机器视觉实验报告（二）

目录

[机器视觉实验报告（二） 1](#_Toc132903833)

[一. 实验目的 2](#_Toc132903834)

[二. 实验原理 2](#_Toc132903835)

[2.1 卷积层 2](#_Toc132903836)

[2.2 池化层 2](#_Toc132903837)

[2.3 全连接层 3](#_Toc132903838)

[2.4 激活函数 4](#_Toc132903839)

[2.5 损失函数 5](#_Toc132903840)

[三. 数据集 6](#_Toc132903841)

[四. 运行说明 6](#_Toc132903842)

[五. 结果说明 9](#_Toc132903843)

1. 实验目的

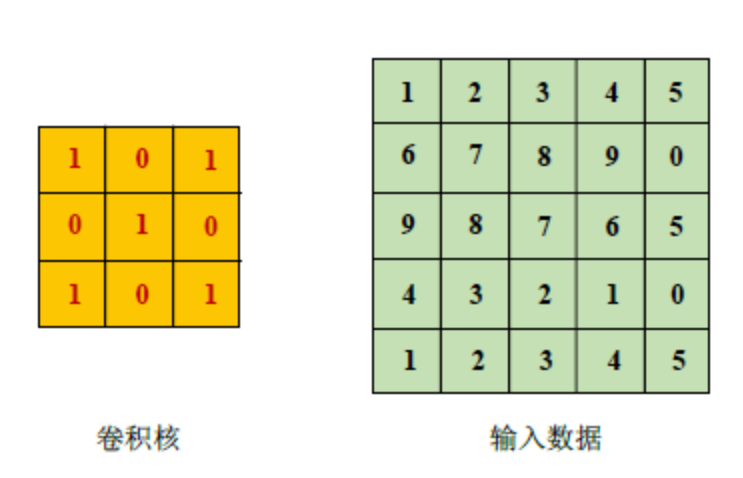
* 熟悉卷积神经网络的基本结构，包括卷积层，池化层，激活函数及全连接层等
* 学习经典神经网络 LeNet 及其他经典特征提取网络
* 使用MNIST数据集进行训练和预测

