浅层神经网络实现对MNIST数据集分类试验

游霄童

21009200158

created： 2023/12/6

1.数据介绍

MNIST数据集来自美国国家标准与技术研究所, National Institute of Standards and Technology (NIST)。训练集（training set）由来自250个不同人手写的数字构成，其中50%是高中学生，50%来自人口普查局（the Census Bureau）的工作人员。测试集（test set）也是同样比例的手写数字数据，但保证了测试集和训练集的作者集不相交。

MNIST数据集一共有7万张图片，其中6万张是训练集，1万张是测试集。每张图片是28×28的$0-9$的手写数字图片组成。每个图片是黑底白字的形式，黑底用0表示，白字用0-1之间的浮点数表示，越接近1，颜色越白。

将28×28维的图片矩阵拉直，转化为1×784维的向量不影响理解：

$$[0,0,0,0.345,0.728,0.310,0.402,0,0,0,\cdots,0,0,0]$$

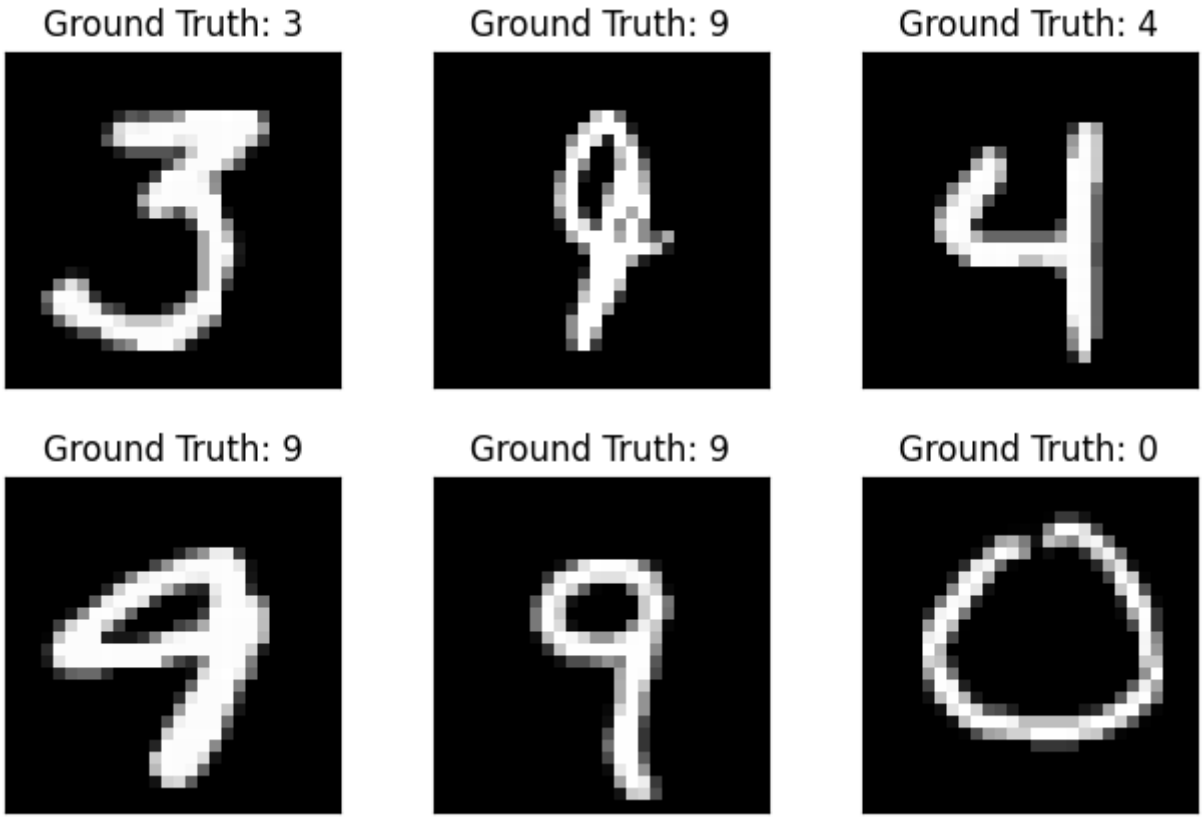
图片的标签以一维数组的one-hot编码形式给出：

$$[0,0,0,0,0,1,0,0,0,0]$$

每个元素表示图片对应的数字出现的概率，显然，该向量标签表示的是数字5 55。

MNIST数据集下载地址是http://yann.lecun.com/exdb/mnist/，它包含了4个部分：

训练数据集：train-images-idx3-ubyte.gz （9.45 MB，包含60,000个样本）。
训练数据集标签：train-labels-idx1-ubyte.gz（28.2 KB，包含60,000个标签）。
测试数据集：t10k-images-idx3-ubyte.gz（1.57 MB ，包含10,000个样本）。
测试数据集标签：t10k-labels-idx1-ubyte.gz（4.43 KB，包含10,000个样本的标签）。



