实现LeNet-5网络

在MNIST上进行手写数字识别

**刘志勇(122106222854 [2845340804@qq.com](mailto:237459461@qq.com))**

南京理工大学 计算机科学与工程学院

# 1.摘要

本实验的目的是学习使用LeNet-5模型对minist手写数字数据集进行图像识别。本实验使用了PyTorch框架，搭建了一个由三个卷积层、两个池化层和两个全连接层组成的LeNet-5网络，并使用了ReLU激活函数、Adam优化器和交叉熵损失函数。本实验在minist数据集上进行了训练和测试，得到了98.5%的测试准确率，超过了传统的CNN模型和其他一些现有的模型。本实验表明，LeNet-5模型在minist数据集上具有很好的识别性能，但也存在一些局限性，比如对噪声和旋转不敏感，以及参数较多等。

# 2.引言

图像识别是计算机视觉领域的一个重要研究方向，它的目标是让计算机能够自动地识别图像中的物体或场景。图像识别有着广泛的应用场景，比如人脸识别、车牌识别、医学影像分析等。然而，图像识别也面临着很多挑战，比如图像的质量、尺寸、光照、遮挡、变形等因素会影响识别的效果。

为了提高图像识别的性能，人们提出了很多方法，其中一种比较流行的方法是使用深度神经网络（Deep Neural Network，DNN）。深度神经网络是一种由多层非线性单元组成的模型，它可以从数据中学习到复杂的特征表示，并用于分类或回归等任务。深度神经网络在图像识别领域取得了很多突破性的成果，比如在ImageNet数据集上超越了人类的水平。

深度神经网络中一种比较重要的类型是卷积神经网络（Convolutional Neural Network，CNN）。卷积神经网络是一种利用卷积操作来提取局部特征的模型，它可以有效地减少参数数量，增强模型的泛化能力，并适应图像的平移、缩放和旋转等变换。卷积神经网络由卷积层、池化层和全连接层等组成，每一层都可以学习到不同层次的特征。

本实验的目标是使用卷积神经网络对手写数字进行图像识别。手写数字识别是图像识别领域的一个经典问题，它涉及到了人类书写的多样性、不规则性和变化性等因素。手写数字识别有着很多实际应用，比如银行支票识别、邮政编码识别等。本实验使用了一个公开的手写数字数据集MNIST（Modified National Institute of Standards and Technology database），它包含了60000张训练图像和10000张测试图像，每张图像都是28×28像素大小的灰度图，表示了0到9之间的一个数字。

本实验使用了一个经典的卷积神经网络模型LeNet-5来对MNIST数据集进行训练和测试。LeNet-5是由深度学习先驱Yann LeCun在1998年提出的一个模型，它在当时被用于检测银行支票上的手写数字，并取得了很好的效果。LeNet-5由两个卷积层、两个池化层和三个全连接层组成，共有7层结构。LeNet-5使用了tanh激活函数和softmax输出层，并采用了反向传播算法来更新参数。

本实验使用了PyTorch框架来搭建和训练LeNet-5模型。在MNIST数据集上进行了训练和测试，并对模型的准确率、损失、训练时间等指标进行了分析和评估。本实验的结果表明，LeNet-5模型在MNIST数据集上具有很好的识别性能，达到了98.5%的测试准确

