实验报告

模型介绍-LeNet-5：

LeNet-5是由Yann LeCun在1998年提出的经典深度卷积神经网络，主要用于解决手写数字识别问题。它是卷积神经网络的开创性工作之一，也是深度学习领域的里程碑之一。

LeNet-5的基本结构包括7层网络结构（不含输入层），其中包括2个卷积层、2个降采样层（池化层）、2个全连接层和输出层。

以下是LeNet-5的各层详细介绍：

1. \*\*输入层\*\*：接收大小为32\*32的手写数字图像。

2. \*\*卷积层C1\*\*：包括6个卷积核，每个卷积核的大小为5\*5，步长为1，填充为0。每个卷积核会产生一个大小为28\*28的特征图。

3. \*\*采样层S2\*\*：采用最大池化操作，每个窗口的大小为2\*2，步长为2。每个池化操作会从4个相邻的特征图中选择最大值，产生一个大小为14\*14的特征图。

4. \*\*卷积层C3\*\*：包括16个卷积核，每个卷积核的大小为5\*5，步长为1，填充为0。每个卷积核会产生一个大小为10\*10的特征图。

5. \*\*采样层S4\*\*：采用最大池化操作，每个窗口的大小为2\*2，步长为2。每个池化操作会从4个相邻的特征图中选择最大值，产生一个大小为5\*5的特征图。

6. \*\*全连接层C5\*\*：将每个大小为5\*5的特征图拉成一个长度为400的向量，并通过一个带有120个神经元的全连接层进行连接。

7. \*\*全连接层F6\*\*：将120个神经元连接到84个神经元。

8. \*\*输出层\*\*：由10个神经元组成，每个神经元对应0-9中的一个数字，并输出最终的分类结果。

LeNet-5的训练过程使用反向传播算法（BP算法），通过最小化误差函数（通常使用交叉熵损失函数）来优化网络的权重和偏置。在实际应用中，通常会对LeNet-5进行一些改进，例如增加网络深度、增加卷积核数量、添加正则化等方法，以进一步提高模型的准确性和泛化能力。

数据集介绍-MNIST：

MNIST（Modified National Institute of Standards and Technology database）是一个大型的手写数字数据库，常用于训练各种图像处理系统。这个数据库也广泛用于机器学习领域的训练和测试。

以下是MNIST的一些主要特点：

1. \*\*数据集大小\*\*：MNIST数据库包含60,000个训练图像和10,000个测试图像。

2. \*\*图像内容\*\*：每个图像都是一个手写数字，从0到9。

3. \*\*图像格式\*\*：所有图像都已经被归一化并居中，以适应28x28像素的边界框，并进行了反锯齿处理，引入了灰度级别。

4. \*\*数据来源\*\*：MNIST是由NIST的原始数据集重新混合而成的。原始的NIST训练数据集来自美国人口普查局的员工，而测试数据集来自美国高中生。

MNIST数据集在1998年由Yann LeCun等人引入，并被视为深度学习领域的"Hello World"项目。由于其数据量小、架构简单，因此非常适合初学者进行实践和学习。