如上图所示就是MNIST数据集中收录的手写图像。

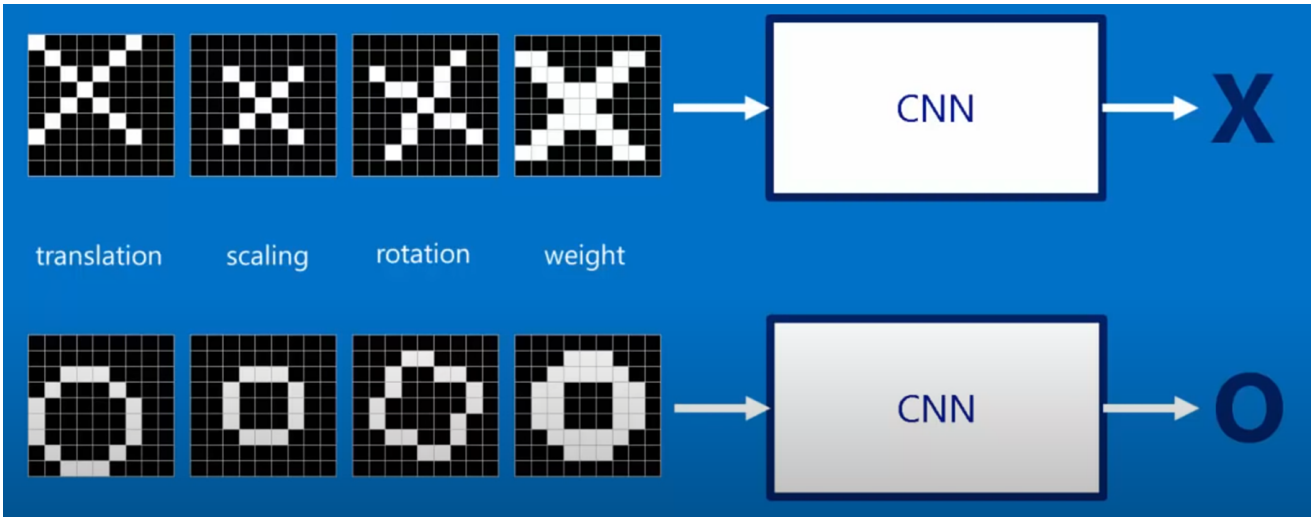
这里采用torchvision将MNIST数据集导入。

```
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=True, download=True,

transform=torchvision.transforms.Compose([torchvision.transforms.ToTenso
r(),torchvision.transforms.Normalize((0.1307,), (0.3081,))])),
                                batch_size=batch_size_train,
shuffle=True)
test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=False, download=True,

transform=torchvision.transforms.Compose([torchvision.transforms.ToTenso
r(),torchvision.transforms.Normalize((0.1307,), (0.3081,))])),
                                batch_size=batch_size_test,
shuffle=True)
```