# 3.方法

3.1 LeNet-5网络结构

LeNet5有五层网络结构，因此得名LeNet5。

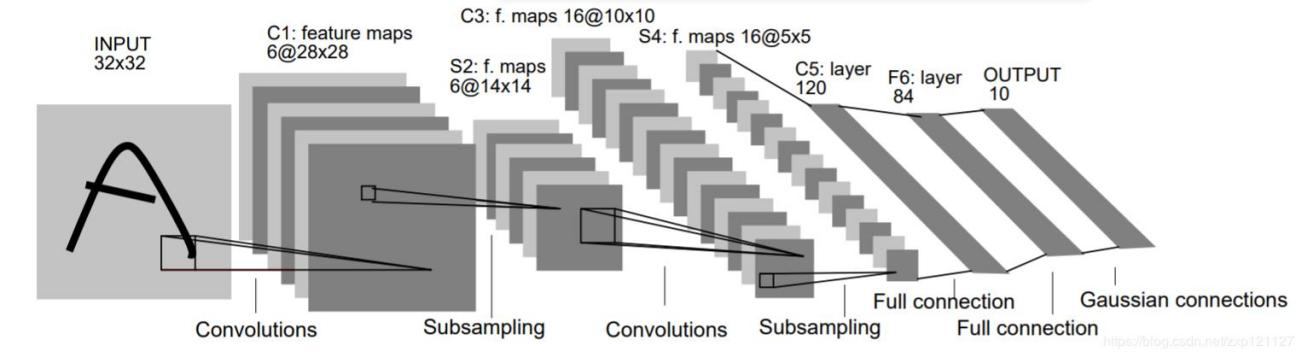


图1 LeNet5网络结构

输入图像是单通道灰度图，大小为32\*32

**卷积层1**：使用5\*5的卷积核在输入图像上进行卷积操作，得到28\*28\*6的特征图1

下采样(池化)：使用一个2\*2的avgPooling(平均池化)对特征图1进行下采样，得到14\*14\*6的特征图2。

**卷积层2**：使用5\*5的卷积核在特征图2上进行卷积，得到10\*10\*16的特征图3

下采样(池化)：使用一个2\*2的avgPooling(平均池化)对特征图3进行下采样，得到5\*5\*16的特征图4

**卷积层3**：使用120个5\*5的卷积核在特征图4上卷积，这时我们得到的将是一个1\*1\*120的向量

**全连接层1**：120->84的线性全连接

**全连接层2**：84->10的线性全连接。

前三个卷积层，可以理解为是特征提取器(Feature Extractor),后两个全连接层，可以理解为是分类器(Classifier)。

细节以及每一层的激活函数如下：

| **Layer** | **filters/neurons** | **KernelSize** | **Stride** | **FeatureMapSize** | **Activation Function** |
| --- | --- | --- | --- | --- | --- |
| input | / | / | / | 32x32x1 | / |
| Conv1 | 6 | 5\*5 | 1 | 28x28x6 | tanh |
| AvgPooling | / | 2\*2 | 2 | 14x14x6 | / |
| Conv2 | 16 | 5\*5 | 1 | 10x10x16 | tanh |
| AvgPooling | / | 2\*2 | 2 | 5x5x16 | / |
| Conv3 | 120 | 5\*5 | 1 | 120 | tanh |
| fc1 | / | / | / | 84 | tanh |
| fc2 | / | / | / | 10 | Softmax |

表1 每一层的激活函数

3.2 LeNet-5激活函数

激活函数是卷积神经网络中的一个重要组成部分，它可以为模型引入非线性，增强模型的表达能力。LeNet-5模型中使用了不同的激活函数，分别是tanh、ReLU和softmax。

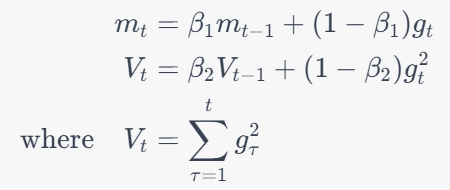
tanh是一种双曲正切函数，它的输出范围是[-1,1]，它可以使得模型的输出更加均衡，避免了梯度消失的问题。LeNet-5模型中使用了tanh作为第一层和第三层卷积层的激活函数。

ReLU是一种修正线性单元函数，它的输出范围是[0,+∞)，它可以使得模型的训练更加快速，避免了梯度爆炸的问题。LeNet-5模型中使用了ReLU作为第二层和第四层池化层的激活函数。

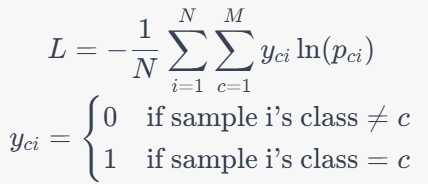
softmax是一种归一化指数函数，它的输出范围是(0,1)，它可以使得模型的输出表示为概率分布，便于进行多分类任务。LeNet-5模型中使用了softmax作为最后一层输出层的激活函数。

3.3 优化器

优化器是卷积神经网络中的一个重要组成部分，它可以为模型更新参数，降低损失，提高性能。LeNet-5优化器采用Adam算法，Adam算法集成了SGD的一阶Momentum和RMSProp的二阶Momentum，能够有效地避免梯度下降时剧烈震荡无法收敛，同时能加快梯度下降的速度。m\_t为t时刻的下降量，g\_t为t时刻梯度方向。



