# 实验：CNN 的简单应用

## 一、 实验内容

（1）编程实现一个包含卷积层（包括激活函数操作）、池化层、全连接层 在内的 5 层简单 CNN，并在 MNIST 数据集进行训练，实现手写体识别功能。 （2）模拟实现 VGG16，对预训练模型进行微调，并在 Fashion-MNIST 数 据集进行训练。

## 二、 实验目标

（1）理解区域卷积神经网络的网络原理。

（2）了解 PyTorch 深度学习环境搭建。

（3）自编程实现一个简单 CNN，对数据集进行训练和测试。

（4）了解如何利用预训练模型加快训练速度，学习如何对现有网络进行微 调。

## 三、 实验内容

#### 3.1简单CNN实验

##### 3.1.1.数据集准备

使用torchvision.datasets 下载数据集

|  |
| --- |
| # 1 prepare dataset batch\_size = 64 # ToTensor 转换图片为张量 # Normalized an tensor image with mean and standard deviation transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]) train\_dataset = datasets.MNIST(root='./data/mnist/', train=True, download=True, transform=transform) train\_loader = DataLoader(train\_dataset, shuffle=True, batch\_size=batch\_size) test\_dataset = datasets.MNIST(root='./data/mnist/', train=False, download=True, transform=transform) test\_loader = DataLoader(test\_dataset, shuffle=False, batch\_size=batch\_size) |

##### 3.1.2网络定义

定义网络结构为

卷积-池化-卷积-池化-全连接

|  |
| --- |
| model = nn.Sequential(  # 卷积1操作  # input: channel=1, out\_channel=10, kernel\_size5\*5;  # output: h\*w=(28-5+0+1)/1\*(28-5+0+1)/1=24\*24  nn.Conv2d(1, 10, kernel\_size=5),  nn.MaxPool2d(2),  nn.ReLU(),  # 卷积2操作 接收池化后的conv1 input(h\*w)=h\*w/2=24/2=12  # input: channel=10, out\_channel=20, kernel\_size5\*5;  # output: h\*w=(12-5+0+1)/1\*(12-5+0+1)/1=8\*8  nn.Conv2d(10, 20, kernel\_size=5),  nn.MaxPool2d(2),  nn.ReLU(),  nn.Flatten(),  # 全连接 接收池化后的conv2 input(h\*w)=h\*w/2=8/2=4  # input: 展开后为20\*(4\*4)=320  # output: 10 手写字10个分类  nn.Linear(320, 10)) |

##### 3.1.3定义损失函数/优化器

采用交叉熵损失函数

Adam优化器

|  |
| --- |
| criterion = torch.nn.CrossEntropyLoss() optimizer = optim.Adam(model.parameters(), lr=0.0001) |

##### 3.1.4定义训练验证方法

采用每300迭代进行一次loss打印以及模型的accuracy验证

|  |
| --- |
| def train(epoch):  running\_loss = 0.0  for step, data in enumerate(train\_loader, 0):  inputs, target = data  optimizer.zero\_grad()  outputs = model(inputs)  loss = criterion(outputs, target)  loss.backward()  optimizer.step()  running\_loss += loss.item()  if step % 300 == 299:  correct = 0  total = 0  with torch.no\_grad():  for l\_data in test\_loader:  images, labels = l\_data  l\_outputs = model(images)  \_, predicted = torch.max(l\_outputs.data, dim=1)  total += labels.size(0)  correct += (predicted == labels).sum().item()  print('Epoch: %d |train loss: %.3f |accuracy: %d %%' % (epoch + 1, running\_loss / 300, 100 \* correct / total))  running\_loss = 0.0 |

##### 3.1.5定义执行函数

定义range为10的测试函数，并保存模型

|  |
| --- |
| def jn\_cnn():  for epoch in range(10):  train(epoch)  torch.save(model, './model/jn\_cnn')  jn\_cnn() |