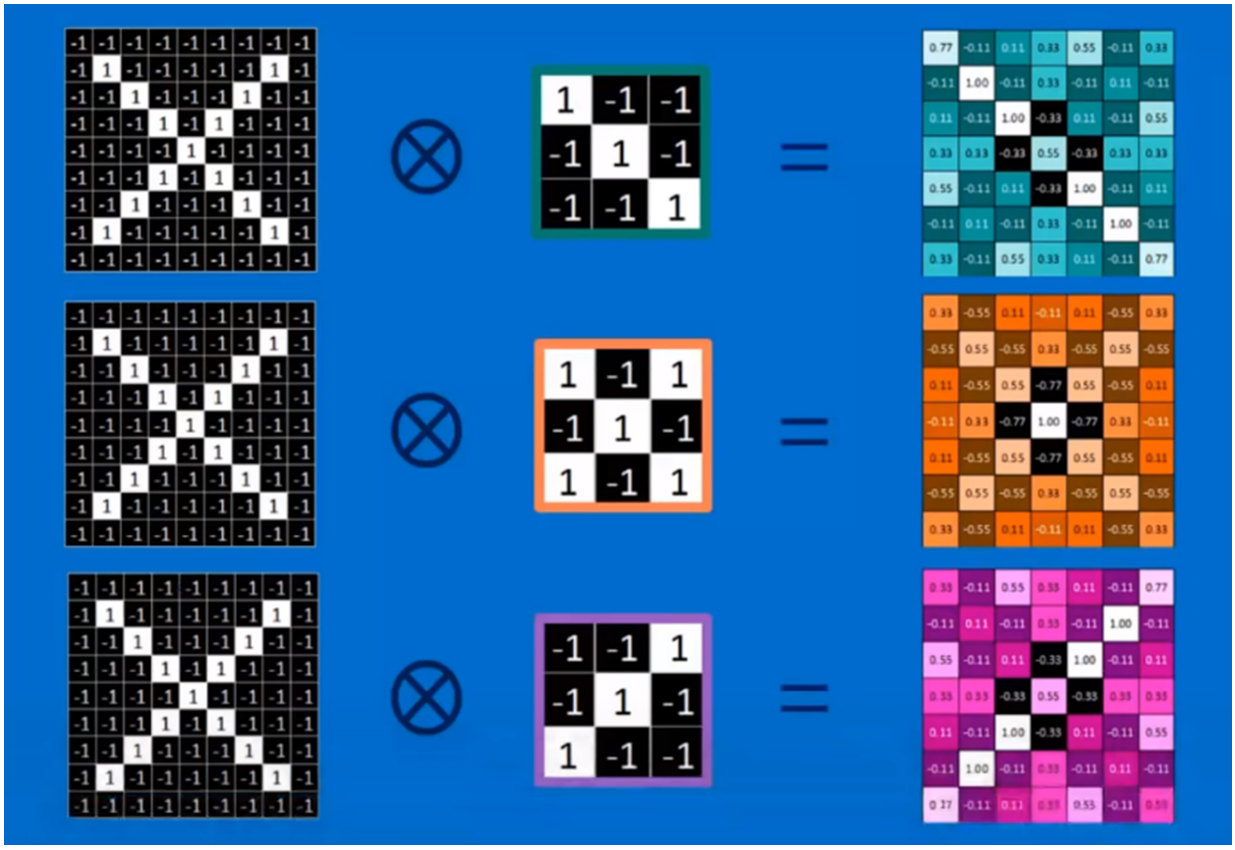
2.CNN



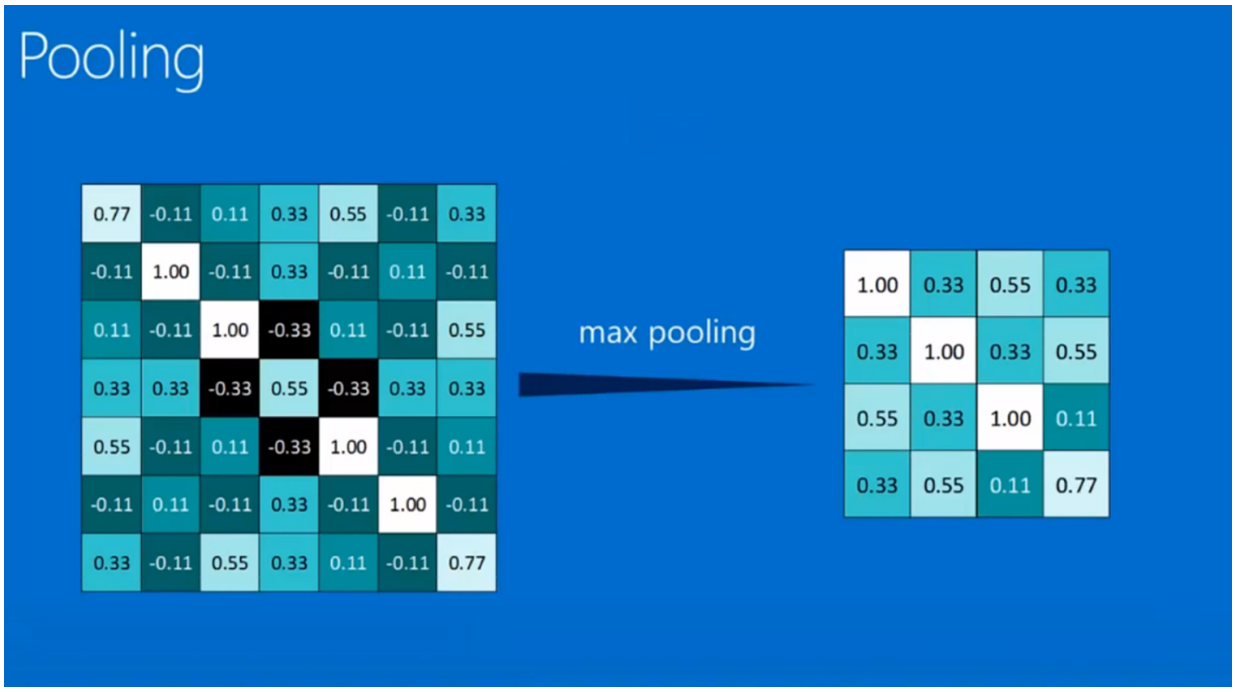
2.1CNN简介

卷积神经网络（Convolutional Neural Network，CNN）是一种深度学习模型，专门用于处理和识别具有层次结构模式的数据，尤其在图像识别领域取得了显著的成果。以下是CNN的一些基本概念和原理：

