

**CENTRAL SOUTH UNIVERSITY**

**深度学习课程实验报告**

题 目 深度学习实验报告

学生姓名 钱兴宇

班级学号 8207211912

指导教师 李仪

设计时间 2023年11月

目录

[第一章 开发环境 1](#_Toc22371)

[1.1 硬件、软件信息 1](#_Toc3169)

[1.2 主要依赖包及其版本 1](#_Toc24436)

[第二章 从零实现MLP 2](#_Toc325)

[2.1 实验分析及思路 2](#_Toc18581)

[2.2 实验过程 2](#_Toc14584)

[2.3 实验结果与分析 8](#_Toc7050)

[第三章 卷积网络实现手写数字识别 9](#_Toc21681)

[3.1 实验分析及思路 9](#_Toc9573)

[3.2 实验过程 9](#_Toc7218)

[3.3 实验结果与分析 14](#_Toc2113)

[第四章 Torch构建ResNet 17](#_Toc227)

[4.1 实验分析及思路 17](#_Toc26320)

[4.2 实验过程 17](#_Toc10935)

[4.3 实验结果与分析 21](#_Toc2113)

[第五章 手写二叉树检测 24](#_Toc3896)

[5.1 实验分析及思路 24](#_Toc31611)

[5.2 实验过程 24](#_Toc6144)

[5.3 实验结果与分析 28](#_Toc17334)

[第六章 总结 31](#_Toc5067)

[参考文献 32](#_Toc1552)

# 开发环境

## 1.1 硬件、软件信息

本次实验硬件、软件信息如下表。

|  |  |
| --- | --- |
| 硬件 | 规格 |
| GPU | RTX3080Ti |
| CPU | I9-12900H |

表 1.1 硬件信息

|  |  |
| --- | --- |
| 软件 | 版本 |
| CUDA | 10.1 |
| Python | 3.9 |
| torch | 2.0.0 |

表1.2 软件信息

## 1.2 主要依赖包及版本

本次实验所使用主要依赖包及其版本如下表。

|  |  |
| --- | --- |
| Package | Version |
| Matplotlib | 3.5.2 |
| Numpy | 1.21.6 |
| Pandas | 1.3.5 |
| Pilow | 9.3.0 |
| Seaborn | 0.12.2 |
| Torch | 2.0.0 |
| Torchvision | 0.15.1 |

表 1.3 依赖包信息

# 从零实现MLP

编写MLP，要求如下：

* 1. 只能使用numpy编写，不能直接使用pytorch中的linear ，sigmoid等函数；
  2. 完成输入层，隐含层和输出层；
  3. 编写损失函数以及激活函数；
  4. 编写训练过程，预测过程；
  5. 回归或分类均可，数据集可自行选择。

## 2.1 实验分析及思路

实验题目要求编写一个多层感知机（MLP），使用numpy库进行实现，不能直接使用pytorch等深度学习框架的函数。实验要求包括输入层、隐含层和输出层的设计，损失函数和激活函数的编写，以及训练和预测过程的实现。此外，可以选择回归或分类任务，并自行选择数据集。

实验思路如下：

1. 数据准备：选择Iris鸢尾花数据集作为实验数据集，并且选择合适比例划分训练集、测试集，再对数据集做适当的预处理。
2. 初始化模型参数：定义MLP结构，包括输入层、隐藏层、输出层的层数和各层的节点数，并对参数进行初始化。
3. 定义激活函数、定义损失函数，编写前向传播和反向传播过程。
4. 定义模型训练过程：先根据权重和偏置计算每一层的输出，再利用损失函数计算当前损失值，通过反向传播得出梯度，从而更新参数。
5. 定义预测过程，待预测样本得到预测值；并在测试集上进行模型评估。

## 2.2 实验过程

本实验选择Iris鸢尾花数据集进行实验。Iris数据集是机器学习和统计领域中经典的数据集之一。它由英国统计学家和生物学家Ronald Fisher于1936年收集整理而成。这个数据集被广泛用于分类问题的演示和实验。Iris数据集包含了150个样本，其中每个样本代表一朵鸢尾花，共涵盖了三个不同品种的鸢尾花：setosa、versicolor和virginica。对于每个样本，有四个特征被测量：花萼长度(sepal length)、花萼宽度(sepal width)、花瓣长度(petal length)和花瓣宽度(petal width)。这些特征被用来预测鸢尾花的品种。通过使用这个数据集，我们可以训练模型来根据花的特征来预测其所属的鸢尾花品种，即将其分为setosa、versicolor和virginica中的一类。 从sklearn库中加载数据集后打乱顺序并按照8：2比例划分为训练集和测试集，并分别将其归一化。下面是具体代码。

1. **import** numpy as np
2. **from** sklearn.datasets **import** load\_iris

5. **def** iris\_data\_load():
6. # 加载Iris数据集
7. iris = load\_iris()
8. X, y = iris.data, iris.target
10. # 随机打乱数据和标签
11. np.random.seed(521)  # 为了可重复性
12. indices = np.arange(X.shape[0])
13. np.random.shuffle(indices)
14. X = X[indices]
15. y = y[indices]
17. # 划分训练集和测试集（例如，80%训练，20%测试）
18. test\_size\_ratio = 0.2
19. test\_size = int(X.shape[0] \* test\_size\_ratio)
20. X\_train = X[:-test\_size]
21. X\_test = X[-test\_size:]
22. y\_train = y[:-test\_size]
23. y\_test = y[-test\_size:]
25. # 手动实现归一化（特征缩放）
26. # 计算训练数据的均值和标准差
27. mean = np.mean(X\_train, axis=0)
28. std = np.std(X\_train, axis=0)
30. # 对训练数据和测试数据应用归一化
31. X\_train\_scaled = (X\_train - mean) / std
32. X\_test\_scaled = (X\_test - mean) / std
34. **return** np.float32(X\_train\_scaled), np.float32(X\_test\_scaled), y\_train, y\_test

Iris鸢尾花数据集是一个分类问题，因此选择交叉熵损失函数。交叉熵损失函数（cross\_entropy\_loss）用于衡量预测结果和真实标签之间的差异。它采用负对数似然的方式计算损失。首先，根据预测结果和真实标签的索引位置，获取对应的预测概率值，然后将其取负对数。接着将所有样本的损失求和，并除以样本数量，得到平均损失。最后返回该平均损失。交叉熵损失函数计算公式如下。

tanh函数的导数在大部分区间内都不为零，这意味着它在整个定义域上都有较好的梯度，有利于梯度下降算法的计算，并且其输出均值接近0，这有助于减少网络训练过程中的偏移偏差此外，tanh函数是一种非线性函数，可以帮助神经网络学习复杂的非线性关系，提高网络的表达能力。使用tanh(x)作为激活函数，计算公式如下。

Tanh(x)函数图像如下。

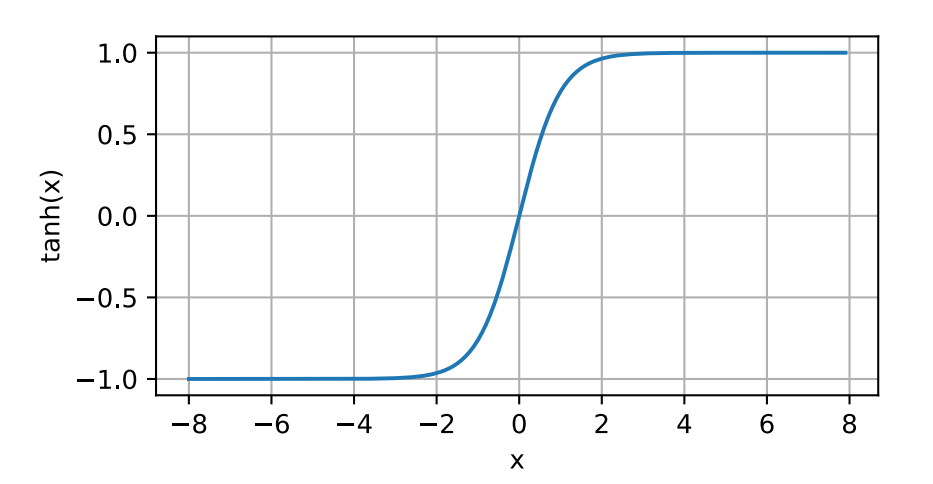


图 2.1 tanh函数图像

交叉熵损失函数、激活函数实现代码如下。

1. **import** numpy as np
2. **def** softmax(x):
3. e\_x = np.exp(x - np.max(x, axis=1, keepdims=True))
4. **return** e\_x / np.sum(e\_x, axis=1, keepdims=True)

7. **def** cross\_entropy\_loss(predictions, labels):
8. m = labels.shape[0]
9. log\_likelihood = -np.log(predictions[range(m), labels])
10. loss = np.sum(log\_likelihood) / m
11. **return** loss

14. **def** derivative\_cross\_entropy(predictions, labels):
15. m = labels.shape[0]
16. grad = predictions.copy()
17. grad[range(m), labels] -= 1
18. grad = grad / m
19. **return** grad

22. **def** tanh(x):
23. # 手动实现tanh函数
24. **return** (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))

27. **def** derivative\_tanh(x):
28. **return** 1 - tanh(x) \*\* 2

MLP（Multilayer Perceptron）是一种人工神经网络模型，也被称为多层感知机。它是最早应用广泛的神经网络模型之一。MLP 由多个神经元（或称为节点）组成，这些神经元按照层次结构排列，包括输入层、隐藏层和输出层。MLP 的每个神经元都与上一层的所有神经元相连，形成一个全连接结构。每个连接都有一个权重，代表了该连接在信息传递中的重要性。此外，每个神经元还有一个偏置项，用于调整神经元的激活阈值。下图是一个单隐藏层MLP示意图。