##### 3.1.6执行效果

从执行结果可以看到，准确率从86%优化到98%

|  |
| --- |
| Epoch: 1 |train loss: 1.485 |accuracy: 86 %  Epoch: 1 |train loss: 0.459 |accuracy: 91 %  Epoch: 1 |train loss: 0.309 |accuracy: 93 %  Epoch: 2 |train loss: 0.242 |accuracy: 94 %  Epoch: 2 |train loss: 0.209 |accuracy: 94 %  Epoch: 2 |train loss: 0.184 |accuracy: 95 %  Epoch: 3 |train loss: 0.166 |accuracy: 95 %  Epoch: 3 |train loss: 0.153 |accuracy: 96 %  Epoch: 3 |train loss: 0.135 |accuracy: 96 %  Epoch: 4 |train loss: 0.135 |accuracy: 96 %  Epoch: 4 |train loss: 0.112 |accuracy: 96 %  Epoch: 4 |train loss: 0.118 |accuracy: 96 %  Epoch: 5 |train loss: 0.110 |accuracy: 96 %  Epoch: 5 |train loss: 0.107 |accuracy: 97 %  Epoch: 5 |train loss: 0.098 |accuracy: 97 %  Epoch: 6 |train loss: 0.096 |accuracy: 97 %  Epoch: 6 |train loss: 0.090 |accuracy: 97 %  Epoch: 6 |train loss: 0.093 |accuracy: 97 %  Epoch: 7 |train loss: 0.086 |accuracy: 97 %  Epoch: 7 |train loss: 0.091 |accuracy: 97 %  Epoch: 7 |train loss: 0.075 |accuracy: 97 %  Epoch: 8 |train loss: 0.082 |accuracy: 97 %  Epoch: 8 |train loss: 0.077 |accuracy: 97 %  Epoch: 8 |train loss: 0.073 |accuracy: 97 %  Epoch: 9 |train loss: 0.072 |accuracy: 97 %  Epoch: 9 |train loss: 0.073 |accuracy: 98 %  Epoch: 9 |train loss: 0.072 |accuracy: 97 %  Epoch: 10 |train loss: 0.072 |accuracy: 98 %  Epoch: 10 |train loss: 0.066 |accuracy: 98 %  Epoch: 10 |train loss: 0.064 |accuracy: 98 % |

#### 3.2VGG16微调实验

##### 3.2.1.了解为何使用预训练模型

使用预训练模型作为初始化，有2个好处

1）加速训练，可以使用更少的训练epoch；

2）好的预训练模型可以避免陷入局部最优点或鞍点。

结论：相对于随机值，更具有实际意义。通过这种方式能够使模型更快的收敛

##### 3.2.2数据集准备

使用torchvision.datasets 下载数据集

|  |
| --- |
| from \_\_future\_\_ import print\_function import torch import torch.nn as nn import torchvision import torchvision.transforms as transforms from torch import optim from torch.utils.data import DataLoader import datetime   device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu") print("device: ", device) # batchSize = 4  ##load data transform = transforms.Compose([transforms.Resize(224), transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])  trainset = torchvision.datasets.FashionMNIST(root='./data/fashion/', train=True, download=True, transform=transform) trainloader = torch.utils.data.DataLoader(trainset, batch\_size=batchSize, shuffle=True, num\_workers=0)  testset = torchvision.datasets.FashionMNIST(root='./data/fashion/', train=False, download=True, transform=transform) testloader = torch.utils.data.DataLoader(testset, batch\_size=batchSize, shuffle=False, num\_workers=0) |

##### 3.2.3导入预训练模型

|  |
| --- |
| # 导入预训练模型 vgg16 = torchvision.models.vgg16(pretrained=True) # 打印vgg16结构 print(vgg16) |

##### 3.2.4网络定义

FASHION数据为1channel，而VGG16第一层为3channel，所以对第一层进行修改

self.features[0] = nn.Conv2d(1, 64, 3, 1, 1)

此外重新修改新的全连接层，使用输出1024为例

|  |
| --- |
| class JnVgg16(nn.Module):  def \_\_init\_\_(self):  super(JnVgg16, self).\_\_init\_\_()   # 预训练vgg16的特征提取层  self.features = vgg16.features  self.features[0] = nn.Conv2d(1, 64, 3, 1, 1)   self.avgpool = vgg16.avgpool  # 添加新的全连接层  self.classifier = nn.Sequential(  nn.Linear(25088, 1024),  nn.ReLU(),  nn.Dropout(p=0.5), # 防止过拟合  nn.Linear(1024, 1024),  nn.ReLU(),  nn.Dropout(p=0.5),  nn.Linear(1024, 10)  )   # 定义前向传播路径  def forward(self, x):  x = self.features(x)  x = self.avgpool(x)  # x = x.view(-1, 25088)  x = x.view(x.size(0), -1)  x = self.classifier(x)  return x |

