

研究生《深度学习》课程

实验报告

|  |  |
| --- | --- |
| **实验名称：** | **PyTorch基本操作实验** |
| **姓 名：** | **束天成** |
| **学 号：** | **22121501** |
| **日 期：** | **2022/7/22** |

## 一、实验内容

本次实验为PyTorch的基本操作实验，本次实验主要包含了基本操作、Logistic 回归实验和Softmax 回归实验，编程的语言为Python，其目的是了解PyTorch的基本操作和编程语言，为后续的研究打下基础。

#### 1.1 PyTorch基本操作实验

PyTorch基本操作实验包含了矩阵的构建、矩阵的运算和梯度计算三个部分，PyTorch的所有运算都建立在矩阵数据运算的基础上，因此构建矩阵、矩阵运算是PyTorch编程中非常重要的环节，梯度计算是对数据进行进一步处理中常见的运算方式，掌握梯度计算对数据处理有非常重要的意义。

#### 1.2 Logistic 回归实验

Logistic回归在广义上是线性回归的一种，是一种用于解决二分类问题的机器学习方法，用于估计某种事物的可能性。本次实验主要需要使用人工构造的数据集并使用两种方法实现Logistic的回归，并从loss以及训练集上的准确率等多个角度对结果进行分析，这对后期数据处理有着较为重要的意义。

#### 1.3 Softmax 回归实验

Softmax回归其实是 logistic 回归的一般形式，logistic 回归用于二分类，而 softmax回归用于多分类，其在数据处理中的适用性和通用性更广。本次实验主要需要使用人工构造的数据集并使用两种方法实现Softmax的回归，并从loss以及训练集上的准确率等多个角度对结果进行分析。由于Softmax用于多分类问题，因此对于复杂问题的通用性更强。

## 二、实验设计

本次实验主要内容都为指定内容，均指定了实验需要的数据矩阵维度，矩阵构造方式，运算函数，使用的方法以及需要分析的内容，不包含过多需要自主设计的模型和方法等，因此不再赘述。

## 三、实验环境及实验数据集

Torch版本：1.5.1

操作系统：Windows10

硬件：

CPU：i5 9300h GPU：GTX1650

数据集：torch.nn，Fashion-MNIST数据集，nn.BCEWithLogitsLoss函数等

## 四、实验过程

实验过程包括在编写代码时一些需要注意的事项，可附代码片段进行说明；还应该包括在训练模型时进行的一些步骤、参数设置等内容。

注：为了代码片段尽量的美观、统一，建议附代码片段时只附加关键的片段，不要全部粘贴，并尽量使用下面提供的网站进行代码高亮等格式转换后再粘贴。

<http://www.planetb.ca/syntax-highlight-word>

#### 4.1 PyTorch基本操作实验

**4.1.1**

使用Tensor初始化一个1×3的矩阵M和一个2×1 的矩阵N，对两矩阵进行减法操作（要求 实现三种不同的形式），给出结果并分析三种方式的不同（如果出现报错，分析报错的原因）， 同时需要指出在计算过程中发生了什么。

1. import torch
2. x = torch.rand(1,3)*#定义矩阵1*
3. y = torch.rand(2,1)*#定义矩阵2*
4. print(x)
5. print(y)
6. print(x-y)*#直接输出相减结果*
7. print(torch.sub(x, y))*#第二种减法形式*
8. y.sub\_(x)*#第三种减法形式*
9. print(y)

**4.1.2**

1.利用 Tensor创建两个大小分别3 ×2和4 × 2 的随机数矩阵，要求服从均值为0，标准差0.01为的正态分布；2.对第二步得到的矩阵进行形状变换得到转置矩阵；3.对上述得到的矩阵和矩阵求矩阵相乘

1. import torch
2. P=torch.normal(0,0.01,size=(3,2))*#创建第一个矩阵*
3. Q=torch.normal(0,0.01,size=(4,2))*#创建第二个矩阵*
4. print(P)
5. print(Q)
6. QT=Q.t()*#矩阵转置*
7. print(QT)
8. S=P.mm(QT)*#对新矩阵进行乘法运算*
9. print(S)

**4.1.3**

给定公式，利用学习所得到的Tensor的相关知识，进行梯度计算。要求计算时中断y2的梯度：

1. import torch
2. x = torch.tensor(1.0, requires\_grad=True)
3. y1 = x \* x*#计算第一个函数*
4. y2 = x \* x \* x*#计算第二个函数*
5. with torch.no\_grad():
6. y2 = x \* x \* x*#中断梯度追踪*
7. print(y3.requires\_grad)
8. print(y2.requires\_grad)
9. print(y1.requires\_grad)
10. y3 = y1 + y2*#计算y3*
11. print(y3)*#输出y3*
12. y3.backward()
13. print(x.grad)*#输出y3梯度*

#### 4.2 Logistic 回归实验

**4.2.1**

要求动手从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，并从loss以及训练集上的准确率等多个角度对结果进行分析。

