



代码：

import torch  
from torch import nn, optim  
from torch.autograd import Variable  
from torch.utils.data import DataLoader  
from torchvision import datasets  
from torchvision import transforms  
  
torch.manual\_seed(1) # 设置随机数种子，确保结果可重复  
batch\_size = 128 # 批处理大小  
learning\_rate = 1e-2 # 学习率  
num\_epoches = 10 # 训练次数  
  
# 下载训练集 MNIST 手写数字训练集  
  
train\_dataset = datasets.MNIST(  
 root='./data',  
 train=True, # 训练集  
 transform=transforms.ToTensor(),  
 download=True)  
  
test\_dataset = datasets.MNIST(  
 root='./data',  
 train=False, # 测试集  
 transform=transforms.ToTensor(),  
 download=True)  
# 数据的批处理，尺寸大小为batch\_size,  
# 在训练集中，shuffle 必须设置为True, 表示次序是随机的  
train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)  
test\_loader = DataLoader(test\_dataset, batch\_size=batch\_size, shuffle=False)  
  
  
# 定义卷积神经网络模型  
class Cnn(nn.Module):  
 def \_\_init\_\_(self, in\_dim, n\_class): # 28x28x1  
 super(Cnn, self).\_\_init\_\_()  
 self.conv = nn.Sequential(  
 nn.Conv2d(in\_dim, 6, 3, stride=1, padding=1), # 28 x28  
 nn.ReLU(True),  
 nn.MaxPool2d(2, 2), # 14 x 14  
 nn.Conv2d(6, 16, 5, stride=1, padding=0), # 10 \* 10\*16  
 nn.ReLU(True), nn.MaxPool2d(2, 2)) # 5x5x16  
  
 self.fc = nn.Sequential(  
 nn.Linear(400, 120), # 400 = 5 \* 5 \* 16  
 nn.Linear(120, 84),  
 nn.Linear(84, n\_class))  
  
 def forward(self, x):  
 out = self.conv(x)  
 out = out.view(out.size(0), 400) # 400 = 5 \* 5 \* 16,  
 out = self.fc(out)  
 return out  
  
  
model = Cnn(1, 10) # 图片大小是28x28, 10  
  
print(model)  
  
# 定义loss和optimizer  
criterion = nn.CrossEntropyLoss()  
optimizer = optim.SGD(model.parameters(), lr=learning\_rate)  
  
# 开始训练  
for epoch in range(num\_epoches):  
 print('epoch {}'.format(epoch + 1))  
 print('\*' \* 10)  
 running\_loss = 0.0  
 running\_acc = 0.0  
 for i,data in enumerate(train\_loader, 1): # 批处理  
 img, label = data  
 img = Variable(img)  
 label = Variable(label)  
 # 前向传播  
 out = model(img)  
 loss = criterion(out, label) # loss  
 running\_loss += loss.item() \* label.size(0) # total loss , 由于loss 是batch取均值的，需要把batch size 乘回去  
 \_,pred = torch.max(out, 1) # 预测结果  
 num\_correct = (pred == label).sum() # 正确结果的num  
 # accuracy = (pred == label).float().mean() #正确率  
 running\_acc += num\_correct.item() # 正确结果的总数  
 # 后向传播  
 optimizer.zero\_grad() # 梯度清零，以免影响其他batch  
 loss.backward() # 后向传播，计算梯度  
 optimizer.step() # 梯度更新  
 # if i % 300 == 0:  
 # print('[{}/{}] Loss: {:.6f}, Acc: {:.6f}'.format(  
 # epoch + 1, num\_epoches, running\_loss / (batch\_size \\* i),  
 # running\_acc / (batch\_size \\* i)))  
 # 打印一个循环后，训练集合上的loss 和 正确率  
 print('Train Finish {} epoch, Loss: {:.6f}, Acc: {:.6f}'.format(  
 epoch + 1, running\_loss / (len(train\_dataset)), running\_acc / (len(  
 train\_dataset))))  
  
 # 模型测试  
 model.eval()  
 eval\_loss = 0  
 eval\_acc = 0  
 for data in test\_loader: # test set 批处理  
 img, label = data  
  
 img = Variable(img, volatile=True) # volatile 确定你是否不调用.backward(),测试中不需要  
 label = Variable(label, volatile=True)  
 out = model(img) # 前向算法  
 loss = criterion(out, label) # 计算 loss  
 eval\_loss += loss.item() \* label.size(0) # total loss  
 \_, pred = torch.max(out, 1) # 预测结果  
 num\_correct = (pred == label).sum() # 正确结果  
 eval\_acc += num\_correct.item() # 正确结果总数  
  
 print('Test Loss: {:.6f}, Acc: {:.6f}'.format(eval\_loss / (len(  
 test\_dataset)), eval\_acc \* 1.0 / (len(test\_dataset))))