机器学习实践作业3 深度学习任务

1 任务目标

本次任务目标为使用python编写深度学习算法中的LeNet5，ResNet18，ResNet34模型实现Fashion-mnist数据集上的图像识别任务，寻找优化参数，同时对比三种模型的准确率（accuracy），精度（precision），召回率（recall），f1-score以及训练时间和执行时间，从中分析它们的优劣。

# 2 实现过程

## 2.0 数据集的读取与使用

使用torchvision来对Fashion-mnist数据集进行读取和使用。使用transforms对数据进行预处理，将像素值从[0,1]范围归一化到均值为0.5，标准差为0.5的范围，同时进行图像增强，具体为添加图像的水平和竖直翻转函数；使用datasets.FashionMNIST()读取训练集和测试集，Dataloader()分批次加载训练集和测试机以备后面训练和测试使用。

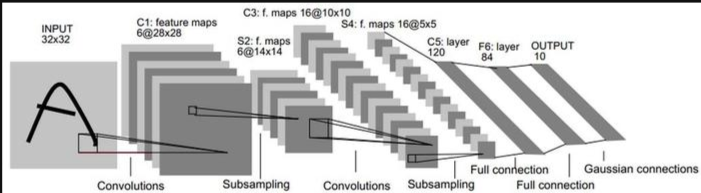
同时，在训练前，将部分训练数据可视化，以便于观察训练数据的特征。打印图片之后发现，该数据集中有用的像素点像素值较高（偏白色），无用的像素点像素值较低（偏黑色），这在后面预测任意实际图片时有很大帮助。

|  |
| --- |
| # 数据预处理  transform=transforms.Compose([transforms.ToTensor(),                              transforms.Normalize((0.5,),(0.5,)),                              transforms.RandomHorizontalFlip(p=0.5),                              transforms.RandomVerticalFlip(p=0.5),                              ])  # 加载数据训练集  trainset=torchvision.datasets.FashionMNIST(root='./DATASET',train=True,download=False,transform=transform)  trainload=torch.utils.data.DataLoader(trainset,shuffle=True,batch\_size=100,num\_workers=0)  # 加载测试数据集  testset=torchvision.datasets.FashionMNIST(root='./DATASET',train=False,download=False,transform=transform)  testload=torch.utils.data.DataLoader(testset,shuffle=True,batch\_size=100,num\_workers=0)  classes=('T-shirt/top', 'Trouser', 'Pullover', 'Dress','Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot') |

## 2.1 LeNet5

### 2.1.1 算法原理

LeNet-5出自论文Gradient-Based Learning Applied to Document Recognition，是一种用于手写体字符识别的非常高效的卷积神经网络。

****

**Fig. 2.** LeNet5 结构示意图

### LeNet-5共有7层，不包含输入，每层都包含可训练参数；每个层有多个Feature Map，每个FeatureMap通过一种卷积滤波器提取输入的一种特征，然后每个FeatureMap有多个神经元。

### 2.1.2 实现思路

本次主要使用pytorch中的torchnn.module模块来构建LeNet5.

