研究生《深度学习》课程

实验报告

实验名称：	PyTorch 基本操作
姓　　名：	唐麒
学　　号：	21120299
上课类型：	专业课
日　　期：	2022 年 9 月 22 日

一、实验内容

本实验报告包括的实验有 PyTorch 基本操作、对逻辑回归（Logistic 回归）和 Softmax 回归分别进行从零实现和利用 torch.nn 实现。实验一和实验二主要通过对 PyTorch 的核心数据结构 Tensor 的基本操作、借助 Tensor 和 Numpy 相关的库实现逻辑回归和 Softmax 回归以达到了解和熟悉 PyTorch 的基本操作，包括初始化、减法、广播、自动求梯度等。而实验三（指利用 torch.nn 实现回归模型）在实验二（指从零实现回归模型）已经对两个机器学习模型及其损失函数、优化器等从零实现的基础上，借助 torch.nn 进一步进行简单实现，从而达到了解利用 torch.nn 搭建模型及其损失函数和优化器并训练的流程。

1.1 PyTorch 基本操作实验

实验一具体内容包括：

（1） 使用 Tensor 初始化一个 1×3 的矩阵 M 和一个 2×1 的矩阵 N，以三种不同的形式对两矩阵进行减法操作。

（2） 利用 Tensor 创建两个大小分别 3×2 和 4×2 的随机数矩阵 P 和 Q（mean=0，std=0.01），分别求矩阵 Q 的转置及其转置与矩阵 P 的乘积。

（3） 给定公式 $y_3 = y_1 + y_2 = x^2 + x^3$，且 $x = 1$。利用 Tensor 的相关知识，求 y_3 对 x 的梯度，即 $\frac{dy_3}{dx}$。要求在计算过程中，在计算 x^3 时中断梯度的追踪。

实验一涉及的内容主要为对 Tensor 的基本操作，包括 Tensor 的初始化，减法、转置、自动求梯度等操作。其中，以三种不同的方式实现两矩阵的减法，可以在实现时更好地了解不同的实现方式在使用和内存开销等方面的不同；而在计算梯度时中断对 x^3 的追踪，可以从实现方式和最终结果认识到自动求梯度的实现方式及对梯度求解的影响。

以上实验内容可以帮助学生更好地了解和熟悉 Tensor 的使用，由于 Tensor 是 PyTorch 中的核心数据结构，因而对 Tensor 基本操作的熟悉程度也对后续模型搭建及训练具有重要意义。

实验涉及的算法主要包括矩阵的减法、乘法和转置及链式求导。

1.2 Logistic 回归实验

Logistic 回归实验包括从零实现和利用 torch.nn 实现两部分。

（1） 从 0 实现 Logistic 回归（只借助 Tensor 和 Numpy 相关的库）在人工构造的数据集上进行训练和测试。

（2） 利用 torch.nn 实现 Logistic 回归在人工构造的数据集上进行训练和测试。

实验二涉及的内容主要为利用不同的方式实现 Logistic 回归，通过从零实现可以帮助学生更好得理解 Tensor 的基本操作，包括初始化、自动求梯度等，而利用 torch.nn 进行实现，可以帮助学生了解和熟悉 Pytorch 和核心类的具体使用及模型训练过程。与利用 torch.nn 进行实现相比，从零实现 Logistic 回归更有利于学生了解 Logistic 回归的模型、损失函数和优化器及其训练的设计与实现，而不是简单地调用封装完善的 API。

以上实验内容首先可以帮助学生建立起利用 PyTorch 进行模型训练的流程图，包括数据生成和加载、模型建立及参数初始化、损失函数和优化器实现和对模型在数据集上的训练和测试。其次，通过不同的方式进行 Logistic 回归的实现，可以更好地理解 Logistic 回归模型的实现和训练原理。

实验涉及的算法主要包括 Logistic 回归、二元交叉熵损失、梯度下降法。

1.3 Softmax 回归实验

Softmax 回归实验包括从零实现和利用 torch.nn 实现两部分。

（1） 从 0 实现 Softmax 回归（只借助 Tensor 和 Numpy 相关的库）在 Fashion-MNIST 数据集上进行训练和测试。

（2） 利用 torch.nn 实现 Softmax 回归在 Fashion-MNIST 数据集上进行训练和测试。

实验三涉及的内容主要为利用不同的方式实现 Softmax 回归，通过从零实现可以帮助学生更好得理解 Tensor 的基本操作，包括初始化、自动求梯度等，而利用 torch.nn 进行实现，可以帮助学生了解和熟悉 Pytorch 和核心类的具体使用及模型训练过程。与利用 torch.nn 进行实现相比，从零实现 Softmax 回归