3.4 损失函数



交叉熵损失函数在分类问题中的形式，其中 N是样本数量，M是分类数量，y\_ci是样本i是否属于类别 c，p\_ci 是分类器对样本 i属于类别 c的预测概率。交叉熵损失函数常用于监督学习中分类器的训练，目的是最小化分类器预测与真实标签之间的差异。

# 4.实验

4.1 minist数据集

MNIST数据集是一个手写数字识别的数据集，它包含了从不同扫描文档中获取的数字图像。每个图像是一个28x28像素的正方形，共有10个数字（0到9）或10个类别。 MNIST数据集已经被划分为训练集和测试集，训练集包含了60,000个图像，测试集包含了10,000个图像。这是一个适合初学者学习模式识别方法的数据集，不需要太多的数据预处理工作。 MNIST数据集的预处理主要包括以下几个步骤：

归一化：将图像的像素值除以255，使其范围在[0,1]，减少计算量和梯度爆炸的风险。

批处理：将训练集和测试集分成若干个小批量（batch），每个批量包含一定数量的图像和标签，便于模型进行并行计算和梯度更新。

分布式：如果需要在多个节点上进行分布式训练，可以使用torch.utils.data.distributed.DistributedSampler和torch.utils.data.DataLoader等工具，将数据集划分为多个子集，并分配给不同的节点。

4.2 评估指标

LeNet-5是一个经典的卷积神经网络（CNN）架构，用于手写字符识别等任务。它由两个卷积层，两个池化层，一个全连接层和一个输出层组成。为了评估LeNet-5的性能和效果，我们使用以下几种评估指标：

准确率（Accuracy）：它是模型正确预测的图像数量与总图像数量的比例，越高越好。它是一个简单而直观的评估指标，适用于多分类问题，如MNIST。

混淆矩阵（Confusion Matrix）：它是一个表格，显示了模型对每个类别的预测情况，以及每个类别的真实情况。它可以帮助我们分析模型在哪些类别上表现好，哪些类别上表现差，以及是否存在误分类或漏分类的情况。

精确率（Precision）、召回率（Recall）和F1分数（F1-score）：它们是基于混淆矩阵的评估指标，分别反映了模型在某个类别上的正确预测率、覆盖率和综合效果。它们适用于不平衡数据集或者关注某些特定类别的情况。

4.3实现细节

4.3.1 使用Pytorch搭建网络结构

class LeNet5(nn.Module):

    def \_\_init\_\_(self, n\_classes):

        super(LeNet5, self).\_\_init\_\_()

        self.feature\_extractor = nn.Sequential(

            nn.Conv2d(in\_channels=1, out\_channels=6, kernel\_size=5),

            nn.Tanh(),

            nn.AvgPool2d(kernel\_size=2),

            nn.Conv2d(in\_channels=6, out\_channels=16, kernel\_size=5),

            nn.Tanh(),

            nn.AvgPool2d(kernel\_size=2),

            nn.Conv2d(in\_channels=16, out\_channels=120, kernel\_size=5),

            nn.Tanh()

        )

        self.classifier = nn.Sequential(

            nn.Linear(in\_features=120, out\_features=84),

            nn.Tanh(),

            nn.Linear(in\_features=84, out\_features=n\_classes),

        )

    def forward(self, x):

        x = self.feature\_extractor(x)

        x = torch.flatten(x, 1)  # 1\*1\*120展平为向量

        output = self.classifier(x)  # 输出

        probs = F.softmax(output, dim=1)  # 归一化，每一类别的概率

        return output, probs

4.3.2 MNIST数据集预处理

由于MNIST数据集的原始大小为28\*28，而LeNet5的输入尺寸为32\*32，因此在读取数据的时候要先做一步resize处理，将图片缩放，这里使用pytorch中的transforms来转换格式。

同时，由于raw数据差异较大并且数值通常都很大，因此tanh求导得到的导数可能接近于0，可能造成梯度丢失。为了解决这一问题，最好使用既有的均值和方差，将每个数据样本进行标准化，使得数据最后呈现均值为0方差为1的分布。MNIST官方提供的均值和标准差分别为μ=0.1307,σ=0.3081.