关键代码：

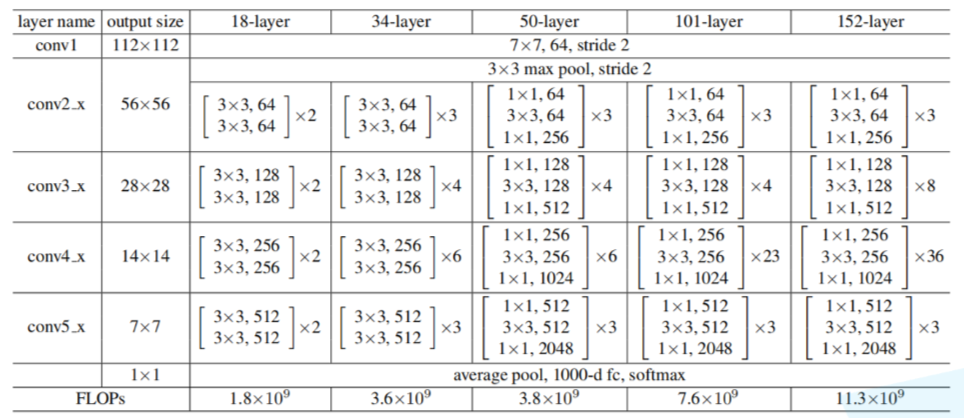
模型定义

|  |
| --- |
| class LeNet(nn.Module):  def \_\_init\_\_(self):  super(LeNet,self).\_\_init\_\_()  # 构建网络  self.conv1=nn.Conv2d(in\_channels=1,out\_channels=6,kernel\_size=5,stride=1)  self.pool1=nn.MaxPool2d(kernel\_size=2,stride=2)  self.conv2=nn.Conv2d(in\_channels=6,out\_channels=16,kernel\_size=5,stride=1)  self.pool2=nn.MaxPool2d(kernel\_size=2,stride=2)  self.fc1=nn.Linear(256,120)  self.fc2=nn.Linear(120,10)  # 前向传播  def forward(self,x):  x=F.relu(self.conv1(x))  x=self.pool1(x)  x=F.relu(self.conv2(x))  x=self.pool2(x)  x=x.view(x.size(0),-1)  #print(x.shape)  x=F.relu(self.fc1(x))  x=self.fc2(x)  return x |

## 2.2 ResNet

### 2.1.1 算法原理

ResNet网络也是经典的卷积网络之一。与VGG中笔直的数据流不同，在ResNet中，独有着“残差连接”这个概念。在某些层中，网络会将未经运算的数据直接加到经过运算的结果上，这部分数据就叫做“残差”。因此，ResNet与其他卷积神经网络相比，更注重了线性变换与非线性变换的平衡，在各种任务中取得了很好的效果，是现在最常用的神经网络之一。常见的ResNet网络有着不同的规格，如ResNet18、ResNet34、ResNet50等。



**Fig. 3.** ResNet 结构

### 2.1.2 实现思路

按照论文中所给结构，使用torch.nn模块搭建ResNet18和ResNet34结构。

**ResNet18**

|  |
| --- |
| class ResBlock(nn.Module):      def \_\_init\_\_(self,in\_channel,out\_channel,stride=1):          super(ResBlock,self).\_\_init\_\_()          # 定义残差块中的两个卷积层          self.left=nn.Sequential(              nn.Conv2d(in\_channel,out\_channel,kernel\_size=3,padding=1,stride=stride,bias=False),              nn.BatchNorm2d(out\_channel),              nn.ReLU(inplace=True),              nn.Conv2d(out\_channel,out\_channel,kernel\_size=3,padding=1,stride=1,bias=False),              nn.BatchNorm2d(out\_channel)          )          # 定义残差块中的shortcut          self.shortcut=nn.Sequential()          if stride!=1 or out\_channel!=in\_channel:              self.shortcut=nn.Sequential(                  nn.Conv2d(in\_channel,out\_channel,kernel\_size=1,stride=stride,bias=False),                  nn.BatchNorm2d(out\_channel)              )      def forward(self,x):          out=self.left(x)          out+=self.shortcut(x)          out=F.relu(out)          return out    class ResNet18(nn.Module):      def \_\_init\_\_(self,ResBlock,numclasses=10):          super(ResNet18,self).\_\_init\_\_()          self.conv1=nn.Sequential(              nn.Conv2d(in\_channels=1,out\_channels=64,kernel\_size=3,stride=1,padding=1,bias=False),              nn.BatchNorm2d(64),              nn.ReLU()          )          self.layer1 = nn.Sequential(ResBlock(64, 64, stride=1))          self.layer2 = nn.Sequential(ResBlock(64, 64, stride=1))          self.layer3 = nn.Sequential(ResBlock(64, 128, stride=2))          self.layer4 = nn.Sequential(ResBlock(128, 128, stride=1))          self.layer5 = nn.Sequential(ResBlock(128, 256, stride=2))          self.layer6 = nn.Sequential(ResBlock(256, 256, stride=1))          self.layer7 = nn.Sequential(ResBlock(256, 512, stride=2))          self.layer8 = nn.Sequential(ResBlock(512, 512, stride=1))          self.fc=nn.Linear(512,numclasses)      def forward(self,x):          out=self.conv1(x)          out=self.layer1(out)          out=self.layer2(out)          out=self.layer3(out)          out=self.layer4(out)          out=self.layer5(out)          out=self.layer6(out)          out=self.layer7(out)          out=self.layer8(out)          out=F.avg\_pool2d(out,4)          out=out.view(out.size(0),-1)          out=self.fc(out)          return out |

