利用全连接神经网络识别MNIST

第一步：导入相关的库

import torch  *#* PyTorch库

import torch.nn as nn

import torchvision

import torchvision.transforms as transforms

第二步：导入MNIST数据集

*#* MNIST数据集 50k+10k  训练数据集 50000个 测试数据集 10000个

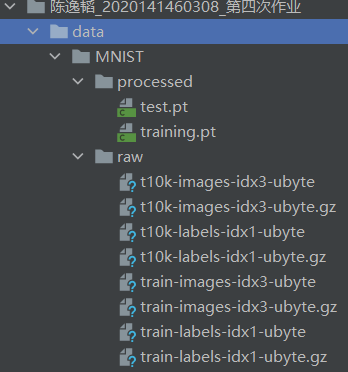
trainDataset = **torchvision**.datasets.MNIST(root='data', train=True, transform=**transforms**.ToTensor(), download=True)

testDataset = **torchvision**.datasets.MNIST(root='data', train=False, transform=**transforms**.ToTensor())

*#* 加载MNIST数据集

trainLoader = **torch**.utils.data.DataLoader(dataset=trainDataset, batch\_size=batchSize, shuffle=True)

testLoader = **torch**.utils.data.DataLoader(dataset=testDataset, batch\_size=batchSize, shuffle=False)



在我的目录下创建了一个data文件夹，下面有一个MNIST，

MNIST下面有两个文件夹，一个是processed，里面有test和training，另一个是raw，里面有ubyte等文件

第三步：设置一些参数

inputSize = 784  *#* 输入大小

hiddenSize = 500  *#* 隐藏层大小

numClasses = 10  *#* 类

learningRate = 0.001  *#* alpha学习率

numEpochs = 50  *#* 训练次数

batchSize = 100  *#* 连接大小

第四步，设置全连接神经网络

*#* 全连接神经网络模型

class **NeuralNetwork**(**nn.Module**):

    def **\_\_init\_\_**(self, inputSize, hiddenSize, numClasses):

**super**(**NeuralNetwork**, self).\_\_init\_\_()  *#* 继承

*#* 一共设置三层，第一层和第三层用线性回归Linear()函数，中间使用ReLU()函数

        self.fc1 = **nn**.Linear(inputSize, hiddenSize)

        self.relu = **nn**.ReLU()

        self.fc2 = **nn**.Linear(hiddenSize, numClasses)

*#* 输入x后输出的output作为下一个的输入，迭代下去，输出output

    def **forward**(self, x):

        output = self.fc1(x)

        output = self.relu(output)

        output = self.fc2(output)

        return output

criterion = **nn**.CrossEntropyLoss()  *#* 定义损失函数

optimizer = **torch**.optim.Adam(model.parameters(), lr=learningRate)  *#* 定义优化器

第五步：对神经网络训练

*#* 进行训练

step = **len**(trainLoader)  *#* 训练步长

for epoch in **range**(numEpochs):

    for i, (images, labels) in **enumerate**(trainLoader):

*#* 移动 tensors 至 GPU 中进行计算

        images = images.reshape(-1, 28 \* 28).to(device)

        labels = labels.to(device)

*#* 定义前向传递

        outputs = model(images)

        loss = criterion(outputs, labels)

*#* 定义反向传播并且优化

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

*#* 每100步显示一下

        if (i + 1) % 100 == 0:

**print**('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'

                  .**format**(epoch + 1, numEpochs, i + 1, step, loss.item()))

第六步：对测试数据集进行测试

*#* 测试模型

with **torch**.no\_grad():

    correct = 0

    total = 0

    for images, labels in testLoader:

*#* 移动 tensors 至 GPU 中进行计算

        images = images.reshape(-1, 28 \* 28).to(device)

        labels = labels.to(device)

        outputs = model(images)

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

        total += labels.size(0)

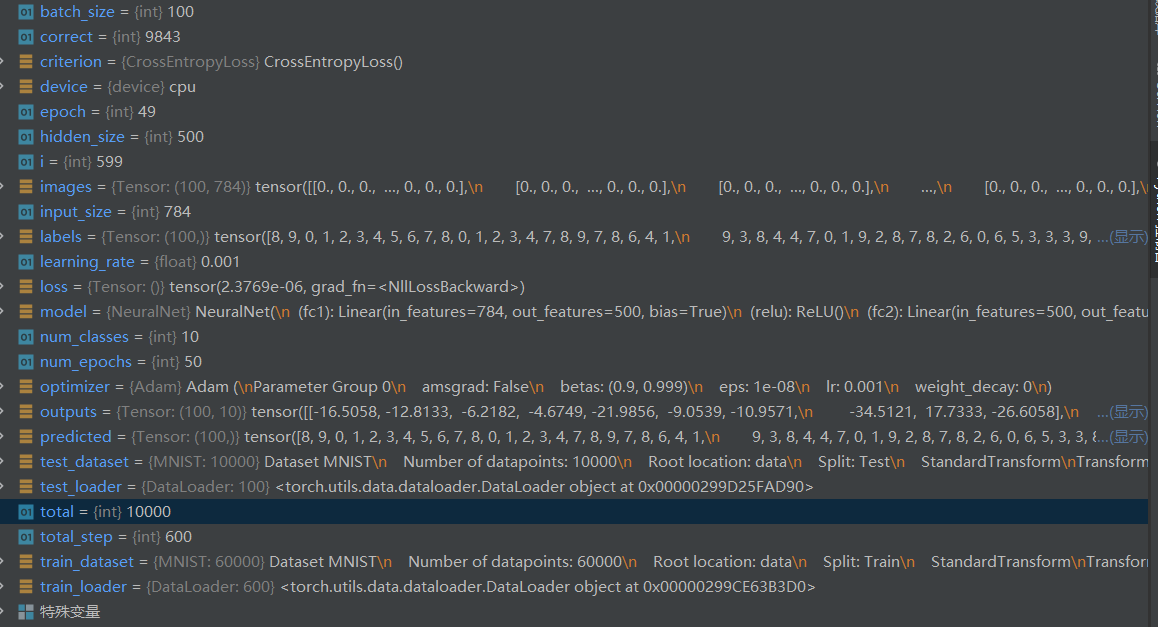
        correct += (predicted == labels).sum().item()

最后一步：打印出训练准确率

**print**('Accuracy of the network on the 10000 test images: {} %'.**format**(100 \* correct / total))



结果如下：



在刚开始的时候loss比较大，到了后面慢慢变小，在27/50的时候就开始变成0.000

之后的28-50之间在0.000-0.002之间反复波动，表示训练到位