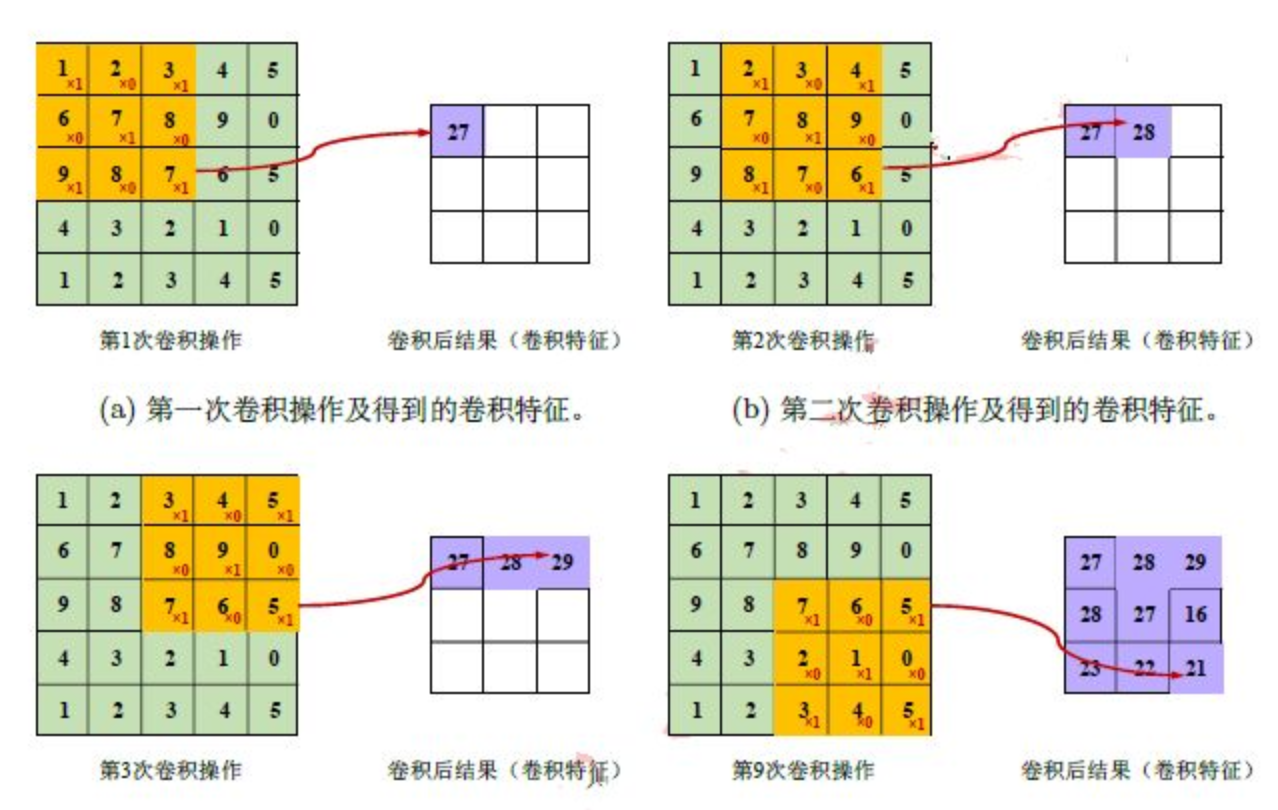
1. 实验原理
   1. 卷积层

卷积运算实际是分析数学中的一种运算方式，在卷积神经网络中通常是仅涉及离散卷积的情形。在深度学习神经网络中，卷积运算是一个重要的组成部分，用于对输入数据进行卷积运算并提取特征。它主要用于处理图像、语音和视频等信号数据，可以有效地提取数据中的局部特征和空间相关性。

在卷积层中，卷积核（也称为滤波器）通过滑动窗口的方式对输入数据进行卷积操作，得到输出特征图。卷积核的参数可以通过反向传播算法进行学习，从而使卷积层能够适应不同的数据集和任务。

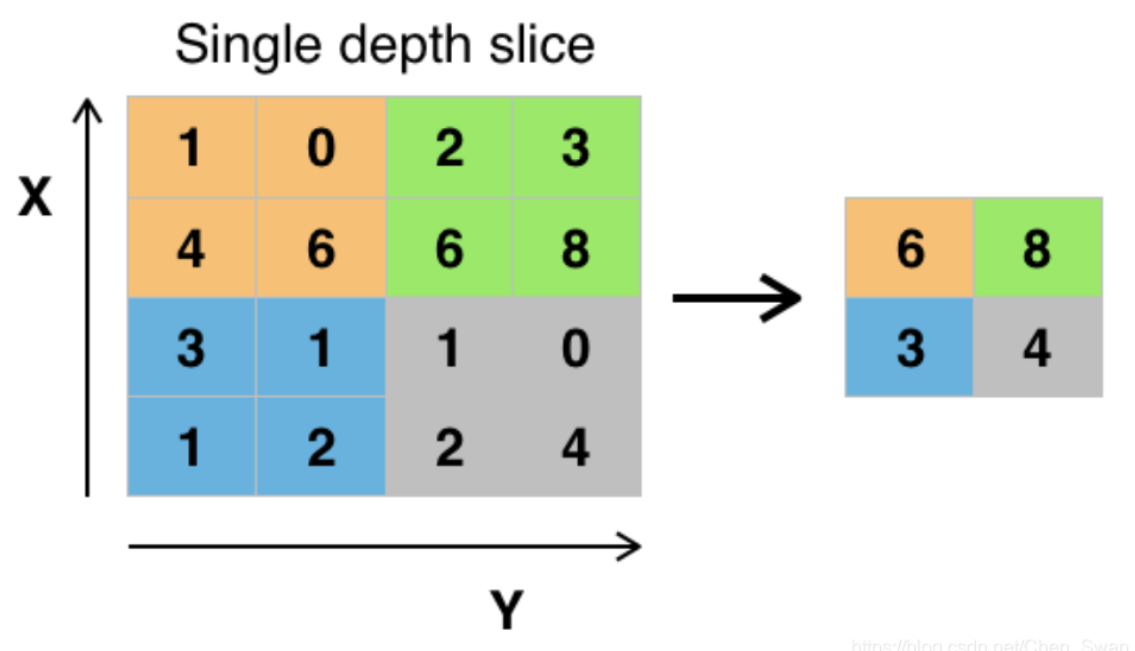
卷积层的主要作用是提取输入数据中的特征，这些特征可以用于后续的分类、回归、目标检测等任务。在图像处理任务中，卷积层可以检测图像中的边缘、角点、纹理等特征，从而帮助神经网络更好地理解图像内容。同时，卷积层还可以通过参数共享的方式减少神经网络中的参数数量，从而降低过拟合的风险。