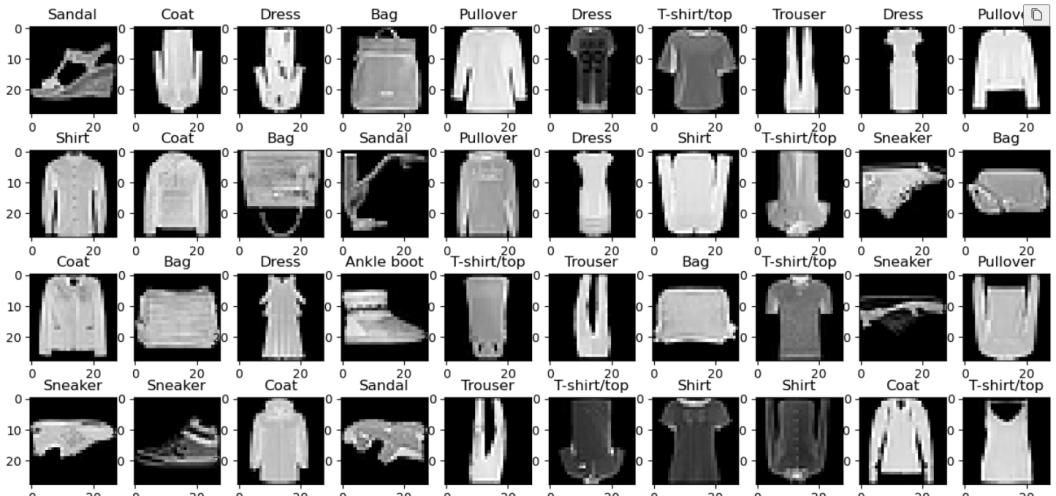
**ResNet34**

|  |
| --- |
| class ResBlock(nn.Module):      def \_\_init\_\_(self,in\_channel,out\_channel,stride=1):          super(ResBlock,self).\_\_init\_\_()          # 定义残差块中的两个卷积层          self.left=nn.Sequential(              nn.Conv2d(in\_channel,out\_channel,kernel\_size=3,padding=1,stride=stride,bias=False),              nn.BatchNorm2d(out\_channel),              nn.ReLU(inplace=True),              nn.Conv2d(out\_channel,out\_channel,kernel\_size=3,padding=1,stride=1,bias=False),              nn.BatchNorm2d(out\_channel)          )          # 定义残差块中的shortcut          self.shortcut=nn.Sequential()          if stride!=1 or out\_channel!=in\_channel:              self.shortcut=nn.Sequential(                  nn.Conv2d(in\_channel,out\_channel,kernel\_size=1,stride=stride,bias=False),                  nn.BatchNorm2d(out\_channel)              )      def forward(self,x):          out=self.left(x)          out+=self.shortcut(x)          out=F.relu(out)          return out    class ResNet34(nn.Module):      def \_\_init\_\_(self,ResBlock,numclasses=10):          super(ResNet34,self).\_\_init\_\_()          self.conv1=nn.Sequential(              nn.Conv2d(in\_channels=1,out\_channels=64,kernel\_size=3,stride=1,padding=1,bias=False),              nn.BatchNorm2d(64),              nn.ReLU()          )          self.layer1 = nn.Sequential(ResBlock(64, 64, stride=1))          self.layer2 = nn.Sequential(ResBlock(64, 64, stride=1))          self.layer3 = nn.Sequential(ResBlock(64, 64, stride=1))          self.layer4 = nn.Sequential(ResBlock(64, 128, stride=2))          self.layer5 = nn.Sequential(ResBlock(128, 128, stride=1))          self.layer6 = nn.Sequential(ResBlock(128, 128, stride=1))          self.layer7 = nn.Sequential(ResBlock(128, 128, stride=1))          self.layer8 = nn.Sequential(ResBlock(128, 256, stride=2))          self.layer9 = nn.Sequential(ResBlock(256, 256, stride=1))          self.layer10 = nn.Sequential(ResBlock(256, 256, stride=1))          self.layer11 = nn.Sequential(ResBlock(256, 256, stride=1))          self.layer12 = nn.Sequential(ResBlock(256, 256, stride=1))          self.layer13 = nn.Sequential(ResBlock(256, 256, stride=1))          self.layer14 = nn.Sequential(ResBlock(256, 512, stride=2))          self.layer15 = nn.Sequential(ResBlock(512, 512, stride=1))          self.layer16 = nn.Sequential(ResBlock(512, 512, stride=1))          self.fc=nn.Linear(512,numclasses)      def forward(self,x):          out=self.conv1(x)          out=self.layer1(out)          out=self.layer2(out)          out=self.layer3(out)          out=self.layer4(out)          out=self.layer5(out)          out=self.layer6(out)          out=self.layer7(out)          out=self.layer8(out)          out=self.layer9(out)          out=self.layer10(out)          out=self.layer11(out)          out=self.layer12(out)          out=self.layer13(out)          out=self.layer14(out)          out=self.layer15(out)          out=self.layer16(out)          out=F.avg\_pool2d(out,4)          out=out.view(out.size(0),-1)          out=self.fc(out)          return out |

# 3 训练结果及对比分析

## 3.0 训练数据组成

每个模型分别使用10个epoch进行训练。，每个epoch里使用enumerate(trainload,start=0)加载训练数据，训练数据样本总量为600000。



**Fig. 4.** 部分训练数据可视化

## 3.1 训练结果评价指标

在本次分类任务中，使用Weight-average方法作为评价指标，即对每类的评价指标进行加权求和,求得整个的评价指标

1. 精度（Accuracy）

Precision = TP / (TP + FP)

1. 召回率（Recall）

Recall = TP / (TP + FN)

1. 准确率（Precision）

Accuracy = (TP + TN) / (TP + FN + FP + TN)

