《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。

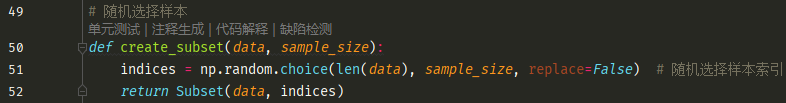


图4-1 随机选取样本

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。



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。

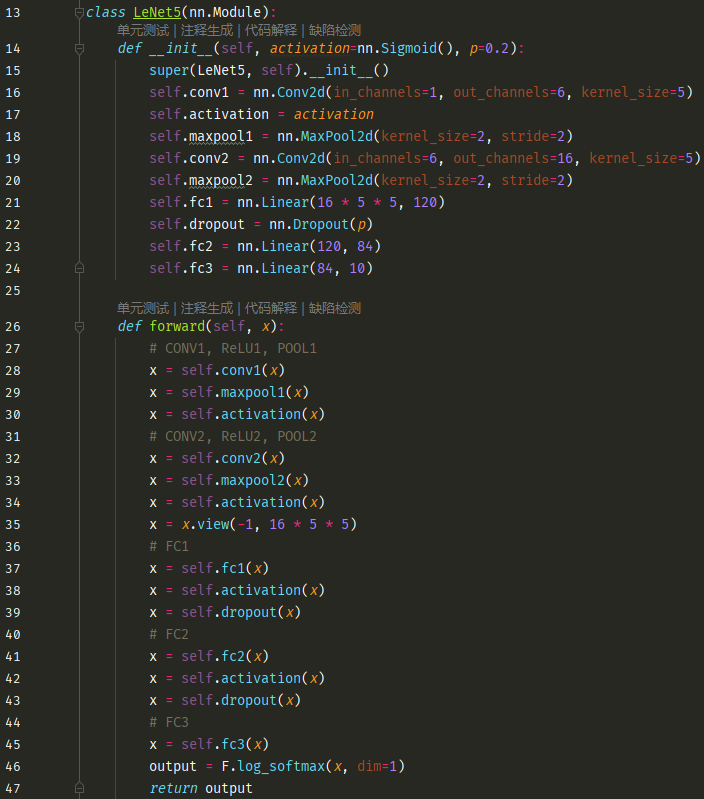


图4-3 LeNet-5模型

**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。



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。

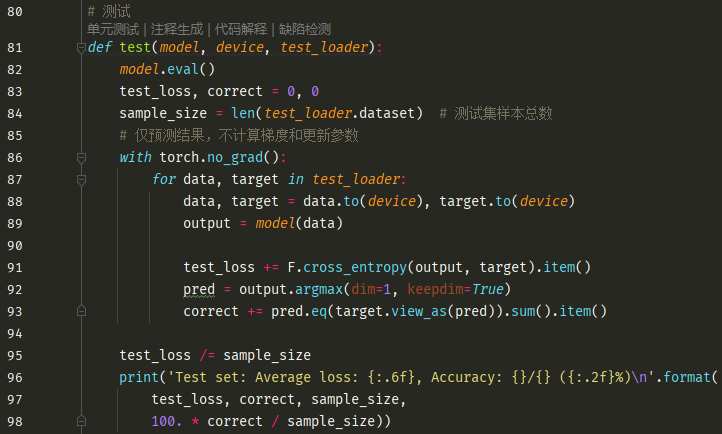


图4-5 模型测试

**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 |

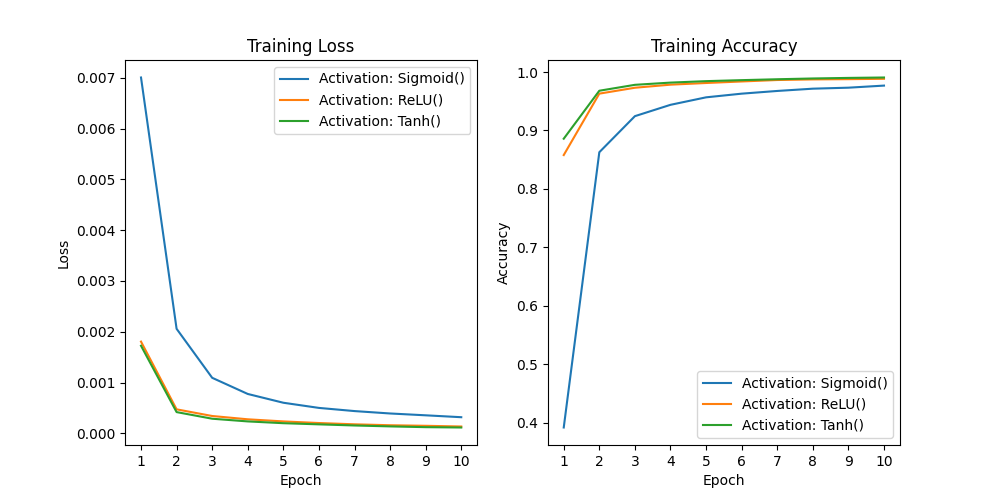


图4-6 不同激活函数训练过程

**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 |

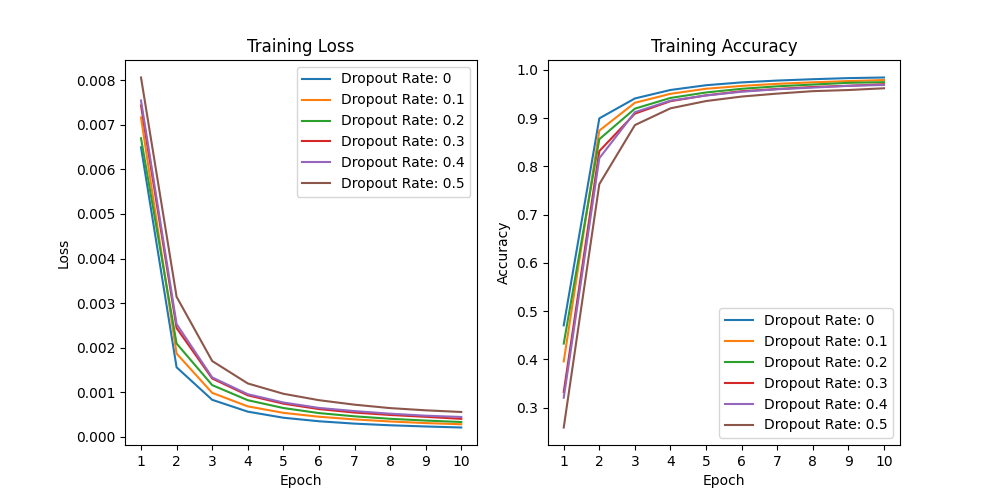


图4-7 不同dropout比例训练过程

**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 |

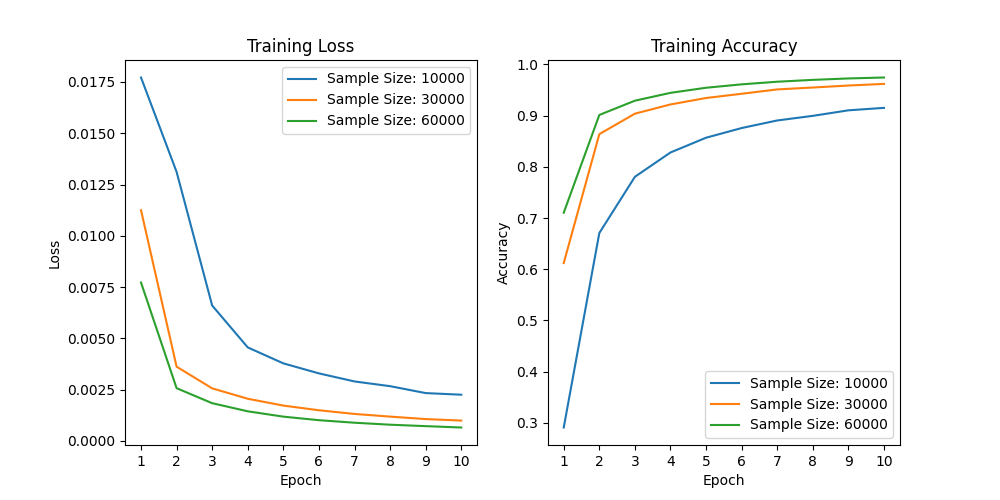


图4-8 不同数据量训练过程

**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 |

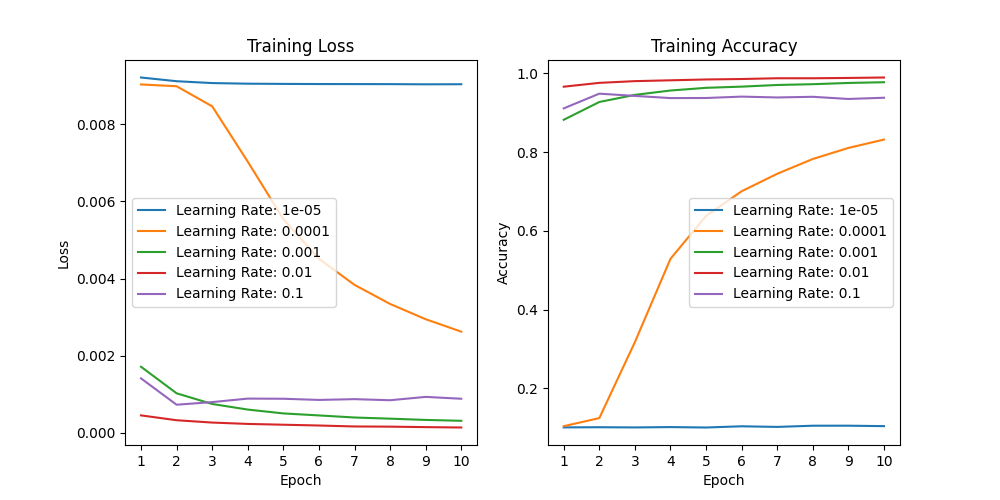


图4-9 不同学习率训练过程

**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 |

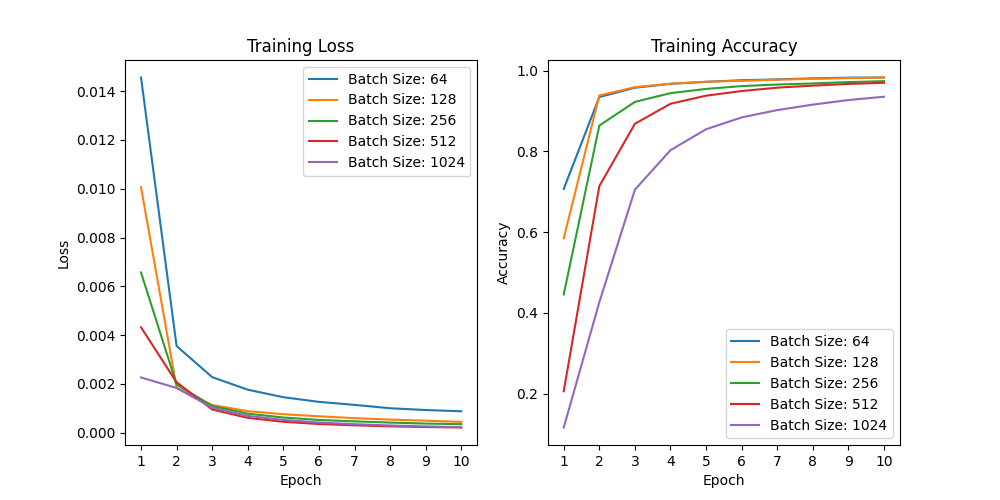


图4-10 不同批次大小训练过程

**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
4. from torch.utils.data import DataLoader
5. from torch.utils.data.dataset import Subset
6. import torch.optim as optim
7. from torchvision import transforms, datasets
8. import matplotlib.pyplot as plt
9. import numpy as np
10. import time
11. class LeNet5(nn.Module):
12. def \_\_init\_\_(self, activation=nn.Sigmoid(), p=0.2):
13. super(LeNet5, self).\_\_init\_\_()
14. self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=6, kernel\_size=5)
15. self.activation = activation
16. self.maxpool1 = nn.MaxPool2d(kernel\_size=2, stride=2)
