**实验3 基于PyTorch手写数字识别**

1. **实验目的**
2. 使用pytorch构建卷积神经网络训练手写数字识别；
3. 使用pytorch进行数据变换和增强；
4. 使用pytorch训练和测试卷积神经网络。
5. **实验内容**
6. 数据预览测试和数据装载；
7. 进行数据处理变换操作；
8. 模型搭建和参数优化；
9. 基于pytorch卷积神经网络的模型训练；
10. 基于pytorch卷积神经网络的模型测试。
11. **实验指导**

**1 . 包导入及所需数据的下载**

torchvision包的主要功能是实现数据的处理、导入、预览等，所以如果需要对计算机视觉的相关问题进行处理，就可以借用在torchvision包中提供的大量的类来完成相应的工作。

代码的开始部分有这两个：

import torch

from torchvision import datasets, transforms # torchvision包的主要功能是实现数据的处理、导入和预览等

torchvision.datasets：实现对数据集的训练集和测试集的下载，只需使用torchvision再加上需要下载的数据集的名称就可以了，比如本例的MNIST

下载数据集的代码如下：

data\_train = datasets.MNIST(

transform=transform,

root="./data/",

train=True,

download=True

)

data\_test = datasets.MNIST(

root="./data/",

transform=transform,

train=True,

download=False

)

①root用于指定数据集在下载之后的存放路径，这里存放在根目录下的data文件夹

②transform用于指定导入数据集是需要对数据进行哪种变换操作

③train用于指定数据集下载完成后需要载入哪部分数据（如果设置为True，则说明载入的是该数据集的训练集部分；如果设置为False，则说明载入的是该数据集的测试集部分）

此处稍微地有些小改动，因为整个导入下载的数据集大约有6万张图片，这是一个极大的数据量，一台配置正常的电脑程序运行的时间需求将会是巨大的（一台正常配置的学生电脑），所以此处将6万张数据集的训练集和测试集都只截取了前1000张用作训练和测试，虽然说精度会降低，使得偏差较大，但是也足够用了，在时间上会有极大的节省，代码如下：

from torch.utils.data import random\_split

data\_train, \_ = random\_split(

dataset=data\_train,

lengths=[1000, 59000],

generator=torch.Generator().manual\_seed(0)

)

data\_test, \_ = random\_split(

dataset=data\_test,

lengths=[1000, 59000],

generator=torch.Generator().manual\_seed(0)

)

调用torch.utils.data import random\_split函数对数据集进行了切割，使得数据量减少，提升了运行速率。

**2. 进行数据处理变换操作**

在torch.transforms中提供了丰富的类对载入的数据进行变换。我们知道，在计算机视觉中处理的数据集有很大一部分是图片类型的，而在PyTorch中实际进行计算的是Tensor数据类型的变量，所以我们首先需要解决的是数据类型转换的问题

对数据进行载入及有相应变化的代码如下：

transform = transforms.Compose(

[transforms.ToTensor(), transforms.Normalize(mean=[0.5], std=[0.5])]

)

我们可以将以上代码中的torchvision.transforms.Compose类看成一种容器，它能够同时对多种数据变换进行组合。传入的参数是一个列表，列表中的元素就开始对载入的数据进行各种变换操作。例如本例：

①转化数据类型为Tensor（张量）

②对均值(mean)和标准差(std)均为0.5的原始数据进行数据标准化变化

**3. 数据预览测试和数据装载**

数据下载完成并载入之后，我们还需对数据进行装载。我们可以将数据的载入理解为对图片的处理，在处理完成后，我们就需要将这些图片打包好送给我们的模型进行训练了，而装载就是这个打包的过程。

代码片如下：

data\_loader\_train = torch.utils.data.DataLoader(dataset=data\_train,

batch\_size=4,

shuffle=True)

data\_loader\_test = torch.utils.data.DataLoader(dataset=data\_test,

batch\_size=4,

shuffle=True)