1. **卷积层（Convolutional Layer）：** CNN的核心是卷积层，它通过卷积操作对输入数据进行特征提取。卷积操作可以理解为将一个滤波器或卷积核（filter或kernel）应用于输入数据的局部区域，通过权重共享来检测输入中的特定特征，如边缘、纹理等。

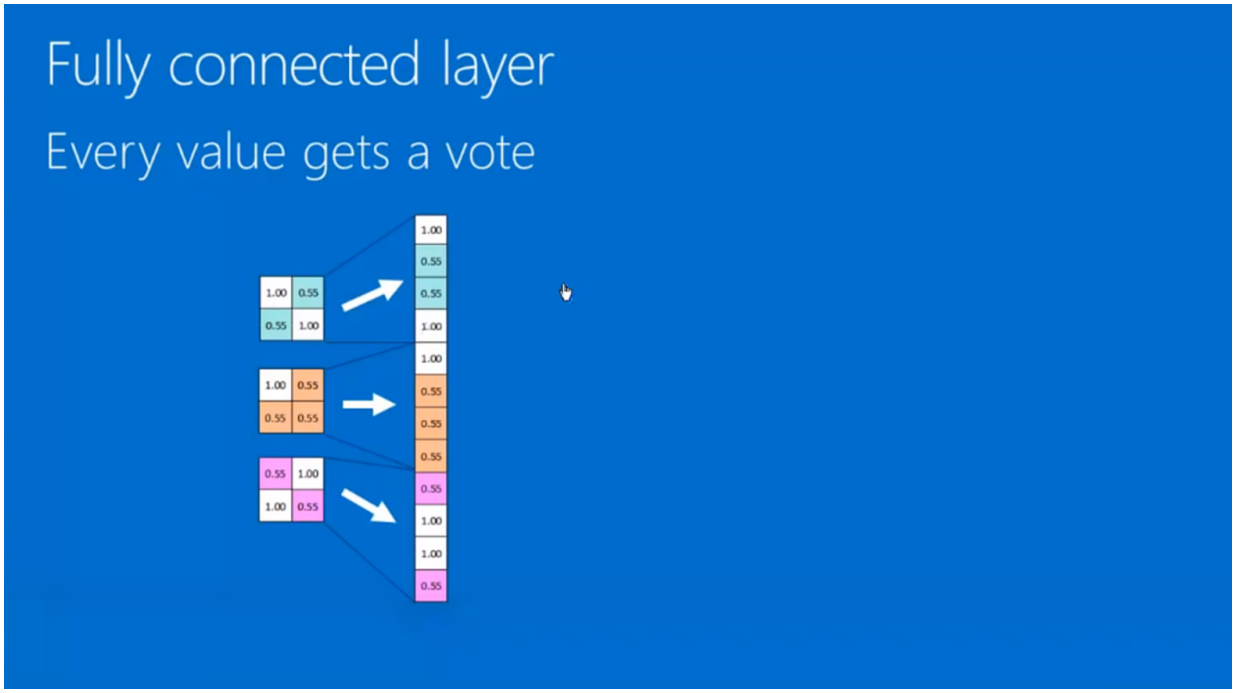


2. **池化层（Pooling Layer）：** 池化层用于降低卷积层输出的空间维度，减少计算复杂度和参数数量，同时保留重要的特征。常用的池化操作有最大池化和平均池化。



3. **激活函数（Activation Function）：** 在卷积层中，激活函数引入非线性，使得网络能够学习复杂的特征映射。常用的激活函数包括ReLU（Rectified Linear Unit）、Sigmoid和Tanh。

4. **全连接层（Fully Connected Layer）：** 在卷积层和全连接层之间，通常会添加一个或多个全连接层，用于将卷积层提取的特征映射与输出进行关联，以进行最终的分类或回归。