代码结构：

Main Directory

│

├── data\_sets/

│ │

│ └── MNIST/

│ ├── train-images-idx3-ubyte.gz

│ ├── train-labels-idx1-ubyte.gz

│ ├── t10k-images-idx3-ubyte.gz

│ └── t10k-labels-idx1-ubyte.gz

│

├── model.py

├── train.py

├── test.py

└── main.py

|  |
| --- |
| model.py  import torch.nn as nn # -------------step4: 构建网络模型-------------------- class LeNet5(nn.Module):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.conv1 = nn.Conv2d(1, 6, kernel\_size=5, stride=1, padding=2)  self.act1 = nn.Tanh()  self.pool1 = nn.AvgPool2d(kernel\_size=2, stride=2)   self.conv2 = nn.Conv2d(6, 16, kernel\_size=5, stride=1, padding=0)  self.act2 = nn.Tanh()  self.pool2 = nn.AvgPool2d(kernel\_size=2, stride=2)   self.conv3 = nn.Conv2d(16, 120, kernel\_size=5, stride=1, padding=0)  self.act3 = nn.Tanh()   self.flat = nn.Flatten()  self.fc1 = nn.Linear(1 \* 1 \* 120, 84)  self.act4 = nn.Tanh()  self.fc2 = nn.Linear(84, 10)   def forward(self, x):  # input 1x28x28, output 6x28x28  x = self.act1(self.conv1(x))  # input 6x28x28, output 6x14x14  x = self.pool1(x)  # input 6x14x14, output 16x10x10  x = self.act2(self.conv2(x))  # input 16x10x10, output 16x5x5  x = self.pool2(x)  # input 16x5x5, output 120x1x1  x = self.act3(self.conv3(x))  # input 120x1x1, output 84  x = self.act4(self.fc1(self.flat(x)))  # input 84, output 10  x = self.fc2(x)  return x |
| Train.py  import torch import torch.nn.functional as F #----------------step6: 定义训练方法----------------------- def train\_model(my\_model, device, trains\_loader, optimizers, epoches):  # 模型训练  my\_model.train()  for batch\_idx, (data, target) in enumerate(trains\_loader):  # 将data和target部署到DEVICE上去  data, target = data.to(device), target.to(device)  # 将梯度初始化为0  optimizers.zero\_grad()  # 训练所得的结果  output = my\_model(data)  # 计算交叉熵损失  loss = F.cross\_entropy(output, target)  # 反向传播  loss.backward()  # 更新参数  optimizers.step()  # 每100batch\_size打印一次log  if batch\_idx % 1000 == 0:  print("Training Epoch:{} \t Loss:{:.5f}".format(epoches, loss.item())) |
| Test.py  import torch import torch.nn.functional as F # ----------------step7: 定义测试方法------------------------ def test\_model(my\_model, device, test\_loder):  my\_model.eval() # 模型验证  correct = 0.0 # 正确率  test\_loss = 0.0 # 测试损失  with torch.no\_grad(): # 测试时不计算梯度，也不进行反向传播  for data, target in test\_loder:  # 将data和target部署到device上  data, target = data.to(device), target.to(device)  # 测试所得的结果  output = my\_model(data)  # 计算交叉熵损失  test\_loss += F.cross\_entropy(output, target).item()  # 找到概率最大的下标  predict = output.argmax(dim=1)  # predict = torch.max(output, dim=1)  correct += predict.eq(target.view\_as(predict)).sum().item() # 累计正确的值  # 计算平均损失  avg\_loss = test\_loss / len(test\_loder.dataset)  # 计算准确率  correct\_ratio = 100 \* correct / len(test\_loder.dataset)  print("Average\_loss in test:{:.5f}\t Accuracy:{:.5f}\n".format(  avg\_loss, correct\_ratio  )) |
| Main.py  import torch import torch.optim as optim from torch.utils.data import DataLoader from torchvision import datasets, transforms from model import LeNet5 from train import train\_model from test import test\_model  # --------------step1: 定义超参数------------------------- DEVICE = torch.device("cuda" if torch.cuda.is\_available() else "cpu") # 是否用GPU EPOCHS = 10 # 数据集的训练次数 BATCH\_SIZE = 16 # 每批处理的数据 16/32/64/128  # -------------step2: 构建transform（对图像做处理）--------- transform = transforms.Compose([  transforms.ToTensor(), # 将图片转成成tensor  transforms.Normalize((0.1307,), (0.3081,)) # 