《模式识别与机器学习 A》实验报告

实验题目：卷积神经网络实验

班级：2203601

学号：2022113416

姓名：刘子康

实验报告内容

一、实验目的

• 掌握卷积神经网络的基本原理与结构，掌握搭建和训练卷积神经网络的方法；

• 采用任意一种课程中介绍过的或者其它卷积神经网络模型（如 Lenet-5、AlexNet 等）用于解决某种媒体类型的模式识别问题；

• 理解不同激活函数、dropout 比例、数据量和超参数对模型性能的影响。

二、实验内容

• 参照 LeNet-5 模型，基于现有框架 Pytorch 构建一个卷积神经网络，实现数据样本的分类和预测。

• 选择 MNIST 数据集进行训练和测试，该数据集为手写体数字图像标准数据集，包含 60000 个训练样本和 10000 个测试样本，每个样本为单通道 28*28 像素灰度图像。

• 尝试选择不同激活函数，使用 dropout 等技巧，分析实验结果和可能原因。

• 使用不同数据量，不同超参数（如学习率和批次大小），比较实验效果，并给出截图和分析。

三、实验环境

• 操作系统：Windows 11

• 编程语言：Python 3.10

• 第三方库：PyTorch 2.4.0+cu118，torchvision0.19.0，Numpy 1.23.4，Matplotlib 3.8.2

• IDE：Pycharm 2022 社区版

四、实验过程、结果及分析

4.1 实验原理

卷积神经网络（Convolutional Neural Networks, CNN）是一类包含卷积计算且具有深度结构的前馈神经网络，是深度学习的代表算法之一，一般由若干卷积层、激活函数、池化层、全连接层组成。

LeNet-5 是由 Yann LeCun 等人于 1998 年提出的一种经典卷积神经网络架构，它主要用于手写数字识别（如 MNIST 数据集）和英文字母识别。LeNet-5 网络的设计为后续的卷积神经网络架构提供了基础，至今仍然对深度学习领域有着重要的影响。

（1）输入层（Input Layer）

LeNet-5 的输入是一个 32*32 的灰度图像。对于 MNIST 数据集识别任务，原始图像大小为 28*28，故通常会通过边缘零填充（Zero-padding）将原始 28*28 的图像扩展到 32*32，以确保卷积操作后不会损失过多信息。

（2）卷积层（Convolutional Layer）

LeNet-5 包含了多个卷积层，用于提取图像的空间特征。卷积操作通过一个卷积核（滤波器）在图像上的滑动，计算加权和，产生一组特征图（feature maps）。

• 第一卷积层（C1）：输入是经过边缘零填充的原始图像，大小为 32*32。该层使用 6 个大小为 5*5 的卷积核，每个卷积核生成一个特征图。经过卷积操作后，输出的特征图大小为 28*28（32-5+1 = 28），此时图像的尺寸变小，特征被初步提取。

• 第二卷积层（C3）：输入是来自 S2 层的 6 个经池化降维后的特征图，大小为

14*14。该层使用 16 个大小为 5*5 的卷积核，不同的卷积核会对不同的输入特征图进行卷积，提取更高层次的特征。经过卷积操作后，输出特征图的大小为 10*10。

（3）池化层（Subsampling / Pooling Layer）

池化层也称为降采样层，用于降低特征图的维度，减少计算量和过拟合风险。LeNet-5 使用了最大池化（Max Pooling），取对应局部区域的最大值，池化窗口为 2*2，步长为 2。

• 第一池化层（S2）：输入是来自 C1 层的 6 个 28*28 特征图，经过池化操作后，每个特征图的大小变为 14*14。

• 第二池化层（S4）：输入是来自 C3 层的 16 个 10*10 特征图，经过池化操作后，每个特征图的大小变为 5*5。

（4）全连接层（Fully Connected Layer）

全连接层用于将局部特征与分类结果相结合，进行最终的决策。

第一全连接层（C5）：输入是来自 S4 层的经池化降维后的 16 个特征图，大小为 5*5。所有特征图被展平为一个 400 维的向量，并通过该全连接层与 120 个神经元连接。

第二全连接层（F6）：将上一层（C5 层）的 120 个神经元连接到 84 个神经元。

（5）输出层（Output Layer）

输出层用于最终的分类。输出层也是一个全连接层，包含 10 个神经元，每个神经元对应一个数字类别（0~9），对应一个 10 维向量。最后使用 Softmax 激活函数，输出一个概率分布，表示输入图像属于每个类别的概率。