5. **卷积神经网络的结构：** CNN通常采用多层的卷积层和池化层交替堆叠的结构，最后连接全连接层。这种层次结构有助于网络学习输入数据的层次性特征。

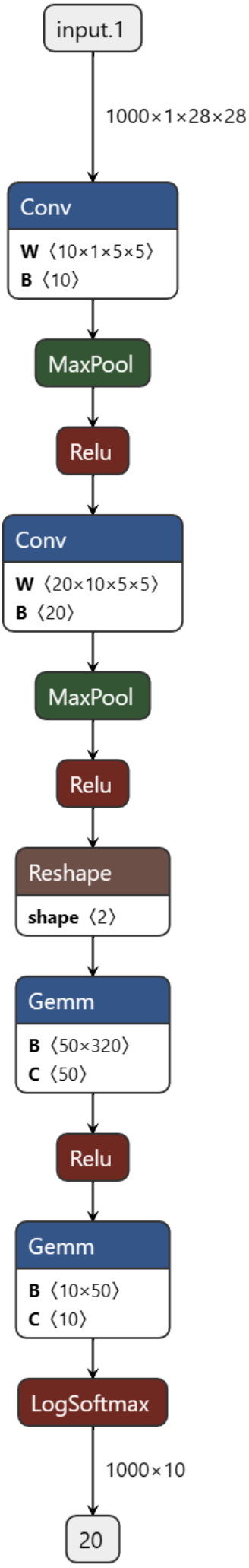
6. **权重共享：** 通过卷积操作，网络可以学习到具有平移不变性的特征，这是因为卷积核在整个输入数据上共享权重，从而减少了需要学习的参数数量。

7. **数据增强：** 在训练CNN时，常常使用数据增强技术，通过对原始数据进行随机变换（如旋转、翻转、缩放等），来增加训练样本的多样性，提高模型的泛化能力。

CNN在计算机视觉领域取得了巨大成功，被广泛应用于图像分类、目标检测、语义分割等任务。其能够自动从数据中学习到特征，从而避免了手工设计特征的繁琐过程，使其成为处理复杂图像数据的强大工具。

2.2网络架构

由于MINIST数据集比较简单，这里采用下图所示的网络架构：


input.1
1000×1×28×28
Conv
W 〈10×1×5×5〉
B 〈10〉
MaxPool
Relu
Conv
W 〈20×10×5×5〉
B 〈20〉
MaxPool
Relu
Reshape
shape 〈2〉
Gemm
B 〈50×320〉
C 〈50〉
Relu
Gemm
B 〈10×50〉
C 〈10〉
LogSoftmax
1000×10
20

此网络采用netron库绘制：

```
import torch.onnx
import netron

onnx_path = "onnx_model_name.onnx" # 文件名

torch.onnx.export(network,example_data, onnx_path) # 导出神经网络模型为onnx格式

netron.start(onnx_path) # 启动netron
```

1. 第一个卷积层，输入通道为1（灰度图像），输出通道为10，卷积核大小为5x5
2. 第二个卷积层，输入通道为10，输出通道为20，卷积核大小为5x5
3. 用于在第二个卷积层后引入dropout，以防止过拟合
4. 第一个全连接层，输入特征数为320，输出特征数为50
5. 第二个全连接层，输入特征数为50，输出特征数为10