1. F1-score

F1 = (2×Precision×Recall) / (Precision+Recall）

1. 误差

loss=loss\_function(outputs,labels)，实际计算时，每一个批次训练后取loss取平均

1. 训练时间，执行时间

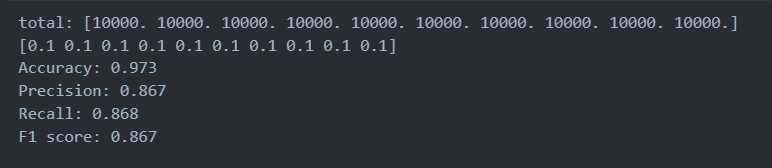
由于使用notebook进行编写，可以直接查看

|  |
| --- |
| # 测试模型  net.eval()  correct = 0  total = 0  TN,TP,FN,FP=np.zeros(10),np.zeros(10),np.zeros(10),np.zeros(10)  # 不进行梯度计算，加快预测速度  with torch.no\_grad():      for data in testload:          images, labels = data          #print(images.shape)          inputs = images          inputs,labels=inputs.to(device),labels.to(device)          outputs = net(inputs)          #获取预测的类别          \_, predicted = torch.max(outputs.data, 1)            '''          Weight-average:对每类的评价指标进行加权求和,求得整个的评价指标          计算每个类别的准确率、召回率、F1值          然后对每个类别的评价指标进行加权求和，求得整个的评价指标          TP:预测为正例的样本中，实际为正例的样本数          FN:预测为负例的样本中,实际为正例的样本数          FP:预测为正例的样本中,实际为负例的样本数          TN:预测为负例的样本中,实际为负例的样本数          '''          TN += labels.shape[0];          for i in range(predicted.shape[0]):              #预测成功(预测为正例且实际为正例，或者预测为负例且实际为负例)              if predicted[i] == labels[i]:                  #对于标签为predicted[i]的类别(TP)和其他类别(TN)来说都预测成功                  TP[predicted[i]]+=1                  TN[predicted[i]]-=1              #预测失败（预测为正例但实际为负例，或者预测为负例而实际为正例）              else:                  #对标签为predicted[i]的类别(FP),对于标签为labels[i]的类别(FN)                  FP[labels[i]]+=1                  FN[predicted[i]]+=1                  TN[labels[i]]-=1                  TN[predicted[i]]-=1    # 计算准确率，召回率，F1值  total=TP+FP+FN+TN  print('total:',total)  accuracy = (TP + TN) / total  precision = TP / (TP + FP)  recall = TP / (TP + FN)  f1 = 2 \* precision \* recall / (precision + recall)  weight=total/total.sum()  print(weight)  total\_accuracy=np.sum(accuracy\*weight)  total\_precision=np.sum(precision\*weight)  total\_recall=np.sum(recall\*weight)  total\_f1=np.sum(f1\*weight)  print('Accuracy: %.3f' % total\_accuracy)  print('Precision: %.3f' % total\_precision)  print('Recall: %.3f' % total\_recall)  print('F1 score: %.3f' % total\_f1) |