LeNet-5 网络结构总结：

• 输入层：32*32 原始图像

• 卷积层：

  • C1：6 个 5*5 卷积核，输出 28*28 特征图

  • C3：16 个 5*5 卷积核，输出 10*10 特征图

• 池化层：

  • S2：输出 6 个 14*14 特征图

  • S4：输出 16 个 5*5 特征图

• 全连接层：

  • C5：400 个神经元

  • F6：120 个神经元

• 输出层：10 个神经元，用于分类

部分层之间使用激活函数 Sigmoid（目前图像领域使用 ReLU、Tanh 较多）进行非线性化映射，使得神经网络可以拟合各种复杂的非线性关系。

4.2 实验过程

4.2.1 数据集预处理

如图 4-1，使用 Numpy 库的 random.choice 随机选择样本索引，使用 torch.utils.data.dataset.Subset 创建子数据集，以便设置不同的训练集数据量。

```
49      # 随机选择样本
        单元测试 | 注释生成 | 代码解释 | 缺陷检测
50      def create_subset(data, sample_size):
51          indices = np.random.choice(len(data), sample_size, replace=False)  # 随机选择样本索引
52          return Subset(data, indices)
```

图 4-1 随机选取样本

如图 4-2，使用 Torchvision 库的 torchvision.datasets.MNIST 下载 MNIST 数据集，并通过 torchvision.transform 模块进行调整图像大小、转换为 Tensor 格式、正则化（0.1307 和 0.3081 为 MNIST 数据集均值和方差）等预处理，以便符合 LeNet-5 的输入格式，降低模型复杂度，减少过拟合风险。使用 PyTorch 库的 torch.utils.data.DataLoader 作为数据集加载

器，并设置批次大小。

```
    transform = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor(),
        transforms.Normalize((0.1307,),(0.3081,))
    ])

    # 获取MNIST数据集
    sample_sizes = [10000, 30000, 60000]
    train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
    test_dataset = datasets.MNIST(root="./data", train=False, download=True, transform=transform)

    # 加载数据集
    batch_sizes = [64, 128, 256, 512, 1024]
    train_dataloader = DataLoader(dataset=create_subset(train_dataset, sample_sizes[2]), batch_size=batch_sizes[1], shuffle=True)
    test_dataloader = DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)
```

图 4-2 获取和加载 MNIST 数据集

4.2.2 LeNet-5 模型搭建

如图 4-3 所示，定义一个 LeNet5 类，以 torch.nn.Module 作为基类。

（1）初始化：接收参数为激活函数和 dropout 比例，按照 LeNet-5 模型的网络结构依次定义各个层和操作。

（2）前向传播：按照 LeNet-5 模型的网络结构依次执行各个层的操作，穿插激活函数和 dropout 操作。

```
class LeNet5(nn.Module):
    单元测试 | 注释生成 | 代码解释 | 缺陷检测
    def __init__(self, activation=nn.Sigmoid(), p=0.2):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.activation = activation
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.dropout = nn.Dropout(p)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    单元测试 | 注释生成 | 代码解释 | 缺陷检测
    def forward(self, x):
        # CONV1, ReLU1, POOL1
        x = self.conv1(x)
        x = self.maxpool1(x)
        x = self.activation(x)
        # CONV2, ReLU2, POOL2
        x = self.conv2(x)
        x = self.maxpool2(x)
        x = self.activation(x)
        x = x.view(-1, 16 * 5 * 5)
        # FC1
        x = self.fc1(x)
        x = self.activation(x)
        x = self.dropout(x)
        # FC2
        x = self.fc2(x)
        x = self.activation(x)
        x = self.dropout(x)
        # FC3
        x = self.fc3(x)
        output = F.log_softmax(x, dim=1)
        return output
```

