`

1.2.2.2 激活函数层

激活函数对于人工神经网络模型学习、理解复杂和非线性的函数来说具有十分重要的作用。它们将非线性特性引入到网络中，增强网络模型的拟合能力。若没有激活函数，无论网络有多少层，输出都是输入的线性组合，缺乏应对非线性模型的能力。

激活函数按元素进行运算，将输出非线性映射。本实验中使用`ReLU`激活函数，公式如下：

$$ReLU(x) = max(0, x)$$

其中，函数将输入中元素值小于0的对应输出置零。反向传播时，当输入大于0，该位置对应的梯度不变，当输入小于0，该位置对应的梯度置零。因此`ReLU`激活函数中第i个输入的偏导公式为：

$$\nabla_{x(i)} L = \begin{cases} \nabla_{y(i)} L & x(i) \geq 0 \\ 0 & x(i) < 0 \end{cases}$$

1.2.2.3 损失函数

损失函数是指将随机事件或其有关随机变量的取值映射为非负实数以表示该随机事件的“风险”或“损失”的函数。在应用中，损失函数通常作为学习准则与优化问题相联系，即通过最小化损失函数求解和评估模型。

本实验中，损失为预测值和真实值的差值，使用交叉熵损失函数，公式如下：

$$crossentropy\ loss = -\frac{1}{p} \sum_{i,j} Y(i,j) ln(\frac{e^{X(i,j)}}{\sum_j e^{X(i,j)}})$$

其中：$X(i,j)$为上一层的输出，$ln(\frac{e^{X(i,j)}}{\sum_j e^{X(i,j)}})$为上层输出的`logsoftmax`结果；$Y(i,j)$为对应真实值的`one hot`矩阵；`p`为`batch`样本数量。

反向传播时，损失函数对输入的偏导公式为：

$$\nabla_x L = \frac{1}{p}(\hat{Y} - Y)$$

1.2.3 神经网络训练

为使计算结果与真实值尽量接近，神经网络训练通常使用梯度下降算法，不断迭代每层参数的梯度，利用梯度对每层参数进行更新。具体而言，给定训练数据与真实标签，首先进行神经网络的前向传播过程，随后利用输出结果与真实标签计算出损失函数值，然后进行神经网络的反向传播，最后利用梯度对相应的参数进行更新。更新参数W的公式为：

$$W = W - \eta \nabla_W L^T$$

其中：$\nabla_W L^T$为参数梯度的转置，η为学习率。

1.3 实验环境

环境：x86_64CPU

操作系统：Ubuntu

环境依赖：

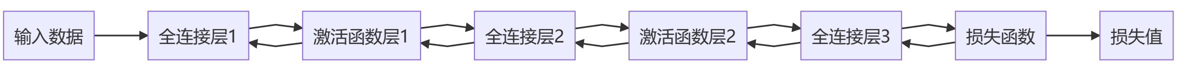
名称	版本
`python`	3.7.5
`numpy`	1.21.5

1.4 实验内容

本实验使用`python`+`numpy`，设计一个由三层全连接层组成的神经网络模型，实现手写数字分类。该网络包含两个隐含层与一个输出层，其中隐含层神经元个数作为超参数自行设置，输出层神经元个数为类别数。本实验中分别为128、64、10。

实验数据集`[MNIST]`(http://yann.lecun.com/exdb/mnist/)，由250个不同的人手写而成，总共有70000张手写数据集。其中训练集有60000张，测试集有10000张。每张图片大小为28×28。

1.5 实验步骤



第一步：通过`Dataloader`组件读入数据，格式为((batch_size, 784)，(batch_size,))，分别对应data(28×28)与label(1)

```
train_data_loader = DataLoader(mnist_npy_dir, batch_size=batch_size,
mode='train')
```

第二步：将data分别经过上述的全连接层与激活函数层的前向传播计算

```
def forward(self, x):
    x = self.fc1.forward(x)
    x = self.relu1.forward(x)
    x = self.fc2.forward(x)
    x = self.relu2.forward(x)
    x = self.fc3.forward(x)
    return x
```

第三步：采用`Crossentropy`计算`data`与`label`的损失

```
criterion = CrossEntropy()
loss = criterion.forward(output, train_labels)
```

第四步：反向传播

```
def backward(self, dloss):
    dh2 = self.fc3.backward(dloss)
    dh2 = self.relu2.backward(dh2)
    dh1 = self.fc2.backward(dh2)
    dh1 = self.relu1.backward(dh1)
    dh1 = self.fc1.backward(dh1)
```