17. self.conv2 = nn.Conv2d(in\_channels=6, out\_channels=16, kernel\_size=5)
18. self.maxpool2 = nn.MaxPool2d(kernel\_size=2, stride=2)
19. self.fc1 = nn.Linear(16 \* 5 \* 5, 120)
20. self.dropout = nn.Dropout(p)
21. self.fc2 = nn.Linear(120, 84)
22. self.fc3 = nn.Linear(84, 10)
23. def forward(self, x):
24. *# CONV1, ReLU1, POOL1*
25. x = self.conv1(x)
26. x = self.maxpool1(x)
27. x = self.activation(x)
28. *# CONV2, ReLU2, POOL2*
29. x = self.conv2(x)
30. x = self.maxpool2(x)
31. x = self.activation(x)
32. x = x.view(-1, 16 \* 5 \* 5)
33. *# FC1*
34. x = self.fc1(x)
35. x = self.activation(x)
36. x = self.dropout(x)
37. *# FC2*
38. x = self.fc2(x)
39. x = self.activation(x)
40. x = self.dropout(x)
41. *# FC3*
42. x = self.fc3(x)
43. output = F.log\_softmax(x, dim=1)
44. return output
45. *# 随机选择样本*
46. def create\_subset(data, sample\_size):
47. indices = np.random.choice(len(data), sample\_size, replace=False)  *# 随机选择样本索引*
48. return Subset(data, indices)
49. *# 训练*
50. def train(model, device, train\_loader, optimizer, epoch):
51. model.train()
52. correct, loss, train\_loss = 0, 0, 0
53. sample\_size = len(train\_loader.dataset) *# 训练集样本总数*
54. for batch\_idx, (data, target) in enumerate(train\_loader):
55. data, target = data.to(device), target.to(device)
56. optimizer.zero\_grad()
57. output = model(data)    *# 前向传播，保存训练结果*
58. pred = output.argmax(dim=1)
59. correct += pred.eq(target.view\_as(pred)).sum().item()
60. loss = F.cross\_entropy(output, target)  *# 交叉熵损失函数*
61. train\_loss += loss.item()
62. loss.backward()     *# 反向传播*
63. optimizer.step()    *# 更新参数*