* 1. 池化层

池化层实际上是一种形式的降采样。有多种不同形式的非线性池化函数，而其中最为常见的有最大池化和平均池化。它是将输入的图像划分为若干个矩形区域，对每个子区域输出最大值或平均值。直觉上，这种机制能够有效地原因在于，在发现一个特征之后，它的精确位置远不及它和其他特征的相对位置的关系重要。池化层会不断地减小数据的空间大小，因此参数的数量和计算量也会下降，这在一定程度上也控制了过拟合。通常来说，CNN的卷积层之间都会周期性地插入池化层。如下图所示，为最大池化操作。



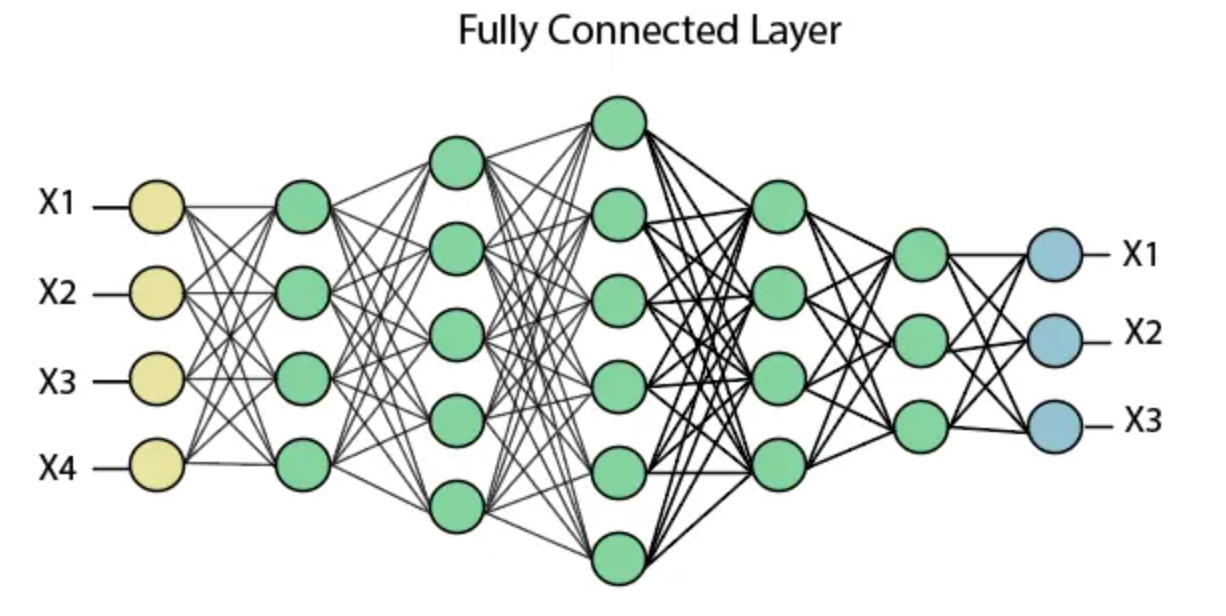
通过池化操作，可以降低输入数据的维度和特征数量，从而减少计算复杂度。此外，池化操作还可以帮助神经网络对输入数据进行平移不变性和缩放不变性的学习，从而提高神经网络的泛化能力。