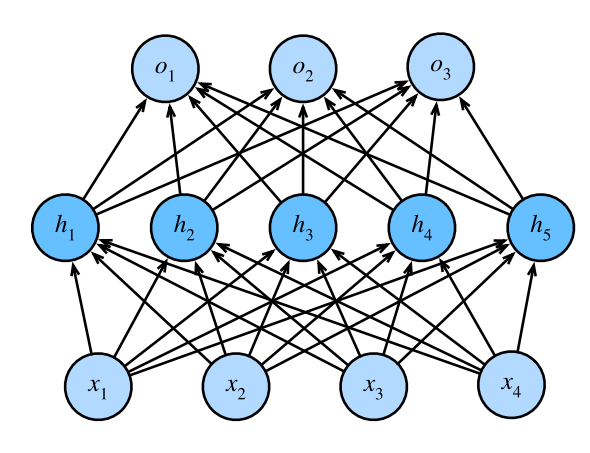


图2.2 多层感知机结构示意图

本实验定义了一个输入层、两个隐藏层、一个输出层的MLP，每一层的节点数由参数指定，具体实现代码如下。

1. # MLP类
2. **class** MLP:
3. **def** \_\_init\_\_(self, input\_size, hidden\_size1, hidden\_size2, output\_size):
4. # 使用Xavier初始化
5. self.W1 = np.random.randn(input\_size, hidden\_size1) / np.sqrt(input\_size)
6. self.b1 = np.zeros(hidden\_size1)
7. self.W2 = np.random.randn(hidden\_size1, hidden\_size2) / np.sqrt(hidden\_size1)
8. self.b2 = np.zeros(hidden\_size2)
9. self.W3 = np.random.randn(hidden\_size2, output\_size) / np.sqrt(hidden\_size2)
10. self.b3 = np.zeros(output\_size)
12. **def** forward(self, X):
13. self.z1 = np.dot(X, self.W1) + self.b1
14. self.a1 = tanh(self.z1)
15. self.z2 = np.dot(self.a1, self.W2) + self.b2
16. self.a2 = tanh(self.z2)
17. self.z3 = np.dot(self.a2, self.W3) + self.b3
18. self.a3 = softmax(self.z3)
19. **return** self.a3

MLP 具有前向传播和反向传播两个步骤。在前向传播过程中，输入数据从输入层经过隐藏层传递到输出层，最终产生预测结果。

在反向传播过程中，根据预测结果和真实标签之间的差异，计算每一层参数的梯度，通过梯度下降算法来更新权重和偏置，以使模型的预测结果逼近真实标签。每一层参数的梯度的计算服从链式法则。

具体实现代码如下。

1. # 对于MLP，我们需要一个训练函数来执行前向和后向传播
2. **def** train(model, X\_train, y\_train, X\_val, y\_val, epochs, batch\_size, learning\_rate):
3. n\_batches = int(np.ceil(X\_train.shape[0] / batch\_size))
5. **for** epoch **in** range(epochs):
6. **for** X\_batch, y\_batch **in** batch\_generator(X\_train, y\_train, batch\_size):
7. # 前向传播
8. output = model.forward(X\_batch)
10. # 计算损失
11. loss = cross\_entropy\_loss(output, y\_batch)
13. # 反向传播
14. dZ3 = derivative\_cross\_entropy(output, y\_batch)
15. dW3 = np.dot(model.a2.T, dZ3)
16. db3 = np.sum(dZ3, axis=0, keepdims=True)
18. dA2 = np.dot(dZ3, model.W3.T)
19. dZ2 = dA2 \* derivative\_tanh(model.z2)
20. dW2 = np.dot(model.a1.T, dZ2)
21. db2 = np.sum(dZ2, axis=0, keepdims=True)
23. dA1 = np.dot(dZ2, model.W2.T)
24. dZ1 = dA1 \* derivative\_tanh(model.z1)
25. dW1 = np.dot(X\_batch.T, dZ1)
26. db1 = np.sum(dZ1, axis=0, keepdims=True)
28. # 更新权重
29. model.W3 -= learning\_rate \* dW3
30. model.b3 -= learning\_rate \* db3.squeeze()
31. model.W2 -= learning\_rate \* dW2
32. model.b2 -= learning\_rate \* db2.squeeze()
33. model.W1 -= learning\_rate \* dW1
34. model.b1 -= learning\_rate \* db1.squeeze()
36. # 这里添加验证逻辑，以监控验证损失和准确性
37. val\_output = model.forward(X\_val)
38. val\_loss = cross\_entropy\_loss(val\_output, y\_val)
39. val\_predictions = np.argmax(val\_output, axis=1)
40. val\_accuracy = np.mean(val\_predictions == y\_val)
41. **print**(
42. f'Epoch {epoch + 1}, Loss: {loss:.4f}, Validation Loss: {val\_loss:.4f}, Validation Accuracy: {val\_accuracy:.4f}')
44. # 加载数据
45. X\_train\_scaled, X\_test\_scaled, y\_train, y\_test = iris\_data\_load()
46. # 训练模型
47. input\_size = X\_train\_scaled.shape[1]
48. hidden\_size1 = 8
49. hidden\_size2 = 116
50. output\_size = len(np.unique(y\_train))  # 根据类别数量来设置
51. model = MLP(input\_size, hidden\_size1, hidden\_size2, output\_size)
53. # 转换标签为one-hot编码
54. y\_train\_one\_hot = np.eye(output\_size)[y\_train]
55. y\_test\_one\_hot = np.eye(output\_size)[y\_test]
57. train(model, X\_train\_scaled, y\_train, X\_test\_scaled, y\_test, epochs=1000, batch\_size=4, learning\_rate=1e-3)
59. # 最后，使用测试集评估模型性能
60. test\_output = model.forward(X\_test\_scaled)
61. test\_loss = cross\_entropy\_loss(test\_output, y\_test)
62. test\_predictions = np.argmax(test\_output, axis=1)
63. test\_accuracy = np.mean(test\_predictions == y\_test)
64. **print**(f'Test Loss: {test\_loss:.4f}, Test Accuracy: {test\_accuracy}’)

## 2.3 实验结果与分析

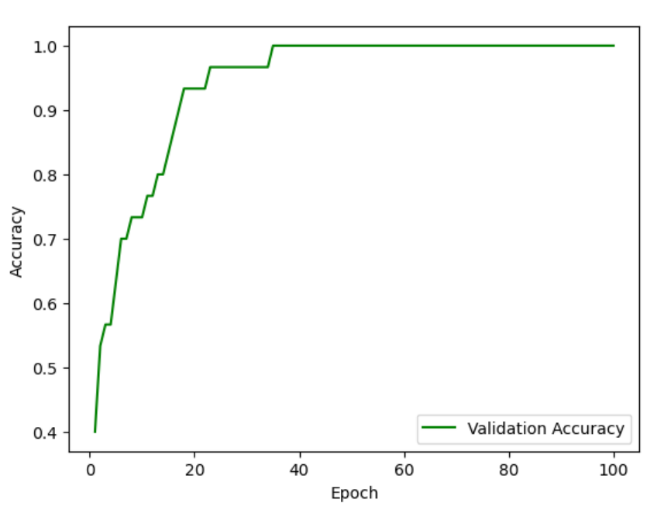
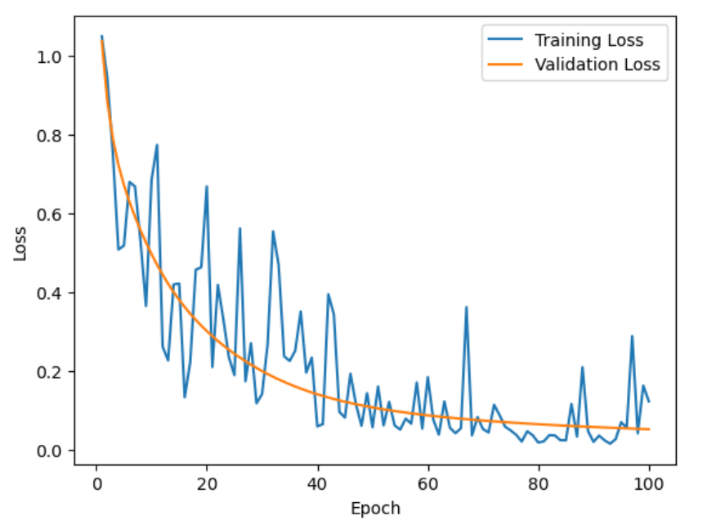
 Hidden\_size=8,hidden\_size2=16,lr=1e-3的条件下本实验结果如下。

图 2.3 训练结果

可以看到随着训练代数的增加，Training\_loss、Validation\_loss整体下降，Valiadation\_accuracy逐步升高，最终取得了较好的分类效果。

# 卷积网络识别手写数字

编写一个卷积网络，对手写体数字进行识别。要求如下：

* 1. 数据集使用torch中自带的样本数据集；
  2. 卷积网络的结构、激活函数、损失函数自定义；
  3. 所有模型中的超参自行设定；
  4. 保存训练结果

提示：分析多种情况并保留分析结果。

## 3.1 实验分析及思路

实验题目要求编写一个卷积网络，对手写体数字进行识别，使用Torch.nn中卷积层、池化层函数构建一个卷积神经网络。通过设定不同超参数比较识别效果分析不同超参数设定对于实验结果的影响，并将训练结果保存。

实验思路如下：

1. 数据准备：选择MNIST手写数字集作为实验数据集，并且选择合适比例划分训练集、测试集，再对数据集做适当的预处理。
2. 网络构建：定义神经网络结构，包括卷积层、池化层的层数和各层的channels、kernel\_szie、stride、padding等参数。
3. 定义激活函数、定义损失函数。
4. 模型训练：设定不同超参数组合进行模型训练，比较不同超参数下模型的性能，保存训练结果。

## 3.2 实验过程

本实验选择MNIST手写数字数据集作为实验数据集。MNIST数据集是一个经典的手写数字识别数据集，由Yann LeCun等人收集和发布。这个数据集常用于测试和评估机器学习算法在图像分类任务上的性能。MNIST数据集包含了大量的手写数字图像，涵盖了0到9这10个数字。每张图像都是28x28像素的灰度图像，像素值的范围是0到255。这些图像已经被预处理并标记。MNIST数据集一般被分为两部分：训练集和测试集。训练集包含了60,000张图像，而测试集包含了10,000张图像。这种划分方式确保了模型在训练和测试阶段使用不同的数据，以便对算法的泛化能力进行评估。为了方便处理和使用，MNIST数据集已经被预处理和标记。预处理包括将图像转换为固定大小的28x28像素，以及对像素值进行归一化处理（通常是将像素值除以255，使其范围在0到1之间）。标记则是指每个图像都有一个对应的标签，表示图像中的数字是0到9中的哪一个。

本实验从Torchvision中Dataset下载MNIST数据集，并且针对本实验进行大小缩放、张量化、标准化等操作，具体代码如下。

1. **from** torch.utils.data **import** DataLoader
2. **from** torchvision **import** datasets, transforms

5. **def** create\_dataset(data\_dir, training=True, batch\_size=64,
6. resize=(32, 32), rescale=1 / (255 \* 0.3081), shift=-0.1307 / 0.3081):
7. # 这里的shift和rescale是根据MNIST数据集的均值和标准差提前计算得到的
8. # 数据处理：数据集加载、缩放、归一化、格式转换、洗牌、批标准化。
9. transform = transforms.Compose([
10. transforms.Resize(resize),
11. transforms.ToTensor(),
12. transforms.Normalize(mean=[shift], std=[rescale])
13. ])
15. dataset = datasets.MNIST(root=data\_dir, train=training, transform=transform, download=True)
16. loader = DataLoader(dataset, batch\_size=batch\_size, shuffle=training)
18. **return** loader

