华为昇腾-昇思MindSpore《AI计算系统》精品课程

HCIA-AI计算系统

精品课程

实验指导手册

版本:1.0



华为技术有限公司

|  |
| --- |
| 版权所有 © 华为技术有限公司 2021。 保留一切权利。  非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。  商标声明  C:\Users\jwx341670\Desktop\华为标志 Huawei Logo 2018\竖版标志Vertical Version\PNG\HW_POS_RBG_Vertical-150ppi.png 和其他华为商标均为华为技术有限公司的商标。  本文档提及的其他所有商标或注册商标，由各自的所有人拥有。  注意  您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。  由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。 |

|  |  |
| --- | --- |
| 华为技术有限公司 | |
| 地址： | 深圳市龙岗区坂田华为总部办公楼 邮编：518129 |
| 网址： | http://[e](http://e.huawei.com/).huawei.com |

目录

[1 基于三层感知机实现手写数字识别 4](#_Toc1019967348)

[1.1实验介绍 4](#_Toc1376747482)

[1.1.1 数据集的介绍 4](#_Toc2006117196)

[1.1.2 数据集的介绍 4](#_Toc1318455272)

[1.1.3 模型知识点的介绍 5](#_Toc1541486758)

[1.2 实验环境要求 7](#_Toc525224298)

[1.3 实验详细设计与实现 7](#_Toc1286987316)

[1.3.1 数据集上传OBS 7](#_Toc940527428)

[1.3.2 实验代码目录介绍 8](#_Toc1964840476)

[1.3.3 实验步骤 8](#_Toc1217840213)

[1.3.4 主函数流程参考 9](#_Toc573820334)

[1.3.5 MLP网络架构 10](#_Toc1996778508)

[1.3.6 算子模块 12](#_Toc1129432287)

[1.4 实验总结 16](#_Toc760491776)

[1.5 实验任务与参考解答任务 16](#_Toc1910095935)

[1.5.1 实验任务 16](#_Toc249340542)

[1.5.2 参考答案 16](#_Toc925894097)

# 基于三层感知机实现手写数字识别

## 1.1实验介绍

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。

### 数据集的介绍

### 数据集的介绍

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。

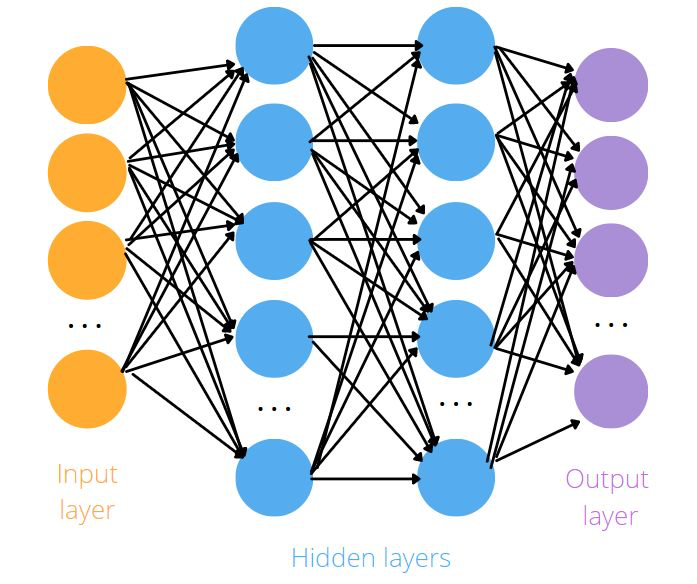
### 模型知识点的介绍

**1.1.3.1 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。



**1.1.3.2 基本算子**

**（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：

其中：输入为，输出为，权重为矩阵，偏置为。

​ 计算反向传播时，给定神经网络损失函数对当前全连接层的输出的偏导，其维度与全连接层的输出相同。根据链式法则，全连接层的权重的梯度 ，偏置的梯度，以及损失函数对输入的偏导。

​ 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

**（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：

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：

**（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：

其中：为上一层的输出，为上层输出的logsoftmax结果；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：

**1.1.3.3 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。更新参数的公式为：

其中：为参数梯度的转置，为学习率。

## 实验环境要求

环境：x86\_64CPU

操作系统：Ubuntu

## 实验详细设计与实现

### 数据集上传OBS

将花卉数据集上传OBS桶，包括flower\_photos\_train和flower\_photos\_test两个数据集。上传步骤详见1.1.3.3。

### 实验代码目录介绍

本实验的代码目录介绍：

./

|——mnist # 数据文件夹

| |——train\_images.npy

| |——train\_labels.npy

| |——val\_images.npy

| |——val\_labels.npy

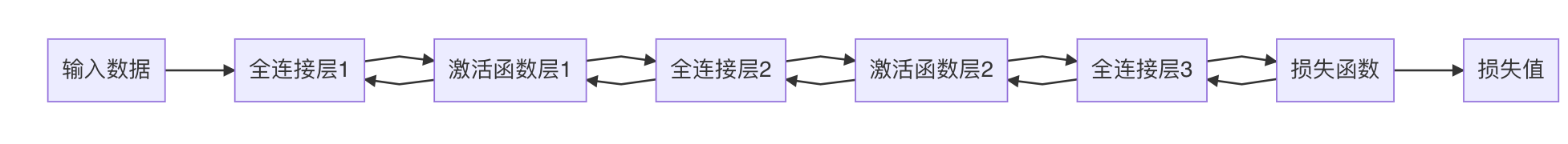
|

|——dataloader.py # 数据加载函数

|——components.py # 各算子模块

|——main.py # 主函数

### 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)

### 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)))

### MLP网络架构

代码：

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'])