对数据的装载使用的是torch.utils.data.DataLoader类，类中的参数：

①batch\_size参数设置了每个包中的图片数据个数，代码中的值是4（此处如果电脑配置不是很高或者想让程序跑的快一点的话可以稍微调低，原本为64，此处我将其调为4）

②dataset参数用于指定我们载入的数据集的名称。 ③将shuffle参数设置为True，在装载的过程中会将数据随机打乱顺序并进行打包。

在装载完成后，我们可以选取其中一个批次的数据进行预览。进行数据预览的代码如下：

images, labels = next(iter(data\_loader\_train))

img = torchvision.utils.make\_grid(images)

img = img.numpy().transpose(1, 2, 0)

std = [0.5]

mean = [0.5]

img = img \* std + mean

print([labels[i] for i in range(4)])

plt.imshow(img)

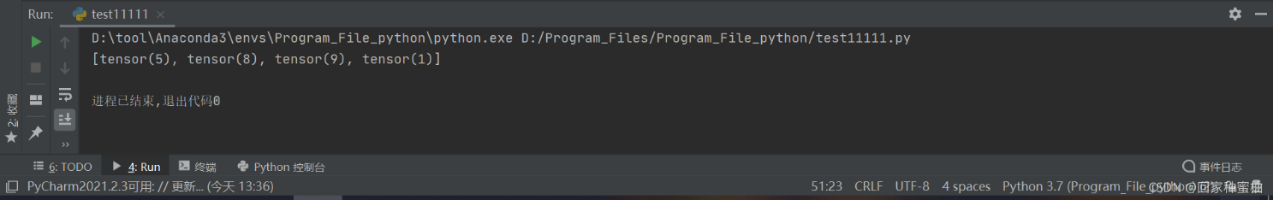
plt.show()

在以上代码中使用了iter和next来获取一个批次的图片数据(images)和其对应的图片标签(abels)。然后使用torchvision.utils中的make\_grid类方法将一个批次的图片构造成网格模式。需要传递给torchvision.utils.make\_grid的参数就是一个批次的装载数据，每个批次的装载数据都是4维的，维度的构成从前往后分别为batch\_size、channel、height、weight，分别对应一个批次中的数据个数、每张图片的色彩通道数、每张图片的高度和宽度。

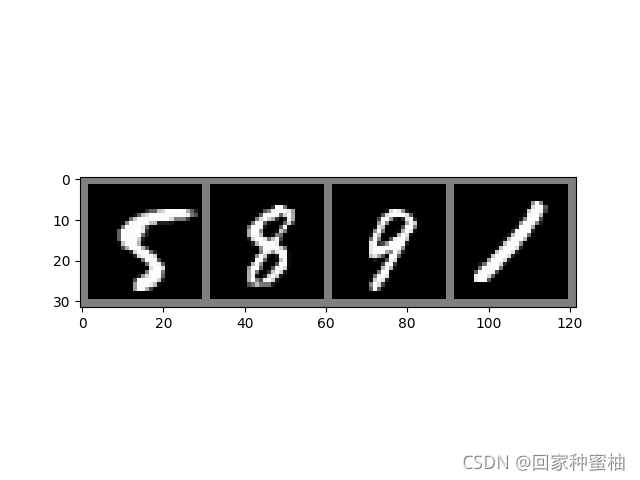
在通过torchvision.utils.make\_grid之后，图片的维度就变成了（channel，height，weight），这个批次的图片全部被整合到了一起，所以在这个维度中对应的值也和之前不一样了，但是色彩通道数保持不变。

若我们想使用Matplotlib将数据显示成正常的图片形式，则使用的数据首先必须是数组，其次这个数组的维度必须是（height、weight、channel）,即色彩通道数在最后面。

所以我们要通过numpy和transpose完成原始数据类型的转换和数据维度的交换，这样才能够使用Matplotlib绘制出正确的图像。在完成数据预览的代码中，我们先打印输出了这个批次中的数据的全部标签，然后才对这个批次中的所有图片数据进行显示。结果如下