1. import torch
2. import matplotlib.pyplot as plt
3. import numpy as np
4. import random#导入需要的模板
5. num\_inputs = 2#定义输入个数
6. n\_data = torch.ones(20, num\_inputs)
7. x1 = torch.normal(2 \* n\_data, 1)
8. y1 = torch.ones(20)
9. x2 = torch.normal(-2 \* n\_data, 1)
10. y2 = torch.zeros(20)
11. aa = torch.cat((x1, x2), 0).type(torch.FloatTensor)#人工构造数据集
12. bb = torch.cat((y1, y2), 0).type(torch.FloatTensor)#人工构造数据集
13. def data\_iter(batch\_size, features, labels):
14. num\_examples = len(features)
15. indices = list(range(num\_examples))
16. random.shuffle(indices)#对数据采取随机读取的方法
17. for i in range(0, num\_examples, batch\_size):
18. j = torch.LongTensor(indices[i: min(i + batch\_size, num\_examples)])
19. yield features.index\_select(0, j), labels.index\_select(0, j)
20. w = torch.tensor(np.random.normal(0, 0.01, (num\_inputs, 1)), dtype=torch.float32)#将w进行初始化
21. b = torch.zeros(1, dtype=torch.float32)#将b进行初始化
22. w.requires\_grad\_(requires\_grad=True)
23. b.requires\_grad\_(requires\_grad=True)
24. def logits(X, w, b):
25. y = torch.mm(X, w) + b
26. return  1/(1+torch.pow(np.e,-y))
27. def logits\_loss(y\_hat, y): #手动实现二元交叉熵损失函数
28. y = y.view(y\_hat.size())
29. return  -y.mul(torch.log(y\_hat))-(1-y).mul(torch.log(1-y\_hat))
30. def sgd(params, lr, batch\_size):
31. for param in params:
32. param.data -= lr \* param.grad / batch\_size
33. def evaluate\_accuracy():
34. acc\_sum,n = 0.0,0
35. for X,y in data\_iter(batch\_size, aa, bb):
36. y\_hat = net(X, w, b)
37. y\_hat = torch.squeeze(torch.where(y\_hat>0.5,torch.tensor(1.0),torch.tensor(0.0)))
38. acc\_sum += (y\_hat==y).float().sum().item()
39. n+=y.shape[0]
40. return acc\_sum/n
41. lr = 0.0005
42. num\_epochs = 10
43. net = logits
44. loss = logits\_loss
45. batch\_size = 5
46. epochlist = np.arange(1,num\_epochs+1)
47. losslist = []
48. for epoch in range(num\_epochs): # 训练模型的迭代周期
49. train\_l\_num, train\_acc\_num,n = 0.0,0.0,0
50. for X, y in data\_iter(batch\_size, aa, bb): # x和y分别是小批量样本的特征和标签
51. y\_hat = net(X, w, b)
52. l = loss(y\_hat, y).sum()
53. l.backward() # 小批量的损失对模型参数求梯度
54. sgd([w, b], lr, batch\_size) # 使用小批量随机梯度下降迭代模型参数
55. w.grad.data.zero\_() # 梯度清零
56. b.grad.data.zero\_() # 梯度清零
57. *#计算每个epoch的loss*
58. train\_l\_num += l.item()
59. *#计算训练样本的准确率*
60. y\_hat = torch.squeeze(torch.where(y\_hat>0.5,torch.tensor(1.0),torch.tensor(0.0)))
61. train\_acc\_num += (y\_hat==y).sum().item()
62. *#每一个epoch的所有样本数*
63. n+= y.shape[0]
64. test\_acc = evaluate\_accuracy()
65. print('epoch %d, loss %.4f,train\_acc %f,test\_acc %f'%(epoch+1,train\_l\_num/n, train\_acc\_num/n, test\_acc))
66. print(aa)
67. print(bb)
68. print( '\n', w)
69. print( '\n', b)

**4.2.2**

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，并对结果进行分析， 并从loss以及训练集上的准确率等多个角度对结果进行分析。