图 4-3 LeNet-5 模型

4.2.3 模型训练及测试

创建模型实例，部署 GPU，使用 CUDA 平台加速模型训练。使用 Adam 作为优化器，使用交叉熵损失函数计算 loss。

（1）训练

如图 4-4，分批次进行训练，首先初始化优化器，接着前向传播并保存模型结果，然后计算预测正确个数，最后计算 loss 并反向传播更新参数。每 50 个批次输出一次 loss。

```python
# 训练
单元测试 | 注释生成 | 代码解释 | 缺陷检测
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    correct, loss, train_loss = 0, 0, 0
    sample_size = len(train_loader.dataset) # 训练集样本总数
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)    # 前向传播，保存训练结果
        pred = output.argmax(dim=1)
        correct += pred.eq(target.view_as(pred)).sum().item()

        loss = F.cross_entropy(output, target)  # 交叉熵损失函数
        train_loss += loss.item()
        loss.backward()     # 反向传播
        optimizer.step()    # 更新参数

        if batch_idx % 50 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), sample_size,
                100. * batch_idx / len(train_loader), loss.item()))

    train_loss /= sample_size
    train_acc = correct / sample_size
    return train_loss, train_acc
```

图 4-4 模型训练

（2）测试

如图 4-5，首先设置模型不更新参数，前向传播一次并保存模型结果，然后计算平均 loss 和准确率，并输出。

```python
# 测试
单元测试 | 注释生成 | 代码解释 | 缺陷检测
def test(model, device, test_loader):
    model.eval()
    test_loss, correct = 0, 0
    sample_size = len(test_loader.dataset)  # 测试集样本总数
    # 仅预测结果，不计算梯度和更新参数
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)

            test_loss += F.cross_entropy(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= sample_size
    print('Test set: Average loss: {:.6f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss, correct, sample_size,
        100. * correct / sample_size))
```

图 4-5 模型测试

4.2.4 输出结果并绘图

分别使用不同激活函数、dropout 比例、数据量、学习率、批次大小，绘制训练过程 loss 和准确率变化，输出测试平均 loss 和准确率。

4.3 实验结果及分析

基准参数：Sigmoid 激活函数、droput 比例 0.2、数据量 60000、学习率 1e-3、批次大小 256，仅改变其中某一方面，比较分析实验结果。

4.3.1 不同激活函数

分别设置激活函数为 Sigmoid 函数、ReLU 函数、Tanh 函数，训练过程 loss 和准确率变化如图 4-6 所示，测试结果平均 loss 和准确率如表 4-1 所示，可以看到最佳激活函数为 ReLU 函数。

Sigmoid 函数优化稳定，但指数运算的计算复杂度较高，且在深层网络中易出现梯度消失的问题，因此适用于较为简单的网络结构；Tanh 函数相比 Sigmoid 函数优点在于均值为 0，不会对梯度产生影响，但仍存在梯度饱和与指数计算的问题；ReLU 函数收敛速度更快，计算简单，且不会出现梯度饱和或消失的问题，但可能导致“神经元坏死”。

表 4-1 不同激活函数测试结果

指标 / 激活函数	准确率	平均 loss 值
Sigmoid 函数	98.21%	0.001818
ReLU 函数	99.01%	0.001016
Tanh 函数	98.79%	0.001144

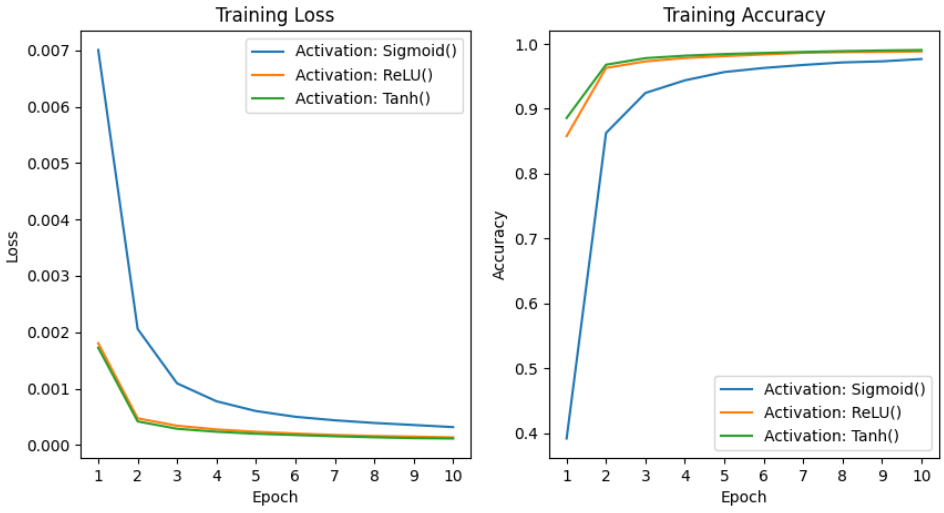

图 4-6 不同激活函数训练过程

4.3.2 不同 dropout 比例

分别设置 dropout 比例为 0、0.1、0.2、0.3、0.4、0.5，训练过程 loss 和准确率变化如图 4-7 所示，测试结果平均 loss 和准确率如表 4-2 所示，可以看到最佳 dropout 比例 0.1。

dropout 技巧主要用于防止模型过拟合，而对应该数据集分类任务，当 dropout 比例逐渐增大时，loss 值逐渐上升，准确率逐渐下降。猜测可能是模型并未出现明显过拟合现象，较高的 dropout 比例反而影响了模型的特征学习。