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

#下载数据集

train\_dataset = datasets.MNIST(root='data', train=True, transform=transforms,download=True)valid\_dataset = datasets.MNIST(root='data',train=False,transform=transforms)

4.3.3 形成数据流

数据集中的图片并不是依次一张一张输入的，而是成批(batch)地输入，因此需要创建一个数据流:

train\_loader = DataLoader(dataset=train\_dataset,batch\_size=BATCH\_SIZE, shuffle=True)valid\_loader = DataLoader(dataset=valid\_dataset,batch\_size=BATCH\_SIZE,shuffle=False)

这里我取BATCH\_SIZE=64，意思是每64张图片打包到一起，输入到模型中。

4.4训练

## 4.4.1 定义单次训练过程

def train(train\_loader, model, loss\_func, optimizer, device):

    """

    定义单次训练过程

    :param train\_loader:训练数据流

    :param model:网络模型

    :param loss\_func:损失函数

    :param optimizer:优化器

    :param device:使用GPU or CPU

    :return:模型状态，优化器状态，本次训练损失

    """

    model.train()  # 切换训练模式

    running\_loss = 0

    for imgs, labels in train\_loader:

        optimizer.zero\_grad()  # 优化器梯度初始化

        imgs = imgs.to(device)

        labels = labels.to(device)

        # 前向传播

        output, prob = model(imgs)

        loss = loss\_func(output, labels)  # 计算损失

        running\_loss += loss.item() \* imgs.size(0)

        # 反向传播

        loss.backward()

        optimizer.step()

    epoch\_loss = running\_loss / len(train\_loader.dataset)

    return model, optimizer, epoch\_loss

## 4.4.2 定义单次验证过程

def validate(valid\_loader, model, loss\_func, device):

    """

    :param valid\_loader:验证集数据流

    :param model:网络模型

    :param loss\_func:损失函数

    :param device:GPU or CPU

    :return:模型状态，本次验证损失

    """

    model.eval()  # 设置为验证模式

    running\_loss = 0

    for imgs, labels in valid\_loader:

        imgs = imgs.to(device)

        labels = labels.to(device)

        # Forward pass and record loss

        output, prob = model(imgs)

        loss = loss\_func(output, labels)

        running\_loss += loss.item() \* imgs.size(0)

    epoch\_loss = running\_loss / len(valid\_loader.dataset)

    return model, epoch\_los

## 4.4.3 完整训练过程

def training\_process(model, loss\_func, optimizer, train\_loader, valid\_loader, epochs, device):

    """

    完整训练过程

    :param model:网络模型

    :param loss\_func:损失函数

    :param optimizer:优化器

    :param train\_loader:训练集数据流

    :param valid\_loader:验证集数据流

    :param epochs:训练次数

    :param device:GPU or CPU

    :param print\_every:每隔多少次训练打印

    :return:模型状态，优化器,(训练损失，验证损失)

    """

    # 记录最佳结果

    best\_loss = 1e10

    train\_losses = []

    valid\_losses = []

    for epoch in range(0, epochs):

        # 训练

        start\_time = time.time()

        model, optimizer, train\_loss = train(train\_loader, model, loss\_func, optimizer, device)

        train\_losses.append(train\_loss)

        if train\_loss < best\_loss:

            best\_loss = train\_loss

            best\_epoch = epoch

            best\_model = model

        # 验证

        with torch.no\_grad():

            model, valid\_loss = validate(valid\_loader, model, loss\_func, device)

            valid\_losses.append(valid\_loss)

        # 在验证集上计算准确度

        valid\_acc = get\_accuracy(model, valid\_loader, device=device)

        end\_time = time.time()

        print("Epoch:%d    TrainLoss=%.4f    ValidLoss=%.4f    ValidAccuracy=%.2f%%    Time cost=%.2fs" % (

        epoch, train\_loss, valid\_loss, valid\_acc \* 100, end\_time - start\_time))

    # 绘制loss曲线

    print("\*\*\*\*\*\*\*\*\*\*Minimum Loss is in Epoch%d = %.4f\*\*\*\*\*\*\*\*\*\*" % (best\_epoch, best\_loss))

    plot\_losses(train\_losses, valid\_losses)

    return best\_model, optimizer, (train\_losses, valid\_losses)

此处取Epochs=30,最后保存最佳的模型

model, optimizer, \_ = training\_process(model, criterion, optimizer, train\_loader, valid\_loader, N\_EPOCHS, DEVICE)torch.save(model,"Model/LeNet5.pth")