1. import torch
2. import matplotlib.pyplot as plt
3. import numpy as np
4. import torch.nn as nn
5. import torch.utils.data as Data
6. import random #导入需要的模板
7. num\_inputs = 2 #定义输入个数
8. n\_data = torch.ones(20, num\_inputs)
9. x1 = torch.normal(2 \* n\_data, 1)
10. y1 = torch.ones(20)
11. x2 = torch.normal(-2 \* n\_data, 1)
12. y2 = torch.zeros(20)
13. aa = torch.cat((x1, x2), 0).type(torch.FloatTensor)#人工构造数据集
14. bb = torch.cat((y1, y2), 0).type(torch.FloatTensor)#人工构造数据集
15. lr = 0.001
16. batch\_size = 10
17. dataset = Data.TensorDataset(aa, bb)
18. data\_iter = Data.DataLoader(
19. dataset=dataset, # torch TensorDataset format
20. batch\_size=batch\_size, # mini batch size
21. shuffle=True, # 是否打乱数据 (训练集一般需要进行打乱)
22. num\_workers=0, # 多线程来读数据， 注意在Windows下需要设置为0
23. )
24. test\_data\_iter = Data.DataLoader(
25. dataset=dataset, # torch TensorDataset format
26. batch\_size=batch\_size, # mini batch size
27. shuffle=True, # 是否打乱数据 (训练集一般需要进行打乱)
28. num\_workers=0, # 多线程来读数据， 注意在Windows下需要设置为0
29. )
30. class LogisticRegression(nn.Module):
31. def \_\_init\_\_(self,n\_features):
32. super(LogisticRegression, self).\_\_init\_\_()
33. self.lr = nn.Linear(n\_features, 1)
34. self.sm = nn.Sigmoid()
35. def forward(self, x):
36. x = self.lr(x)
37. x = self.sm(x)
38. return x
39. logistic\_model = LogisticRegression(num\_inputs)
40. criterion = nn.BCELoss()
41. optimizer = torch.optim.SGD(logistic\_model.parameters(), lr=1e-3)
42. from torch.nn import init
43. init.normal\_(logistic\_model.lr.weight, mean=0, std=0.01)
44. init.constant\_(logistic\_model.lr.bias, val=0) #也可以直接修改bias的data： net[0].bias.data.fill\_(0)
45. print(logistic\_model.lr.weight)
46. print(logistic\_model.lr.bias)
47. def evaluate\_accuracy():
48. acc\_sum,n = 0.0,0
49. for X,y in test\_data\_iter:
50. y\_hat = logistic\_model(X)
51. y\_hat = torch.squeeze(torch.where(y\_hat>0.5,torch.tensor(1.0),torch.tensor(0.0)))
52. acc\_sum += (y\_hat==y).float().sum().item()
53. n+=y.shape[0]
54. return acc\_sum/n
55. num\_epochs = 10
56. for epoch in range( num\_epochs):
57. train\_l\_num, train\_acc\_num,n = 0.0,0.0,0
58. for X, y in data\_iter:
59. y\_hat = logistic\_model(X)
60. l = criterion(y\_hat, y.view(-1, 1))
61. optimizer.zero\_grad() # 梯度清零，等价于logistic\_model.zero\_grad()
62. l.backward()
63. optimizer.step()
64. train\_l\_num += l.item()
65. y\_hat = torch.squeeze(torch.where(y\_hat>0.5,torch.tensor(1.0),torch.tensor(0.0)))
66. train\_acc\_num += (y\_hat==y).sum().item()
67. n+= y.shape[0]
68. test\_acc = evaluate\_accuracy()
69. print('epoch %d, loss %.4f,train\_acc %f,test\_acc %f'%(epoch+1,train\_l\_num/n, train\_acc\_num/n, test\_acc))

#### 4.3 Softmax 回归实验

**4.3.1**

要求动手从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，并从loss、训练集以及测试集上的准确率等多个角度对结果进行分析

1. import torch
2. import numpy as np
3. import random
4. from IPython import display
5. from matplotlib import pyplot as plt
6. import torchvision
7. import torchvision.transforms as transforms
8. mnist\_train = torchvision.datasets.FashionMNIST(root='./Datasets/FashionMNIST', train=True,
9. download=True, transform=transforms.ToTensor())
10. mnist\_test = torchvision.datasets.FashionMNIST(root='./Datasets/FashionMNIST', train=False,
11. download=True, transform=transforms.ToTensor())
12. batch\_size = 256
13. train\_iter = torch.utils.data.DataLoader(mnist\_train, batch\_size=batch\_size, shuffle=True,
14. num\_workers=0)
15. test\_iter = torch.utils.data.DataLoader(mnist\_test, batch\_size=batch\_size, shuffle=False,
16. num\_workers=0)
17. print(test\_iter)
18. num\_inputs = 784
19. num\_outputs = 10
21. W = torch.tensor(np.random.normal(0,0.01,(num\_inputs,num\_outputs)),dtype=torch.float)
22. b = torch.zeros(num\_outputs,dtype = torch.float)
24. W.requires\_grad\_(requires\_grad = True)
25. b.requires\_grad\_(requires\_grad = True)
26. def softmax(X):
27. X\_exp = X.exp()
28. partition = X\_exp.sum(dim = 1, keepdim=True)
29. return X\_exp / partition
30. def net(X):
31. *#torch.mm  矩阵相乘  view（）改变矩阵维度为1行 num\_input列*
32. f\_x = torch.mm(X.view((-1,num\_inputs)),W) + b
33. return softmax(f\_x)
34. def cross\_entropy(y\_hat, y):
35. return -torch.log(y\_hat.gather(1, y.view(-1,1)))
36. def sgd(params, lr, batch\_size):
37. for param in params:
38. param.data -= lr \* param.grad / batch\_size *# 注意这里更改param时用的param.data*
39. def evaluate\_accuracy(data\_iter,net):
40. acc\_sum,n = 0.0,0
41. for X,y in data\_iter:
42. *#print(len(X)) 小批量数据集 每个X中有 256个图像*
43. *#print((net(X).argmax(dim=1)==y).float().sum().item())*
44. acc\_sum += (net(X).argmax(dim=1)==y).float().sum().item()
45. n+=y.shape[0]
46. return acc\_sum/n
47. def train(net, train\_iter,test\_iter,loss,num\_epochs,batch\_size,params=None,lr=None,optimizer = None):
48. for epoch in range(num\_epochs):
49. *#模型训练次数 num\_epochs次*
50. train\_l\_num, train\_acc\_num,n = 0.0,0.0,0
51. for X,y in train\_iter:
52. *#X 为小批量256个图像 1\*28\*28 y为标签*
53. *# 计算X softmax下的值   与损失函数值*
54. y\_hat = net(X)
55. l = loss(y\_hat,y).sum()
56. l.backward()
57. sgd([W, b], lr, batch\_size) *# 使用小批量随机梯度下降迭代模型参数*
58. W.grad.data.zero\_() *# 梯度清零*
59. b.grad.data.zero\_()
60. *#计算每个epoch的loss*
61. train\_l\_num += l.item()
62. *#计算训练样本的准确率*
63. train\_acc\_num += (y\_hat.argmax(dim=1)==y).sum().item()
64. *#每一个epoch的所有样本数*
65. n+= y.shape[0]
66. *#计算测试样本的准确率*
67. test\_acc = evaluate\_accuracy(test\_iter,net)
68. print('epoch %d, loss %.4f,train\_acc %.3f,test\_acc %.3f'%(epoch+1,train\_l\_num/n, train\_acc\_num/n, test\_acc))
69. num\_epochs ,lr = 30,0.1
70. train(net, train\_iter, test\_iter, cross\_entropy, num\_epochs,batch\_size, [W, b], lr)