更有利于学生了解 Softmax 回归的模型、损失函数和优化器及其训练的设计与实现，而不是简单地调用封装完善的 API。

Softmax 回归与 Logistic 回归的实现有所不同，一方面表现在模型的不同，包括 Softmax 回归为多分类，而 Logistic 回归为二分类，且损失函数不同等；另一方面表现在输入数据不同，Logistic 回归的输入为人工生成的数据集，每条数据的维度为 2，而 Softmax 回归的输入数据为 Fashion-MNIST 数据集，每一条数据都为一个大小为 28×28 的图像。

以上实验内容首先可以帮助学生建立起利用 PyTorch 进行模型训练的流程图，包括数据生成和加载、模型建立及参数初始化、损失函数和优化器实现和对模型在数据集上的训练和测试。其次，通过不同的方式进行 Softmax 回归的实现，可以更好地理解 Softmax 回归模型的实现和训练原理。

实验涉及的算法主要包括 Softmax 回归、交叉熵损失、梯度下降法。

二、实验设计

本次实验所包含的内容皆为指定内容，涉及部分数据加载、模型及其损失函数和优化器的实现，故此部分省略，在报告的后续内容中进行介绍。

三、实验环境及实验数据集

实验所用的开发环境如表 1 所示。

表 1 实验环境

开发环境	版本
操作系统	macOS Big Sur 11.5.1
开发工具	Jupyter Notebook
Python	3.8.2
PyTorch	1.8.0

本次实验所涉及的数据集包括：人工构造的数据集和 Fashion-MNIST 数据集(一个多类图像分类数据集)。

人工构造的数据集是利用 torch 随机生成的二分类数据集 x_1 和 x_2，分别对应的标签 y_1 和 y_2。实验要求中给出的示例为 50 个样本，在这里为了更

好地划分训练集和测试集，我们将样本数调整为 150，训练集与测试集按照 7:3 进行划分。构造数据可视化如图 1 所示。

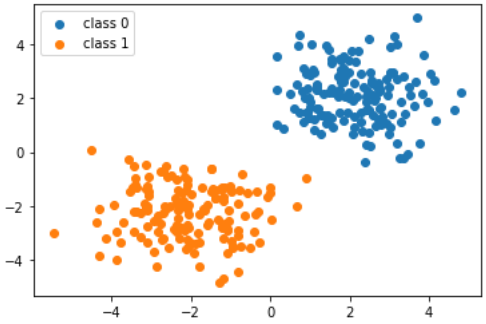

图 1 人工构造数据集可视化图

Fashion-MNIST 数据集是一个替代 MNIST 手写数字集的图像数据集，涵盖了来自 10 种类别的共 7 万个不同商品的正面图片。Fashion-MNIST 数据集的大小、格式和训练集/测试集划分与原始的 MNIST 完全一致，可以直接用它来测试模型性能，且不需要改动任何的代码。

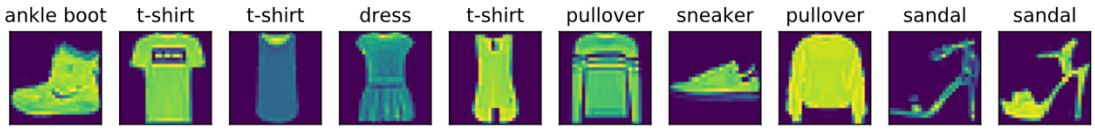

图 2 Fashion-MNIST 数据集（部分）

如图 2 所示，10 个类别分别为 dress(连衣裙)、coat(外套)、sandal(凉鞋)、shirt(衬衫)、sneaker(运动鞋)、bag(包)和 ankle boot(短靴)。

四、实验过程

本小节将对实验的具体内容进行详细说明。

4.1 PyTorch 基本操作实验

4.1.1 基本操作实验 1

使用 Tensor 初始化一个 1×3 的矩阵 M 和一个 2×1 的矩阵 N

```
M = torch.rand(1,3)
N = torch.rand(2,1)
```

两个矩阵初始化后的结果如表 2 所示。

表 2 矩阵及其值

张量	值
M	tensor([[0.0270, 0.2390, 0.7372]])
N	tensor([[0.2716], [0.2433]])

