实验报告

实验任务一

图像中目标的边缘检测

代码:

```
#训练补全部分(自定义损失函数，训练epoch，学习率等)
#TODU
loss = nn.MSELoss()
#TODU
epoch = 100
trainer = torch.optim.SGD(conv2d.parameters(), lr=0.2)
for num in range(epoch):
    y_hat = conv2d.forward(X)
    trainer.zero_grad()
    l = loss(y_hat, Y)
    l.sum().backward()
    trainer.step()
    if (num+1) % 10 == 0:
        print('epoch %d, loss %lf'%(num+1, l))
#TODU
#
```

实验结果:

```
epoch 10, loss 0.069720
epoch 20, loss 0.021275
epoch 30, loss 0.006558
epoch 40, loss 0.002022
epoch 50, loss 0.000623
epoch 60, loss 0.000192
epoch 70, loss 0.000059
epoch 80, loss 0.000018
epoch 90, loss 0.000006
epoch 100, loss 0.000002

tensor([[ 0.9977, -0.9977]])
```

实验任务二

1. 在CIFAR数据集上实现CNN

代码:

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        #TODO: 实现模型结构
        #TODO 实现self.conv1:卷积层
```

```
        self.conv1 = nn.Conv2d(3,32, kernel_size=3, padding=1)
        #TODO 实现self.conv2:卷积层
        self.conv2 = nn.Conv2d(32,64, kernel_size=3, padding=1)
        #TODO 实现self.pool: MaxPool2d
        self.pool = nn.MaxPool2d(kernel_size=2)
        #TODO 实现self.fc1: 线性层
        self.fc1 = nn.Linear(64*8*8,500)
        #TODO 实现self.fc2：线性层
        self.fc2 = nn.Linear(500,10)
        #TODO 实现 self.dropout: Dropout层
        self.dropout = nn.Dropout2d(0.2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

def train(model, train_loader, test_loader, device):
    num_epochs = 15
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)
            #TODO:实现训练部分，完成反向传播过程
            #TODO: optimizer梯度清除
            optimizer.zero_grad()
            #TODO: 模型输入
            outputs = model(inputs)
            #TODO: 计算损失
            loss = criterion(outputs, labels).sum()
            #TODO: 反向传播
            loss.backward()
            #TODO: 更新参数
            optimizer.step()

            running_loss += loss.item()
            if i % 100 == 99:  # 每100个batch打印一次损失
                print(
                  f'Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1}/{len(train_loader)}],
Loss: {running_loss / 100:.4f}')
                running_loss = 0.0