### 算子模块

**1.3.6.1 全连接层**

全连接层前向传播：

全连接层反向传播：

其中，以第一层全连接层为例：

前向过程中，输入的shape为(64,784)，权重矩阵的shape为(128,784)，偏置的shape为(128,)经过堆叠扩充至(64,128)，输出的shape为(64,128)；

反向过程中，输入梯度的shape为(64,128)，权重梯度的shape (784,128)，输出梯度的shape为(128)，输出梯度的shape为(64,784)。

偏置梯度为全1的向量1与相乘。即

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。

代码：

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):

# 根据公式编写全连接层的前向传播过程

self.inputs = inputs

bias = np.stack([self.bias for \_ in range(inputs.shape[0])]) if self.has\_bias else 0

outputs = np.dot(inputs, self.weight.T) + bias

return outputs

def backward(self, in\_grad):

# 根据公式编写全连接层的反向传播过程

self.grad\_weight = np.dot(self.inputs.T, in\_grad)

self.grad\_bias = np.matmul(np.ones([in\_grad.shape[0]]), in\_grad)

out\_grad = np.dot(in\_grad, self.weight)

return out\_grad

def update\_params(self, lr):

# 根据公式编写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

**1.3.6.2 激活函数层**

ReLU层前向传播：

ReLU层反向传播：

其中：为输入梯度，为输出梯度；逐元素判断梯度是否。

代码：

class ReluLayer(object):

def \_\_init\_\_(self):

self.inputs = None

def forward(self, inputs):

# 根据公式编写激活函数ReLU的前向传播过程

self.inputs = inputs

outputs = np.maximum(self.inputs, 0)

return outputs

def backward(self, in\_grad):

# 根据公式编写激活函数ReLU的反向传播过程

b = self.inputs

b[b > 0] = 1

b[b < 0] = 0

out\_grad = np.multiply(b, in\_grad)

return out\_grad

**1.3.6.3 损失函数**

交叉熵前向传播：

其中：（64,10）为上一层的输出，求得上层输出的softmax后取对数得到softmax（64,10）；（64,10）为对应真实值的编码；（64,）为样本数量。

交叉熵后向传播：

其中：y（64,10）为前向传播时的softmax输出，（64,10）为真实值的one hot编码，两者shape相同；（64,10）为输出梯度。

代码：

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):

# 根据公式编写交叉熵损失函数的前向传播过程

self.softmax\_out = self.\_softmax(inputs, dim=self.dim)

self.batch\_size, out\_size = self.softmax\_out.shape

self.label\_onehot = np.eye(out\_size)[labels]

log\_softmax = np.log(self.softmax\_out)

outputs = -np.sum(self.label\_onehot \* log\_softmax) / labels.shape[0]

return outputs

def backward(self, in\_grad):

# 根据公式编写交叉熵损失函数的反向传播过程

out\_grad = (self.softmax\_out - self.label\_onehot) / self.batch\_size

return out\_grad

实验运行：

# 实现代码

vim components.py

# 运行实验

python3.7 main.py

## 实验总结

本实验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。希望通过公式推导能够让同学们从理论基础层面感知神经网络最重要的前向和后向过程。并不借助api自己按照公式完成网络的传播过程。

## 实验任务与参考解答任务

### 实验任务

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。完成整个三层神经网络的训练和推理，准确率达到92%以上。有能力的同学可以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，试着能不能准确率达到98%以上。

### 参考答案

补全1.3.6.1 全连接层部分；

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):

# 根据公式编写全连接层的前向传播过程

self.inputs = inputs

bias = np.stack([self.bias for \_ in range(inputs.shape[0])]) if self.has\_bias else 0

outputs = np.dot(inputs, self.weight.T) + bias

return outputs

def backward(self, in\_grad):

# 根据公式编写全连接层的反向传播过程

self.grad\_weight = np.dot(self.inputs.T, in\_grad)

self.grad\_bias = np.matmul(np.ones([in\_grad.shape[0]]), in\_grad)

out\_grad = np.dot(in\_grad, self.weight)

return out\_grad

def update\_params(self, lr):

# 根据公式编写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

补全1.3.6.2激活函数层部分：

class ReluLayer(object):

def \_\_init\_\_(self):

self.inputs = None

def forward(self, inputs):

# 根据公式编写激活函数ReLU的前向传播过程

self.inputs = inputs

outputs = np.maximum(self.inputs, 0)

return outputs

def backward(self, in\_grad):

# 根据公式编写激活函数ReLU的反向传播过程

b = self.inputs

b[b > 0] = 1

b[b < 0] = 0

out\_grad = np.multiply(b, in\_grad)

return out\_grad

补全1.3.6.3损失函数部分：

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):

# 根据公式编写交叉熵损失函数的前向传播过程

self.softmax\_out = self.\_softmax(inputs, dim=self.dim)

self.batch\_size, out\_size = self.softmax\_out.shape

self.label\_onehot = np.eye(out\_size)[labels]

log\_softmax = np.log(self.softmax\_out)

outputs = -np.sum(self.label\_onehot \* log\_softmax) / labels.shape[0]

return outputs

def backward(self, in\_grad):

# 根据公式编写交叉熵损失函数的反向传播过程

out\_grad = (self.softmax\_out - self.label\_onehot) / self.batch\_size

return out\_grad

精度达到98%以上有多种做法：例如调学习率，各层的神经元增加，增加epoch数，改进初始化方法。