1.5.1 主函数流程参考

```
if __name__ == '__main__':
    # 设置
    mnist_npy_dir = './mnist'   # 数据集地址
    epochs = 5 # 训练轮次
    batch_size = 32 # batch size
    lr = 0.01 # 学习率
    print_freq = 100 # 打印频率
    train_data_loader = DataLoader(mnist_npy_dir, batch_size=batch_size,
mode='train')
    val_data_loader = DataLoader(mnist_npy_dir, batch_size=batch_size,
mode='val')

    model = MlpMnistModel(input_size=784, hidden1=128, hidden2=64, out_size=10)
# 初始化模型
    criterion = CrossEntropy() # 初始化损失函数

    best_loss = 999
    for idx_epoch in range(epochs):
        # 训练
        train_data_loader.shuffle_data() # 每一个新轮次打乱一次数据
        for id_1 in range(train_data_loader.batch_nums):
            train_data, train_labels = train_data_loader.get_data(id_1) # 读取训练
数据

            output = model.forward(train_data) # 前向传播
            loss = criterion.forward(output, train_labels) # 计算损失
```

```
            dloss = criterion.backward() # 损失函数反向
            model.backward(dloss) # 反向传播
            model.step(lr) # 参数更新

            if id_1 % print_freq == 0:
                print('Train Epoch %d, iter %d, loss: %.6f' % (idx_epoch, id_1,
loss))
        # 验证
        mean_val_loss = []
        pred_results = np.zeros([val_data_loader.input_data.shape[0]]) # 保存推理
结果
        for id_2 in range(val_data_loader.batch_nums):
            val_data, val_labels = val_data_loader.get_data(id_2) # 读取验证数据
            prob = model.forward(val_data) # 前向传播（即推理）
            val_loss = criterion.forward(prob, val_labels) # 计算损失
            mean_val_loss.append(val_loss)
            pred_labels = np.argmax(prob, axis=1) # 获取推理结果
            pred_results[id_2 * val_labels.shape[0]:(id_2 + 1) *
val_labels.shape[0]] = pred_labels # 保存推理结果

            if id_2 % print_freq == 0:
                print('Val Epoch %d, iter %d, loss: %.6f' % (idx_epoch, id_2,
val_loss))

        accuracy = np.mean(pred_results == val_data_loader.input_label) # 计算准确
率
        mean_val_loss = np.array(mean_val_loss).mean() # 计算平均损失
        print('Val Epoch: %d, Loss: %f, Acc: %f' % (idx_epoch, mean_val_loss,
accuracy))

        # 保存最优模型
        if mean_val_loss <= best_loss:
            best_loss = mean_val_loss
            model.save_model(os.path.join('ckpts', 'epoch_%d_loss_%.6f.npy' %
(idx_epoch, mean_val_loss)))
```

1.5.2 MLP网络架构

```
class MlpMnistModel(object):
    def __init__(self, input_size, hidden1, hidden2, out_size):
        self.input_size = input_size
        self.hidden1 = hidden1
        self.hidden2 = hidden2
        self.out_size = out_size

        # 初始化网络中各组件
        self.fc1 = FullyConnectLayer(self.input_size, self.hidden1)
        self.fc2 = FullyConnectLayer(self.hidden1, self.hidden2)
        self.fc3 = FullyConnectLayer(self.hidden2, self.out_size)
        self.relu1 = ReluLayer()
        self.relu2 = ReluLayer()

        # 定义需要更新参数的组件列表
        self.update_layer_list = [self.fc1, self.fc2, self.fc3]

    def forward(self, x):
        # 前向传播流程
```

```
        x = self.fc1.forward(x)
        x = self.relu1.forward(x)
        x = self.fc2.forward(x)
        x = self.relu2.forward(x)
        x = self.fc3.forward(x)
        return x

    def backward(self, dloss):
        # 反向传播流程
        dh2 = self.fc3.backward(dloss)
        dh2 = self.relu2.backward(dh2)
        dh1 = self.fc2.backward(dh2)
        dh1 = self.relu1.backward(dh1)
        dh1 = self.fc1.backward(dh1)

    def step(self, lr):
        # 参数更新
        for layer in self.update_layer_list:
            layer.update_params(lr)

    def save_model(self, param_dir):
        # 保存权重和偏置
        params = {}
        params['w1'], params['b1'] = self.fc1.weight, self.fc1.bias
        params['w2'], params['b2'] = self.fc2.weight, self.fc2.bias
        params['w3'], params['b3'] = self.fc3.weight, self.fc3.bias
        np.save(param_dir, params)

    def load_model(self, params):
        # 加载权重和偏置
        self.fc1.load_params(params['w1'], params['b1'])
        self.fc2.load_params(params['w2'], params['b2'])
        self.fc3.load_params(params['w3'], params['b3'])
```

1.5.3 算子模块

1.5.3.1 全连接层

全连接层前向传播:

$$Y = XW^T + b$$

全连接层反向传播:

$$\nabla_W L = X^T \nabla_Y L$$
$$\nabla_b L = 1 * \nabla_Y L$$
$$\nabla_X L = \nabla_Y L W$$

其中，以第一层全连接层为例：

前向过程中，输入X 的`shape`为(64,784)，权重矩阵W的`shape`为(128,784)，偏置b的`shape`为(128,)经过堆叠扩充至(64,128)，输出Y的`shape`为(64,128)；

反向过程中，输入梯度$\nabla_Y L$的`shape`为(64,128)，权重梯度$\nabla_W L$的`shape`(784,128)，输出梯度$\nabla_b L$的`shape`为(128)，输出梯度$\nabla_X L$的`shape`为(64,784)。