表 4-2 不同 dropout 比例测试结果

指标 / dropout 比例	准确率	平均 loss 值
0	98.33%	0.001649
0.1	98.40%	0.001649
0.2	98.28%	0.001782

0.3	98.08%	0.001956
0.4	97.90%	0.002137
0.5	97.70%	0.002227

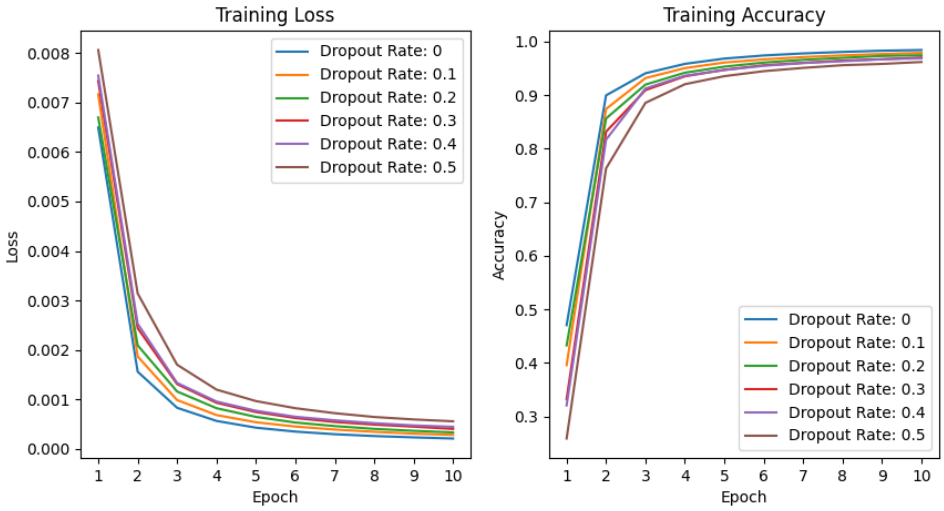

图 4-7 不同 dropout 比例训练过程

4.3.3 不同数据量

分别设置数据量为 10000、30000、60000，训练过程 loss 和准确率变化如图 4-8 所示，测试结果平均 loss 和准确率如表 4-3 所示，可以看到最佳数据量为 60000。

在模型提取特征能力足够的前提下，更多的数据样本可以帮助模型更好地学习数据分布和多样化的特征，减少过拟合的风险，并提高模型的泛化能力。

表 4-3 不同数据量测试结果

指标 / 数据量	准确率	平均 loss 值
10000	93.30%	0.006967
30000	97.09%	0.003013
60000	98.20%	0.001876