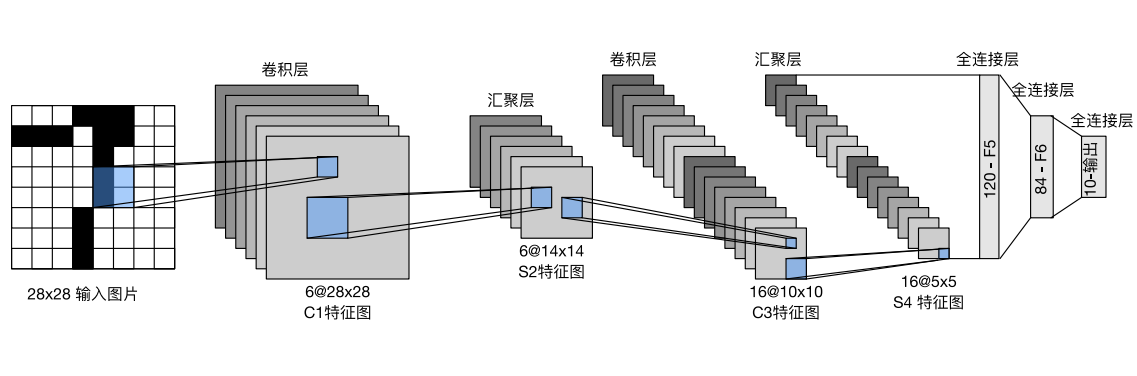
卷积神经网络（Convolutional Neural Network，CNN）是一种特殊的神经网络结构，主要用于处理具有网格结构的数据，例如图像、视频和声音等。与传统的全连接神经网络相比，卷积神经网络引入了卷积层（Convolutional Layer）和池化层（Pooling Layer），这两个关键组件使得CNN在处理图像等数据具有局部感知、参数共享的优势。卷积神经网络通常由多个卷积层、激活函数、池化层和全连接层组成。在前向传播过程中，输入数据会通过一系列的卷积和池化操作来提取特征，并最终通过全连接层进行分类或回归等任务。LeNet卷积神经网络结构示例如下图。

图 3.1 LeNet示意图

本实验网络结构实现代码如下。

1. **class** LeNet5(nn.Module):
2. """模型定义：算子初始化（参数设置），网络构建。"""
4. **def** \_\_init\_\_(self, activation='relu', dropout\_rate=None):
5. super(LeNet5, self).\_\_init\_\_()
6. self.conv1 = nn.Conv2d(1, 6, 5, stride=1, padding=0)
7. self.conv2 = nn.Conv2d(6, 16, 5, stride=1, padding=0)
8. self.activation = activation
9. self.relu = nn.ReLU()
10. self.sigmoid = nn.Sigmoid()
11. self.pool = nn.MaxPool2d(kernel\_size=2, stride=2)
12. self.dropout\_rate = dropout\_rate
13. self.dropout = nn.Dropout(p=dropout\_rate) **if** dropout\_rate **else** None
14. self.fc1 = nn.Linear(400, 120)
15. self.fc2 = nn.Linear(120, 84)
16. self.fc3 = nn.Linear(84, 10)
17. self.softmax = nn.Softmax(dim=1)
19. **def** forward(self, x):
20. act = self.relu **if** self.activation == 'relu' **else** self.sigmoid
21. x = act(self.conv1(x))
22. x = self.pool(x)
23. x = act(self.conv2(x))
24. x = self.pool(x)
26. x = x.view(x.size(0), -1)  # Flatten
27. x = self.fc1(x)
28. **if** self.dropout:
29. x = self.dropout(x)
30. x = self.fc2(x)
31. **if** self.dropout:
32. x = self.dropout(x)
33. x = self.fc3(x)
34. x = self.softmax(x)
36. **return** x

上述神经网络可以选择激活函数。Sigmoid函数，也称为逻辑函数，是一种常用的非线性函数，通常用于将输入值映射到0和1之间，但是由于Sigmoid函数在远离零点时梯度接近于零，可能导致梯度消失的问题。其数学表达式及图像如下。

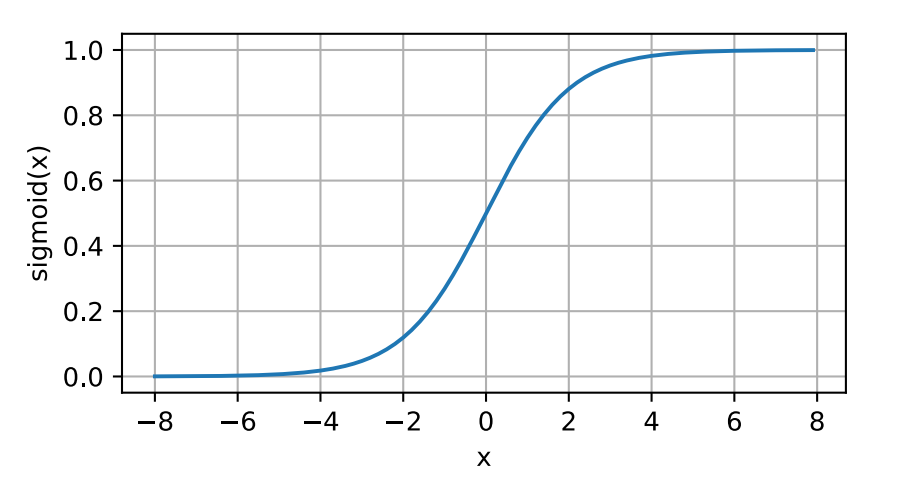


图 3.2 Sigmoid函数图像

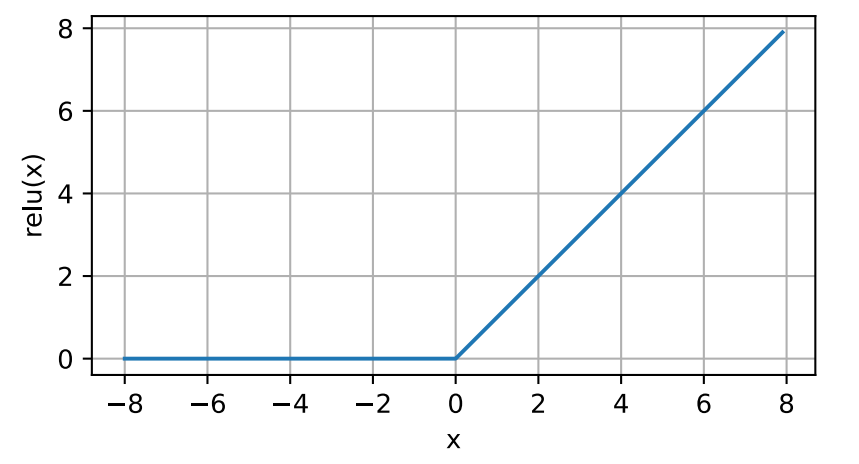
ReLU(Rectified Linear Unit)，也即修正线性单元，是一种常用的神经网络激活函数。在正区间上，ReLU函数的导数恒为1，这意味着它不会饱和，有助于解决梯度消失的问题。当输入为负时，ReLU函数的输出为0，因此可以通过设置部分权重为0来实现稀疏性，有助于模型的稀疏表示和计算效率。ReLU函数的计算非常简单，只需比较输入和0的大小即可。由于ReLU函数具有上述优点，并且在实践中表现良好，因此在深度学习中被广泛应用，特别是在卷积神经网络（CNN）中。其数学表达式及图像如下。

图 3.3 ReLU函数图像

在上述神经网络构建过程中可以选择是否加入Dropout技术。Dropout是一种常用的正则化技术，旨在减少神经网络中的过拟合问题。在训练过程中，Dropout随机地将一部分神经元的输出置为0，即将其“丢弃”，从而强制网络学习更加鲁棒和泛化的特征。，Dropout通过在每个训练批次中随机屏蔽（dropout）一些神经元的输出来实现。可以将Dropout视为在训练过程中对神经网络进行了多个子网络的集成，因为每个子网络只包含原始网络的一部分神经元。使用Dropout可以通过减少神经元之间的复杂共适应关系，强制网络学习更加独立的特征表示，从而减少过拟合，迫使网络中的神经元独立地学习有用的特征，而不依赖于其他神经元的存在。

在深度学习中，超参数（Hyperparameters）是指那些不能直接从训练数据中学习得到的参数。它们用于控制模型的结构和训练过程，而不是作为模型的参数进行学习。

学习率（Learning Rate）：控制模型在每次迭代更新参数时的步长或速度。较大的学习率可以使模型更快地收敛，但可能会导致不稳定或无法收敛；较小的学习率可以提高模型的稳定性，但训练速度可能会变慢。

批量大小（Batch Size）：指定在每次参数更新时用于训练的样本数量。较大的批量大小可以加快训练速度，但会增加内存需求；较小的批量大小可以提高模型的泛化能力，但训练过程可能会更慢。 批量大小（Batch Size）：指定在每次参数更新时用于训练的样本数量。较大的批量大小可以加快训练速度，但会增加内存需求；较小的批量大小可以提高模型的泛化能力，但训练过程可能会更慢。 正则化参数（Regularization Parameter）：用于控制正则化项的权重，以防止模型过拟合。较大的正则化参数可以增加正则化效果，但可能导致模型欠拟合。