64. if batch\_idx % 50 == 0:
65. print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
66. epoch, batch\_idx \* len(data), sample\_size,
67. 100. \* batch\_idx / len(train\_loader), loss.item()))
68. train\_loss /= sample\_size
69. train\_acc = correct / sample\_size
70. return train\_loss, train\_acc
71. *# 测试*
72. def test(model, device, test\_loader):
73. model.eval()
74. test\_loss, correct = 0, 0
75. sample\_size = len(test\_loader.dataset)  *# 测试集样本总数*
76. *# 仅预测结果，不计算梯度和更新参数*
77. with torch.no\_grad():
78. for data, target in test\_loader:
79. data, target = data.to(device), target.to(device)
80. output = model(data)
81. test\_loss += F.cross\_entropy(output, target).item()
82. pred = output.argmax(dim=1, keepdim=True)
83. correct += pred.eq(target.view\_as(pred)).sum().item()
84. test\_loss /= sample\_size
85. print('Test set: Average loss: {:.6f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
86. test\_loss, correct, sample\_size,
87. 100. \* correct / sample\_size))
88. if \_\_name\_\_ == '\_\_main\_\_':
89. totalLoss, totalAccuracy = [], []
90. epochs = 10
91. transform = transforms.Compose([
92. transforms.Resize((32, 32)),
93. transforms.ToTensor(),
94. transforms.Normalize((0.1307,),(0.3081,))
95. ])
96. *# 获取MNIST数据集*
97. sample\_sizes = [10000, 30000, 60000]
98. train\_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
99. test\_dataset = datasets.MNIST(root="./data", train=False, download=True, transform=transform)
100. *# 加载数据集*
101. batch\_sizes = [64, 128, 256, 512, 1024]
102. train\_dataloader = DataLoader(dataset= create\_subset(train\_dataset, sample\_sizes[2]), batch\_size=batch\_sizes[1], shuffle=True)