分别用三种形式对两个矩阵进行减法操作：

```
# 减法形式一
M – N

# 减法形式二
result = torch.rand(2,3)
torch.sub(M,N,out=result)

# 减法形式三：inplace
M.sub_(N)
```

其中，减法形式一和减法形式二获得的结果相同，均为：

tensor([[-0.2446, -0.0326, 0.4656],
[-0.2163, -0.0042, 0.4940]])

而减法形式三在运行过程中报错，报错信息为：

RuntimeError: output with shape [1, 3] doesn't match the broadcast shape [2, 3]

4.1.2 基本操作实验 2

（1） 利用 Tensor 创建两个大小分别 3 × 2 和 4 × 2 的随机数矩阵 P 和 Q（mean=0，std=0.01)

```
mean = 0
std = 0.01
P = torch.normal(mean,std,size=(3,2))
Q = torch.normal(mean,std,size=(4,2))
```

两个矩阵初始化后的结果如表 3 所示。

表 3 矩阵及其值

张量	值
P	tensor([[-0.0111, 0.0123], [-0.0119, -0.0064], [-0.0007, 0.0063]])
Q	tensor([[0.0114, 0.0128], [0.0167, -0.0100], [-0.0115, 0.0084], [0.0016, -0.0133]])

（2） 求矩阵 Q 的转置。

```
Qt = Q.t()
```

（3） 矩阵 Q 的转置与矩阵 P 的乘积。

```
outer_product = torch.mm(P,Qt)
```

4.1.3 基本操作实验 3

给定公式 $y_3 = y_1 + y_2 = x^2 + x^3$，且 $x = 1$。利用 Tensor 的相关知识，求 y_3 对 x 的梯度，即 $\frac{dy_3}{dx}$。要求在计算过程中，在计算 x^3 时中断梯度的追踪。

```
x = torch.tensor([1.0], requires_grad=True)

y1 = x * x
with torch.no_grad():
    y2 = x * y1

y3 = y1 + y2
y3.backward()
x.grad
```

4.2 Logistic 回归实验

分类问题中，由于输出目标 y 是一些离散的标签，而 f 值域为实数，因此无法直接进行预测，需要引入一个非线性的决策函数 g(·) 来预测输出目标。

从机器学习的角度来看，自变量就是样本的特性向量 $x \in R^d$(每一维对应一个自变量)，因变量是标签 y，这里 y 是离散值。假设空间时一组参数化的线性函数

$$g(f(x; w, b)) \tag{1}$$

其中，g(·) 可以是符号函数，权重向量 w 和偏置 b 是逻辑回归需要学习的参数。

逻辑回归总结如下：

（1） 模型

$$p(y = 1 \mid \boldsymbol{x}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \triangleq \frac{1}{1 + exp(-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})} \tag{2}$$

（2） 学习准则：交叉熵

$$l(\theta) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log \widehat{y^{(i)}} + \left(1 - y^{(i)}\right) \log\left(1 - \widehat{y^{(i)}}\right) \tag{3}$$

（3） 优化算法：梯度下降

$$\theta_j := \theta_j - \alpha \frac{1}{m}\sum_{i=1}^{m} (h_\theta(x_i) - y_i) x_i^j \tag{4}$$

使用 Tensor 和 Numpy 相关库来实现一个逻辑回归，具体的步骤有:

（1） 生成和读取数据集

在模型训练的时候，需要遍历数据集并不断读取小批量的数据样本。这里本实验定义一个函数 data_iter() 它每次返回 batch_size（批量大小）个随机样本的特征和标签。

```
# 随机生成实验所需要的二分类数据集
n_data = torch.ones(150, 2) # 数据的基本形态
x1 = torch.normal(2 * n_data, 1) # shape=(150, 2)
y1 = torch.zeros(150, 1) # 类型 0 shape=(150, 1)
x2 = torch.normal(-2 * n_data, 1) # 类型 1 shape=(150, 2)
y2 = torch.ones(150, 1) # 类型 1 shape=(150, 1)
# 注意 x, y 数据的数据形式是一定要像下面一样 (torch.cat 是在合并数据)
x = torch.cat((x1, x2), 0).type(torch.FloatTensor)
y = torch.cat((y1, y2), 0).type(torch.FloatTensor)