本实验训练时可以自行选择不同的超参数组合，达到不同训练效果。训练日志借助python中logging库进行记录。具体训练代码如下。

1. **def** train(data\_dir, loss\_type='ce', activation='relu', dropout\_rate=0.1, lr=0.01, momentum=0.9, num\_epochs=10):
2. ds\_train = create\_dataset(data\_dir, training=True)
3. ds\_eval = create\_dataset(data\_dir, training=False)
5. net = LeNet5(activation=activation, dropout\_rate=dropout\_rate)
6. device = device\_setting()
7. net.to(device)
8. loss = nn.CrossEntropyLoss() **if** loss\_type == 'ce' **else** nn.NLLLoss()
10. opt = optim.SGD(net.parameters(), lr=lr, momentum=momentum)
12. train\_losses = []
13. val\_accuracies = []
15. **for** epoch **in** range(num\_epochs):
16. running\_loss = 0.0
17. **for** i, data **in** enumerate(ds\_train, 0):
18. inputs, labels = data
19. inputs, labels = inputs.to(device), labels.to(device)  # 将输入和标签移动到所选设备
20. opt.zero\_grad()
21. outputs = net(inputs)
22. loss\_output = loss(outputs, labels)
23. loss\_output.backward()
24. opt.step()
26. running\_loss += loss\_output.item()
27. train\_losses.append(running\_loss)
29. logging.info(f'Epoch {epoch + 1}, Loss: {running\_loss:.4f}')
31. # 每个epoch结束后，计算在验证集上的准确率
32. correct = 0
33. total = 0
34. with torch.no\_grad():
35. **for** data **in** ds\_eval:
36. images, labels = data
37. images, labels = images.to(device), labels.to(device)
38. outputs = net(images)
39. \_, predicted = torch.max(outputs.data, 1)
40. total += labels.size(0)
41. correct += (predicted == labels).sum().item()
42. acc = 100 \* correct / total
43. val\_accuracies.append(acc)
44. logging.info(f'Accuracy: {acc:.2f} %')
45. **return** net, train\_losses, val\_accuracies

## 3.3 实验结果与分析

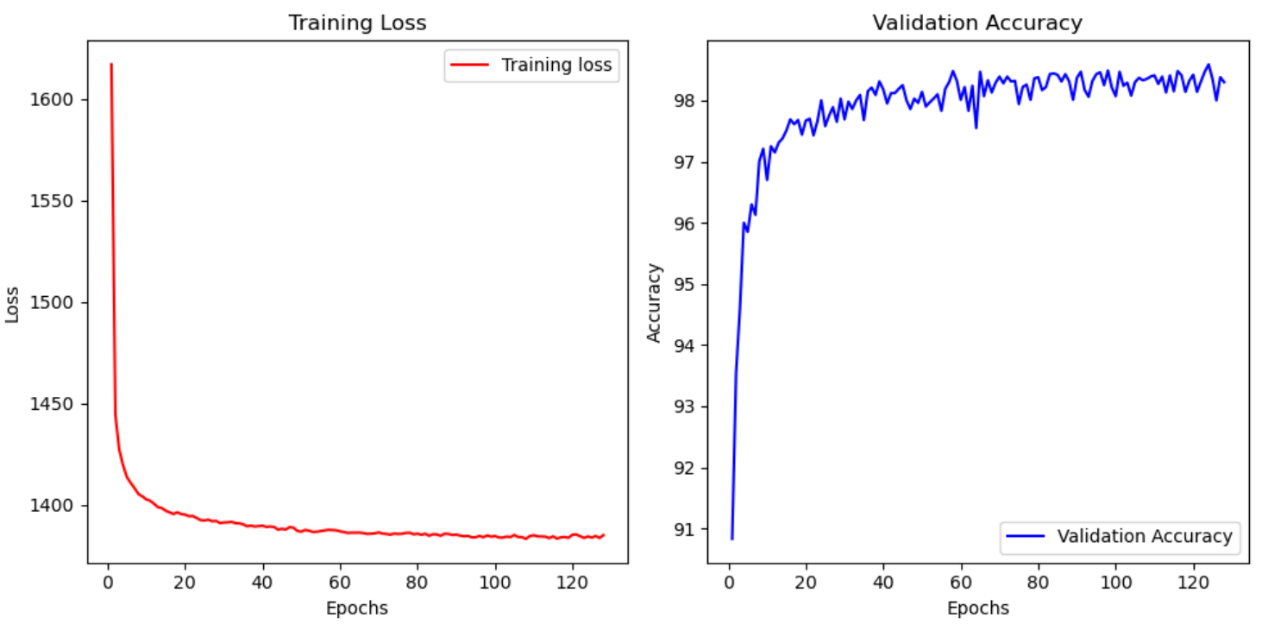


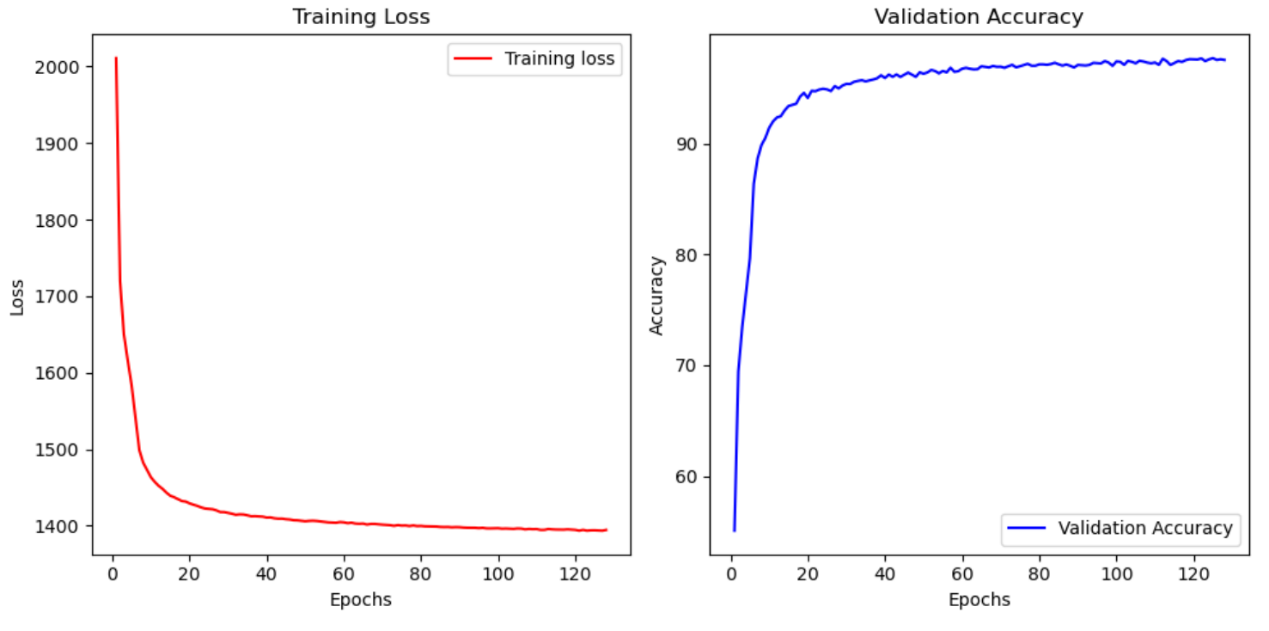
图 3.4 lr=1e-3,relu训练结果

图 3.5 lr=e-4 relu训练结果

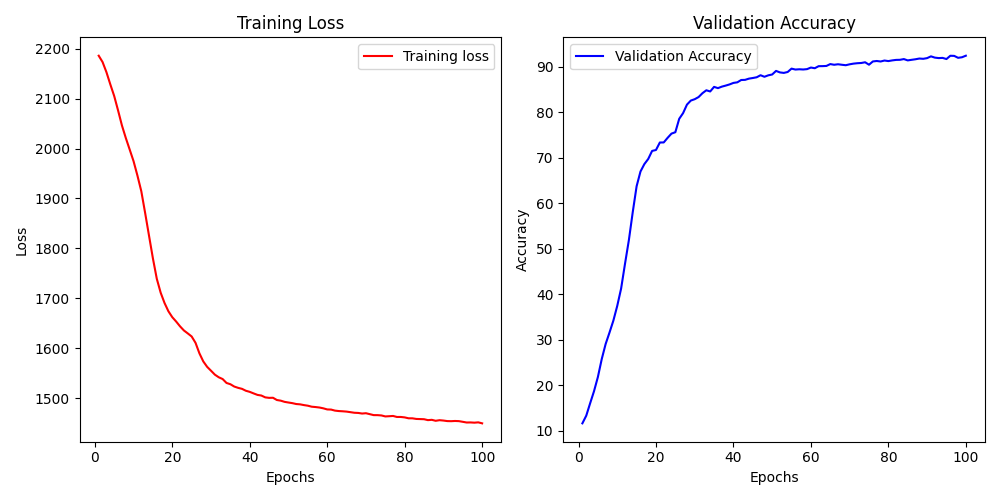
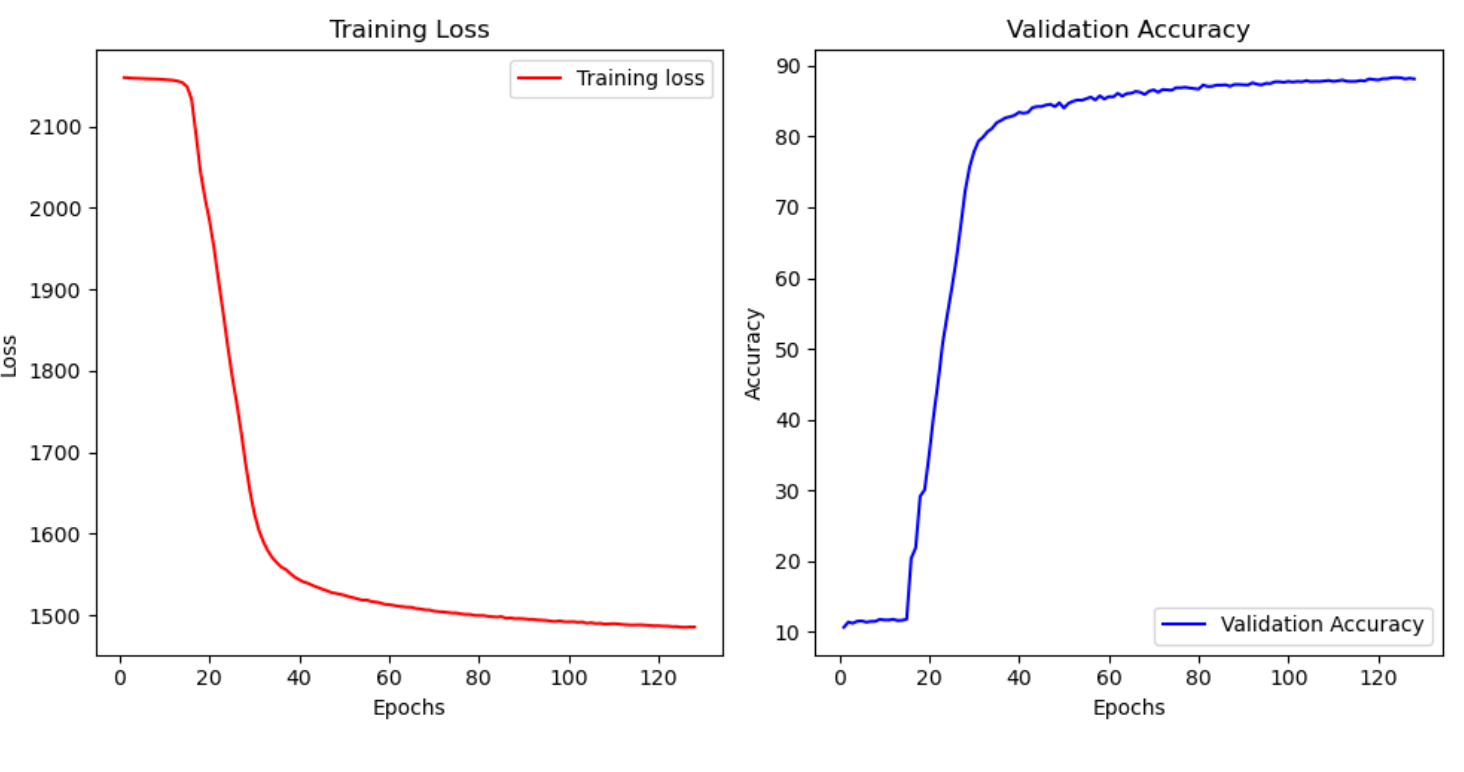