* 1. 全连接层

全连接层是神经网络中最基本的层之一，通常被用来将前面的卷积层或池化层的输出展平成一个向量，然后将这个向量作为输入，传递给后面的一系列全连接层进行分类、回归等任务。

全连接层的主要作用是学习输入特征之间的非线性关系，使得模型可以对输入进行更复杂的分类或回归。具体来说，全连接层会将输入特征进行线性组合，然后通过一个非线性激活函数进行转换。这种转换可以帮助模型捕捉输入之间的复杂关系，从而提高模型的表现力和泛化能力。

在深度神经网络中，全连接层通常被用于模型的最后几层，用来将模型的输出映射到目标类别或回归值。由于全连接层的参数量较大，容易造成过拟合，因此在一些深度学习任务中，可以通过使用卷积神经网络等结构来减少全连接层的数量，从而提高模型的泛化能力。



* 1. 激活函数

激活函数是神经网络中非常重要的一部分，它的作用有以下几点：

1. 引入非线性

神经网络的每一层都是线性变换，如果没有激活函数的话，多个线性层叠加后仍然只能得到一个线性函数，无法对复杂的数据进行学习和表示。通过引入非线性的激活函数，神经网络可以学习更加复杂的函数，提高了网络的表达能力。

1. 实现特征映射

激活函数可以将输入的信号映射到另一个空间中，从而产生新的特征。例如，ReLU激活函数可以将负数映射为0，从而实现了特征的选择和稀疏性。

1. 提高网络的稳定性和收敛速度

激活函数可以使得神经网络更加稳定和鲁棒。例如，Sigmoid函数和Tanh函数都具有饱和性质，可以缓解梯度爆炸的问题，而ReLU函数可以缓解梯度消失的问题，并且加速网络的收敛速度。

1. 实现输出的归一化

在多分类问题中，Softmax函数可以将神经网络的输出转换为概率分布，使得输出值之和为1，从而方便了模型的解释和比较。

深度学习中常用的激活函数主要有以下几种：

1. Sigmoid函数

Sigmoid函数是一种常用的激活函数，将输入的值映射到0~1的范围内，具有平滑且单调递增的性质。但是，Sigmoid函数在输入值较大或较小时容易出现梯度消失的问题。

1. Tanh函数

Tanh函数也是一种常用的激活函数，将输入的值映射到-1~1的范围内，具有类似Sigmoid函数的性质，但是可以产生正负值的输出。但是，Tanh函数同样也存在梯度消失的问题。

1. ReLU函数

ReLU函数（Rectified Linear Unit）是一种常用的非线性激活函数，对于输入值大于0的部分直接输出，小于0的部分输出0。ReLU函数的计算速度很快，不会产生梯度消失的问题。但是，ReLU函数在输入值小于0的时候会出现“神经元死亡”的问题，即这些神经元将不再参与网络的训练。

1. LeakyReLU函数

LeakyReLU函数是对ReLU函数的一种改进，当输入值小于0时，不再输出0，而是输出一个非零的小值。这样可以解决ReLU函数的“神经元死亡”问题。

其中，α是一个小于1的超参数，通常取0.01。

1. Softmax函数

Softmax函数是一种常用的激活函数，将输入的多维向量映射到0~1的范围内，并且使所有元素之和为1。通常用于多分类问题中，将神经网络的输出转换为概率分布。

其中，n是向量的维度。

* 1. 损失函数

损失函数是神经网络中非常重要的一部分，它的作用是用来评估神经网络输出与真实值之间的差距。具体来说，损失函数是一个数值，表示网络的预测值与真实值之间的差异程度。训练过程的目标就是通过优化损失函数来使得预测值尽可能接近真实值。