103. test\_dataloader = DataLoader(dataset=test\_dataset, batch\_size=32, shuffle=False)
104. *# 创建模型，部署GPU*
105. activation = [nn.Sigmoid(), nn.ReLU(), nn.Tanh()]   *# 不同激活函数*
106. dropout\_p = [0, 0.1, 0.2, 0.3, 0.4, 0.5]    *# 不同dropout比例*
107. device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")
108. model = LeNet5(activation=activation[1], p=dropout\_p[1]).to(device)
109. *# 优化器*
110. lrs = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1]    *# 不同学习率*
111. optimizer = optim.Adam(model.parameters(), lr=lrs[1])
112. *# 训练*
113. Loss, Accuracy = [], []
114. print(f"Start Time: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())}")
115. for epoch in range(1, epochs+1):
116. loss, acc = train(model, device, train\_dataloader, optimizer, epoch)
117. Loss.append(loss)
118. Accuracy.append(acc)
119. print(f"End Time: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())}")
120. totalLoss.append(Loss)
121. totalAccuracy.append(Accuracy)
122. *# 测试*
123. test(model, device, test\_dataloader)
124. *# 可视化*
125. x\_ticks = np.arange(1, epochs+1)
126. fig, axes = plt.subplots(1, 2, figsize=(10, 5))
127. axes[0].plot(x\_ticks, totalLoss[0])
128. axes[0].set\_title('Training Loss')
129. axes[0].set\_xlabel('Epoch')
130. axes[0].set\_ylabel('Loss')
131. axes[0].set\_xticks(x\_ticks)
132. axes[0].legend()
133. axes[1].plot(x\_ticks, totalAccuracy[0])
134. axes[1].set\_title('Training Accuracy')
135. axes[1].set\_xlabel('Epoch')
136. axes[1].set\_ylabel('Accuracy')
137. axes[1].set\_xticks(x\_ticks)
138. axes[1].legend()
139. 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.