图 3.6 lr=1e-5 relu训练结果

由上面三组训练曲线可知，当学习率较小时，收敛速度比较慢，往往需要训练较多代数准确率和损失值才会收敛。当学习率增大时，收敛速度会增快。但是在学习率增大过程中模型会更容易出现震荡，可能导致模型的性能不太稳定。

图 3.8 lr=1e-4 sigmoid训练结果

将图3.8与图3.5对比可知，使用sigmoid函数作为激活函数收敛速度要比使用ReLU函数作为激活函数要慢。并且使用Sigmoid函数作为激活函数在训练初期损失值下降不明显，会存在一个平台期。Sigmoid函数的导数在输入很大或很小的时候会趋近于0，导致梯度消失的问题。在反向传播过程中，当梯度接近于零时，神经网络的参数几乎不会被更新，导致训练变得比较慢。相比之下，ReLU函数在正区域导数为常数，减少了出现梯度消失的风险，不会出现类似的平台期现象。

# Torch实现ResNet

使用pytroch实现ResNet。要求如下：

* 1. 层数不限，建议50层
  2. 数据集可以使用MNIST
  3. 完成训练和预测

## 4.1 实验分析及思路

实验题目要求使用Pytorch实现ResNet，并在自定的数据集上进行训练、测试。可以尝试构建不同层数的ResNet并将训练结果保存。

实验思路如下：

1. 数据准备：选择数据集，并且选择合适比例划分训练集、测试集，再对数据集做适当的预处理。
2. 模型构建：根据ResNet结构，构建残差模块，再通过堆叠不同层数的残差模块从而实现不同层数的ResNet。
3. 训练模型：选择损失函数、优化器，定义超参数，进行模型训练，并且保存训练结果。

## 4.2 实验过程

本实验依旧采用MNIST手写数字数据集作为实验数据集，数据读取、预处理、打包操作与前文所述相同。

本实验利用Torch构建了不同层数的ResNet。ResNet是一种深度残差网络，它是由微软亚洲研究院的研究人员提出的。ResNet的全称是Residual Network，它通过引入残差学习的方式来解决深度神经网络训练中的梯度消失和梯度爆炸等问题。

ResNet是由残差模块(Residual Block)堆叠而成。每个残差块由两部分组成：主路径（Main Path）和残差连接（Residual Connection）。主路径通常由一系列的卷积层、批量归一化层和非线性激活函数（如ReLU）组成。这些层用于提取输入数据的特征和进行非线性变换。残差连接则是将输入数据与主路径的输出进行相加，形成一个跨层连接。这个连接使得输入数据可以直接传递到后续层，而不需要经过额外的多个卷积层。其中一种残差模块的结构如下图。残差模块的实现代码如下。

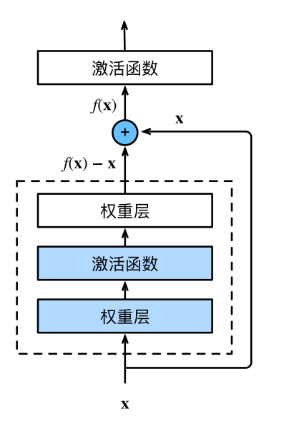
1. **class** ResBlock(nn.Module):

图 4.‑1 残差模块结构示意图

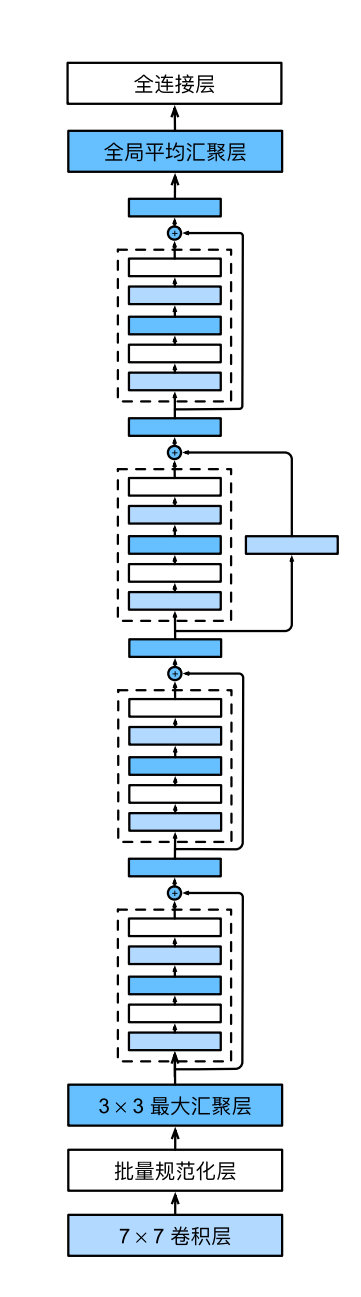
1. **def** \_\_init\_\_(self, n\_chans):
2. super(ResBlock, self).\_\_init\_\_()
3. self.conv = nn.Conv2d(n\_chans, n\_chans, kernel\_size=3, padding=1, bias=False)
4. self.batch\_norm = nn.BatchNorm2d(num\_features=n\_chans)
5. # 批范数层会抵消偏置的影响, 因此它通常被排除在外
6. torch.nn.init.kaiming\_normal\_(self.conv.weight, nonlinearity='relu')
7.         torch.nn.init.constant\_(self.batch\_norm.weight, 0.5)

图 4.2残差网络结构示意图

1. torch.nn.init.zeros\_(self.batch\_norm.bias)
3. **def** forward(self, x):
4. out = self.conv(x)
5. out = self.batch\_norm(out)
6. out = torch.relu(out)
7. **return** out + x

由残差模块堆叠而成的ResNet结构如下图。实现代码如下。

1. **class** ResNet(nn.Module):
2. **def** \_\_init\_\_(self, n\_chans1=32, n\_blocks=100):
3. super().\_\_init\_\_()
4. self.n\_chans1 = n\_chans1  # 通道数
5. self.conv1 = nn.Conv2d(1, n\_chans1, kernel\_size=3, padding=1)
6. self.resblocks = nn.Sequential(\*(n\_blocks \* [ResBlock(n\_chans=n\_chans1)]))
7. self.fc1 = nn.Linear(8 \* 8 \* n\_chans1, 32)
8. self.fc2 = nn.Linear(32, 10)
10. **def** forward(self, x):
11. out = F.max\_pool2d(torch.relu(self.conv1(x)), 2)  # nc\*28\*28 -> nc\*14\*14
12. out = self.resblocks(out)  # nc\*14\*14 -> nc\*14\*14
13. out = F.max\_pool2d(out, 2)  # nc\*14\*14 -> nc\*7\*7
14. out = out.view(-1, 8 \* 8 \* self.n\_chans1)
15. out = torch.relu(self.fc1(out))
16. out = self.fc2(out)
17. **return** out

可以通过修改残差模块的对叠层数构建不同层数的ResNet。训练具体代码如下。

1. **def** train(data\_dir, lr=0.01, num\_epochs=20):
2. ds\_train = create\_dataset(data\_dir, training=True)
3. ds\_eval = create\_dataset(data\_dir, training=False)
4. device = device\_setting(num\_cores=20)
5. net = ResNet().to(device)
6. loss = nn.CrossEntropyLoss()
7. opt = optim.Adam(net.parameters(), lr=lr)
9. train\_losses = []
10. val\_accuracies = []
12. **for** epoch **in** range(1, 1+num\_epochs):
13. running\_loss = 0.0
15. **for** i, data **in** enumerate(ds\_train, 0):
16. inputs, labels = data
17. inputs, labels = inputs.to(device), labels.to(device)  # 将输入和标签移动到所选设备
18. opt.zero\_grad()
19. outputs = net(inputs)
20. loss\_output = loss(outputs, labels)
21. loss\_output.backward()
22. opt.step()
24. running\_loss += loss\_output.item()
26. train\_losses.append(running\_loss)
27. logging.info(f'Epoch {epoch + 1}, Loss: {running\_loss:.4f}')
29. # 每个epoch结束后，计算在验证集上的准确率
30. correct = 0
31. total = 0
32. with torch.no\_grad():
33. **for** data **in** ds\_eval:
34. images, labels = data
35. images, labels = images.to(device), labels.to(device)
36. outputs = net(images)
37. \_, predicted = torch.max(outputs.data, 1)
38. total += labels.size(0)
39. correct += (predicted == labels).sum().item()
40. acc = 100 \* correct / total
41. val\_accuracies.append(acc)
42. logging.info(f'Accuracy: {acc:.2f} %')
43. **return** net, train\_losses, val\_accuracies

46. setup\_logger(timestamp)
48. # 定义超参数
49. hyperparameters = {
50. 'data\_dir': 'MNIST/MNIST\_Data',
51. 'lr': 3e-5,
52. 'num\_epochs': 100,
53. }
55. # 记录训练的开始时间
56. logging.info(f"Training with: \n{hyperparameters}\n")
58. # 训练模型
59. net, train\_losses, val\_acc = (
60. train(hyperparameters['data\_dir'], hyperparameters['lr'], hyperparameters['num\_epochs']))
62. # 绘制训练历史记录
63. title = f"ResNet\_{timestamp}(lr={hyperparameters['lr']}, epochs={hyperparameters['num\_epochs']})"
64. plot\_history(train\_losses, val\_acc, title)
66. # 将超参数保存到 JSON
67. with open(f'resnet.hyperparameters.json', 'w') as f:
68. json.dump(hyperparameters, f, indent=4)
70. # 保存模型参数
71. torch.save(net.state\_dict(), f'resnet\_{timestamp}.pth')