**4.3.2**

利用torch.nn实现 softmax 回归在Fashion-MNIST数据集上进行训练和测试，并从loss，训 练集以及测试集上的准确率等多个角度对结果进行分析

1. import torch
2. import numpy as np
3. import random
4. from torch import nn
5. from IPython import display
6. from matplotlib import pyplot as plt
7. import torchvision
8. import torchvision.transforms as transforms
9. from torch.nn import init
10. mnist\_train = torchvision.datasets.FashionMNIST(root='./Datasets/FashionMNIST', train=True,
11. download=True, transform=transforms.ToTensor())
12. mnist\_test = torchvision.datasets.FashionMNIST(root='./Datasets/FashionMNIST', train=False,
13. download=True, transform=transforms.ToTensor())
14. *#读取数据*
15. batch\_size = 256
16. train\_iter = torch.utils.data.DataLoader(mnist\_train, batch\_size=batch\_size, shuffle=True,
17. num\_workers=0)
18. test\_iter = torch.utils.data.DataLoader(mnist\_test, batch\_size=batch\_size, shuffle=False,
19. num\_workers=0)
20. print(test\_iter)
21. *#定义输入和输出*
22. num\_inputs = 784
23. num\_outputs = 10
24. *#定义网络模型*
25. class LinearNet(nn.Module):
26. def \_\_init\_\_(self, num\_inputs, num\_outputs):
27. super(LinearNet, self).\_\_init\_\_()
28. self.linear = nn.Linear(num\_inputs, num\_outputs)
29. def forward(self, x): *# x shape: (batch, 1, 28, 28)*
30. y = self.linear(x.view(x.shape[0], -1))
31. return y
32. net = LinearNet(num\_inputs, num\_outputs)
33. *# 初始化参数w和b*
34. init.normal\_(net.linear.weight, mean=0, std=0.01)
35. init.constant\_(net.linear.bias, val=0)
36. *#nn模块实现交叉熵损失函数--包含了softmax函数*
37. cross\_entropy = nn.CrossEntropyLoss()
38. *#nn模块实现交叉熵损失函数--包含了softmax函数*
39. cross\_entropy = nn.CrossEntropyLoss()
40. optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
41. *#测试集准确率*
42. def evaluate\_accuracy(data\_iter,net):
43. acc\_sum,n = 0.0,0
44. for X,y in data\_iter:
45. *#print(len(X)) 小批量数据集 每个X中有 256个图像*
46. *#print((net(X).argmax(dim=1)==y).float().sum().item())*
47. acc\_sum += (net(X).argmax(dim=1)==y).float().sum().item()
48. n+=y.shape[0]
49. return acc\_sum/n
50. def train(net, train\_iter, test\_iter, loss, num\_epochs, batch\_size,params=None, lr=None, optimizer=None):
51. for epoch in range(num\_epochs):
52. train\_l\_sum, train\_acc\_sum, n = 0.0, 0.0, 0
53. for X, y in train\_iter:
54. y\_hat = net(X)
55. l = loss(y\_hat, y).sum()
56. optimizer.zero\_grad() *# 梯度清零*
57. l.backward() *# 计算梯度*
58. optimizer.step()  *# 随机梯度下降算法, 更新参数*
59. train\_l\_sum += l.item()
60. *#训练集准确率*
61. train\_acc\_sum += (y\_hat.argmax(dim=1) == y).sum().item()
62. n += y.shape[0]
63. test\_acc = evaluate\_accuracy(test\_iter, net)
64. print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'% (epoch + 1, train\_l\_sum / n, train\_acc\_sum / n, test\_acc))
65. num\_epochs = 30
66. train(net, train\_iter, test\_iter, cross\_entropy, num\_epochs,batch\_size,None,None, optimizer)