```

```
#读取数据
num_inputs = 2
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    random.shuffle(indices) # 样本的读取顺序是随机的
    for i in range(0, num_examples, batch_size):
        j = torch.LongTensor(indices[i: min(i + batch_size, num_examples)]) # 最后一次
        可能不足一 个batch
        yield features.index_select(0, j), labels.index_select(0, j)
```

此外，还可以通过以下方法生成训练集和测试集：

```
def shuffle_data(X, y, seed=None):
    if seed:
        np.random.seed(seed)
    idx = np.arange(X.shape[0])
    np.random.shuffle(idx)
    return X[idx], y[idx]

def train_test_split(X, y, test_size=0.2, shuffle=True, seed=None):
    if shuffle:
        X, y = shuffle_data(X, y, seed)
    n_train_samples = int(X.shape[0] * (1 - test_size))
    x_train, x_test = X[:n_train_samples], X[n_train_samples:]
    y_train, y_test = y[:n_train_samples], y[n_train_samples:]
    return x_train, x_test, y_train, y_test

X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, seed=1)
```

（2） 构建模型

```
def logistic_regression(x, w, b):
    return 1 / (1 + torch.exp(-1 * (torch.mm(x, w) + b)))
```

（3） 初始化模型参数

在构建模型之前，需要将权重和偏置初始化。本实验将权重初始化成均值为0、标准差为 0.01 的正态随机数，偏置初始化为 0。 在后面的模型训练中，需要对这些参数求梯度来迭代参数的值，因此要设置 requires_grad = True

```
w = torch.tensor(np.random.normal(0, 0.01, (num_inputs, 1)), dtype=torch.float32)
b = torch.zeros(1, dtype=torch.float32)
w.requires_grad_(requires_grad=True)
b.requires_grad_(requires_grad=True)
```

（4） 定义损失函数和优化算法

本实验使用二元交叉熵损失来定义逻辑回归的损失函数。

```
def bce_loss(y_hat, y):
    return -1 * (y * torch.log10(y_hat) + (1 - y) * torch.log10(1 - y_hat))
```

以下的 sgd 函数实现了小批量随机梯度下降算法。它通过不断迭代模型参数来优化损失函数。这里自动求梯度模块计算得来提速是一个批量样本的梯度和。将它除以批量大小来得到平均值。

```
def sgd(params, lr, batch_size):
    for param in params:
        param.data -= lr * param.grad / batch_size
```

（5） 训练模型

在训练过程中，模型将会多次迭代更新参数。在每次迭代中，根据当前读取的小批量数据样本（特征 x 和标签 y），通过调用反向函数 backward 计算小批量随机梯度，并调用优化算法 sgd 迭代模型参数。超参数如表 4 所示。

```
lr = 0.03
num_epochs = 15
batch_size = 10
net = logistic_regression
loss = bce_loss
for epoch in range(num_epochs): # 训练模型一共需要 num_epochs 个迭代周期
```

```
# 在每一个迭代周期中，会使用训练数据集中所有样本一次
    acc = 0.0
    for X, Y in data_iter(batch_size, X_train, y_train): #x 和y 分别是小批量样本的特征
和标签
        y_pred = net(X, w, b)
        l = loss(y_pred, Y).sum() #l 是有关小批量x 和y 的损失
        l.backward() # 小批量的损失对模型参数求梯度
        sgd([w, b], lr, batch_size) # 使用小批量随机梯度下降迭代模型参数
        w.grad.data.zero_() # 梯度清零
        b.grad.data.zero_()
        mask = y_pred.ge(0.5).float()
        correct = (mask == Y).sum()
        acc += correct.data.item()
    train_l = loss(net(x, w, b), y)
    print('epoch %d, loss %f, acc %f' % (epoch + 1, train_l.mean().item(), acc /
    len(y_train)))
```

表 4 从零实现逻辑回归超参数

超参数	值
Batch size	10
Learning rate	0.03
# of epoch	15

利用 torch.nn 实现逻辑回归的步骤与从零实现相同，不同的是 torch.nn 提供了更为简洁的实现方式，如数据的加载、模型及损失函数和优化器的定义等。

（1） 读取数据集

PyTorch 提供了 data 库来读取数据。

```
batch_size = 10
system = platform.system()
num_workers = 4 if system == 'Linux' else 0
dataset = Data.TensorDataset(X_train, y_train)
data_iter = Data.DataLoader(
    dataset=dataset, # torch TensorDataset format
    batch_size=batch_size, # mini batch size
    shuffle=True, # 是否打乱数据（训练集一般需要进行打乱）
    num_workers=num_workers, # 多线程来读数据，注意在 Windows 下需要设置为 0
)
```