## 3.3 实验结果与分析

本实验通过堆叠不同层数的残差模块构建了ResNet16、ResNet34、ResNet50三种模型，通过在MNIST数据集上训练、验证来比较不同层数的残差网络性能变化。训练结果如下图。

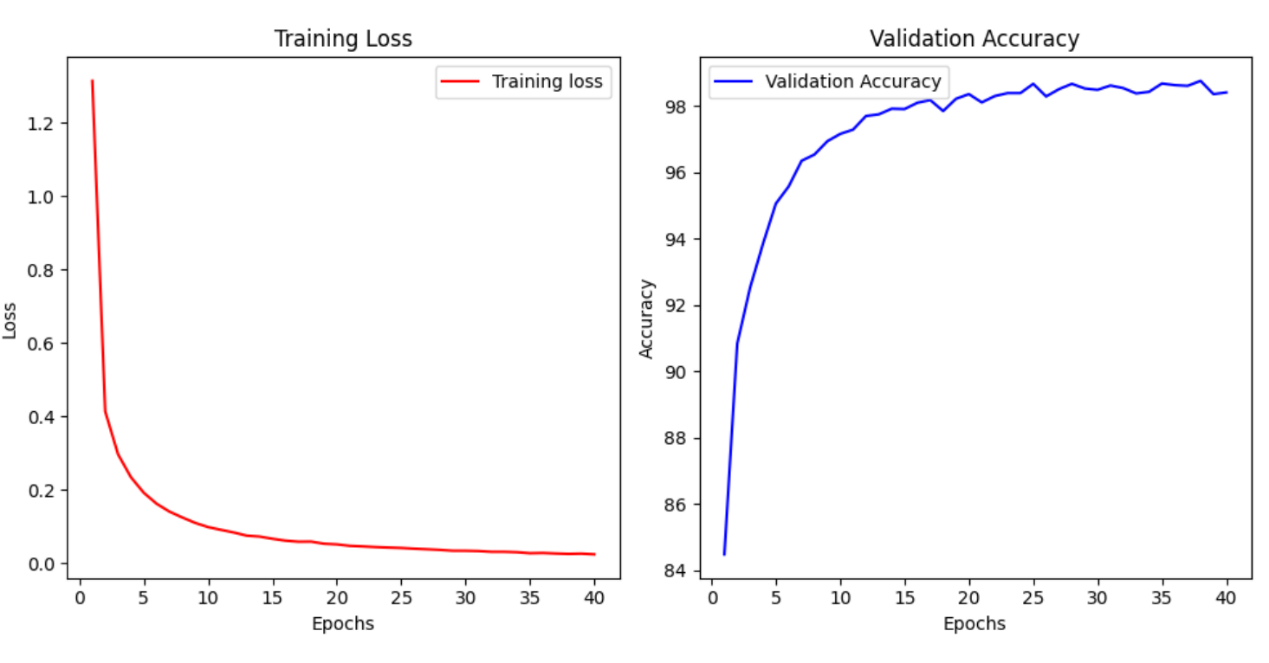


图 4.3 ResNet16训练结果

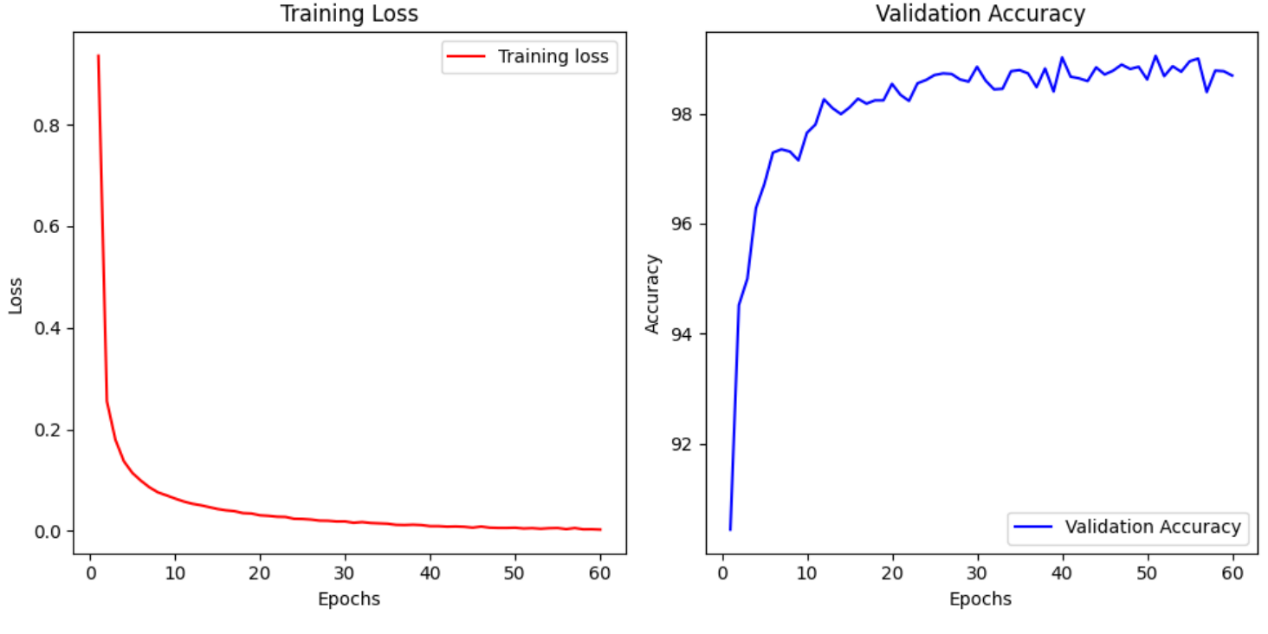


图 4.4 ResNet34训练结果

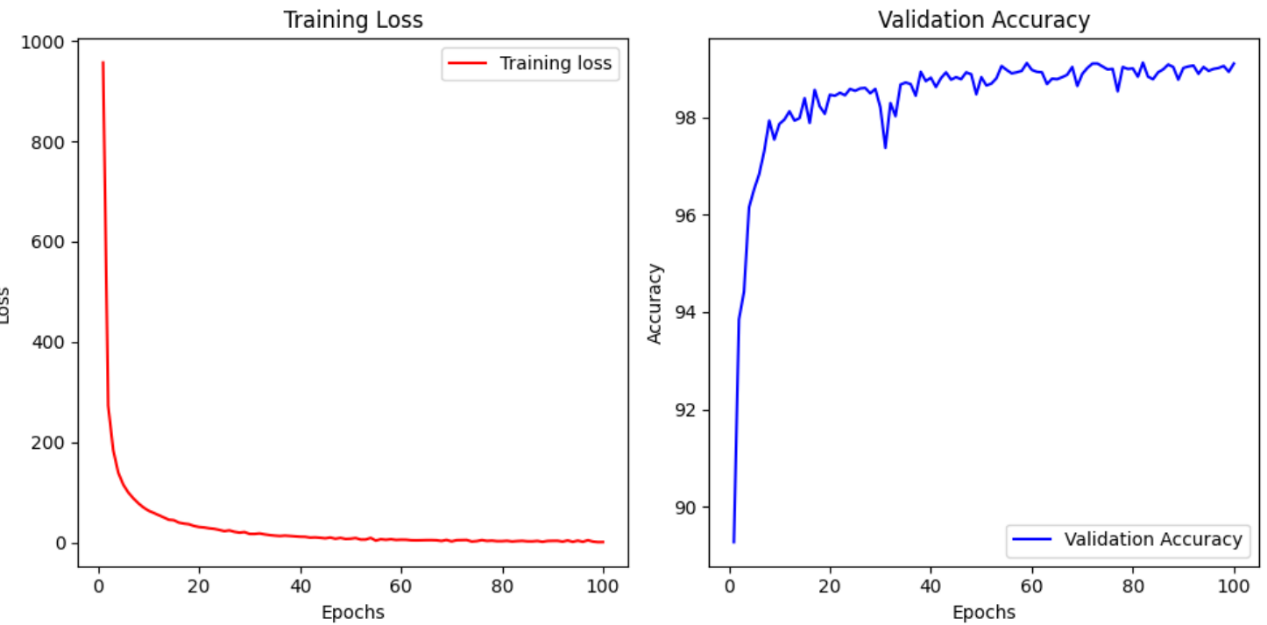


图 4.5 ResNet50训练结果

对比上面三组不同网络层数的训练曲线，在训练初期，层数较多的网络由于参数量较大、非线性变换较多，因此能够更好地拟合训练数据，表现出较高的训练准确率。相比之下，层数较少的网络可能因为表达能力有限而表现不佳。因此层数较少的网络性能显然不如层数较多的网络。但是随着训练代数的增多，层数较少的网络和层数较多的网络性能差距越来越小，最后得到的网络准确率往往只相差一到两个百分点。但是层数较多的网络通常具有更复杂的结构，参数之间的相互作用更加复杂，这可能导致训练过程中出现梯度爆炸或梯度消失的问题，从而导致训练的震荡风险增加。此外，层数较多的网络对于初始化、学习率等超参数的选择也更为敏感，容易陷入局部最优点或者不稳定的训练状态。

# 手写二叉树检测

手绘树结构，通过目标检测+图像处理完成数据结构中树的手绘图转换成二叉树。前提：

* 1. 在白纸上绘制一颗二叉树（噪声很小，结构化环境）
  2. 节点均为大写字母+圈组成。
  3. 关系均为箭头表示。
  4. 层次相对标准。
  5. 可以通过手绘图变成二叉树的内存表示

## 5.1 实验分析及思路

本实验要求使用目标检测和图像处理的方法检测手绘的一颗二叉树并且将其转换成二叉树数据结构存储起来。

可以先通过一个圆圈检测网络将图片中所有圆圈检测并分割成若干个节点图片，再将其图片、位置信息送入手写字母识别网络，得到每个节点的字母与位置信息。通过位置信息将其还原成二叉树，并且存储在内存中。

## 实验过程

本实验使用EMNIST手写数据集训练字母识别网络。EMNIST数据集是一个用于手写字符识别的数据集，它是基于美国国家标准与技术研究所（NIST）提供的NIST Special Database 19和Special Database 20而构建的。EMNIST的全称是Extended MNIST，它扩展了经典的MNIST数据集，提供了更多的样本和类别。EMNIST数据集包含了62个类别，其中包括10个数字（0-9）、26个大写字母（A-Z）和26个小写字母（a-z）。每个样本是一个28x28像素的灰度图像，表示手写字符。这些手写字符来自不同的人群，具有一定的变化和多样性，从而更贴近实际应用场景。下载数据集后进行预处理，并且划分为训练集和测试集，具体代码如下。