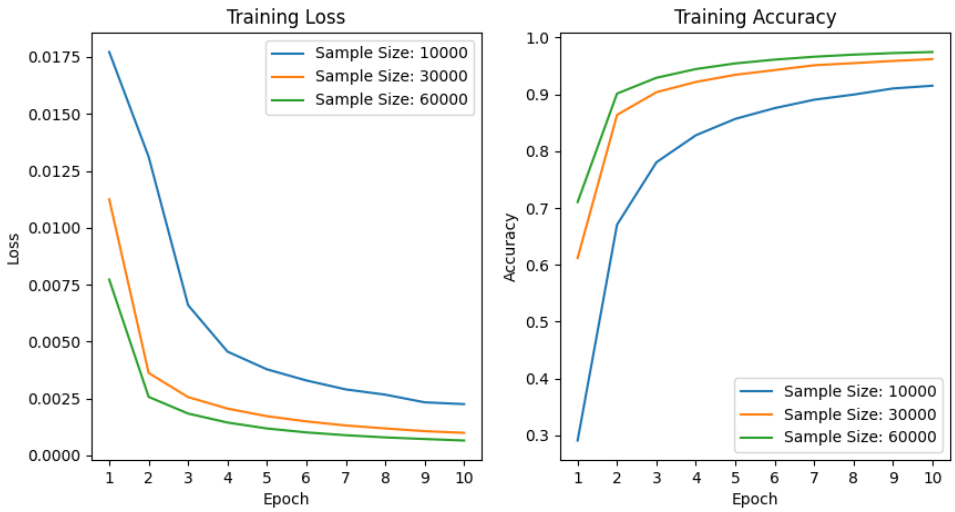

图 4-8 不同数据量训练过程

4.3.4 不同学习率

分别设置学习率为 1e-5、1e-4、1e-3、1e-2、1e-1，训练过程 loss 和准确率变化如图 4-9 所示，测试结果平均 loss 和准确率如表 4-4 所示，可以看到最佳学习率为 1e-2。

当学习率较低时（如 1e-5），模型收敛速度过慢，不能很好地学习到数据特征，故 loss 值很高且不下降，准确率很低且不上升；当学习率为 1e-4 时，随 epoch 增加，loss 值迅速下降，准确率迅速上升，继续增加 epoch 可使模型完全收敛；当学习率大于等于 1e-3 时，模型在第一个 epoch 就已经收敛，之后 loss 值和准确率均保持平稳，但 1e-1 时准确率整体略有下降，可能是学习率过大导致参数更新波动较大，模型无法完全收敛。

表 4-4 不同学习率测试结果

指标 / 学习率	准确率	平均 loss 值
1e-5	11.35%	0.072015
1e-4	86.86%	0.017792
1e-3	98.44%	0.001632
1e-2	98.66%	0.001267
1e-1	93.89%	0.006499

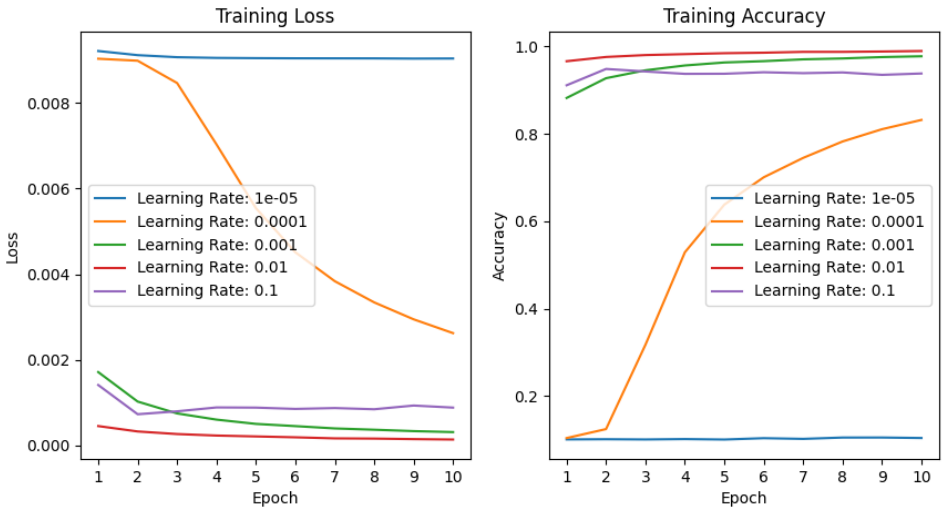

图 4-9 不同学习率训练过程

4.3.5 不同批次大小

分别设置批次大小为 64、128、256、512、1024，训练过程 loss 和准确率变化如图 4-10 所示，测试结果平均 loss 和准确率如表 4-5 所示，可以看到最佳批次大小为 128。

较小的 Batch Size 可以加快每轮训练的速度，更好地拟合复杂的数据分布，提高模型精度，且使得训练更加随机化，有助于跳出局部极小值，从而提高最终模型的泛化能力，但由于每次更新都是基于少量样本，也存在着梯度波动较大，收敛速度变慢的问题。

较大的 Batch Size 可以充分利用现代 GPU 的强大并行计算能力，加速整体训练过程，且可以获得更稳定的梯度估计，优化过程更加直接地朝向全局极值前进，收敛速度更快，但过大可能会导致陷入局部极小值，影响最终的模型性能。