在分类任务中，损失函数通常选择交叉熵损失函数，该损失函数可以衡量预测结果的正确性。对于二分类问题，常用的交叉熵损失函数为二元交叉熵损失函数（Binary Cross Entropy），对于多分类问题，常用的交叉熵损失函数为分类交叉熵损失函数（Categorical Cross Entropy）。

* 1. 优化器

优化器通过调整神经网络中的参数来最小化损失函数，从而提高网络的性能和泛化能力。优化器的主要作用包括以下几个方面：

1. 参数更新

优化器通过更新神经网络中的参数来使得损失函数尽可能小。更新参数的方法通常是基于梯度的优化算法，包括梯度下降、随机梯度下降、Adam等等。

1. 学习率调整

学习率是优化器中一个重要的超参数，它决定了参数更新的步长和速度。在训练过程中，学习率的调整非常重要，因为不合适的学习率会导致训练不稳定或者陷入局部最优解。优化器通常会提供不同的学习率调整策略，例如学习率衰减、自适应学习率等。

1. 正则化

优化器也可以实现一些正则化方法，例如L1正则化、L2正则化等。这些方法可以帮助减小模型的复杂度，提高模型的泛化能力。

1. 批量处理

优化器通常会支持批量处理，即将训练数据分成多个小批量，每个小批量作为一次训练的输入。批量处理可以提高训练效率和稳定性。

总之，优化器在神经网络中扮演着非常重要的角色，通过调整参数来最小化损失函数，提高网络的性能和泛化能力。不同的优化器在不同的任务中表现不同，需要根据具体情况选择合适的优化器和超参数。

1. 数据集

MNIST数据集是一个经典的手写数字图像数据集，包含60000张训练图像和10000张测试图像，每张图像大小为28x28像素，以灰度图像形式存储。MNIST数据集中的每个图像都表示一个0到9之间的数字。训练集和测试集都包含对应的标签，即每个图像表示的数字。



1. 运行说明

class LeNet(nn.Module):

def \_\_init\_\_(self):

super(LeNet, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 6, kernel\_size=5, padding=2)

self.maxpool1 = nn.MaxPool2d(kernel\_size=2, stride=2)

self.conv2 = nn.Conv2d(6, 16, kernel\_size=5)

self.maxpool2 = nn.MaxPool2d(kernel\_size=2, stride=2)

self.fc1 = nn.Linear(16\*5\*5, 120)

self.fc2 = nn.Linear(120, 84)

self.fc3 = nn.Linear(84, 10)

def forward(self, x):

x = self.maxpool1(torch.relu(self.conv1(x)))

x = self.maxpool2(torch.relu(self.conv2(x)))

x = x.view(-1, 16\*5\*5)

x = torch.relu(self.fc1(x))

x = torch.relu(self.fc2(x))

x = self.fc3(x)

return x

基于pytorch的LeNet实现，对于传入的图像经过两次卷积、激活、最大池化，然后传入三个线性层进行输出，最终输出的结果为传入图像在十个分类下各自的概率。

train\_transform = Compose([

ToTensor(),

Normalize((0.1307,), (0.3081,))

])

train\_data = torchvision.datasets.MNIST(root="./data",train=True,transform=train\_transform, download=True)

test\_data = torchvision.datasets.MNIST(root="./data",train=False,transform=train\_transform,download=True)

train\_loader = DataLoader(train\_data, batch\_size=config['batch\_size'], shuffle=True, pin\_memory=True)

test\_loader = DataLoader(test\_data, batch\_size=config['batch\_size'], pin\_memory=True)