## 五、实验结果

实验结果包括程序运行结果以及对结果的分析，尽量用图表展示实验结果，并且通过结果进行相关的分析。

#### 5.1 PyTorch基本操作实验

**5.1.1 实验4.1.1运行结果**

1. tensor([[0.0305, 0.5506, 0.8343]])
2. tensor([[0.4280],
3. [0.4952]])
4. tensor([[-0.3975,  0.1227,  0.4063],
5. [-0.4647,  0.0554,  0.3391]])
6. tensor([[-0.3975,  0.1227,  0.4063],
7. [-0.4647,  0.0554,  0.3391]])
8. ---------------------------------------------------------------------------
9. RuntimeError                              Traceback (most recent call last)
10. <ipython-input-3-a27a74325f95> in <module>
11. 7 print(x-y)
12. 8 print(torch.sub(x, y))
13. ----> 9 y.sub\_(x)
14. 10 print(y)
15. RuntimeError: output with shape [2, 1] doesn't match the broadcast shape [2, 3]

**分析：**构建了矩阵M和矩阵N以后，使用print函数能正常输出预期结果。使用第一种方法和第二种方法进行矩阵相减运算时触发了Tensor运算的广播机制，因此前两种方法都可以正常输出运算结果。但是第三种运算并没有正常触发广播机制，因此进行运算时出现了维度不一致的报错。改进方法是定义一个新的空矩阵进行两个已有矩阵的运算，即可正常输出结果，结果如下：

1. tensor([[0.9562, 0.6112, 0.9202]])
2. tensor([[0.1051],
3. [0.0494]])
4. tensor([[0.8511, 0.5061, 0.8151],
5. [0.9068, 0.5618, 0.8708]])
6. tensor([[0.8511, 0.5061, 0.8151],
7. [0.9068, 0.5618, 0.8708]])
8. tensor([[0.8511, 0.5061, 0.8151],
9. [0.9068, 0.5618, 0.8708]])

**5.1.2 实验4.1.2运行结果**

1. tensor([[-0.0007,  0.0038],
2. [ 0.0103,  0.0083],
3. [ 0.0002,  0.0034]])
4. tensor([[-0.0002,  0.0127],
5. [ 0.0196, -0.0032],
6. [-0.0074,  0.0032],
7. [ 0.0066,  0.0158]])
8. tensor([[-0.0002,  0.0196, -0.0074,  0.0066],
9. [ 0.0127, -0.0032,  0.0032,  0.0158]])
10. tensor([[ 4.8016e-05, -2.5900e-05,  1.7237e-05,  5.4497e-05],
11. [ 1.0270e-04,  1.7536e-04, -4.9595e-05,  1.9827e-04],
12. [ 4.3418e-05, -7.5094e-06,  9.6251e-06,  5.4912e-05]])

**分析：**构建了矩阵P和矩阵Q以后，使用print函数能正常输出预期结果。在进行过转置运算后矩阵均可以正常实现矩阵转置，由于转制后的矩阵进行运算矩阵运算的条件，因此在运算时不会报错，可以正常输出结果。

**5.1.3 实验4.1.3运行结果**

1. True
2. False
3. True
4. tensor(2., grad\_fn=<AddBackward0>)
5. tensor(2.)

**分析：**进行运算后，由于函数y1和y2都能正常输出值，区别在于y1正常进行梯度追踪，y2中断了梯度追踪，y3是y1和y2之和，因此y1和y3应该都具有梯度属性，y2不具备梯度属性，y3输出应该只具有y1的梯度，根据实验结果可得，输出的数据和预期结果相符合。