偏置梯度$\nabla_b L$为全1的向量1与$\nabla_Y L$相乘。即$\nabla_b L(128,) = 1(1,64) * \nabla_Y L(64,128)$

注：**numpy.matmul()**函数返回两个数组的矩阵乘积。在numpy中存在广播机制，如果任一参数是1-D数组，则通过在其维度上附加1来将其提升为矩阵，在乘法后将其删除。

```
class FullyConnectLayer(object):
    def __init__(self, in_features, out_features, has_bias=True):
        # 初始化权重和偏置
        self.weight = np.random.normal(loc=0, scale=0.01, size=(out_features,
in_features))
        self.bias = np.zeros(out_features) if has_bias else None
        self.has_bias = has_bias # 是否使用偏置，默认为True

        self.inputs = None
        self.grad_weight = None
        self.grad_bias = None

    def forward(self, inputs):
        # TODO 根据公式编写全连接层的前向传播过程
        self.inputs = inputs
        return outputs

    def backward(self, in_grad):
        # TODO 根据公式编写全连接层的反向传播过程
        return out_grad

    def update_params(self, lr):
        # 全连接层的参数更新过程
        self.weight = self.weight - lr * self.grad_weight.T
        if self.has_bias:
            self.bias = self.bias - lr * self.grad_bias

    def load_params(self, weight, bias):
        # 加载权重和偏置
        assert self.weight.shape == weight.shape
        self.weight = weight
        if self.has_bias:
            assert self.bias.shape == bias.shape
            self.bias = bias
```

1.5.3.2 激活函数层

`ReLU` **层前向传播**：

$$y = max(0, x)$$

`ReLU` **层反向传播**：

$$\nabla_{x(i)}L = \begin{cases} \nabla_{y(i)}L & x(i) \geq 0 \\ 0 & x(i) < 0 \end{cases}$$

其中：$\nabla_{y(i)}L$为输入梯度，$\nabla_{x(i)}L$为输出梯度；逐元素判断梯度是否≥ 0。

```python
class ReluLayer(object):
    def __init__(self):
        self.inputs = None

    def forward(self, inputs):
        # TODO 根据公式编写激活函数ReLU的前向传播过程
        self.inputs = inputs
        return outputs

    def backward(self, in_grad):
        # TODO 根据公式编写激活函数ReLU的反向传播过程
        return out_grad
```

1.5.3.3 损失函数

交叉熵前向传播:

$$softmax = \frac{e^{X(i,j)}}{\sum_j e^{X(i,j)}}$$

$$logsoftmax = ln(softmax)$$

$$loss = -\frac{1}{p}\sum_{i,j} Y(i,j)logsoftmax$$

其中：$X(i,j)$（64,10）为上一层的输出，求得上层输出的 $softmax$ 后取对数得到 $logsoftmax$（64,10）；

$Y(i,j)$（64,10）为对应真实值的 $onehot$ 编码；p（64,）为 $batch$ 样本数量。

交叉熵反向传播:

$$\nabla_x L = \frac{1}{p}(\hat{Y} - Y)$$

其中：$\hat{Y}$（64,10）为前向传播时的 $softmax$ 输出，Y（64,10）为真实值的 $onehot$ 编码，两者 `shape` 相同；$\nabla_x L$（64,10）为输出梯度。

```python
class CrossEntropy(object):
    def __init__(self, dim=1):
        self.softmax_out = None
        self.label_onehot = None
        self.batch_size = None
        self.dim = dim

    def _softmax(self, inputs, dim=1):
        input_exp = np.exp(inputs)
        partsum = np.sum(input_exp, axis=dim)
        partsum = np.repeat(np.expand_dims(partsum, axis=dim), 
inputs.shape[dim], axis=dim)
        result = input_exp / partsum
        return result

    def forward(self, inputs, labels):
        # TODO 根据公式编写交叉熵损失函数的前向传播过程
```

```
        self.batch_size, out_size = self.softmax_out.shape
        self.label_onehot = np.eye(out_size)[labels]
        return outputs

    def backward(self, in_grad):
        # TODO 根据公式编写交叉熵损失函数的反向传播过程
        return out_grad
```

1.5.4 实验运行

代码目录：

```
EXP
|--mnist              # 数据文件夹
|    |--train_images.npy
|    |--train_labels.npy
|    |--val_images.npy
|    |--val_labels.npy
|
|--dataloader.py      # 数据加载函数
|--components.py      # 各算子模块
|--main.py            # 主函数
```

代码运行：

```
# 实现代码
vim components.py
# 运行实验
python3.7 main.py
```

1.6 评分指标

评分指标：

60'：根据提示，完成三个算子模块的前向传播与反向传播编写，通过测试用例检验。

80'：根据提示，完成整个三层神经网络的训练和推理，准确率达到92%以上。

100'：在此神经网络的基础上，设计自己的神经网络结构，并进行训练和推理，准确率达到98%以上。

1.7 实验思考

1、为提升神经网络预测精度，通常有哪些做法，这些做法的会给模型带来怎样的影响？

2、为什么全连接层权重需要进行初始化？如果不进行初始化会有怎样的结果？