具体操作如下：

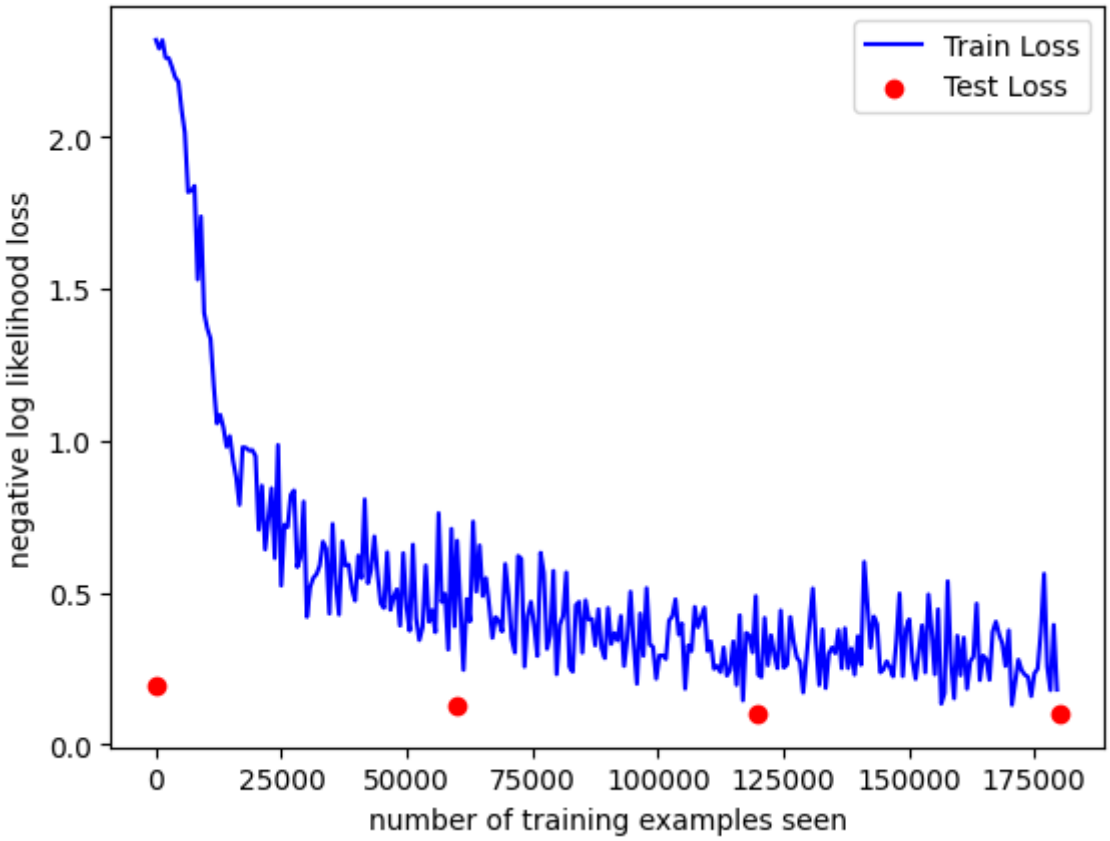
- 输入x经过第一个卷积层，然后通过ReLU激活函数和2x2的最大池化层
- 输入x经过第二个卷积层，然后通过ReLU激活函数、2x2的最大池化层和dropout
- 将x展平为一维向量，用于连接到全连接层
- 输入x经过第一个全连接层，然后通过ReLU激活函数
- 在训练时进行dropout，以防止过拟合
- 输入x经过第二个全连接层，输出最终结果
- 使用log_softmax作为输出层的激活函数

3.结果分析

采用超参数如下所示

```
n_epochs = 3
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.01
momentum = 0.5
log_interval = 10
random_seed = 1
torch.manual_seed(random_seed)
```

具体结果如下：



Train Epoch: 1 [0/60000 (0%)] Loss: 2.319280
Train Epoch: 1 [640/60000 (1%)] Loss: 2.290954
Train Epoch: 1 [1280/60000 (2%)] Loss: 2.318535
Train Epoch: 1 [1920/60000 (3%)] Loss: 2.261000
Train Epoch: 1 [2560/60000 (4%)] Loss: 2.259137
Train Epoch: 1 [3200/60000 (5%)] Loss: 2.229576
Train Epoch: 1 [3840/60000 (6%)] Loss: 2.196151
Train Epoch: 1 [4480/60000 (7%)] Loss: 2.181698
Train Epoch: 1 [5120/60000 (9%)] Loss: 2.088444
Train Epoch: 1 [5760/60000 (10%)] Loss: 2.014765
Train Epoch: 1 [6400/60000 (11%)] Loss: 1.818533
Train Epoch: 1 [7040/60000 (12%)] Loss: 1.822772
Train Epoch: 1 [7680/60000 (13%)] Loss: 1.838733
Train Epoch: 1 [8320/60000 (14%)] Loss: 1.530556
Train Epoch: 1 [8960/60000 (15%)] Loss: 1.738141
Train Epoch: 1 [9600/60000 (16%)] Loss: 1.420971
Train Epoch: 1 [10240/60000 (17%)] Loss: 1.369237
Train Epoch: 1 [10880/60000 (18%)] Loss: 1.335397
Train Epoch: 1 [11520/60000 (19%)] Loss: 1.182360
Train Epoch: 1 [12160/60000 (20%)] Loss: 1.056649

.....

.....