##### 3.2.5定义损失函数/优化器

采用交叉熵损失函数

SGD优化器 lr为0.01 使用gpu运行

|  |
| --- |
| jn\_vgg = JnVgg16() jn\_vgg.to(device) # net = vgg16 print(jn\_vgg) criterion = nn.CrossEntropyLoss() # optimizer = optim.SGD(jn\_vgg.parameters(), lr=0.05, momentum=0.9) optimizer = optim.SGD(jn\_vgg.parameters(), lr=0.01) |

##### 3.2.6定义训练函数

采用每1000迭代进行一次loss打印以及模型的accuracy验证

|  |
| --- |
| def train(epoch):  running\_loss = 0.0  start = datetime.datetime.now()  total\_step = len(trainloader)  for step, data in enumerate(trainloader, 0):  inputs, target = data   optimizer.zero\_grad()  if 'cpu' != device.type:  inputs = inputs.cuda()  target = target.cuda()  outputs = jn\_vgg(inputs)  loss = criterion(outputs, target)  loss.backward()  optimizer.step()  running\_loss += loss.item()  if step % 1000 == 999:  end = datetime.datetime.now()   interval = end - start   print('Epoch: %d |step: %d / %d |train loss: %.3f |cost time: %d s' % (epoch + 1, step, total\_step, running\_loss / 1000, interval.seconds))  running\_loss = 0.0  start = datetime.datetime.now() |

##### 3.2.7定义测试函数

|  |
| --- |
| def test(epoch):  model = jn\_vgg  model.to(device)  correct = 0  total = 0  with torch.no\_grad():  for l\_data in testloader:  images, labels = l\_data  if 'cpu' != device.type:  images = images.cuda()  labels = labels.cuda()  l\_outputs = model(images)  \_, predicted = torch.max(l\_outputs.data, dim=1)  total += labels.size(0)  correct += (predicted == labels).sum().item()  print('Epoch: %d |accuracy: %d %% ' % (epoch + 1, 100 \* correct / total)) |

##### 3.2.8定义执行函数

采用epoch 5

|  |
| --- |
| def jn\_vgg16\_run():  for epoch in range(5):  train(epoch)  test(epoch)   jn\_vgg16\_run() |