（2） 构建模型

```
class LogisticNet(torch.nn.Module):
    def __init__(self, features):
        super(LogisticNet, self).__init__()
        self.linear = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()
    def forward(self, x):
        y = self.linear(x)
        y = self.sigmoid(y)
        return y

logistic_model = LogisticNet(2)
```

（3） 定义损失函数和优化算法

```
criterion = nn.BCELoss() #选用 BCE 损失函数,该损失函数是用于 2 分类问题的损失函数
optimizer = torch.optim.SGD(logistic_model.parameters(),lr=1e-3,momentum=0.9) #采用随机梯度下降的方法
```

（4） 训练模型

超参数如表 5 所示。

```
num_epochs = 20
for epoch in range(num_epochs): # 训练模型一共需要 num_epochs 个迭代周期
# 在每一个迭代周期中，会使用训练数据集中所有样本一次
    for X, Y in data_iter:
        if torch.cuda.is_available():
            x_data=Variable(X).cuda()
            y_data=Variable(Y).cuda()
        else:
            x_data=Variable(X)
            y_data=Variable(Y)
        out=logistic_model(x_data) #根据逻辑回归模型拟合出的y 值
        loss=criterion(out,y_data) #计算损失函数
        print_loss=loss.data.item() #得出损失函数值
        mask=out.ge(0.5).float() #以 0.5 为阈值进行分类
        correct=(mask==y_data).sum() #计算正确预测的样本个数
```

```
            acc=correct.item()/x_data.size(0) #计算精度
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        #每隔 20 轮打印一下当前的误差和精度
        if (epoch+1)%5==0:
            print('*'*10)
            print('epoch {}'.format(epoch+1)) #误差
            print('loss is {:.4f}'.format(print_loss))
            print('acc is {:.4f}'.format(acc)) #精度
```

表 5 利用 torch.nn 实现逻辑回归超参数

超参数	值
Batch size	10
Learning rate	0.001
# of epoch	20

4.3 Softmax 回归实验

Softmax 回归也称为多项或多类的 Logistic 回归, 与线性回归或逻辑回归相比的主要不同在于 Softmax 回归的输出值个数等于标签里的类别数。以具有 4 种特征和 3 种输出类别的数据集为例，Softmax 回归需要学习的参数包括 12 个权重向量 w 和 3 个偏置 b。

Softmax 回归总结如下：

（1） 模型

$$p(y=1 \mid \boldsymbol{x})=\sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \triangleq \frac{1}{1+exp(-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})} \tag{5}$$

（2） 学习准则：交叉熵

$$\mathrm{l}(\theta)=-\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log \widehat{y^{(i)}}+\left(1-y^{(i)}\right) \log\left(1-\widehat{y^{(i)}}\right) \tag{6}$$

（3） 优化算法：梯度下降

$$\theta_{\mathrm{j}} \coloneqq \theta_{\mathrm{j}}-\alpha \frac{1}{\mathrm{m}}\sum_{\mathrm{i}=1}^{\mathrm{m}}\left(\mathrm{h}_{\theta}\left(\mathrm{x}_{\mathrm{i}}\right)-\mathrm{y}_{\mathrm{i}}\right) \mathrm{x}_{\mathrm{i}}^{\mathrm{j}} \tag{7}$$

使用 Tensor 和 Numpy 相关库来实现一个 Softmax 回归，具体的步骤有:

（1） 获取数据集

通过 torchvision 来下载 Fashion-MNIST 数据集，通过参数 train 来指定获取训练数据集或测试数据集。另外还指定了参数 transform 将数据集中的图片转为 Tensor 类型。

```
mnist_train = torchvision.datasets.FashionMNIST(root='~/Datasets/FashionMNIST',
train=True, download=True, transform=transforms.ToTensor())
mnist_test = torchvision.datasets.FashionMNIST(root='~/Datasets/FashionMNIST',
train=False, download=True, transform=transforms.ToTensor())
```

```
batch_size = 256
system = platform.system()
num_workers = 4 if system == 'Linux' else 0
train_iter = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True,
num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=False,
num_workers=num_workers)
```