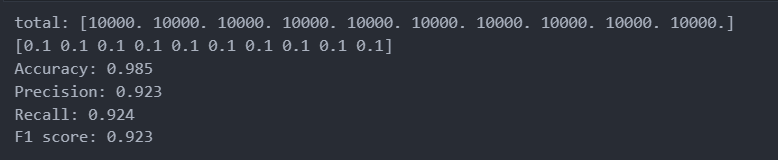
## 3.2 寻找最优参数

经过尝试，三种模型loss函数使用nn.CrossEntropyLoss()，优化器都使用Adam算法，lr=0.001。

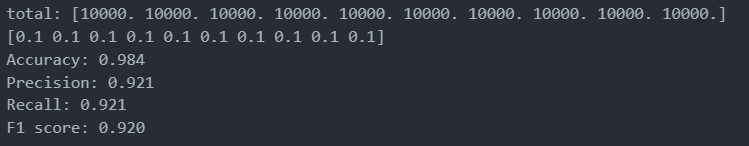
## 3.3 不同模型的对比



**Fig. 5**.Lenet5训练结果



**Fig. 6.**ResNet18 训练结果

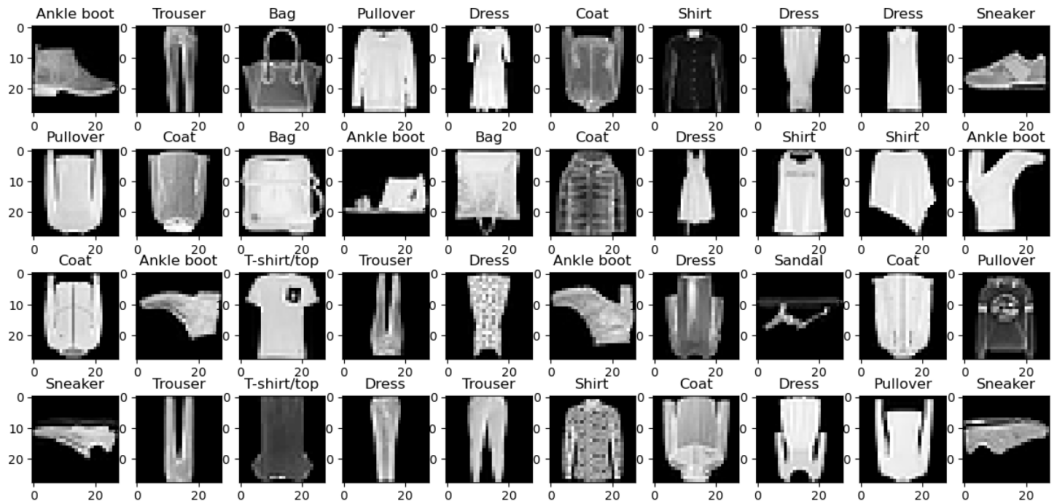


**Fig. 6.**ResNet34 训练结果

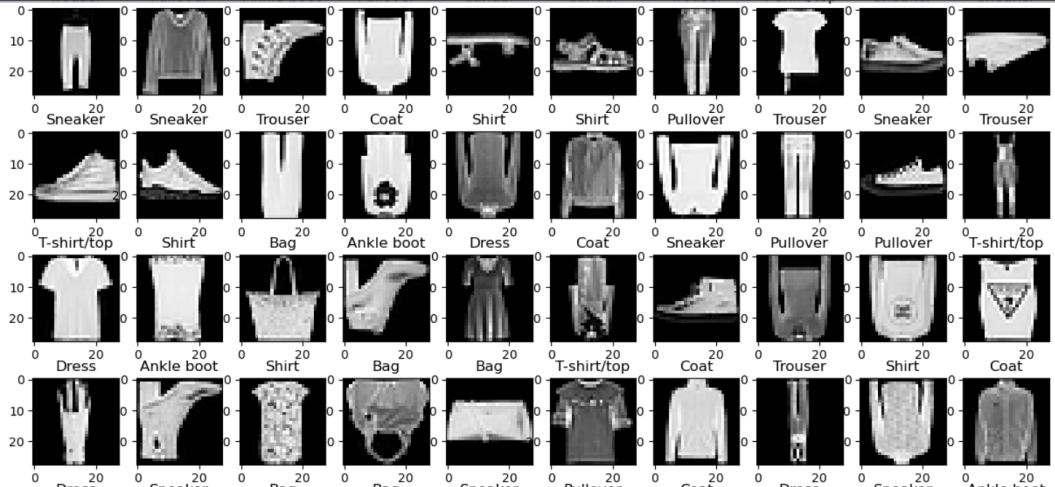
**Table1.** 三种模型评价指标对比（训练三次取平均）

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 算法 | Loss | Accuracy | Precision | Recall | F1-socre | 训练时间 | 执行时间 |
| LeNet5 | 0.337 | 0.973 | 0.867 | 0.868 | 0.867 | 2m25.5s | 4.7s |
| ResNet18 | 0.165 | 0.985 | 0.923 | 0.924 | 0.923 | 10m0.6s | 5.7s |
| ResNet34 | 0.197 | 0.984 | 0.921 | 0.921 | 0.920 | 17m4.7s | 7.9s |

通过对比可以发现，模型越复杂，训练和执行的时间越长。对于ResNet来说，ResNet18和ResNet34的训练结果怕相差无几（但ResNet34的loss比ResNet18大了20%），这可能是因为ResNet34还训练的epoch较少，还未达到最佳水平。理论上来说，模型越复杂，能够达到的效果就越好，但同时需要训练的数据越多，需要训练的时间越长。