这里首先简单定义了图像增强方式，由于数字的特殊性质，因此不能进行翻转等操作，只能简单的转换为tensor类型，并根据数据的均值和方差进行归一化，以加速计算。使用pytorch内置的MNIST数据集进行加载并下载到本地方便后续处理，并设置batch\_size等参数。

criterion = nn.CrossEntropyLoss(label\_smoothing=0.3)

def trainer(model, trainloader, testloader, writer, criterion, config):

    device = config['device']

    model = model.to(device)

    criterion = criterion.to(device)

    optimizer = optim.Adam(model.parameters(), lr=config['learning\_rate'], weight\_decay=1e-4)

    last\_loss = -10005

    best\_accuracy = 0

    batches = len(trainloader.dataset)//config['batch\_size']

    for epoch in range(config['epochs']):

        model.train()

        for batch, (imgs, labels) in enumerate(trainloader):

            imgs, labels = imgs.to(device), labels.to(device)

            optimizer.zero\_grad()

            preds = model(imgs)

            loss = criterion(preds, labels)

            loss.backward()

            optimizer.step()

            if last\_loss<-10000:

                last\_loss = loss.item()

            else:

                last\_loss = last\_loss\*0.99 + loss.item()

            writer.add\_scalar("Loss/Train", last\_loss, batch+epoch\*batches)

        valid\_pred\_true, valid\_set\_size = 0, 0

        test\_loss\_sum = 0.

        whole\_pred = torch.tensor([]).to(device)

        whole\_target = torch.tensor([]).to(device)

        for imgs, labels in testloader:

            imgs, labels = imgs.to(device), labels.to(device)

            with torch.no\_grad():

                pred = model(imgs)

                loss = criterion(pred, labels)

                \_, predicted = torch.max(pred.data, dim=1)

                whole\_pred = torch.cat((whole\_pred, predicted))

                whole\_target = torch.cat((whole\_target, labels))

            for i in range(len(labels)):

                if (int(predicted[i]) == int(labels[i])):

                    valid\_pred\_true += 1

                valid\_set\_size += 1

            test\_loss\_sum += loss.item()

        accuracy = valid\_pred\_true / valid\_set\_size

        if best\_accuracy<accuracy:

            torch.save(model.state\_dict(), config['save\_path'])

        writer.add\_scalar("Loss/Test", last\_loss, epoch)

        writer.add\_scalar('Results/Precision', accuracy, epoch)

        print("epoch: {:}, Train Loss = {:.5f}, Test Loss = {:.5f}, Precision: {:.5f}".format(epoch, last\_loss, test\_loss\_sum, accuracy))

trainer函数负责模型的训练，设置损失函数为交叉熵损失函数，并使用label\_smoothing增强模型的鲁棒性，使用Adam作为优化器，设置初始学习率为0.001。在记录loss时，对loss进行平滑处理，取前一次的loss的99%加上这次loss的1%以得到较为平滑的loss曲线方便后续分析。

在一个epoch训练完成之后，使用测试集进行测试并保存准确率较高的模型。最后打印这个epoch训练时的训练损失、测试损失以及测试准确率。

all\_preds = []

all\_targets = []

model.load\_state\_dict(torch.load('mnist.pt'))

model = model.to(config['device'])

with torch.no\_grad():

    for images, labels in test\_loader:

        images = images.to(config['device'])

        labels = labels.to(config['device'])

        preds = model(images)

        all\_preds.extend(torch.argmax(preds, dim=1).cpu().numpy())

        all\_targets.extend(labels.cpu().numpy())

conf\_mat = confusion\_matrix(all\_targets, all\_preds)

print(conf\_mat)

import matplotlib.pyplot as plt

import seaborn as sns

classes = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

plt.figure(figsize=(8, 8))

sns.heatmap(conf\_mat, annot=True, cmap='Blues', xticklabels=classes, yticklabels=classes, fmt='g')

plt.xlabel('Predicted Label')