标准化 => x' = （x-μ）/σ ])   def load\_data():  # -------------step3: 下载并加载数据集------------------  # 下载数据集  train\_set = datasets.MNIST("data\_sets", train=True, download=True, transform=transform)  test\_set = datasets.MNIST("data\_sets", train=False, download=True, transform=transform)  # 加载数据集  train\_loader = DataLoader(train\_set, batch\_size=BATCH\_SIZE, shuffle=True)  test\_loader = DataLoader(test\_set, batch\_size=BATCH\_SIZE, shuffle=True)   return train\_loader, test\_loader   if \_\_name\_\_ == '\_\_main\_\_':  train\_loader, test\_loader = load\_data()  # ----------------step:5 定义优化器--------------------------  model = LeNet5().to(DEVICE)  optimizer = optim.Adam(model.parameters())   # --------------step8: 训练模型----------------------------  for epoch in range(1, EPOCHS + 1):  train\_model(model, DEVICE, train\_loader, optimizer, epoch)  test\_model(model, DEVICE, test\_loader) |
| Result:  Training Epoch:1 Loss:2.29138  Training Epoch:1 Loss:0.05695  Training Epoch:1 Loss:0.00455  Training Epoch:1 Loss:0.14211  Average\_loss in test:0.00401 Accuracy:97.91000  Training Epoch:2 Loss:0.04114  Training Epoch:2 Loss:0.23524  Training Epoch:2 Loss:0.03082  Training Epoch:2 Loss:0.36369  Average\_loss in test:0.00364 Accuracy:98.16000  Training Epoch:3 Loss:0.21357  Training Epoch:3 Loss:0.00141  Training Epoch:3 Loss:0.01133  Training Epoch:3 Loss:0.01068  Average\_loss in test:0.00252 Accuracy:98.77000  Training Epoch:4 Loss:0.00116  Training Epoch:4 Loss:0.00983  Training Epoch:4 Loss:0.27756  Training Epoch:4 Loss:0.02450  Average\_loss in test:0.00258 Accuracy:98.78000  Training Epoch:5 Loss:0.00082  Training Epoch:5 Loss:0.00319  Training Epoch:5 Loss:0.00837  Training Epoch:5 Loss:0.05007  Average\_loss in test:0.00232 Accuracy:98.89000  Training Epoch:6 Loss:0.00040  Training Epoch:6 Loss:0.00046  Training Epoch:6 Loss:0.00333  Training Epoch:6 Loss:0.01039  Average\_loss in test:0.00227 Accuracy:98.81000  Training Epoch:7 Loss:0.00084  Training Epoch:7 Loss:0.06549  Training Epoch:7 Loss:0.00835  Training Epoch:7 Loss:0.11384  Average\_loss in test:0.00217 Accuracy:98.92000  Training Epoch:8 Loss:0.00927  Training Epoch:8 Loss:0.00031  Training Epoch:8 Loss:0.00143  Training Epoch:8 Loss:0.00040  Average\_loss in test:0.00248 Accuracy:98.78000  Training Epoch:9 Loss:0.01932  Training Epoch:9 Loss:0.05961  Training Epoch:9 Loss:0.02729  Training Epoch:9 Loss:0.00036  Average\_loss in test:0.00213 Accuracy:98.90000  Training Epoch:10 Loss:0.00373  Training Epoch:10 Loss:0.04423  Training Epoch:10 Loss:0.36702  Training Epoch:10 Loss:0.02052  Average\_loss in test:0.00216 Accuracy:98.92000 |
| 结果分析：实验结果显示，使用LeNet-5在MNIST数据集上进行训练和测试的效果非常好。从第一轮到第十轮，模型的测试准确率从97.91%提高到了98.92%，这表明模型的性能在逐步提升。  同时，我们也可以看到，每轮训练的平均损失值也在逐步下降，从第一轮的0.00401降低到了第十轮的0.00216，这说明模型在逐步学习并优化。  但是，需要注意的是，虽然模型的准确率在提高，但是在第10轮时，训练损失出现了一次较大的波动（从0.00373上升到0.36702），这可能是由于模型在某些特定的样本上出现了过拟合。为了解决这个问题，可以尝试使用一些正则化技术，如L1、L2正则化或者dropout等。 |
| # optimizer = optim.Adam(model.parameters()) optimizer = torch.optim.SGD(model.parameters(), lr=0.01, weight\_decay=1e-5)  Average\_loss in test:0.00262 Accuracy:98.71000  Training Epoch:9 Loss:0.43823  Training Epoch:9 Loss:0.00249  Training Epoch:9 Loss:0.02240  Training Epoch:9 Loss:0.01226  Average\_loss in test:0.00248 Accuracy:98.65000  Training Epoch:10 Loss:0.00938  Training Epoch:10 Loss:0.02003  Training Epoch:10 Loss:0.00876  Training Epoch:10 Loss:0.01816  Average\_loss in test:0.00223 Accuracy:98.94000 |