1. **print**(f"All classes: {cdata.classes}")
2. **print**(f"Data size: {cdata.data.shape}")
4. # Transforming to 4D tensor for conv layers
5. images = cdata.data.view([124800, 1, 28, 28]).float()
6. **print**(f"Tensor shape: {images.shape}")
8. torch.sum(cdata.targets == 0), len(torch.unique(cdata.targets))
10. **import** copy
12. letterCategories = cdata.classes[1:]
13. labels = copy.deepcopy(cdata.targets) - 1
14. **print**(labels.shape)
15. **print**(torch.sum(labels == 0))
17. **print**(torch.sum(labels == 0))
18. #%%
19. # Before normalizing
20. plt.hist(images[:10, :, :, :].view(1, -1).detach(), 40)
21. plt.title("Raw Values")
22. plt.show()
24. # Normalize
25. images /= torch.max(images)
27. # After normalizing
28. plt.hist(images[:10, :, :, :].view(1, -1).detach(), 40)
29. plt.title("Normalized Values")
30. plt.show()
32. X\_train, X\_test, y\_train, y\_test = train\_test\_split(images, labels, test\_size=0.01)
34. train\_data = TensorDataset(X\_train, y\_train)
35. test\_data = TensorDataset(X\_test, y\_test)
36. batch\_size = 128
37. train\_dl = DataLoader(train\_data, batch\_size=batch\_size, shuffle=True, drop\_last=True)
38. test\_dl = DataLoader(test\_data, batch\_size=len(test\_data))

使用Pytorh框架构建分类模型，具体代码如下。

1. **class** EMNISTNet(nn.Module):
2. **def** \_\_init\_\_(self, print\_toggle):
3. super().\_\_init\_\_()
4. self.print\_toggle = print\_toggle
6. # Conv1
7. self.conv1 = nn.Conv2d(in\_channels=1, out\_channels=64, kernel\_size=3, padding=1)
8. self.bnorm1 = nn.BatchNorm2d(num\_features=64)
10. # Conv2
11. self.conv2 = nn.Conv2d(in\_channels=64, out\_channels=128, kernel\_size=3)
12. self.bnorm2 = nn.BatchNorm2d(num\_features=128) # Input: number of channels
14. # Conv3
15. self.conv3 = nn.Conv2d(in\_channels=128, out\_channels=256, kernel\_size=3)
16. self.bnorm3 = nn.BatchNorm2d(num\_features=256) # Input: number of channels
18. self.fc1 = nn.Linear(in\_features=2\*2\*256, out\_features=256)
19. self.fc2 = nn.Linear(in\_features=256, out\_features=64)
20. self.fc3 = nn.Linear(in\_features=64, out\_features=26)
22. **def** forward(self, x):
23. **if** self.print\_toggle:
24. **print**(f"Input: {list(x.shape)}")
26. # First Block: conv -> max\_pool -> bnorm -> relu
27. x = F.max\_pool2d(self.conv1(x), 2)
28. x = F.leaky\_relu((self.bnorm1(x)))
29. x = F.dropout(x, p=0.25, training=self.training)
30. **if** self.print\_toggle:
31. **print**(f"First Block: {list(x.shape)}")
33. # Second Block: conv -> max\_pool -> bnorm -> relu
34. x = F.max\_pool2d(self.conv2(x), 2)
35. x = F.leaky\_relu((self.bnorm2(x)))
36. x = F.dropout(x, p=0.25, training=self.training)
37. **if** self.print\_toggle:
38. **print**(f"Second Block: {list(x.shape)}")
40. # Third Block: conv -> max\_pool -> bnorm -> relu
41. x = F.max\_pool2d(self.conv3(x), 2)
42. x = F.leaky\_relu((self.bnorm3(x)))
43. x = F.dropout(x, p=0.25, training=self.training)
44. **if** self.print\_toggle:
45. **print**(f"Second Block: {list(x.shape)}")
47. # Reshape for linear layer
48. n\_units = x.shape.numel() / x.shape[0]
49. x = x.view(-1, int(n\_units))
50. **if** self.print\_toggle:
51. **print**(f"Vectorized: {list(x.shape)}")
53. # Linear layers
54. x = F.leaky\_relu(self.fc1(x))
55. x = F.dropout(x, p=0.5, training=self.training)
56. x = F.leaky\_relu(self.fc2(x))
57. x = F.dropout(x, p=0.5, training=self.training)
58. x = self.fc3(x)
59. **if** self.print\_toggle:
60. **print**(f"Final Output: {list(x.shape)}")
62. **return** x
64. **def** make\_the\_model(print\_toggle):
66. model = EMNISTNet(print\_toggle)
68. loss\_fun = nn.CrossEntropyLoss()
69. optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001)
71. **return** model, loss\_fun, optimizer

训练代码如下。

1. **def** train\_the\_model():
2. epochs = 10
3. model, loss\_fun, optimizer = make\_the\_model(False)
5. model = model.to(device)
7. train\_loss = torch.zeros(epochs)
8. test\_loss = torch.zeros(epochs)
9. train\_acc = torch.zeros(epochs)
10. test\_acc = torch.zeros(epochs)
12. **for** epoch\_i **in** tqdm(range(epochs)):
13. batch\_loss = []
14. batch\_acc = []
16. model.train()
17. **for** X, y **in** tqdm(train\_dl):
18. X, y = X.to(device), y.to(device)
20. y\_hat = model(X)
21. loss = loss\_fun(y\_hat, torch.squeeze(y))
22. batch\_loss.append(loss.item())
24. optimizer.zero\_grad()
25. loss.backward()
26. optimizer.step()
28. acc = torch.mean((torch.argmax(y\_hat, axis=1) == y).float()).item()
29. batch\_acc.append(acc)
31. train\_acc[epoch\_i] = np.mean(batch\_acc)
32. train\_loss[epoch\_i] = np.mean(batch\_loss)
34. model.eval()
35. X, y = next(iter(test\_dl))
36. X, y = X.to(device), y.to(device)
37. with torch.no\_grad():
38. y\_hat = model(X)
39. loss = loss\_fun(y\_hat, torch.squeeze(y))
40. acc = torch.mean((torch.argmax(y\_hat, axis=1) == y).float()).item()
42. test\_acc[epoch\_i] = acc
43. test\_loss[epoch\_i] = loss.item()
45. **return** train\_loss, test\_loss, train\_acc, test\_acc, model