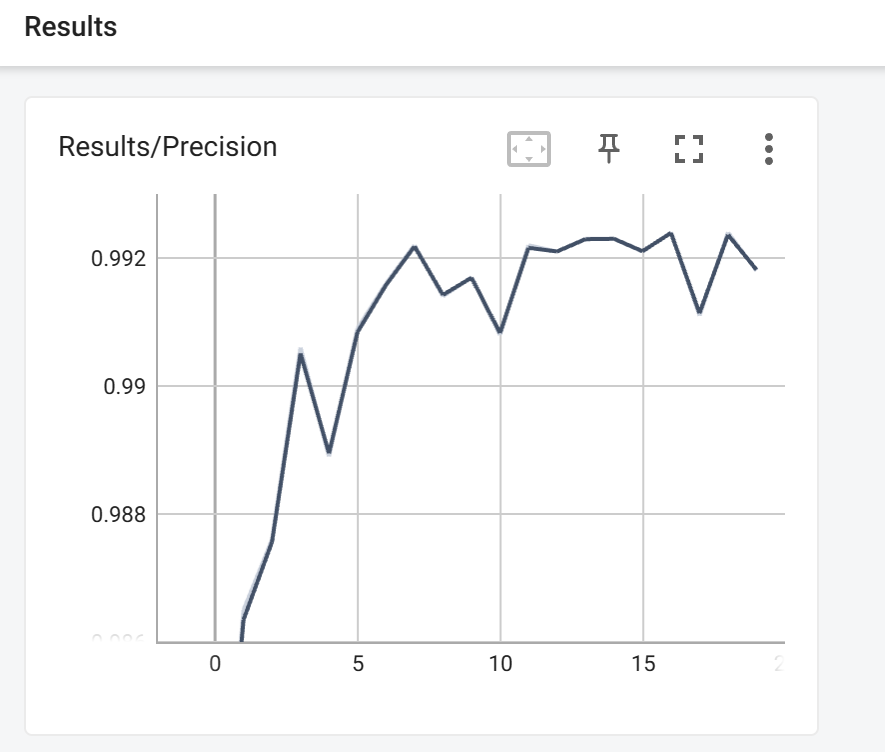
plt.ylabel('True Label')

plt.savefig('confusion\_matrix.jpg')

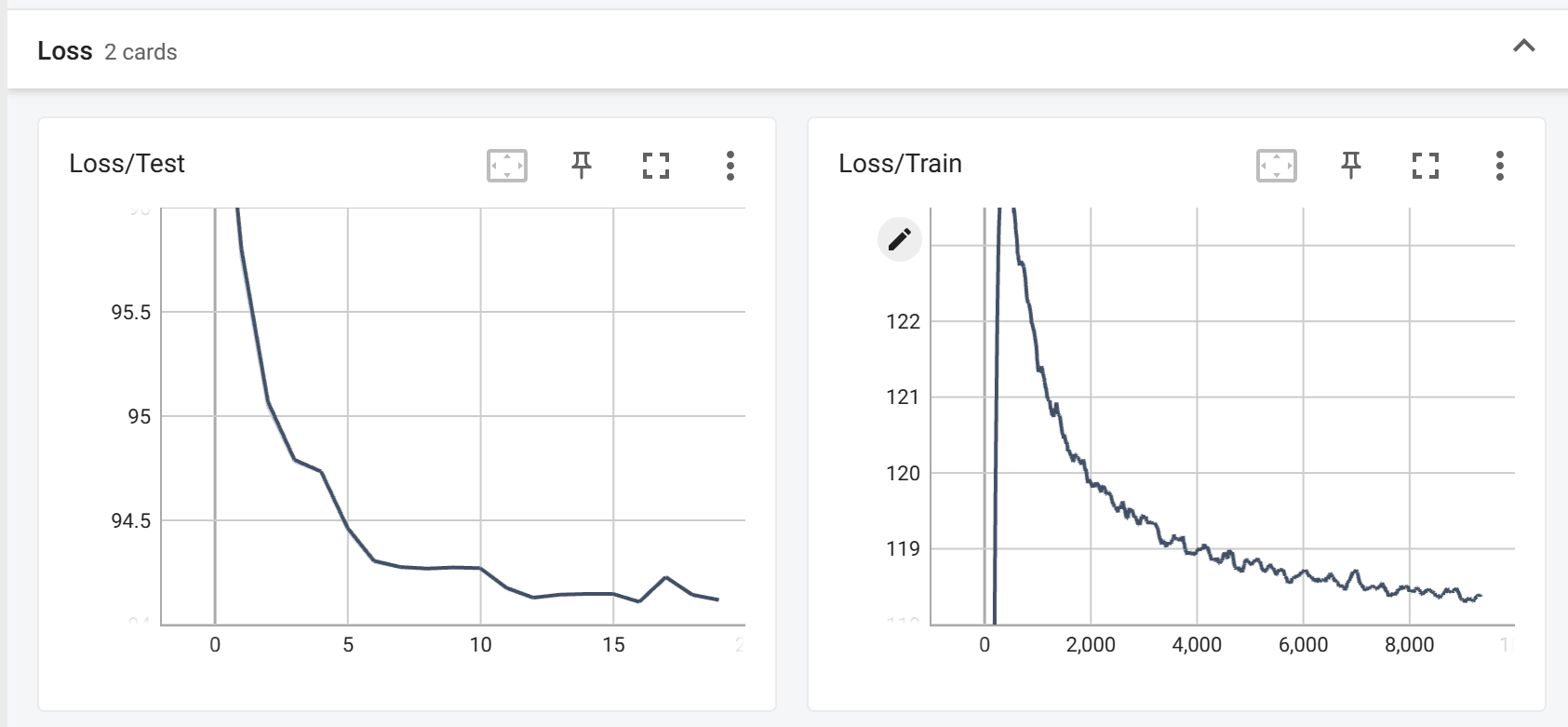
在模型训练完成之后，使用模型在测试集上的预测结果和真实标签绘制混淆矩阵以分析模型。