****

**Fig.7.**LeNet5模型部分预测结果可视化



**Fig.8.**ResNet18模型部分预测结果可视化

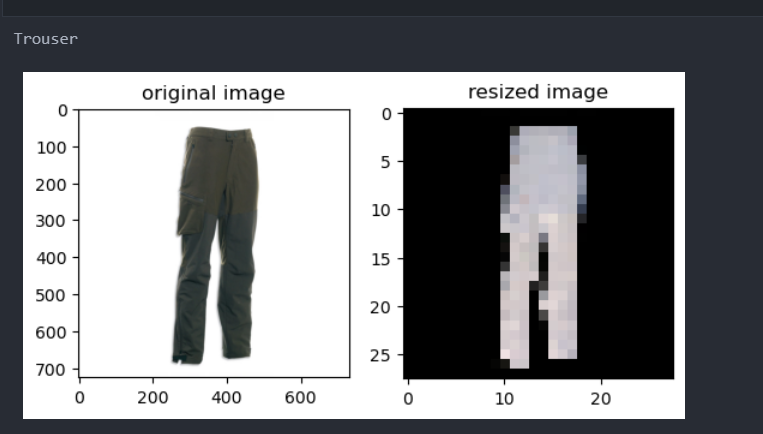
## 3.4 分类真实图片

可以使用模型来进行真实图片的预测，下面以LeNet5为例。

首先需要在训练时保存训练好的参数。

然后定义分类器，加载模型参数，就可以使用模型进行计算，计算时需要注意输入的维度是4维，需要对输入的图片添加一个维度。

|  |
| --- |
| #保存模型参数  save\_path = './ckpt/Lenet1.pth'  torch.save(net.state\_dict(), save\_path) |
| import cv2  classes = ('T-shirt/top', 'Trouser', 'Pullover', 'Dress','Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot')  #加载模型  path = './ckpt/Lenet1.pth'  model = LeNet()  model.load\_state\_dict(torch.load(path))  #预测图片  transform = transforms.Compose([transforms.ToTensor(),                                  transforms.Normalize((0.5,),(0.5,)),                                  transforms.Resize((28,28)),                                  #转化为灰度图                                  transforms.Grayscale(num\_output\_channels=1)                                  ])  im = cv2.imread('./source/trouser2.jpg')  im = cv2.cvtColor(im, cv2.COLOR\_BGR2RGB)  #预处理图片  resized\_image = cv2.resize(im, (28, 28))  resized\_image = 255 - resized\_image  input = transform(im)  #增加一个维度  input = input.unsqueeze(0)  # 把im转化为tensor  input = input.clone().detach().requires\_grad\_(True)  plt.subplot(1,2,1)  plt.imshow(im)  plt.title('original image')  plt.subplot(1,2,2)  plt.imshow(resized\_image)  plt.title('resized image')  with torch.no\_grad():      outputs = model(input)      \_, predict = torch.max(outputs.data, 1) # 索引即classed中的类别  print(classes[predict]) |