（2） 初始化模型参数

由于每个样本为 28×28 的图像，因而模型的输入向量长度为 784，该向量的每个元素对应图像中的每个像素。由于图像有 10 个类别，即模型的输出个数为 10，因此 Softmax 回归的权重和偏差参数分别为 784×10 和 1×10。

```
num_inputs = 784
num_outputs = 10
W = torch.normal(0, 0.1, (num_inputs, num_outputs), dtype=torch.float32) # 784*10
b = torch.normal(0, 0.01, (1, num_outputs), dtype=torch.float32) # 偏差参数 1*10

W.requires_grad_(requires_grad=True)
b.requires_grad_(requires_grad=True)
```

（3） 实现 Softmax 运算

给定一个矩阵 X，PyTorch 可以只对其中同一列(dim=0)或同一行(dim=1)的元素求和，并在结果中保留行和列这两个维度(keepdim=True)。

```
def softmax(X):
    X_exp = X.exp()
    partition = X_exp.sum(dim = 1,keepdim = True)
    return X_exp / partition
```

（4） 定义模型

```
def softmax_net(X):
    return softmax(torch.mm(X.view((-1,num_inputs)),W) + b)
```

（5） 定义损失函数

```
def cross_entropy(y_hat, y):
    return - torch.log(y_hat.gather(1,y.view(-1,1)))
```

（6） 定义优化算法

由于从零实现 Softmax 回归和从零实现逻辑回归采用的优化算法相同，故具体代码不在此进行展示。

（7） 计算分类准确率

分类准确率即正确预测数量与总预测数量之比。

```
def accuracy(y_hat,y):
    return (y_hat.argmax(dim=1)==y).sum().item()

def evaluate_accuracy(data_iter,net):
    acc_sum,n = 0.0,0
    for x,y in data_iter:
        acc_sum += (net(x).argmax(dim=1)==y).float().sum().item()
        n += y.shape[0]
    return acc_sum / n
```

（8） 训练模型

超参数如表 6 所示。

```
num_inputs = 784
num_epochs = 15
```

```
lr = 0.1
net = softmax_net
loss = cross_entropy
optimizer = sgd
for epoch in range(num_epochs):
    train_l_sum, train_acc_sum, test_acc, n = 0.0, 0.0, 0.0, 0
    for x, y in train_iter:
        y_hat = net(x)
        l = loss(y_hat, y).sum()
        l.backward() # 求梯度
        sgd([W, b], lr, batch_size) # 参数更新
        W.grad.data.zero_()
        b.grad.data.zero_() # 梯度清零
        train_l_sum += l.item()
        train_acc_sum += accuracy(y_hat, y)
        n += y.shape[0]
    test_acc += evaluate_accuracy(test_iter, net)
    print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f' % (epoch + 1, train_l_sum /
    n, train_acc_sum / n, test_acc))
```

表 6 从零实现 Softmax 回归超参数

超参数	值
Batch size	256
Learning rate	0.1
# of epoch	15

（9） 预测

```
x,y = iter(test_iter).next()
true_labels = get_fashion_mnist_labels(y.numpy())
pred_labels = get_fashion_mnist_labels(net(x).argmax(dim=1).numpy())
titles=[true+'\n'+pred for true, pred in zip(true_labels,pred_labels)]
show_fashion_mnist(x[0:10],titles[0:10])
```

利用 torch.nn 实现 Softmax 回归的步骤与从零实现相同，不同的是 torch.nn 提供了更为简洁的实现方式，如模型及损失函数和优化器的定义等，下面将对两者不同的代码实现进行展示，相同步骤不再赘述。

（1） 定义和初始化模型

```
from torch.nn import init
num_inputs = 784
num_outputs = 10

class Flatten(nn.Module):
        def __init__(self):
                super(Flatten,self).__init__()
        def forward(self,x):
                return x.view(x.shape[0],-1)

net = nn.Sequential(
      OrderedDict([
            ('flatten',Flatten()),
            ('linear',nn.Linear(num_inputs,num_outputs))
        ]
        )
)

init.normal_(net.linear.weight,mean=0,std=0.01)
init.constant_(net.linear.bias,val=0)
```

（2） 定义损失函数和优化器

```
loss=nn.CrossEntropyLoss()
optimizer=torch.optim.SGD(net.parameters(),lr=0.1)
```