#### 5.2 Logistic 回归实验

**5.2.1 实验4.2.1运行结果**

1. epoch 1, loss 0.7132,train\_acc 0.000000,test\_acc 0.000000
2. epoch 2, loss 0.7049,train\_acc 0.000000,test\_acc 0.000000
3. epoch 3, loss 0.6967,train\_acc 0.125000,test\_acc 0.900000
4. epoch 4, loss 0.6887,train\_acc 0.975000,test\_acc 1.000000
5. epoch 5, loss 0.6808,train\_acc 1.000000,test\_acc 1.000000
6. epoch 6, loss 0.6730,train\_acc 1.000000,test\_acc 1.000000
7. epoch 7, loss 0.6654,train\_acc 1.000000,test\_acc 1.000000
8. epoch 8, loss 0.6580,train\_acc 1.000000,test\_acc 1.000000
9. epoch 9, loss 0.6506,train\_acc 1.000000,test\_acc 1.000000
10. epoch 10, loss 0.6434,train\_acc 1.000000,test\_acc 1.000000
11. tensor([[ 1.8809,  1.1965],
12. [ 3.0081,  2.4303],
13. [ 2.4967,  0.9777],
14. [ 1.0196,  3.9064],
15. [ 1.7284,  1.3719],
16. [ 2.1854,  1.9097],
17. [ 1.7848,  0.9357],
18. [ 0.9906, -0.0168],
19. [ 2.2404,  0.9121],
20. [ 1.5281,  0.7345],
21. [ 1.9127,  1.6078],
22. [ 2.5034,  4.0133],
23. [ 1.5753,  0.8903],
24. [ 2.9513,  3.0152],
25. [ 3.3529,  1.4548],
26. [ 2.0861,  1.7741],
27. [ 3.6954,  1.8050],
28. [ 3.9431,  1.8753],
29. [ 2.6096,  3.4752],
30. [ 1.8023,  2.9966],
31. [-0.9877, -0.8622],
32. [-0.3483, -3.8689],
33. [-2.3075, -3.8316],
34. [-0.8467, -3.1742],
35. [-0.5870, -1.9295],
36. [-2.4078, -1.6827],
37. [-1.3419, -1.2953],
38. [-3.7345, -3.8704],
39. [-1.2352, -2.1746],
40. [-1.7332, -2.8962],
41. [-2.8115, -2.7985],
42. [-0.9951, -0.9064],
43. [-2.2489, -0.4578],
44. [-2.7144, -1.8151],
45. [-0.4393, -4.5418],
46. [-1.4527, -2.3678],
47. [-1.6124, -3.1521],
48. [-2.7330, -1.5429],
49. [-1.4250, -1.6272],
50. [-1.2331,  0.1828]])
51. tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
52. 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
53. 0., 0., 0., 0.])
54. tensor([[0.0281],
55. [0.0273]], requires\_grad=True)
56. tensor([-1.4793e-05], requires\_grad=True)

**分析：**根据结果分析可得，随着迭代次数的增加，函数的损失值不断减少，训练精度逐渐上升，模型的准确性随着迭代次数的增加不断提高，和预期结果相符合。

**5.2.2 实验4.2.2运行结果**

1. Parameter containing:
2. tensor([[-0.0213, -0.0021]], requires\_grad=True)
3. Parameter containing:
4. tensor([0.], requires\_grad=True)
5. epoch 1, loss 0.0715,train\_acc 0.000000,test\_acc 0.000000
6. epoch 1, loss 0.0713,train\_acc 0.000000,test\_acc 0.000000
7. epoch 1, loss 0.0714,train\_acc 0.000000,test\_acc 0.000000
8. epoch 1, loss 0.0712,train\_acc 0.000000,test\_acc 0.000000
9. epoch 2, loss 0.0706,train\_acc 0.000000,test\_acc 0.000000
10. epoch 2, loss 0.0708,train\_acc 0.000000,test\_acc 0.025000
11. epoch 2, loss 0.0707,train\_acc 0.000000,test\_acc 0.050000
12. epoch 2, loss 0.0706,train\_acc 0.000000,test\_acc 0.075000
13. epoch 3, loss 0.0701,train\_acc 0.000000,test\_acc 0.150000
14. epoch 3, loss 0.0701,train\_acc 0.000000,test\_acc 0.200000
15. epoch 3, loss 0.0700,train\_acc 0.066667,test\_acc 0.350000
16. epoch 3, loss 0.0699,train\_acc 0.125000,test\_acc 0.400000
17. epoch 4, loss 0.0697,train\_acc 0.400000,test\_acc 0.475000
18. epoch 4, loss 0.0697,train\_acc 0.350000,test\_acc 0.575000
19. epoch 4, loss 0.0694,train\_acc 0.533333,test\_acc 0.650000
20. epoch 4, loss 0.0693,train\_acc 0.550000,test\_acc 0.750000
21. epoch 5, loss 0.0690,train\_acc 0.600000,test\_acc 0.825000
22. epoch 5, loss 0.0688,train\_acc 0.800000,test\_acc 0.850000
23. epoch 5, loss 0.0687,train\_acc 0.800000,test\_acc 0.875000
24. epoch 5, loss 0.0687,train\_acc 0.825000,test\_acc 0.925000
25. epoch 6, loss 0.0685,train\_acc 0.800000,test\_acc 0.925000
26. epoch 6, loss 0.0683,train\_acc 0.900000,test\_acc 0.950000
27. epoch 6, loss 0.0682,train\_acc 0.900000,test\_acc 0.975000
28. epoch 6, loss 0.0681,train\_acc 0.925000,test\_acc 0.975000
29. epoch 7, loss 0.0679,train\_acc 0.900000,test\_acc 0.975000
30. epoch 7, loss 0.0678,train\_acc 0.950000,test\_acc 0.975000
31. epoch 7, loss 0.0676,train\_acc 0.966667,test\_acc 0.975000
32. epoch 7, loss 0.0675,train\_acc 0.975000,test\_acc 0.975000
33. epoch 8, loss 0.0675,train\_acc 0.900000,test\_acc 0.975000
34. epoch 8, loss 0.0673,train\_acc 0.950000,test\_acc 0.975000
35. epoch 8, loss 0.0670,train\_acc 0.966667,test\_acc 0.975000
36. epoch 8, loss 0.0669,train\_acc 0.975000,test\_acc 0.975000
37. epoch 9, loss 0.0661,train\_acc 1.000000,test\_acc 0.975000
38. epoch 9, loss 0.0663,train\_acc 0.950000,test\_acc 0.975000
39. epoch 9, loss 0.0662,train\_acc 0.966667,test\_acc 1.000000
40. epoch 9, loss 0.0663,train\_acc 0.975000,test\_acc 1.000000
41. epoch 10, loss 0.0658,train\_acc 1.000000,test\_acc 1.000000
42. epoch 10, loss 0.0656,train\_acc 1.000000,test\_acc 1.000000
43. epoch 10, loss 0.0657,train\_acc 1.000000,test\_acc 1.000000
44. epoch 10, loss 0.0657,train\_acc 1.000000,test\_acc 1.000000