……

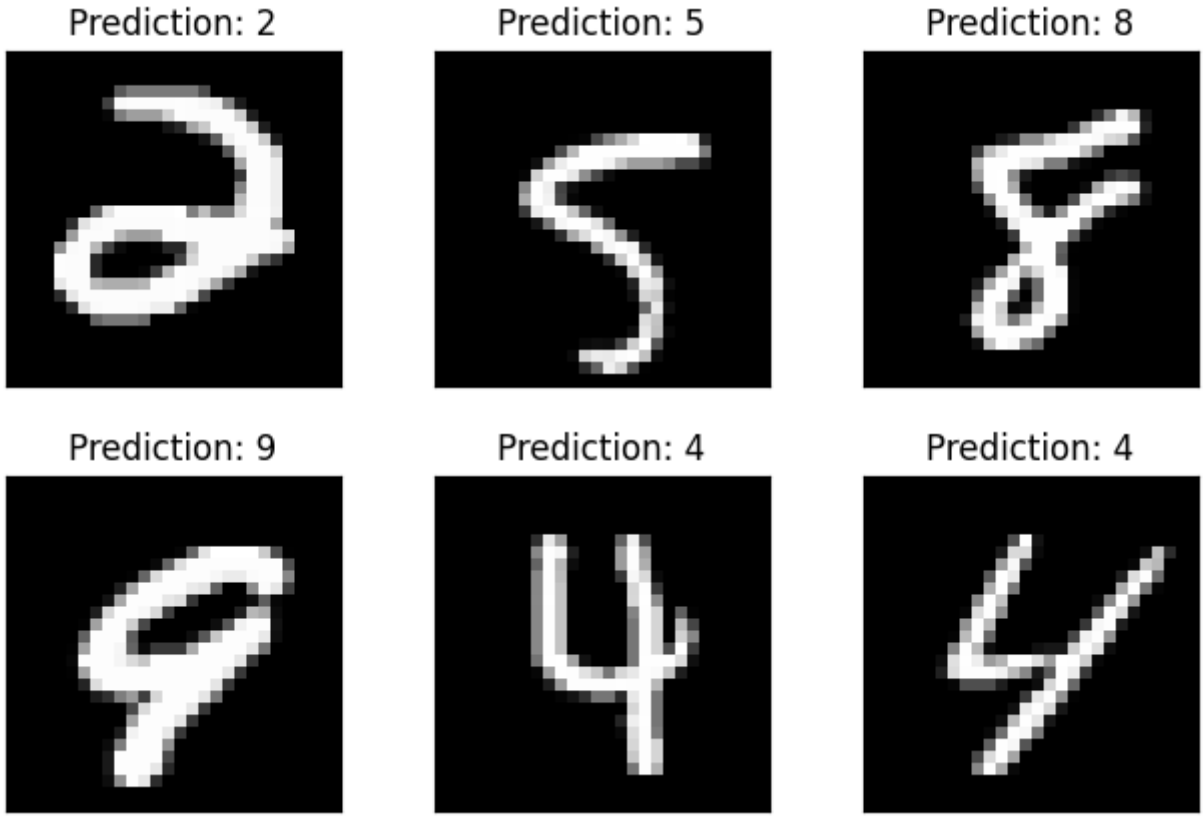
```
Train Epoch: 3 [49920/60000 (83%)]   Loss: 0.375541
Train Epoch: 3 [50560/60000 (84%)]   Loss: 0.130251
Train Epoch: 3 [51200/60000 (85%)]   Loss: 0.203908
Train Epoch: 3 [51840/60000 (86%)]   Loss: 0.278938
Train Epoch: 3 [52480/60000 (87%)]   Loss: 0.244604
Train Epoch: 3 [53120/60000 (88%)]   Loss: 0.229257
Train Epoch: 3 [53760/60000 (90%)]   Loss: 0.222509
Train Epoch: 3 [54400/60000 (91%)]   Loss: 0.159071
Train Epoch: 3 [55040/60000 (92%)]   Loss: 0.233795
Train Epoch: 3 [55680/60000 (93%)]   Loss: 0.249439
Train Epoch: 3 [56320/60000 (94%)]   Loss: 0.370515
Train Epoch: 3 [56960/60000 (95%)]   Loss: 0.561829
Train Epoch: 3 [57600/60000 (96%)]   Loss: 0.255670
Train Epoch: 3 [58240/60000 (97%)]   Loss: 0.178493
Train Epoch: 3 [58880/60000 (98%)]   Loss: 0.393723
Train Epoch: 3 [59520/60000 (99%)]   Loss: 0.180310

Test set: Avg. loss: 0.1018, Accuracy: 9671/10000 (97%)
```