##### 3.2.9执行效果

从执行结果可以看到，准确率从86%优化到98%，采用GPU运行大大加快了执行效率

|  |
| --- |
| device: cuda:0  VGG(  (features): Sequential(  (0): Conv2d(3, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (1): ReLU(inplace=True)  (2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (3): ReLU(inplace=True)  (4): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (5): Conv2d(64, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (6): ReLU(inplace=True)  (7): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (8): ReLU(inplace=True)  (9): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (10): Conv2d(128, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (11): ReLU(inplace=True)  (12): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (13): ReLU(inplace=True)  (14): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (15): ReLU(inplace=True)  (16): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (17): Conv2d(256, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (18): ReLU(inplace=True)  (19): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (20): ReLU(inplace=True)  (21): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (22): ReLU(inplace=True)  (23): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (24): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (25): ReLU(inplace=True)  (26): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (27): ReLU(inplace=True)  (28): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (29): ReLU(inplace=True)  (30): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  )  (avgpool): AdaptiveAvgPool2d(output\_size=(7, 7))  (classifier): Sequential(  (0): Linear(in\_features=25088, out\_features=4096, bias=True)  (1): ReLU(inplace=True)  (2): Dropout(p=0.5, inplace=False)  (3): Linear(in\_features=4096, out\_features=4096, bias=True)  (4): ReLU(inplace=True)  (5): Dropout(p=0.5, inplace=False)  (6): Linear(in\_features=4096, out\_features=1000, bias=True)  )  )  JnVgg16(  (features): Sequential(  (0): Conv2d(1, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (1): ReLU(inplace=True)  (2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (3): ReLU(inplace=True)  (4): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (5): Conv2d(64, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (6): ReLU(inplace=True)  (7): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (8): ReLU(inplace=True)  (9): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (10): Conv2d(128, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (11): ReLU(inplace=True)  (12): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (13): ReLU(inplace=True)  (14): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (15): ReLU(inplace=True)  (16): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (17): Conv2d(256, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (18): ReLU(inplace=True)  (19): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (20): ReLU(inplace=True)  (21): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (22): ReLU(inplace=True)  (23): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  (24): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (25): ReLU(inplace=True)  (26): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (27): ReLU(inplace=True)  (28): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))  (29): ReLU(inplace=True)  (30): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)  )  (avgpool): AdaptiveAvgPool2d(output\_size=(7, 7))  (classifier): Sequential(  (0): Linear(in\_features=25088, out\_features=1024, bias=True)  (1): ReLU()  (2): Dropout(p=0.5, inplace=False)  (3): Linear(in\_features=1024, out\_features=1024, bias=True)  (4): ReLU()  (5): Dropout(p=0.5, inplace=False)  (6): Linear(in\_features=1024, out\_features=10, bias=True)  )  )  Epoch: 1 |step: 999 / 15000 |train loss: 0.972 |cost time: 41 s  Epoch: 1 |step: 1999 / 15000 |train loss: 0.554 |cost time: 41 s  Epoch: 1 |step: 2999 / 15000 |train loss: 0.452 |cost time: 40 s  Epoch: 1 |step: 3999 / 15000 |train loss: 0.392 |cost time: 41 s  Epoch: 1 |step: 4999 / 15000 |train loss: 0.368 |cost time: 40 s  Epoch: 1 |step: 5999 / 15000 |train loss: 0.351 |cost time: 41 s  Epoch: 1 |step: 6999 / 15000 |train loss: 0.331 |cost time: 40 s  Epoch: 1 |step: 7999 / 15000 |train loss: 0.334 |cost time: 40 s  Epoch: 1 |step: 8999 / 15000 |train loss: 0.310 |cost time: 40 s  Epoch: 1 |step: 9999 / 15000 |train loss: 0.288 |cost time: 40 s  Epoch: 