1. 结果说明

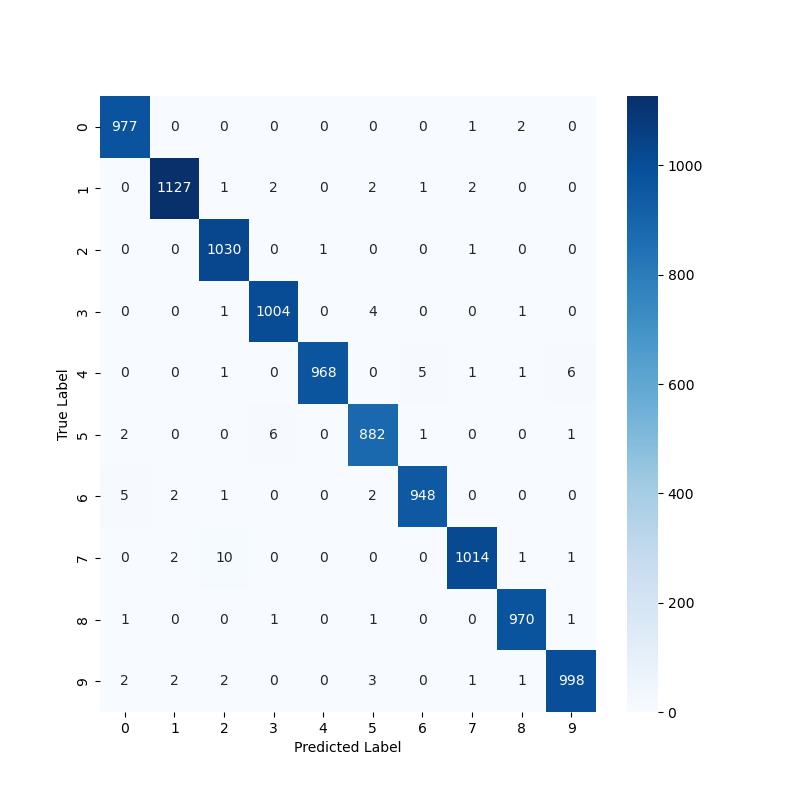
通过20个epoch的训练，我们使用LeNet模型在mnist得到最高99.24%的准确率。



测试集准确率



损失曲线



混淆矩阵

通过混淆矩阵可以清晰看到模型在分辨2和7、3和5、4和9存在不足。后续可以根据这些问题进行改进。