（3） 训练模型

超参数如表 7 所示。

```
num_epochs = 15
lr = 0.1
for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, test_acc, n = 0.0, 0.0, 0.0, 0
        for x, y in train_iter:
                y_hat = net(x)
                l = loss(y_hat, y).sum()
```

```
            optimizer.zero_grad()
            l.backward() # 求梯度
            optimizer.step()
            train_l_sum += l.item()
            train_acc_sum += accuracy(y_hat, y)
            n += y.shape[0]
        test_acc += evaluate_accuracy(test_iter, net)
        print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f' % (epoch + 1, train_l_sum /
        n, train_acc_sum / n, test_acc))
```

表 7 从零实现 Softmax 回归超参数

超参数	值
Batch size	256
Learning rate	0.1
# of epoch	15

五、实验结果

本小节将对上述实验结果进行展示并分析。

5.1 PyTorch 基本操作实验

（1） 三种方式实现矩阵相减

减法形式一和减法形式二获得的结果相同，均为：

tensor([[-0.2446, -0.0326, 0.4656],
[-0.2163, -0.0042, 0.4940]])

而减法形式三在运行过程中报错，报错信息为：

RuntimeError: output with shape [1, 3] doesn't match the broadcast shape [2, 3]

这是由于减法形式三是原地操作（inplace），在相减时对矩阵进行广播后，计算结果和原矩阵的形状不同，故而报错。

（2） 矩阵转置和乘法

表 8 矩阵及其值

张量	值
Q	tensor([[0.0114, 0.0128], [0.0167, -0.0100], [-0.0115, 0.0084],

	[0.0016, -0.0133]])
Q 的转置	tensor([[0.0114, 0.0167, -0.0115, 0.0016], [0.0128, -0.0100, 0.0084, -0.0133]])

The shape of Q is torch.Size([4, 2])

The shape of Q after transposing is torch.Size([2, 4])

矩阵 Q 的转置与矩阵 P 的乘积

tensor([[3.0825e-05, -3.0886e-04, 2.3096e-04, -1.8182e-04],

[-2.1912e-04, -1.3524e-04, 8.3214e-05, 6.6380e-05],

[7.2738e-05, -7.4042e-05, 6.0398e-05, -8.4600e-05]])

（3） 自动求梯度

x.grad = tensor([2.])

$\frac{dy_3}{dx}$ 是 2 而不是 5：由于 y_2 被 torch.no_grad() 包裹，所以与 y_2 有关的梯度不会回传，只有与 y_1 有关的梯度才会回传。

5.2 Logistic 回归实验

从零实现 Logistic 回归和利用 torch.nn 实现 Logistic 回归的可视化结果分别如图 3 (a) 和 (b) 所示。

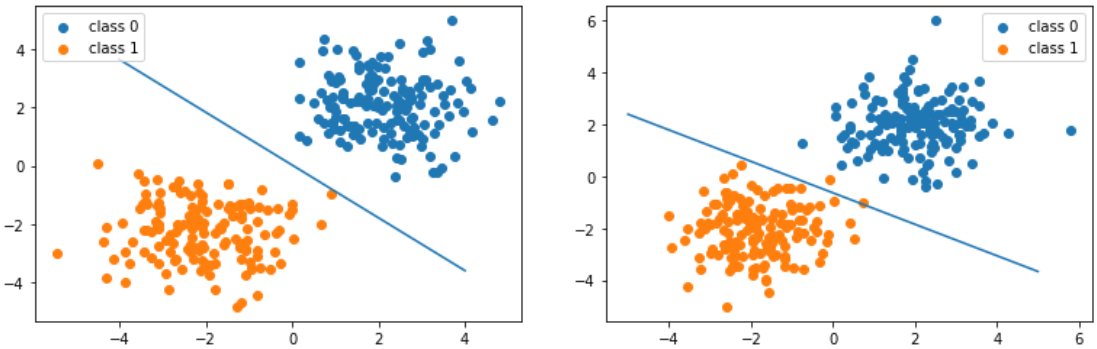

图 3 Logistic 回归实验结果可视化

无论是从零开始实现的 Logistic 回归还是利用 torch.nn 实现的 Logistic 回归，在训练过程中可以看到模型可以较快收敛，训练集上的准确率可以很快到达 1（基本为 1 个 epoch 后），而在测试集上的准确率同样可以到达 1（多次训练后，不同的模型参数在测试集上均能达到 0.97 及以上）。出现该结果的原因一方面是 Logistic 回归可以很好地拟合数据特征从而实现准确的分类，而另一方