效果图如下，可以看到，打印输出的首先是4张图片对应的标签，然后是4张图片的预览效果



plt.show()的话如果是使用PyCham编译的话一定要加上去，不然会出现显示不出图像的情况。

**4. 模型搭建和参数优化**

在顺利完成数据装载之后，我们就可以开始编写卷积神经网络的搭建和参数优化的代码了。

**卷积层使用torch.nn.Conv2d类方法来搭建；**  
**激活层使用torch.nn.ReLU()类方法来搭建；**  
**池化层使用torch.nn.MaxPool2d类方法来搭建；**  
**全连接层使用torch.nn.Linear类方法来搭建**

实现卷积神经网络模型搭建的代码如下：

class Model(torch.nn.Module):

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

super(Model, self).\_\_init\_\_()

self.conv1 = torch.nn.Sequential(

torch.nn.Conv2d(1, 64, kernel\_size=3, stride=1, padding=1),

torch.nn.ReLU(),

torch.nn.Conv2d(64, 128, kernel\_size=3, stride=1, padding=1),

torch.nn.ReLU(),

torch.nn.MaxPool2d(stride=2, kernel\_size=2)

)

self.dense = torch.nn.Sequential(

torch.nn.Linear(14 \* 14 \* 128, 1024),

torch.nn.ReLU(),

torch.nn.Dropout(p=0.5),

torch.nn.Linear(1024, 10)

)

def forward(self, x):

x = self.conv1(x) # 卷积处理

x = x.view(-1, 14\*14\*128) # 对参数实行扁平化处理

x = self.dense(x)

return x

我们选择搭建一个在结构层次上有所简化的卷积神经网络模型，在结构上使用了两个卷积层：一个最大池化层和两个全连接层。

torch.nn.Conv2d():用于搭建卷积神经网络的卷积层，主要的输入参数有输入通道数、输出通道数、卷积核大小、卷积核移动步长和Padding值。其中，输入通道数的数据类型是整型，用于确定输入数据的层数；输出通道数的数据类型也是整型，用于确定输出数据的层数；卷积核大小的数据类型是整型，用于确定卷积核的大小；卷积核移动步长的数据类型是整型，用于确定卷积核每次滑动的步长；Paddingde的数据类型是整型，值为0时代表不进行边界像素的填充，如果值大于0，那么增加数字所对应的边界像素层数。

torch.nn.MaxPool2d():用于实现卷积神经网络中的最大池化层，主要的输入参数时池化窗口的大小、池化窗口移动步长和Paddingde值。同样：池化窗口大小的数据类型是整型，用于确定池化窗口的大小。池化窗口步长的数据类型也是整型，用于确定池化窗口每次移动的步长。Paddingde值和在torch.nn.Conv2d中定义的Paddingde值的用法和意义时一样的。

torch.nn.Dropout():torch.nn.Dropout类用于防止卷积神经网络在训练的过程中发生过拟合，其工作原理简单来说就是在模型训练的过程中，以一定的随机概率将卷积神经网络模型的部分参数归零，以达到减少相邻两层神经连接的目的。

**代码前向传播forward函数中的内容：**

首先，经过self.conv1进行卷积处理；然后进行x.view(-1 ,14 \* 14 \*128)，对参数实现扁平化因为之后紧挨着就是全连接层，所以如果不进行扁平化处理，则全连接层的实际输出的参数维度和其定义输入的维度将不匹配，程序会报错；最后，通过self.dense定义的全连接进行最后的分类。在编辑完搭建卷积神经网络模型的代码之后，我们就可以开始对模型进行训练和对参数进行优化了。首先，定义在训练之前使用哪种损失函数和优化函数：

model = Model()

cost = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters())

# 损失函数： 交叉熵

# 优化函数： Adam自适应优化算法，需要优化的参数实在Model中生成的全部参数，

#因为没有定义学习速率的值，所以使用默认值

最后，卷积神经网络模型进行模型训练和参数优化的代码如下：

epochs\_n = 5

for epoch in range(epochs\_n):

running\_loss = 0.0

running\_correct = 0

print("Epoch{}/{}".format(epoch, epochs\_n))

print("-" \* 10)

for data in data\_loader\_train:

X\_train, y\_train = data

X\_train, y\_train = Variable(X\_train), Variable(y\_train)

outputs = model(X\_train)

\_,pred = torch.max(outputs.data, 1)

optimizer.zero\_grad()

loss = cost(outputs, y\_train)

loss.backward()

optimizer.step()

running\_loss += loss.data

running\_correct += torch.sum(pred == y\_train.data)

testing\_correct = 0

for data in data\_loader\_test:

X\_test, y\_test = data

X\_test, y\_test = Variable(X\_test), Variable(y\_test)

outputs = model(X\_test)

\_, pred = torch.max(outputs.data, 1)

testing\_correct += torch.sum(pred == y\_test.data)

print("Loss is:{:.4f},Train Accuracy is:{:.4f}%, Test Accuracy is:{:.4f}".format(running\_loss / len(data\_train),100 \* running\_correct / len(data\_train),100 \* testing\_correct / len(data\_test)))

\*关于模型搭建的改动

在此处对上面模型进行了优化改动，大大优化了运行的时间，但是对应也减少了一些训练精度。原理就是，卷积层的运算量不会太大，但全连接层的运算量比较大，所以降低全连接的参数量，以及降低图像特征图的尺寸。

class Model(torch.nn.Module):

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

super(Model, self).\_\_init\_\_()

self.conv1 = torch.nn.Sequential(

torch.nn.Conv2d(1, 64, kernel\_size=3, stride=2, padding=1),

torch.nn.ReLU(),

torch.nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1),

torch.nn.ReLU(),

# torch.nn.MaxPool2d(stride=2, kernel\_size=2)

)

self.dense = torch.nn.Sequential(

# torch.nn.Linear(14 \* 14 \* 128, 1024),

torch.nn.Linear(7 \* 7 \* 128, 512),

torch.nn.ReLU(),

# torch.nn.Dropout(p=0.5),

torch.nn.Dropout(p=0.8),

torch.nn.Linear(512, 10)

)

def forward(self, x):

x = self.conv1(x) # 卷积处理

# x = x.view(-1, 14\*14\*128) # 对参数实行扁平化处理

x = x.view(-1, 7\*7\*128) # 对参数实行扁平化处理

x = self.dense(x)

return x

为了验证我们训练的模型是不是真的已如结果显示的一样准确，则最好的方法就是随机选取一部分测试集中的图片，用训练好的模型进行预测，看看和真实值有多大偏差，并对结果进行可视化，测试的代码如下：

X\_test, y\_test = next(iter(data\_loader\_test))

inputs = Variable(X\_test)

pred = model(inputs)

\_, pred = torch.max(pred,1)

print("Predict Label is:", [i for i in pred.data])

print("Real Label is:", [i for i in y\_test])

img = torchvision.utils.make\_grid(X\_test)

img = img.numpy().transpose(1,2,0)

std = [0.5, 0.5, 0.5]