**分析：**根据结果分析可得，随着迭代次数的增加，函数的损失值不断减少，训练精度逐渐上升，模型的准确性随着迭代次数的增加不断提高，和预期结果相符合。

#### 5.3 Softmax 回归实验

**5.3.1 实验4.3.1运行结果**

1. <torch.utils.data.dataloader.DataLoader object at 0x000001BC2D2A1988>
2. epoch 1, loss 2.3442,train\_acc 0.000,test\_acc 0.192
3. epoch 1, loss 2.2477,train\_acc 0.050,test\_acc 0.227
4. epoch 1, loss 2.5763,train\_acc 0.133,test\_acc 0.215
5. epoch 1, loss 2.2638,train\_acc 0.225,test\_acc 0.240
6. epoch 1, loss 2.3536,train\_acc 0.200,test\_acc 0.202
7. epoch 1, loss 2.2479,train\_acc 0.200,test\_acc 0.256
8. epoch 1, loss 2.4571,train\_acc 0.186,test\_acc 0.367
9. epoch 1, loss 2.4093,train\_acc 0.212,test\_acc 0.264
10. epoch 1, loss 2.3457,train\_acc 0.211,test\_acc 0.183
11. epoch 1, loss 2.4236,train\_acc 0.200,test\_acc 0.318
12. epoch 1, loss 2.3814,train\_acc 0.191,test\_acc 0.388
13. epoch 1, loss 2.3315,train\_acc 0.200,test\_acc 0.415
14. epoch 1, loss 2.3451,train\_acc 0.200,test\_acc 0.405
15. epoch 1, loss 2.2799,train\_acc 0.229,test\_acc 0.343
16. epoch 1, loss 2.2514,train\_acc 0.227,test\_acc 0.401
17. epoch 1, loss 2.2333,train\_acc 0.250,test\_acc 0.305
18. epoch 1, loss 2.2661,train\_acc 0.247,test\_acc 0.380
19. epoch 1, loss 2.2308,train\_acc 0.261,test\_acc 0.429
20. epoch 1, loss 2.2335,train\_acc 0.263,test\_acc 0.431
21. epoch 1, loss 2.2239,train\_acc 0.265,test\_acc 0.562
22. epoch 1, loss 2.2273,train\_acc 0.267,test\_acc 0.594
23. epoch 1, loss 2.2172,train\_acc 0.286,test\_acc 0.609
24. epoch 1, loss 2.1705,train\_acc 0.304,test\_acc 0.647
25. epoch 1, loss 2.1154,train\_acc 0.317,test\_acc 0.633
26. epoch 1, loss 2.0827,train\_acc 0.328,test\_acc 0.510
27. epoch 1, loss 2.0654,train\_acc 0.335,test\_acc 0.595
28. epoch 1, loss 2.0286,train\_acc 0.348,test\_acc 0.463
29. epoch 1, loss 1.9944,train\_acc 0.361,test\_acc 0.573
30. epoch 1, loss 1.9703,train\_acc 0.369,test\_acc 0.637
31. epoch 1, loss 1.9492,train\_acc 0.377,test\_acc 0.582
32. epoch 1, loss 1.9194,train\_acc 0.384,test\_acc 0.543
33. epoch 1, loss 1.9123,train\_acc 0.391,test\_acc 0.609
34. epoch 1, loss 1.8818,train\_acc 0.397,test\_acc 0.484
35. epoch 1, loss 1.8775,train\_acc 0.397,test\_acc 0.501
36. epoch 1, loss 1.8880,train\_acc 0.394,test\_acc 0.556
37. epoch 1, loss 1.8822,train\_acc 0.400,test\_acc 0.551
38. epoch 1, loss 1.8741,train\_acc 0.403,test\_acc 0.612
39. epoch 1, loss 1.8567,train\_acc 0.405,test\_acc 0.666
40. epoch 1, loss 1.8316,train\_acc 0.413,test\_acc 0.606
41. epoch 1, loss 1.8085,train\_acc 0.420,test\_acc 0.655
42. epoch 1, loss 1.7851,train\_acc 0.427,test\_acc 0.659
43. epoch 1, loss 1.7624,train\_acc 0.431,test\_acc 0.670
44. epoch 1, loss 1.7415,train\_acc 0.442,test\_acc 0.695
45. epoch 1, loss 1.7268,train\_acc 0.445,test\_acc 0.678
46. epoch 1, loss 1.7097,train\_acc 0.449,test\_acc 0.676
47. epoch 1, loss 1.7019,train\_acc 0.454,test\_acc 0.642