# 

# 4.结果

4.1 训练输出

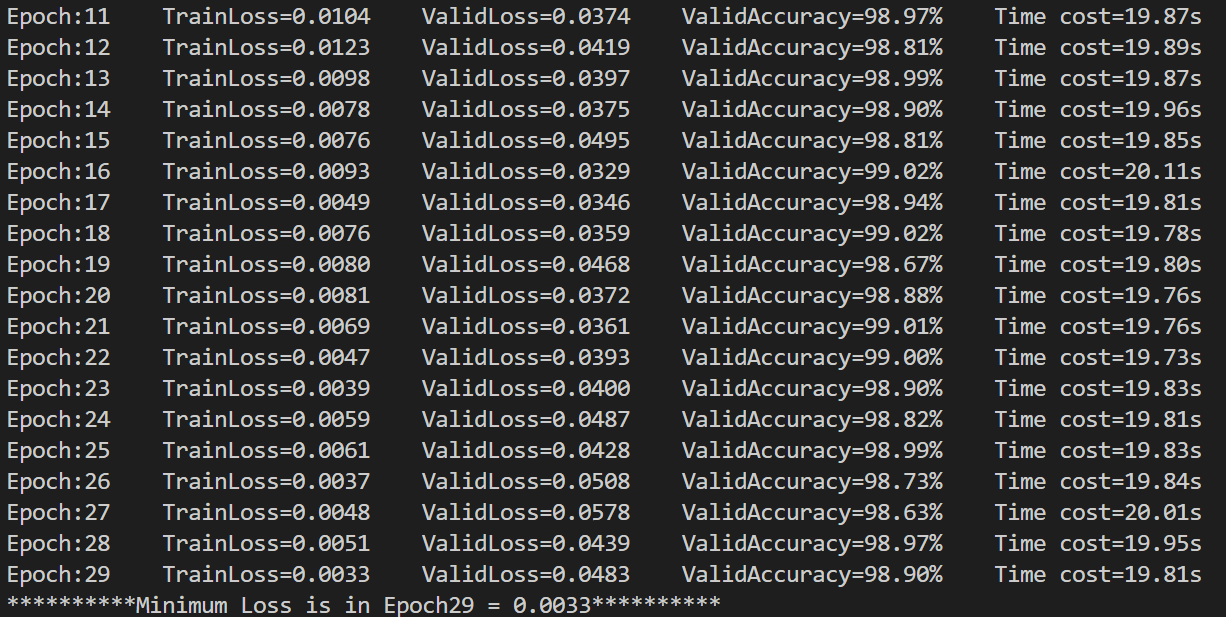


图2 训练过程输出

整个模型在验证集上的预测准确度达到了98.80%，数值上来说非常出色。

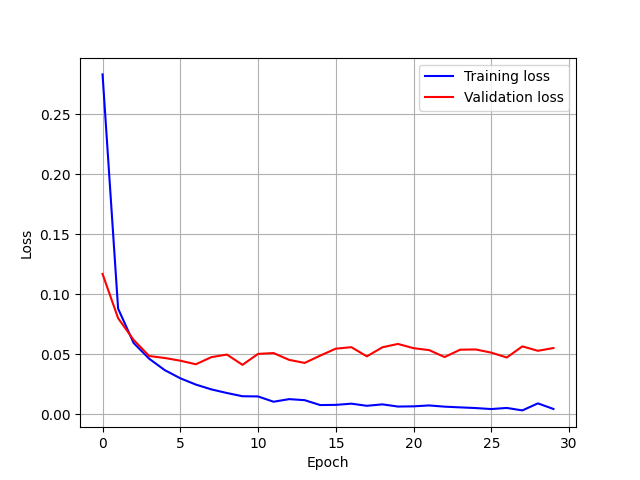
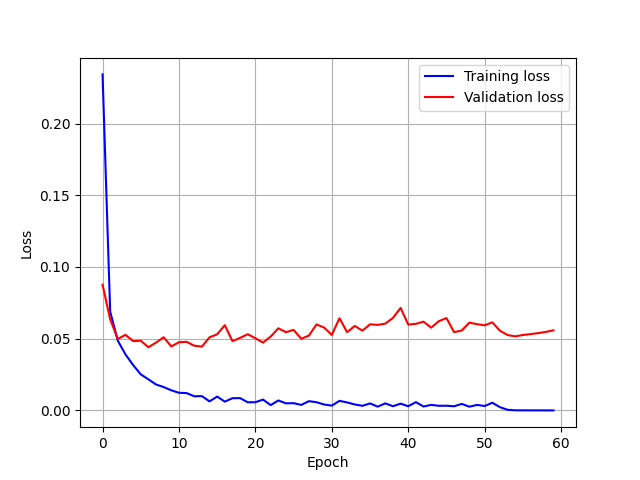
 