表 4-5 不同批次大小测试结果

指标 / 批次大小	准确率	平均 loss 值
64	98.45%	0.001438

128	98.61%	0.001431
256	97.80%	0.001987
512	97.82%	0.002242
1024	95.04%	0.005335

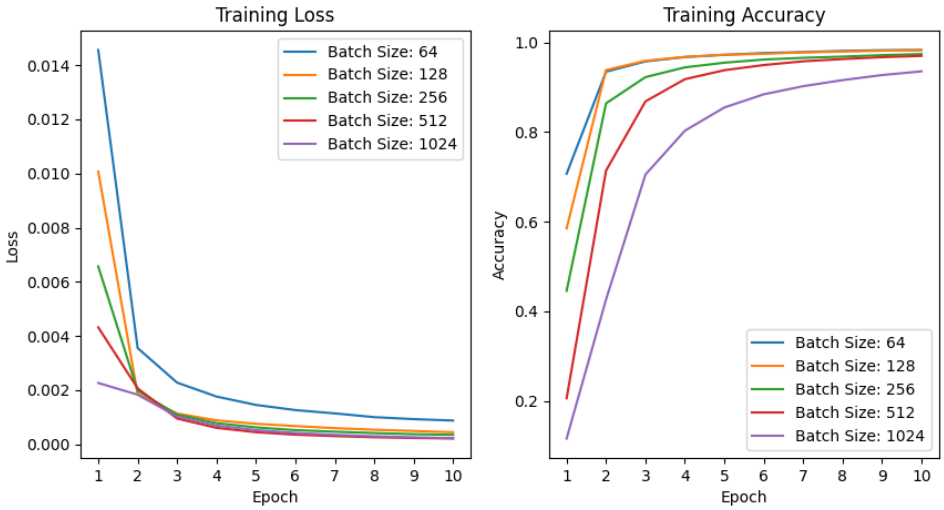

图 4-10 不同批次大小训练过程

五、实验总体结论

此次试验参照 LeNet-5 模型，基于 PyTorch 搭建了一个卷积神经网络，并实现了对 MNIST 数据集的训练和测试，达到了较好的性能和分类效果。并且使用不同激活函数、dropout 比例、数据量和超参数进行训练，分析了其对模型性能的影响。

由实验结果可以得到，当激活函数为 ReLU，dropout 比例为 0.1，数据量为 60000，学习率为 1e-2，批次大小为 128 时，模型达到最佳性能。

卷积神经网络具有局部连接和局部卷积、参数共享、多卷积核、池化处理等特点。

（1）局部连接和局部卷积：卷积神经网络生成的特征图中的每个元素，只需要和输入图像中的局部区域中的部分像素有连接，因而连接的参数数目得以大幅压缩，大大提高了模型训练的效率；

（2）参数共享：卷积操作的参数共享使得提取特征的方式与位置无关，并且大大减少了参数的数量，从而有助于提高卷积计算的效率；

（3）多卷积核：多个卷积核可以充分提取特征，生成多通道图像；

（4）池化处理：池化处理则不仅可以压缩特征数目，降低模型的复杂度，减少过拟合风险，提高分类器泛化能力，还能提供平移和旋转不变性。

作为经典的卷积神经网络，LeNet-5 的提出已经有 26 年，其为之后的 AlexNet、VGGNet、GoogLeNet、ResNet 等诸多模型的网络架构提供了基础，具有深远影响。在 LeNet-5 基础上，通过修改激活函数、加入 dropout 技巧、扩展网络深度、修改卷积模块、跳跃连接等方式，可达到更好的模型性能。