**分析：**根据结果分析可得，随着迭代次数的增加，函数的损失值不断减少，训练精度逐渐上升，模型的准确性虽然有着上下波动的趋势，但是按照整体情况来看是随着迭代次数的增加不断提高，基本上符合预期结果。

**5.3.2 实验4.3.2运行结果**

1. <torch.utils.data.dataloader.DataLoader object at 0x000001B3C7544B88>
2. epoch 1, loss 0.0031, train acc 0.748, test acc 0.764
3. epoch 2, loss 0.0022, train acc 0.812, test acc 0.804
4. epoch 3, loss 0.0021, train acc 0.825, test acc 0.813
5. epoch 4, loss 0.0020, train acc 0.832, test acc 0.821
6. epoch 5, loss 0.0019, train acc 0.837, test acc 0.822
7. epoch 6, loss 0.0019, train acc 0.840, test acc 0.828
8. epoch 7, loss 0.0018, train acc 0.843, test acc 0.831
9. epoch 8, loss 0.0018, train acc 0.845, test acc 0.829
10. epoch 9, loss 0.0018, train acc 0.847, test acc 0.831
11. epoch 10, loss 0.0017, train acc 0.850, test acc 0.833
12. epoch 11, loss 0.0017, train acc 0.849, test acc 0.833
13. epoch 12, loss 0.0017, train acc 0.851, test acc 0.836
14. epoch 13, loss 0.0017, train acc 0.851, test acc 0.833
15. epoch 14, loss 0.0017, train acc 0.852, test acc 0.827
16. epoch 15, loss 0.0017, train acc 0.853, test acc 0.830
17. epoch 16, loss 0.0017, train acc 0.854, test acc 0.829
18. epoch 17, loss 0.0017, train acc 0.855, test acc 0.835
19. epoch 18, loss 0.0017, train acc 0.855, test acc 0.837
20. epoch 19, loss 0.0016, train acc 0.856, test acc 0.831
21. epoch 20, loss 0.0016, train acc 0.856, test acc 0.836
22. epoch 21, loss 0.0016, train acc 0.856, test acc 0.839
23. epoch 22, loss 0.0016, train acc 0.858, test acc 0.840
24. epoch 23, loss 0.0016, train acc 0.858, test acc 0.839
25. epoch 24, loss 0.0016, train acc 0.859, test acc 0.838
26. epoch 25, loss 0.0016, train acc 0.859, test acc 0.841
27. epoch 26, loss 0.0016, train acc 0.859, test acc 0.839
28. epoch 27, loss 0.0016, train acc 0.859, test acc 0.839
29. epoch 28, loss 0.0016, train acc 0.860, test acc 0.841
30. epoch 29, loss 0.0016, train acc 0.861, test acc 0.841
31. epoch 30, loss 0.0016, train acc 0.861, test acc 0.840

**分析：**根据结果分析可得，采用torch.nn函数进行softmax回归运算，其总的运行时间相比于前一种方法有所减少，结果呈现为函数的损失值不断减少，训练精度逐渐上升，模型的准确性虽然有着上下波动的趋势，但是按照整体情况来看是随着迭代次数的增加不断提高，基本上符合预期结果。

## 六、实验心得体会

这次实验让我初步了解了PyTorch的基本原理和操作，学会了数据构建，对简单的数据矩阵进行运算，并学会了以计算机语言进行梯度计算。此外，我还学会了以PyTorch方法进行Logistic回归和Softmax回归，其实验结果也较为准确，这个过程将我初步带入深度学习的大门，了解了深度学习的基本知识，强化了我的编程能力，对我的个人能力有着较大的提高。但是与此同时我也发现了部分回归模型还需要参考网上的代码，这也是现阶段的不足之处。在接下来的学习中我会再接再厉，力争做得更好。

## 七、参考文献

[1] https://blog.csdn.net/qq\_37534947/article/details/108179408

[2] 动⼿学深度学习-PyTorch

## 八、附录

需要补充说明的内容，如无可略。

**实验报告填写要求**

1. 正文要求小四号宋体，行间距1.5倍；

2. 英文要求小四号Times New Roman；

3. 在实验内容、实验过程、实验结果三部分需要针对当次实验不同的实验内容分别填写（模版以实验一为例），实验设计中如有必要也可以分开填写；

4. 实验报告配图的每幅图应有编号和标题，编号和标题应位于图下方处，居中，中文用五号宋体；

5. 表格应为三线表，每个表格应有编号和标题，编号和标题应写在表格上方正中，距正文段前0.5倍行距。表格中量与单位之间用“/”分隔，编号与标题中的中文用五号宋体；

6. 图、表、公式、算式等，一律用阿拉伯数字分别依序连续编排序号。其标注形式应便于互相区别，可分别为：图1、表2、公式(5)等。