1 |step: 10999 / 15000 |train loss: 0.289 |cost time: 40 s  Epoch: 1 |step: 11999 / 15000 |train loss: 0.273 |cost time: 40 s  Epoch: 1 |step: 12999 / 15000 |train loss: 0.254 |cost time: 40 s  Epoch: 1 |step: 13999 / 15000 |train loss: 0.249 |cost time: 40 s  Epoch: 1 |step: 14999 / 15000 |train loss: 0.249 |cost time: 40 s  Epoch: 1 |accuracy: 89 %  Epoch: 2 |step: 999 / 15000 |train loss: 0.232 |cost time: 40 s  Epoch: 2 |step: 1999 / 15000 |train loss: 0.235 |cost time: 40 s  Epoch: 2 |step: 2999 / 15000 |train loss: 0.214 |cost time: 40 s  Epoch: 2 |step: 3999 / 15000 |train loss: 0.238 |cost time: 40 s  Epoch: 2 |step: 4999 / 15000 |train loss: 0.238 |cost time: 40 s  Epoch: 2 |step: 5999 / 15000 |train loss: 0.225 |cost time: 40 s  Epoch: 2 |step: 6999 / 15000 |train loss: 0.220 |cost time: 40 s  Epoch: 2 |step: 7999 / 15000 |train loss: 0.227 |cost time: 40 s  Epoch: 2 |step: 8999 / 15000 |train loss: 0.223 |cost time: 40 s  Epoch: 2 |step: 9999 / 15000 |train loss: 0.219 |cost time: 40 s  Epoch: 2 |step: 10999 / 15000 |train loss: 0.238 |cost time: 40 s  Epoch: 2 |step: 11999 / 15000 |train loss: 0.207 |cost time: 40 s  Epoch: 2 |step: 12999 / 15000 |train loss: 0.207 |cost time: 40 s  Epoch: 2 |step: 13999 / 15000 |train loss: 0.219 |cost time: 40 s  Epoch: 2 |step: 14999 / 15000 |train loss: 0.194 |cost time: 40 s  Epoch: 2 |accuracy: 92 %  Epoch: 3 |step: 999 / 15000 |train loss: 0.179 |cost time: 40 s  Epoch: 3 |step: 1999 / 15000 |train loss: 0.163 |cost time: 40 s  Epoch: 3 |step: 2999 / 15000 |train loss: 0.189 |cost time: 40 s  Epoch: 3 |step: 3999 / 15000 |train loss: 0.165 |cost time: 40 s  Epoch: 3 |step: 4999 / 15000 |train loss: 0.176 |cost time: 40 s  Epoch: 3 |step: 5999 / 15000 |train loss: 0.167 |cost time: 40 s  Epoch: 3 |step: 6999 / 15000 |train loss: 0.175 |cost time: 40 s  Epoch: 3 |step: 7999 / 15000 |train loss: 0.180 |cost time: 40 s  Epoch: 3 |step: 8999 / 15000 |train loss: 0.197 |cost time: 40 s  Epoch: 3 |step: 9999 / 15000 |train loss: 0.173 |cost time: 40 s  Epoch: 3 |step: 10999 / 15000 |train loss: 0.189 |cost time: 40 s  Epoch: 3 |step: 11999 / 15000 |train loss: 0.187 |cost time: 40 s  Epoch: 3 |step: 12999 / 15000 |train loss: 0.189 |cost time: 40 s  Epoch: 3 |step: 13999 / 15000 |train loss: 0.179 |cost time: 40 s  Epoch: 3 |step: 14999 / 15000 |train loss: 0.165 |cost time: 40 s  Epoch: 3 |accuracy: 92 %  Epoch: 4 |step: 999 / 15000 |train loss: 0.142 |cost time: 40 s  Epoch: 4 |step: 1999 / 15000 |train loss: 0.145 |cost time: 40 s  Epoch: 4 |step: 2999 / 15000 |train loss: 0.154 |cost time: 40 s  Epoch: 4 |step: 3999 / 15000 |train loss: 0.165 |cost time: 40 s  Epoch: 4 |step: 4999 / 15000 |train loss: 0.158 |cost time: 40 s  Epoch: 4 |step: 5999 / 15000 |train loss: 0.143 |cost time: 40 s  Epoch: 4 |step: 6999 / 15000 |train loss: 0.155 |cost time: 40 s  Epoch: 4 |step: 7999 / 15000 |train loss: 0.141 |cost time: 40 s  Epoch: 4 |step: 8999 / 15000 |train loss: 0.149 |cost time: 40 s  Epoch: 4 |step: 9999 / 15000 |train loss: 0.140 |cost time: 40 s  Epoch: 4 |step: 10999 / 15000 |train loss: 0.150 |cost time: 40 s  Epoch: 4 |step: 11999 / 15000 |train loss: 0.155 |cost time: 40 s  Epoch: 4 |step: 12999 / 15000 |train loss: 0.153 |cost time: 40 s  Epoch: 4 |step: 13999 / 15000 |train loss: 0.160 |cost time: 40 s  Epoch: 4 |step: 14999 / 15000 |train loss: 0.141 |cost time: 40 s  Epoch: 4 |accuracy: 93 %  Epoch: 5 |step: 999 / 15000 |train loss: 0.112 |cost time: 40 s  Epoch: 5 |step: 1999 / 15000 |train loss: 0.118 |cost time: 40 s  Epoch: 5 |step: 2999 / 15000 |train loss: 0.128 |cost time: 40 s  Epoch: 5 |step: 3999 / 15000 |train loss: 0.121 |cost time: 40 s  Epoch: 5 |step: 4999 / 15000 |train loss: 0.128 |cost time: 40 s  Epoch: 5 |step: 5999 / 15000 |train loss: 0.124 |cost time: 40 s  Epoch: 5 |step: 6999 / 15000 |train loss: 0.136 |cost time: 40 s  Epoch: 5 |step: 7999 / 15000 |train loss: 0.146 |cost time: 40 s  Epoch: 5 |step: 8999 / 15000 |train loss: 0.124 |cost time: 40 s  Epoch: 5 |step: 9999 / 15000 |train loss: 0.123 |cost time: 40 s  Epoch: 5 |step: 10999 / 15000 |train loss: 0.114 |cost time: 40 s  Epoch: 5 |step: 11999 / 15000 |train loss: 0.139 |cost time: 40 s  Epoch: 5 |step: 12999 / 15000 |train loss: 0.127 |cost time: 40 s  Epoch: 5 |step: 13999 / 15000 |train loss: 0.135 |cost time: 40 s  Epoch: 5 |step: 14999 / 15000 |train loss: 0.118 |cost time: 40 s  Epoch: 5 |accuracy: 93 % |