## 实验结果及分析

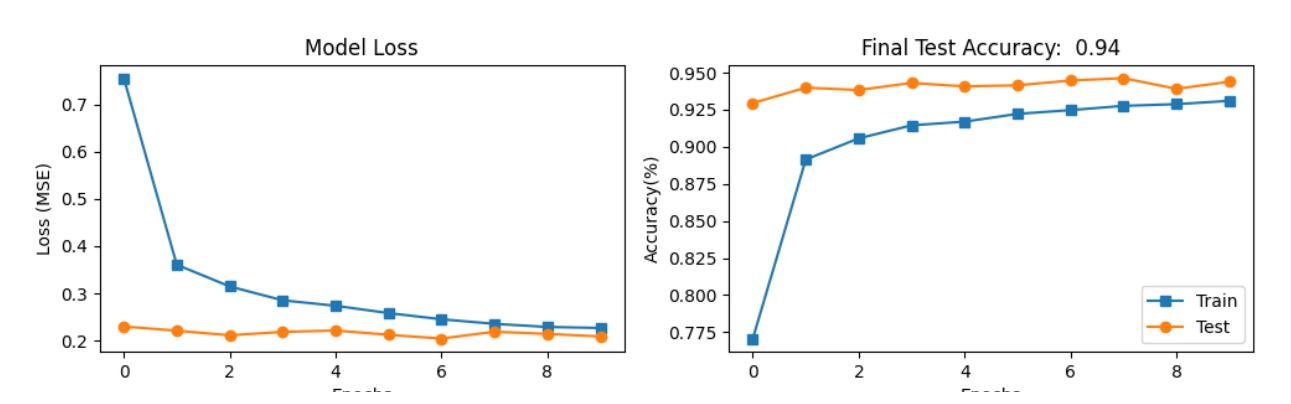


图5.五‑1 模型训练结果

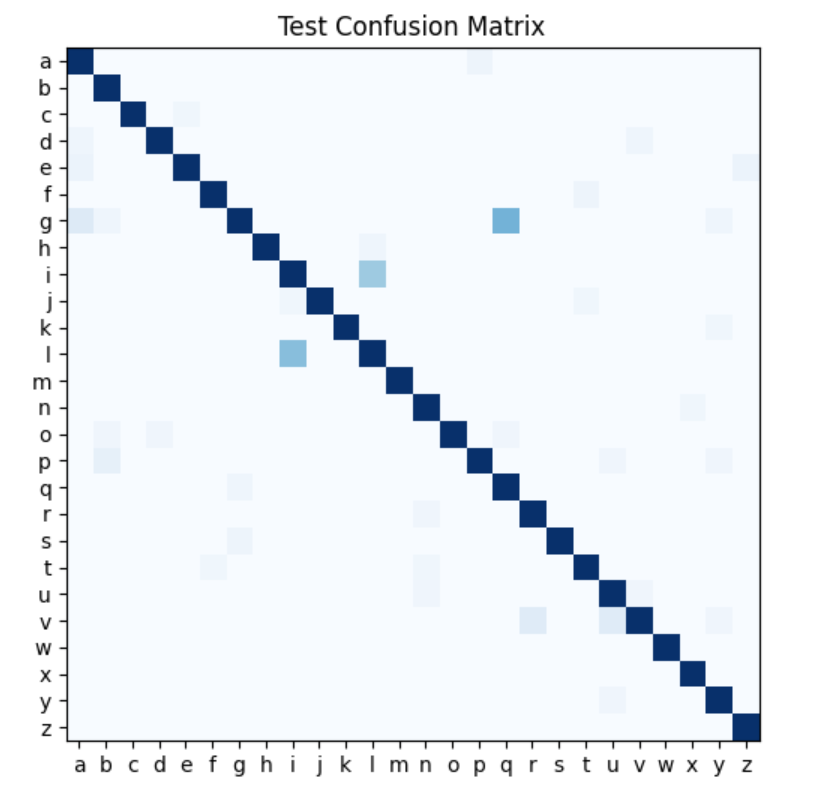


图 5.2 26个字母混淆矩阵

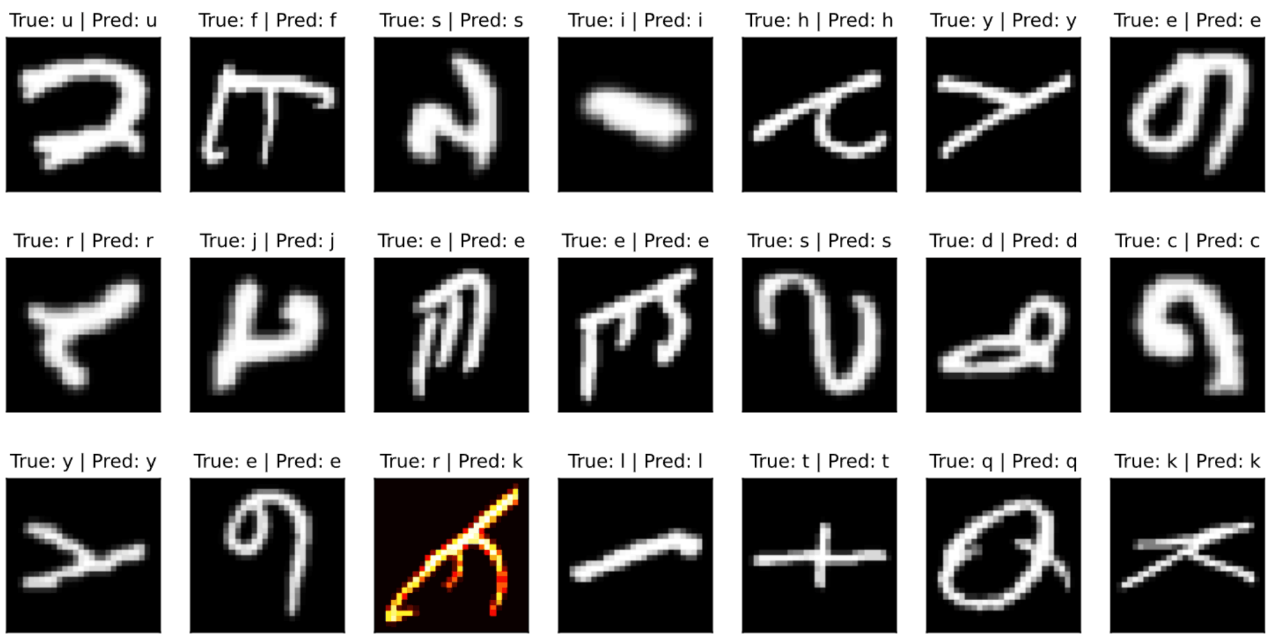


图 5.3 识别示例

# 总结

本学期的深度学习实验主要关注多层感知机、神经网络、现代卷积网络的模型构建， 其中包括了图像分类、手写数字识别、手写字母识别等多个部分。通过本次实验，我不仅深入理解了深度学习技术的应用和原理，还掌握了开发框架Pytorch的使用方法和技巧。

在实验一中，我从零开始构建了一个多层感知机模型，利用Numpy库逐步编写了激活函数、损失函数、前向计算、反向传播等过程，对于多层感知机模型的原理有了进一步的认识与理解，并将其应用于Iris鸢尾花数据集分类中，取得比较好的效果。

在实验二中，我利用Pytorch框架中的卷积层、池化层搭建了一个简单的卷积神经网络，并且通过对学习率、Dropout概率、迭代次数等超参数的不同组合训练分析了超参数的选择策略。

在实验三中，我利用Pytorch框架构建了经典的残差模块，并通过调整残块模块的堆叠高度获得了不同层数的ResNet。在MNIST数据集上观察了不同层数ResNet的训练效果，并且使用训练后的模型做了预测任务。

在提高实验中，我们面对一个新的问题时通过上网查询资料、组内讨论等方式提出了许多方案。虽然在实施过程中遇到了各种各样的问题，其中最重要的问题就是实验数据集难以获取。为了解决这个问题我们选择将问题拆分了两个子问题：圆圈分割、字母识别。虽然最后没能够完全实现题目要求，但是在不断求索的过程中我们收获了很多知识，也提高了动手解决问题的能力。

在本次实验的过程中，我遇到了很多课堂上没有提及的问题，通过查阅资料、咨询老师、请教同学等方法最终克服了一个个困难，最终完成了本次 深度学习实验。也让我对于平时所学的理论知识有了更多的思考与理解。

总的来说，本学期的深度学习实验让我更加全面、深入地了解和掌握 了深度学习技术的应用和发展，不仅提升了我的理论能力，还增强了我的实践能力，为我未来的研究和工作打下了坚实的基础。

最后，非常感谢老师一直以来的指导！

## 参考文献

[1] [Focal Loss for Dense Object Detection.](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P3exm-VbkOyc-pO-ZYCbPbTwvumivUdMWFxQpgPmvuQOe9bXmCSyn2pzZe-lJDrcBF3xCAUNQ-VGp5qWxN1w2HM_cgsExhEcX0ZIpOPF8ebbQu1SDnonBnkMWnhn7VZg0eMfKw5uVH18rAanbfR9327pHJvJ8aa9Bg=&uniplatform=NZKPT&language=CHS). Lin Tsung-Yi;;Goyal Priya;;Girshick Ross;;He Kaiming;;Dollar Piotr.IEEE transactions on pattern analysis and machine intelligence,2020

[2] [Leveraging Prior-Knowledge for Weakly Supervised Object Detection Under a Collaborative Self-Paced Curriculum Learning Framework](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P3exm-VbkOyc-pO-ZYCbPbTwvumivUdMWHkm3VOHR8yPaVCFysWn-u3YiijaDTSjpUO6bjDYCunG155tl3fTSGN9KzBYn6hMFvi8lIzzokYC6zot8DgXQuY6HZ1zuOR1mE0CfIjzhnjiweLzp-gCFo84QB0H0IOczs=&uniplatform=NZKPT&language=CHS). Dingwen Zhang;Junwei Han;Long Zhao;Deyu Meng.International Journal of Computer Vision,2019

[3] [Additive Margin Softmax for Face Verification](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P3exm-VbkOyc-pO-ZYCbPbTwvumivUdMWExFqCmXAmlhol7-o3u9IHojNGKTVV4adlbVw03XZp5aIhffAvw9XhVWKKdCNX-vZfPvpPGIQjlvd-PY4Gl89vf6ApUQhHbjDePImXZQtfxK3gw7YQFjMEDQB8AU0fdWXo=&uniplatform=NZKPT&language=CHS). Wang Feng;Cheng Jian;Liu Weiyang;Liu Haijun.IEEE Signal Processing Letters,2018

[4] [Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P3exm-VbkOyc-pO-ZYCbPbTwvumivUdMWFxQpgPmvuQOSxxTFWbf0RRsZgDwkQ4ZDnbwFO0m4f6eM2qe0J2mra3ALkSllWJ-zNcvQxWRNvARWBGc4-RJV0Ug6JZZJI_0gjXFPNoWf8uFy0kUfxzjrQw8F0PP6BLF1M=&uniplatform=NZKPT&language=CHS). Ren Shaoqing;;He Kaiming;;Girshick Ross;;Sun Jian.IEEE transactions on pattern analysis and machine intelligence,2017

[5] [Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning.](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P3exm-VbkOyc-pO-ZYCbPbTwvumivUdMWG_8pIOpXC1nKcj4X7upp3_9tB4f_3GT1qVqS3qtuFCPEan1SH17FVggQREhIIhyh4Gd3afKM8x-TmSYDQJQXvxWmiO3dAriDCSqCpOA9ixZ-xpbx82KgLyqzMQggeDpHo=&uniplatform=NZKPT&language=CHS). Pavlo Molchanov;Stephen Tyree;Tero Karras;Timo Aila;Jan Kautz.CoRR,2016

[6] [Learning Deep Embeddings with Histogram Loss.](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P2Yl_GPfuGoFt-QzzN7Md3AYpMBp0MLLH3KEvlYH340Xq0KkfTZua3kJJhxpduvUJR5mBfhyi9aRPQHwt7VTeGtyBsNUyQLVtU_XkarIKRMq2XiaSIDNxqISw-5xNITukJQal4N9LZaR3QQBwJV8meE9WbLlfrpZAo=&uniplatform=NZKPT&language=CHS). Evgeniya Ustinova;Victor S. Lempitsky.CoRR,2016

[7] [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P2Yl_GPfuGoFt-QzzN7Md3AYpMBp0MLLH3KEvlYH340XoAj6L4CgaT0NucXqVRPwD_FbvSlhiPDovk6AfvYw6DfPRSRaoQdADC5VH8ilTVnNQB-HPNE-EZuvu0zexSCV4ULn3oiUrFVWEvEaNM_3SqZc-SoWGt0_pM=&uniplatform=NZKPT&language=CHS). Sergey Ioffe;Christian Szegedy.CoRR,2015

[8] [Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P2Yl_GPfuGoFt-QzzN7Md3AYpMBp0MLLH3KEvlYH340XhxEl8ODdatlkom2HtKrsLU0gZDqH1wibAiaaxUjzf5YGCciTA2fOKYGgvqe0a6iZxRMZFZI51BTjk4-1ZjbwiLYxh6lEPePkvd-ErB3Ji0X0SJRPatYqDc=&uniplatform=NZKPT&language=CHS). Djork-Arné Clevert;Thomas Unterthiner;Sepp Hochreiter.CoRR,2015

[9] [Rethinking the Inception Architecture for Computer Vision.](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P2Yl_GPfuGoFt-QzzN7Md3AYpMBp0MLLH3KEvlYH340XhIZp0fU1y1CKCYwi2zBlxzYuIOnEICteqN8J0AmA_34mS-2iBubFu6pZiRbmeChPNaVLKB-MmOsH7rlC67tJIt2j8fzLQYZDtOv-K03vOmV_Ym2df-ZtXI=&uniplatform=NZKPT&language=CHS). Christian Szegedy;Vincent Vanhoucke;Sergey Ioffe;Jonathon Shlens;Zbigniew Wojna.CoRR,2015

[10] [Highway Networks.](https://kns.cnki.net/kcms2/article/abstract?v=Jk1LZv7y6P2Yl_GPfuGoFt-QzzN7Md3AYpMBp0MLLH3KEvlYH340XiCVmwXvqtY-SgGvvZ0gZepaXZkLvD_0CcEoZDGPwlKE4o1F3cafgWe9VmoZZFTAKFr5axrhs2ceUOi8MoMaIHL5gjMAfKu70RpQlGu8BxUMMzhJKm7XhVs=&uniplatform=NZKPT&language=CHS). Rupesh Kumar Srivastava;Klaus Greff;Jürgen Schmidhuber.CoRR,2015