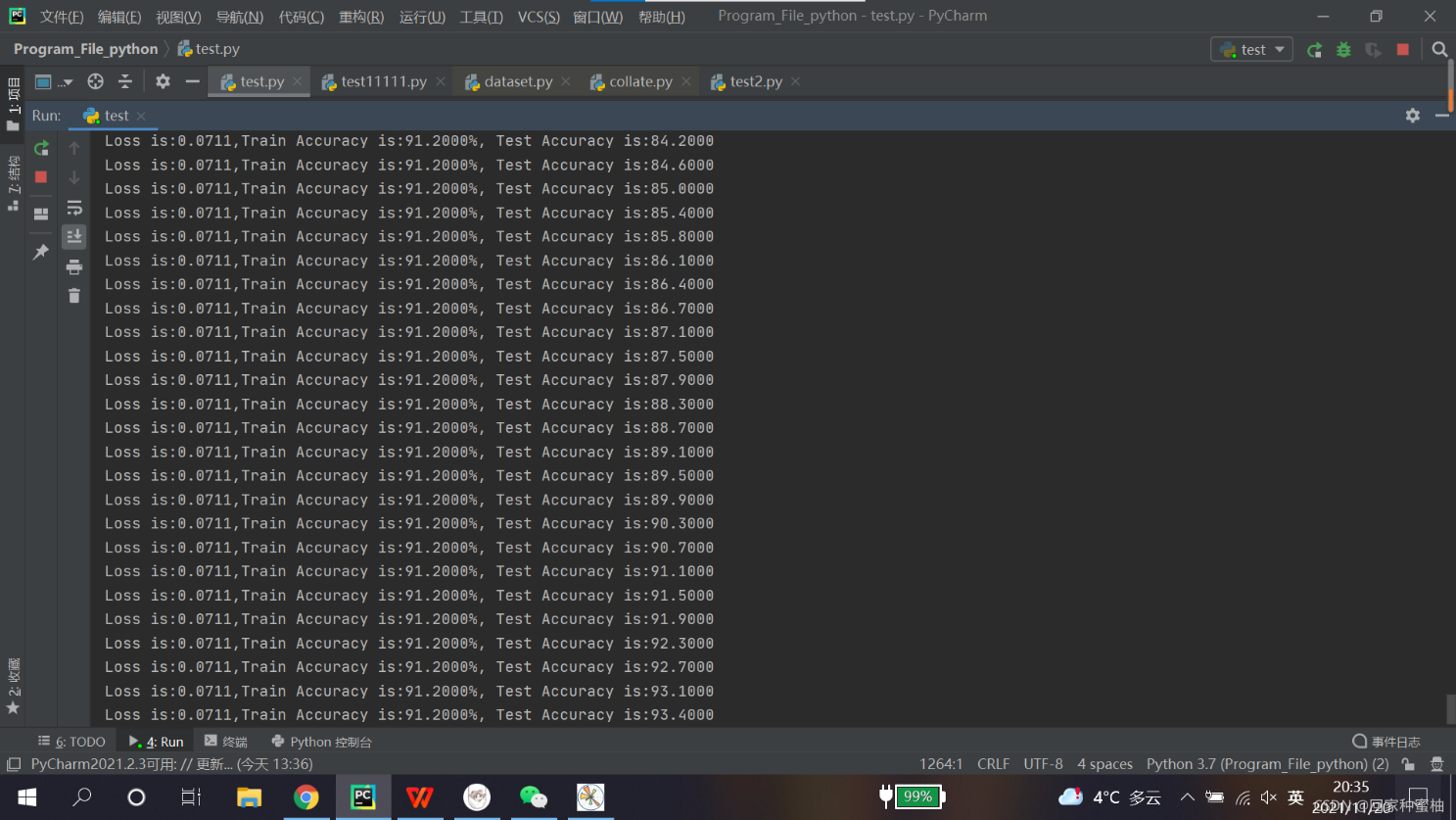
mean = [0.5, 0.5, 0.5]

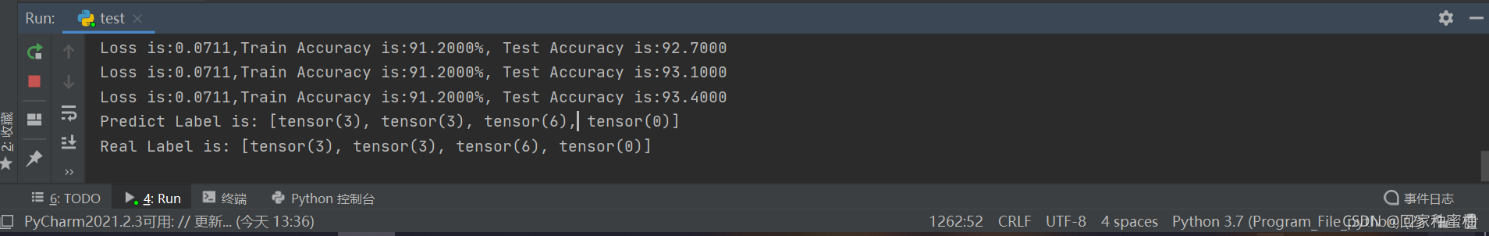
img = img\*std+mean

plt.imshow(img)