最后的准确稳定在

$$96.71\%$$

抽取预测的结果如下，可以看到还是不错的。



4.附录

```
import torch
import torchvision
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
#%%
n_epochs = 3
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.01
momentum = 0.5
log_interval = 10
random_seed = 1
torch.manual_seed(random_seed)
#%%
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=True, download=True,

transform=torchvision.transforms.Compose([torchvision.transforms.ToTenso
r(),torchvision.transforms.Normalize((0.1307,), (0.3081,))])),
                                   batch_size=batch_size_train,
shuffle=True)
test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=False, download=True,

transform=torchvision.transforms.Compose([torchvision.transforms.ToTenso
r(),torchvision.transforms.Normalize((0.1307,), (0.3081,))])),
                                   batch_size=batch_size_test,
shuffle=True)
#%%
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
print(example_targets)
print(example_data.shape)
#%%

#%%
fig = plt.figure()
for i in range(6):
  plt.subplot(2,3,i+1)
  plt.tight_layout()
  plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
  plt.title(f"Ground Truth: {example_targets[i]}")
  plt.xticks([])
  plt.yticks([])
```

```
plt.show()
#%%
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # 第一个卷积层，输入通道为1（灰度图像），输出通道为10，卷积核大小为5x5
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)

        # 第二个卷积层，输入通道为10，输出通道为20，卷积核大小为5x5
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)

        # 用于在第二个卷积层后引入dropout，以防止过拟合
        self.conv2_drop = nn.Dropout2d()

        # 第一个全连接层，输入特征数为320，输出特征数为50
        self.fc1 = nn.Linear(320, 50)

        # 第二个全连接层，输入特征数为50，输出特征数为10
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        # 输入x经过第一个卷积层，然后通过ReLU激活函数和2x2的最大池化层
        x = F.relu(F.max_pool2d(self.conv1(x), 2))

        # 输入x经过第二个卷积层，然后通过ReLU激活函数、2x2的最大池化层和dropout
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))

        # 将x展平为一维向量，用于连接到全连接层
        x = x.view(-1, 320)

        # 输入x经过第一个全连接层，然后通过ReLU激活函数
        x = F.relu(self.fc1(x))

        # 在训练时进行dropout，以防止过拟合
        x = F.dropout(x, training=self.training)

        # 输入x经过第二个全连接层，输出最终结果
        x = self.fc2(x)

        # 使用log_softmax作为输出层的激活函数
        return F.log_softmax(x)
#%%
network = Net()
optimizer = optim.SGD(network.parameters(), lr=learning_rate,
                      momentum=momentum)
#%%
import torch.onnx
import netron
```

```
onnx_path = "onnx_model_name.onnx" # 文件名

torch.onnx.export(network,example_data, onnx_path) # 导出神经网络模型为onnx
格式

netron.start(onnx_path) # 启动netron
#%%
train_losses = []
train_counter = []
test_losses = []
test_counter = [i*len(train_loader.dataset) for i in range(n_epochs +
1)]
#%%
def train(epoch):
    # 将神经网络设置为训练模式
    network.train()

    # 遍历训练数据集的每个批次
    for batch_idx, (data, target) in enumerate(train_loader):
        # 清零梯度，以防止梯度累积
        optimizer.zero_grad()

        # 前向传播，获取模型输出
        output = network(data)

        # 计算损失，使用负对数似然损失（NLL Loss）
        loss = F.nll_loss(output, target)

        # 反向传播，计算梯度
        loss.backward()

        # 优化器更新模型参数
        optimizer.step()

        # 打印训练信息，每隔一定批次打印一次
        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss:
{:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

            # 记录训练损失和计数器
            train_losses.append(loss.item())
            train_counter.append((batch_idx * 64) + ((epoch - 1) *
len(train_loader.dataset)))

            # 保存模型和优化器的状态字典
            torch.save(network.state_dict(), './model.pth')
```

```
            torch.save(optimizer.state_dict(), './optimizer.pth')

#%%
def test():
  network.eval()
  test_loss = 0
  correct = 0
  with torch.no_grad():
    for data, target in test_loader:
      output = network(data)
      test_loss += F.nll_loss(output, target, size_average=False).item()
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
  test_loss /= len(test_loader.dataset)
  test_losses.append(test_loss)
  print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{}
({:.0f}%)\n'.format(test_loss, correct, len(test_loader.dataset),100. *
correct / len(test_loader.dataset)))
#%%
test()
#%%
for epoch in range(1, n_epochs + 1):
    train(epoch)
    test()
#%%

#%%
fig = plt.figure()
plt.plot(train_counter, train_losses, color='blue')
plt.scatter(test_counter, test_losses, color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('number of training examples seen')
plt.ylabel('negative log likelihood loss')
plt.show()
#%%
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
with torch.no_grad():
    output = network(example_data)
fig = plt.figure()
for i in range(6):
    plt.subplot(2, 3, i + 1)
    plt.tight_layout()
    plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
    plt.title("Prediction: {}".format(output.data.max(1, keepdim=True)
[1][i].item()))
    plt.xticks([])
    plt.yticks([])
plt.show()
```