面也同数据集规模有关，若数据集过大，Logistic 回归模型可能不能很好地拟合数据特征，从而欠拟合；反之亦然。此外，相对于多分类任务而言，二分类任务更加简单，因而可以取得很好的性能。

5.3 Softmax 回归实验

从零实现 Softmax 回归和利用 torch.nn 实现 Softmax 回归的可视化结果分别如图 4 (a) 和 (b) 所示。

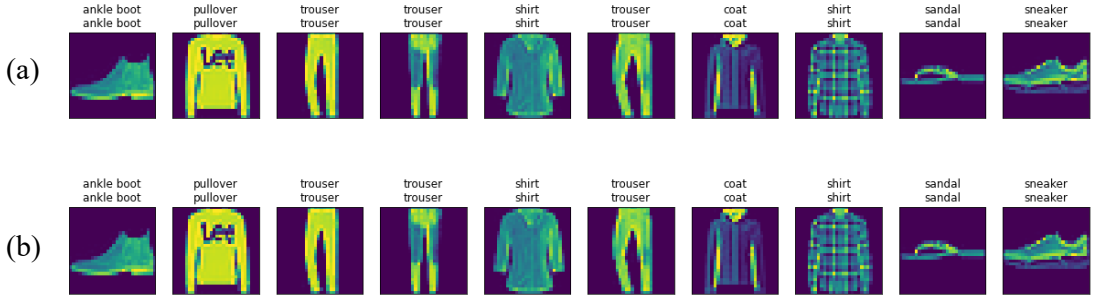

图 4 Softmax 回归实验结果可视化

如表 9 所示，无论是从零开始实现的 Softmax 回归还是利用 torch.nn 实现的 Softmax 回归，在训练过程中可以看到损失函数不断下降，训练集上的准确率逐渐提升，测试集上的准确率也在相应地提升。可以发现，训练集上的准确率与损失成负相关，基本呈现出损失下降而训练集上准确率上升的关系。而测试集上的准确率不一定与训练集上的准确率（或损失）有线性相关关系，即训练集上的准确率上升而测试集上的准确率可能下降，尤其表现在训练后期。这是由于多分类的任务更难，不再是非黑即白的问题，对模型的学习能力或特征表示能力要求更高。此外，随着数据集规模增大，模型有可能过拟合，即学到了训练集中的噪音，从而造成了模型在训练集和测试集上性能之间的差距。

表 9 Softmax 回归训练指标

	从零实现			利用 torch.nn 实现		
	Loss	Train Acc	Test Acc	Loss	Train Acc	Test Acc
1	0.8205	0.722	0.781	0.0031	0.749	0.789
2	0.5813	0.805	0.798	0.0022	0.813	0.809
3	0.5351	0.820	0.814	0.0021	0.825	0.814

4	0.5097	0.828	0.818	0.0020	0.832	0.808
5	0.4927	0.834	0.824	0.0019	0.836	0.822
6	0.4793	0.838	0.824	0.0019	0.841	0.828
7	0.4703	0.841	0.826	0.0018	0.843	0.824
8	0.4632	0.842	0.828	0.0018	0.845	0.829
9	0.4571	0.844	0.830	0.0018	0.847	0.830
10	0.4523	0.846	0.828	0.0018	0.848	0.826
11	0.4473	0.848	0.831	10.0017	0.850	0.830
12	0.4439	0.847	0.833	0.0017	0.851	0.828
13	0.4399	0.849	0.834	0.0017	0.851	0.837
14	0.4363	0.851	0.836	0.0017	0.852	0.838
15	0.4355	0.851	0.836	0.0017	0.854	0.836

六、实验心得体会

通过本次实验，对 PyTorch 的基本操作有了更扎实的理解和掌握，尤其是实验中对自动求梯度的设计，解决了我在之前自学过程中的迷惑，从而在今后的深度学习编程中可以更好地利用 Tensor 数据结构及其基本操作。

此外，通过从零开始编写和利用 torch.nn 两种方式分别实现了逻辑回归和 Softmax 回归，在很大程度上加深了我对这两个模型的理解，真正做到了知其然并知其所以然。

而在实验报告撰写的过程中，可以对实验的目的、问题和解决方案进行明确的梳理，进一步在编码实验的基础上获得了提升，做到了理论和实践的齐头并进。

七、参考文献

[1] 北京交通大学 《深度学习》课程组，实验 1 PyTorch 基本操作实验，北京交通大学《深度学习》课件