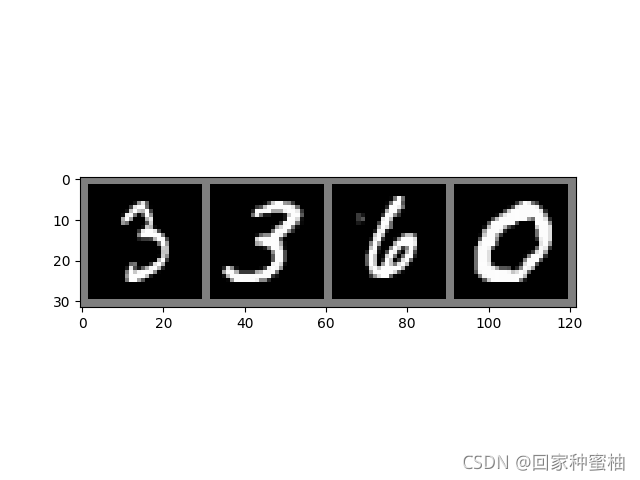
plt.show()

用于测试的数据标签结果输出如下：





在输出结果中  
**第1个结果是我们训练好的模型的预测值，第2个结果是这4个测试数据的真实值。**  
对测试数据进行可视化，如下图所示：



以看到，在上图可视化的这部分测试集图片，模型的预测结果和真实结果是完全一致的。当然如果想选取更多的测试集进行可视化，则只需将batch\_size设置的更大，但考虑对应程序的运行速度将会略微降低。

import torch

import numpy

import torchvision

import matplotlib.pyplot as plt

from torchvision import datasets, transforms # torchvision包的主要功能是实现数据的处理、导入和预览等

from torch.autograd import Variable

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean=[0.5], std=[0.5])])

data\_train = datasets.MNIST(

transform=transform,

root="./data/",

train=True,

download=True

)

data\_test = datasets.MNIST(

root="./data/",

transform=transform,

train=True,

download=False

)

from torch.utils.data import random\_split

data\_train, \_ = random\_split(

dataset=data\_train,

lengths=[1000, 59000],

generator=torch.Generator().manual\_seed(0)

)

data\_test, \_ = random\_split(

dataset=data\_test,

lengths=[1000, 59000],

generator=torch.Generator().manual\_seed(0)

)

data\_loader\_train = torch.utils.data.DataLoader(dataset=data\_train,

batch\_size=4,

shuffle=True)

data\_loader\_test = torch.utils.data.DataLoader(dataset=data\_test,

batch\_size=4,

shuffle=True)

# images, labels = next(iter(data\_loader\_train))

#

# img = torchvision.utils.make\_grid(images)

# img = img.numpy().transpose(1, 2, 0)

#

# std = [0.5]

# mean = [0.5]

# img = img \* std + mean

#

# print([labels[i] for i in range(64)])

# plt.imshow(img)

# plt.show()

# class Model(torch.nn.Module):

#

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

# super(Model, self).\_\_init\_\_()

# self.conv1 = torch.nn.Sequential(

# torch.nn.Conv2d(1, 64, kernel\_size=3, stride=1, padding=1),

# torch.nn.ReLU(),

# torch.nn.Conv2d(64, 128, kernel\_size=3, stride=1, padding=1),

# torch.nn.ReLU(),

# torch.nn.MaxPool2d(stride=2, kernel\_size=2)

# )

#

# self.dense = torch.nn.Sequential(

# torch.nn.Linear(14 \* 14 \* 128, 1024),

# torch.nn.ReLU(),

# torch.nn.Dropout(p=0.5),

# torch.nn.Linear(1024, 10)

# )

#

# def forward(self, x):

# x = self.conv1(x) # 卷积处理

# x = x.view(-1, 14\*14\*128) # 对参数实行扁平化处理

# x = self.dense(x)

# return x

class Model(torch.nn.Module):