六、完整实验代码

```
1.  import torch
2.  import torch.nn as nn
3.  import torch.nn.functional as F
```

```
from torch.utils.data import DataLoader
from torch.utils.data.dataset import Subset
import torch.optim as optim
from torchvision import transforms, datasets
import matplotlib.pyplot as plt
import numpy as np
import time


class LeNet5(nn.Module):
    def __init__(self, activation=nn.Sigmoid(), p=0.2):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.activation = activation
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.dropout = nn.Dropout(p)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # CONV1, ReLU1, POOL1
        x = self.conv1(x)
        x = self.maxpool1(x)
        x = self.activation(x)
        # CONV2, ReLU2, POOL2
        x = self.conv2(x)
        x = self.maxpool2(x)
        x = self.activation(x)
        x = x.view(-1, 16 * 5 * 5)
        # FC1
        x = self.fc1(x)
        x = self.activation(x)
        x = self.dropout(x)
        # FC2
        x = self.fc2(x)
        x = self.activation(x)
        x = self.dropout(x)
        # FC3
        x = self.fc3(x)
        output = F.log_softmax(x, dim=1)
        return output
```

```
48.
49. # 随机选择样本
50. def create_subset(data, sample_size):
51.     indices = np.random.choice(len(data), sample_size, replace=False)  # 随机选择样本索引
52.     return Subset(data, indices)
53.
54. # 训练
55. def train(model, device, train_loader, optimizer, epoch):
56.     model.train()
57.     correct, loss, train_loss = 0, 0, 0
58.     sample_size = len(train_loader.dataset) # 训练集样本总数
59.     for batch_idx, (data, target) in enumerate(train_loader):
60.         data, target = data.to(device), target.to(device)
61.         optimizer.zero_grad()
62.         output = model(data)    # 前向传播，保存训练结果
63.         pred = output.argmax(dim=1)
64.         correct += pred.eq(target.view_as(pred)).sum().item()
65.
66.         loss = F.cross_entropy(output, target)  # 交叉熵损失函数
67.         train_loss += loss.item()
68.         loss.backward()     # 反向传播
69.         optimizer.step()    # 更新参数
70.
71.         if batch_idx % 50 == 0:
72.             print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
73.                 epoch, batch_idx * len(data), sample_size,
74.                 100. * batch_idx / len(train_loader), loss.item()))
75.
76.     train_loss /= sample_size
77.     train_acc = correct / sample_size
78.     return train_loss, train_acc
79.
80. # 测试
81. def test(model, device, test_loader):
82.     model.eval()
83.     test_loss, correct = 0, 0
84.     sample_size = len(test_loader.dataset)  # 测试集样本总数
85.     # 仅预测结果，不计算梯度和更新参数
86.     with torch.no_grad():
87.         for data, target in test_loader:
88.             data, target = data.to(device), target.to(device)
89.             output = model(data)
90.
91.             test_loss += F.cross_entropy(output, target).item()
```

```
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= sample_size
    print('Test set: Average loss: {:.6f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss, correct, sample_size,
        100. * correct / sample_size))

if __name__ == '__main__':
    totalLoss, totalAccuracy = [], []
    epochs = 10
    transform = transforms.Compose([
        transforms.Resize((32, 32)),
        transforms.ToTensor(),
        transforms.Normalize((0.1307,),(0.3081,))
    ])

    # 获取MNIST数据集
    sample_sizes = [10000, 30000, 60000]
    train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
    test_dataset = datasets.MNIST(root="./data", train=False, download=True, transform=transform)

    # 加载数据集
    batch_sizes = [64, 128, 256, 512, 1024]
    train_dataloader = DataLoader(dataset= create_subset(train_dataset, sample_sizes[2]), batch_size=batch_sizes[1], shuffle=True)
    test_dataloader = DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)

    # 创建模型，部署GPU
    activation = [nn.Sigmoid(), nn.ReLU(), nn.Tanh()]   # 不同激活函数
    dropout_p = [0, 0.1, 0.2, 0.3, 0.4, 0.5]    # 不同dropout比例
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = LeNet5(activation=activation[1], p=dropout_p[1]).to(device)

    # 优化器
    lrs = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]    # 不同学习率
    optimizer = optim.Adam(model.parameters(), lr=lrs[1])

    # 训练
    Loss, Accuracy = [], []
    print(f"Start Time: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())}")
    for epoch in range(1, epochs+1):
```

```
        loss, acc = train(model, device, train_dataloader, optimizer, epoch)
        Loss.append(loss)
        Accuracy.append(acc)
    print(f"End Time: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())}")
    totalLoss.append(Loss)
    totalAccuracy.append(Accuracy)

    # 测试
    test(model, device, test_dataloader)

    # 可视化
    x_ticks = np.arange(1, epochs+1)
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].plot(x_ticks, totalLoss[0])
    axes[0].set_title('Training Loss')
    axes[0].set_xlabel('Epoch')
    axes[0].set_ylabel('Loss')
    axes[0].set_xticks(x_ticks)
    axes[0].legend()
    axes[1].plot(x_ticks, totalAccuracy[0])
    axes[1].set_title('Training Accuracy')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('Accuracy')
    axes[1].set_xticks(x_ticks)
    axes[1].legend()

    plt.show()
```

七、参考文献

[1] 刘远超. 深度学习基础: 高等教育出版社, 2023.