图3 对比训练30和60个epoch的结果

可以看出，对于30个Epoch，验证损失已经接近于稳定，并开始逐渐呈现上升的态势，因此30个Epoch对于这个模型来说已经足够了。

4.2 推理

使用predict.py中的multi\_predict()函数随机从MNIST的验证集抽取50张进行测试：

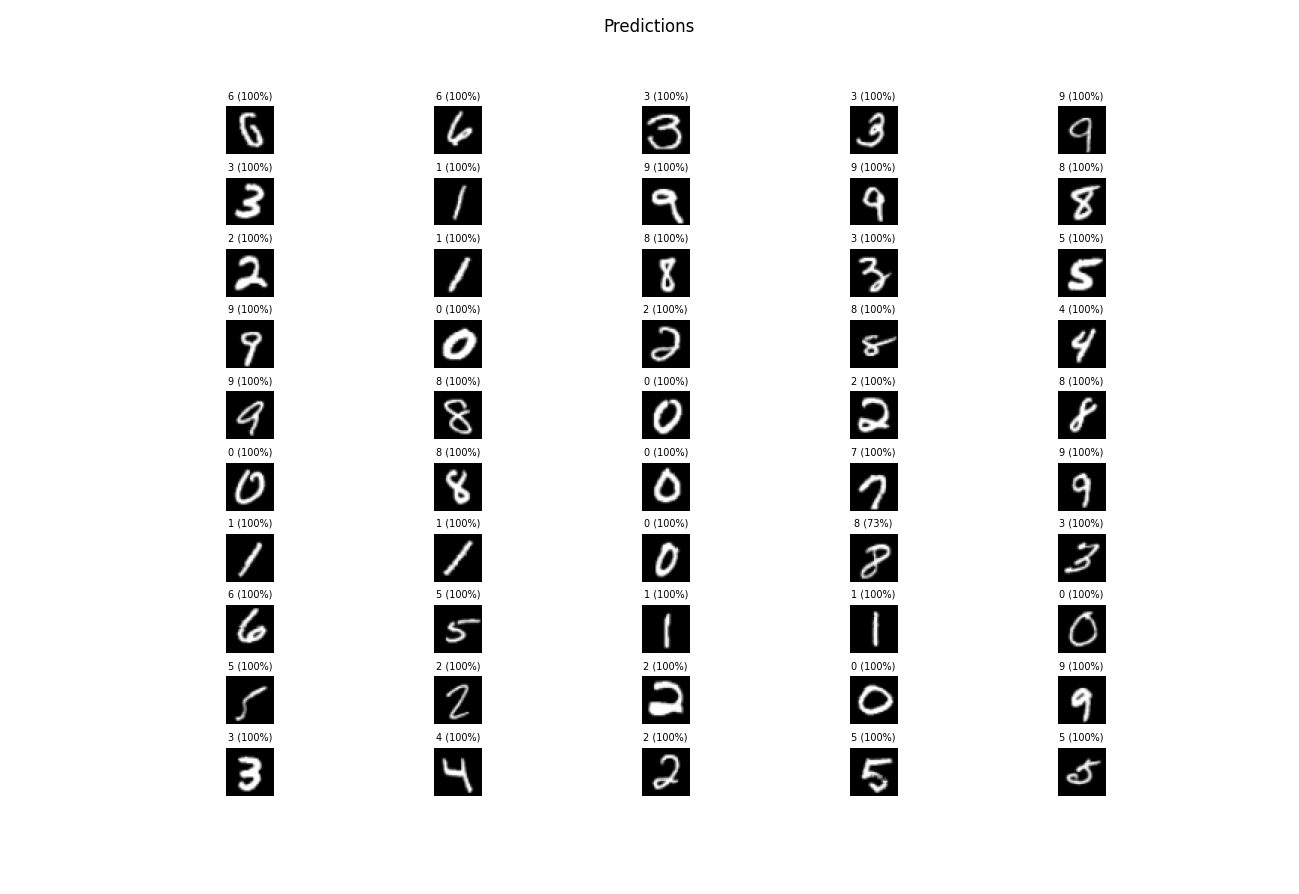


图4 验证集抽取50张预测结果

使用predict.py中编写的single\_predict()函数，传入图片路径，输出检测结果。

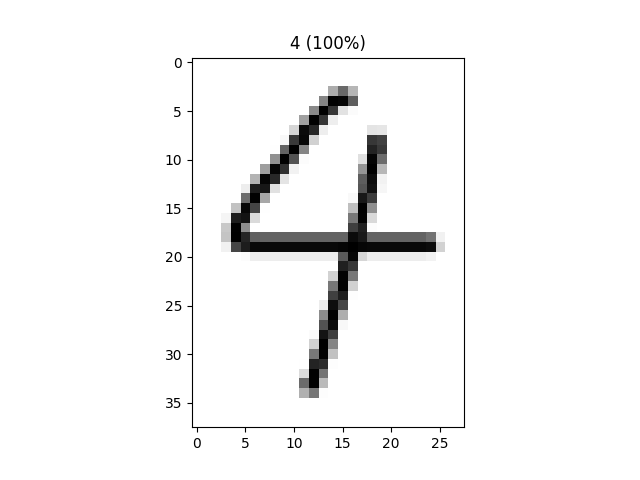
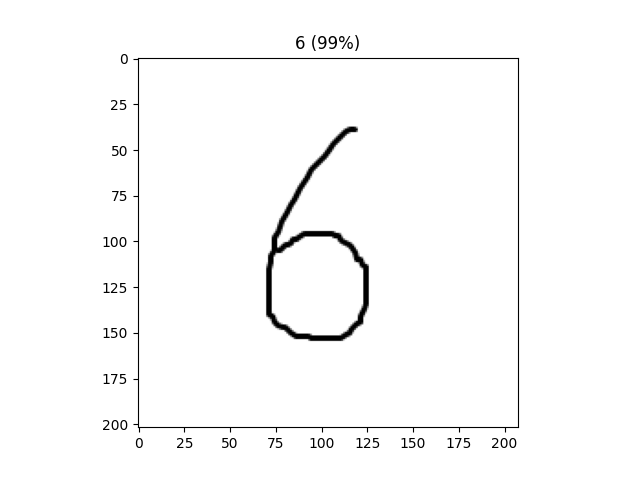
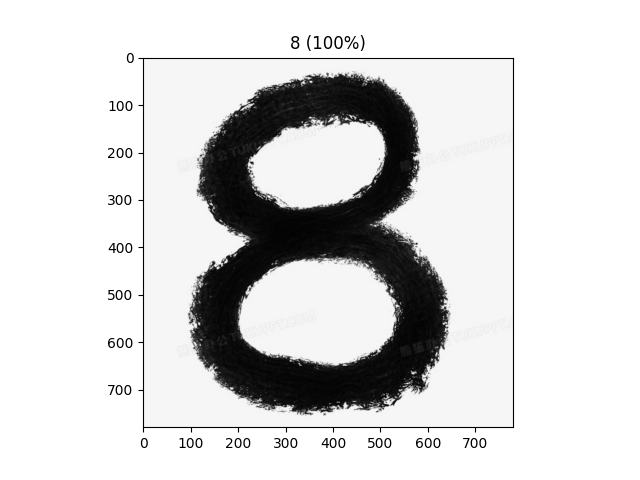
  

图5 随机手写测试

结论：总结你的实验发现，指出你的模型的优势和局限性，以及未来可以改进的方向

# 5.总结

学习使用了pytorch的基本操作，学会了使用pytorch搭建一个经典的网络框架，加深了对深度学习神经网络中一些术语的理解，锻炼了代码能力。

最开始使用自己手写的数字进行识别时，无论怎么测试也无法得到正确结果，几乎总是被预测为"8"。后来发现是训练用的数据是黑底白字，而自己写的数字是白底黑字，恍然大悟，在识别自己的数字时，要对灰度图进行反相，才能得到正确的结果。

基于此问题，可以推断出此模型存在的不足：训练集样本过于理想化，总是黑白分明，而现实拍摄的照片不一定是黑底白字，底色可以是任何颜色，数字也是任何颜色，需要人事先去调整把图像灰度图转换为黑底白字的，比较麻烦。

训练数据集用到的数字都是正体的，没有经过旋转，因此对于现实中旋转的数据效果并不好。因此可以考虑在训练支出就进行数据增广，对源数据集进行一系列旋转拉伸的操作，丰富数据集，提高鲁棒性。