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

super(Model, self).\_\_init\_\_()

self.conv1 = torch.nn.Sequential(

torch.nn.Conv2d(1, 64, kernel\_size=3, stride=2, padding=1),

torch.nn.ReLU(),

torch.nn.Conv2d(64, 128, kernel\_size=3, stride=2, padding=1),

torch.nn.ReLU(),

# torch.nn.MaxPool2d(stride=2, kernel\_size=2)

)

self.dense = torch.nn.Sequential(

# torch.nn.Linear(14 \* 14 \* 128, 1024),

torch.nn.Linear(7 \* 7 \* 128, 512),

torch.nn.ReLU(),

# torch.nn.Dropout(p=0.5),

torch.nn.Dropout(p=0.8),

torch.nn.Linear(512, 10)

)

def forward(self, x):

x = self.conv1(x) # 卷积处理

# x = x.view(-1, 14\*14\*128) # 对参数实行扁平化处理

x = x.view(-1, 7 \* 7 \* 128) # 对参数实行扁平化处理

x = self.dense(x)

return x

model = Model()

cost = torch.nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters())

epochs\_n = 5

for epoch in range(epochs\_n):

running\_loss = 0.0

running\_correct = 0

print("Epoch{}/{}".format(epoch, epochs\_n))

print("-" \* 10)

for data in data\_loader\_train:

X\_train, y\_train = data

X\_train, y\_train = Variable(X\_train), Variable(y\_train)

outputs = model(X\_train)

\_, pred = torch.max(outputs.data, 1)

optimizer.zero\_grad()

loss = cost(outputs, y\_train)

loss.backward()

optimizer.step()

running\_loss += loss.data

running\_correct += torch.sum(pred == y\_train.data)

testing\_correct = 0

for data in data\_loader\_test:

X\_test, y\_test = data

X\_test, y\_test = Variable(X\_test), Variable(y\_test)

outputs = model(X\_test)

\_, pred = torch.max(outputs.data, 1)

testing\_correct += torch.sum(pred == y\_test.data)

print("Loss is:{:.4f},Train Accuracy is:{:.4f}%, Test Accuracy is:{:.4f}".format(running\_loss / len(data\_train),

100 \* running\_correct / len(

data\_train),

100 \* testing\_correct / len(

data\_test)))

X\_test, y\_test = next(iter(data\_loader\_test))

inputs = Variable(X\_test)

pred = model(inputs)

\_, pred = torch.max(pred, 1)

print("Predict Label is:", [i for i in pred.data])

print("Real Label is:", [i for i in y\_test])

img = torchvision.utils.make\_grid(X\_test)

img = img.numpy().transpose(1, 2, 0)

std = [0.5, 0.5, 0.5]

mean = [0.5, 0.5, 0.5]

img = img \* std + mean

plt.imshow(img)

plt.show()

最后，关于这类代码的运行时间的需求都是巨大的，所以短时间内出不来很正常，尽量别中途中断程序，若你想检测程序是否运行：

epochs\_n = 5

for epoch in range(epochs\_n):

running\_loss = 0.0

running\_correct = 0

print("Epoch{}/{}".format(epoch, epochs\_n))

print("-" \* 10)

iter = 0

for data in data\_loader\_train:

iter+=1

print(iter)

X\_train, y\_train = data

X\_train, y\_train = Variable(X\_train), Variable(y\_train)

outputs = model(X\_train)

\_, pred = torch.max(outputs.data, 1)

optimizer.zero\_grad()

loss = cost(outputs, y\_train)

loss.backward()

optimizer.step()

running\_loss += loss.data

running\_correct += torch.sum(pred == y\_train.data)

testing\_correct = 0

for data in data\_loader\_test:

X\_test, y\_test = data

X\_test, y\_test = Variable(X\_test), Variable(y\_test)

outputs = model(X\_test)

\_, pred = torch.max(outputs.data, 1)

testing\_correct += torch.sum(pred == y\_test.data)

print("Loss is:{:.4f},Train Accuracy is:{:.4f}%, Test Accuracy is:{:.4f}".format(running\_loss / len(data\_train),

100 \* running\_correct / len(

data\_train),

100 \* testing\_correct / len(

data\_test)))