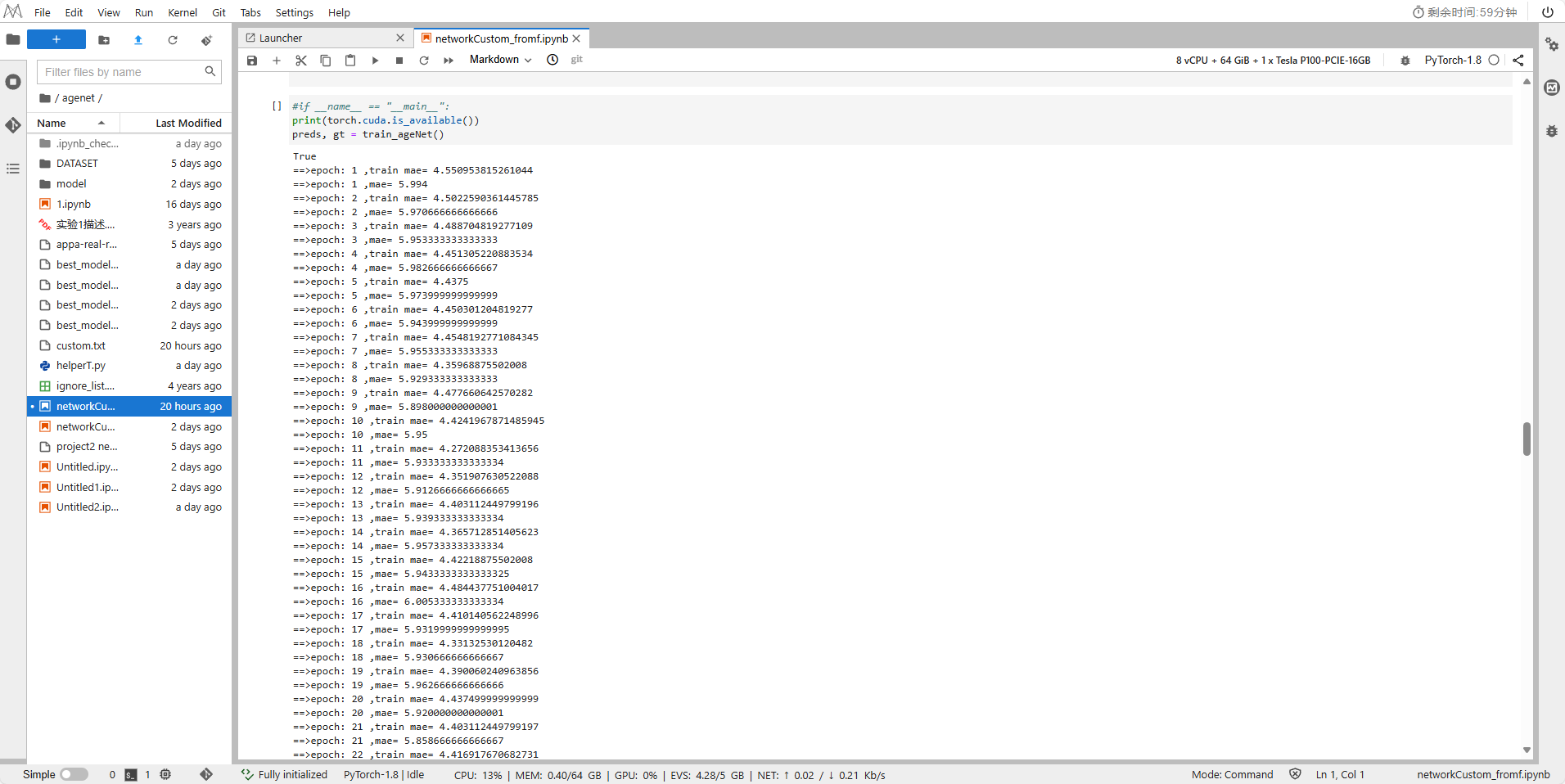


**Fig. 9.**Lenet5预测真实图片

# 4 总结

深度学习是当今机器学习非常热门的一个研究方向，使用深度学习可以解决很多棘手的问题。但是，模型本身具体如何运作，仍然是一个黑盒子，我们只能猜测模型可能通过卷积来识别输入的局部特征等操作，这些中间步骤当前都是无法观测的。

本次实验中对每个模型仅训练了10个epoch，实际在训练数据集时可能使用更多的epoch。Fashion-Mnist是一个输入尺寸较小的训练集合，所以本次实验使用3050laptop显卡也能较快的跑玩10个epoch。对于深度学习，也可以使用华为云等云服务器，上面可以使用Tesla显卡，训练速度更快。



**Fig. 10.**使用华为云训练模型

References

1.[机器学习——二分类、多分类的精确率和召回率\_多分类的准确率和召回率-CSDN博客](https://blog.csdn.net/Alex_81D/article/details/115233536?ops_request_misc=%7B%22request%5Fid%22%3A%22171465328016800225571707%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=171465328016800225571707&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-2-115233536-null-null.142^v100^control&utm_term=%E5%A4%9A%E5%88%86%E7%B1%BB%E9%97%AE%E9%A2%98%E7%B2%BE%E5%BA%A6%E5%8F%AC%E5%9B%9E%E7%8E%87%E8%AF%AF%E5%B7%AE&spm=1018.2226.3001.4187)2.[详解深度学习之经典网络架构（一）：LeNet\_lenet的网络结构-CSDN博客](https://blog.csdn.net/chenyuping333/article/details/82177677)

3.LeCun, Yann. "Generalization and network design strategies." Connectionism in perspective 19.143-155 (1989): 18.

4.[Pytorch模型训练&保存/加载（搭建完整流程）\_pytorch训练模型-CSDN博客](https://blog.csdn.net/FUTEROX/article/details/122724634)

5.He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.