        #每个epoch结束后在测试集上评估模型
        model.eval()
```

```
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        print(f'Test Accuracy: {100 * correct / total:.2f}%')
```

结果:

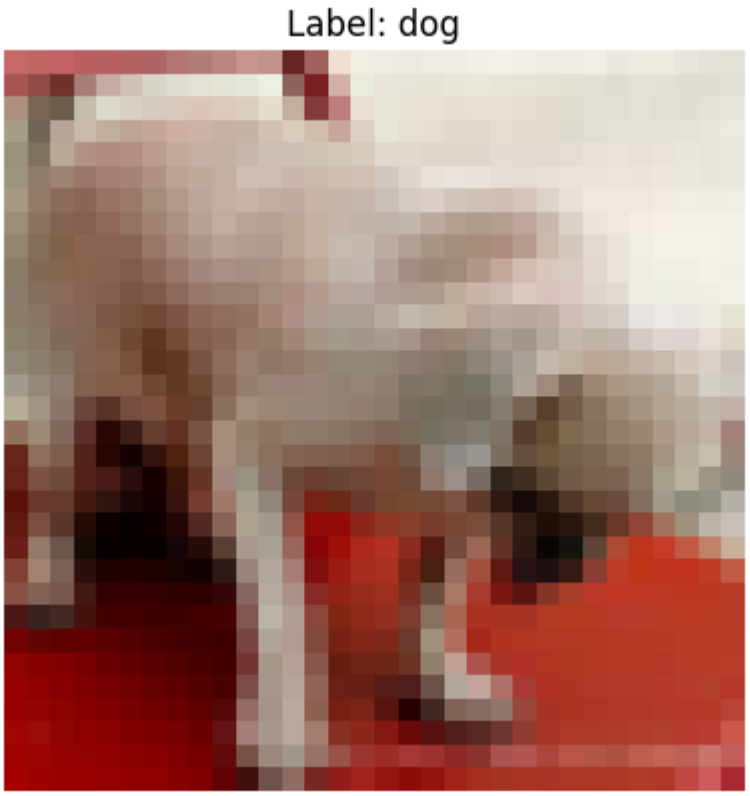
```
Epoch [1/15], Step [100/1563], Loss: 1.9610
Epoch [1/15], Step [200/1563], Loss: 1.5819
Epoch [1/15], Step [300/1563], Loss: 1.4527
Epoch [1/15], Step [400/1563], Loss: 1.3865
Epoch [1/15], Step [500/1563], Loss: 1.3535
Epoch [1/15], Step [600/1563], Loss: 1.2826
Epoch [1/15], Step [700/1563], Loss: 1.2283
Epoch [1/15], Step [800/1563], Loss: 1.1969
Epoch [1/15], Step [900/1563], Loss: 1.1606
Epoch [1/15], Step [1000/1563], Loss: 1.1399
Epoch [1/15], Step [1100/1563], Loss: 1.1619
Epoch [1/15], Step [1200/1563], Loss: 1.1164
Epoch [1/15], Step [1300/1563], Loss: 1.1098
Epoch [1/15], Step [1400/1563], Loss: 1.0721
Epoch [1/15], Step [1500/1563], Loss: 1.0497
Test Accuracy: 63.89%
Epoch [2/15], Step [100/1563], Loss: 0.9338
Epoch [2/15], Step [200/1563], Loss: 0.9477
Epoch [2/15], Step [300/1563], Loss: 0.9458
Epoch [2/15], Step [400/1563], Loss: 0.9555
Epoch [2/15], Step [500/1563], Loss: 0.9217
Epoch [2/15], Step [600/1563], Loss: 0.8886
Epoch [2/15], Step [700/1563], Loss: 0.8842
Epoch [2/15], Step [800/1563], Loss: 0.8535
Epoch [2/15], Step [900/1563], Loss: 0.8831
Epoch [2/15], Step [1000/1563], Loss: 0.8827
Epoch [2/15], Step [1100/1563], Loss: 0.8875
Epoch [2/15], Step [1200/1563], Loss: 0.8531
Epoch [2/15], Step [1300/1563], Loss: 0.8832
Epoch [2/15], Step [1400/1563], Loss: 0.8736
Epoch [2/15], Step [1500/1563], Loss: 0.8728
Test Accuracy: 68.81%
Epoch [3/15], Step [100/1563], Loss: 0.7404
Epoch [3/15], Step [200/1563], Loss: 0.6924
Epoch [3/15], Step [300/1563], Loss: 0.7136
Epoch [3/15], Step [400/1563], Loss: 0.7030
Epoch [3/15], Step [500/1563], Loss: 0.6810
Epoch [3/15], Step [600/1563], Loss: 0.6891
Epoch [3/15], Step [700/1563], Loss: 0.7495
```

```
Epoch [3/15], Step [800/1563], Loss: 0.7566
Epoch [3/15], Step [900/1563], Loss: 0.7171
Epoch [3/15], Step [1000/1563], Loss: 0.6705
Epoch [3/15], Step [1100/1563], Loss: 0.7089
Epoch [3/15], Step [1200/1563], Loss: 0.7538
Epoch [3/15], Step [1300/1563], Loss: 0.7252
Epoch [3/15], Step [1400/1563], Loss: 0.7519
Epoch [3/15], Step [1500/1563], Loss: 0.7421
Test Accuracy: 71.22%
Epoch [4/15], Step [100/1563], Loss: 0.5407
Epoch [4/15], Step [200/1563], Loss: 0.5583
Epoch [4/15], Step [300/1563], Loss: 0.5568
Epoch [4/15], Step [400/1563], Loss: 0.5628
Epoch [4/15], Step [500/1563], Loss: 0.5971
Epoch [4/15], Step [600/1563], Loss: 0.5699
Epoch [4/15], Step [700/1563], Loss: 0.5586
Epoch [4/15], Step [800/1563], Loss: 0.5743
Epoch [4/15], Step [900/1563], Loss: 0.5557
Epoch [4/15], Step [1000/1563], Loss: 0.5943
Epoch [4/15], Step [1100/1563], Loss: 0.5644
Epoch [4/15], Step [1200/1563], Loss: 0.5954
Epoch [4/15], Step [1300/1563], Loss: 0.5616
Epoch [4/15], Step [1400/1563], Loss: 0.5540
Epoch [4/15], Step [1500/1563], Loss: 0.5941
Test Accuracy: 72.33%
Epoch [5/15], Step [100/1563], Loss: 0.3966
Epoch [5/15], Step [200/1563], Loss: 0.4226
Epoch [5/15], Step [300/1563], Loss: 0.4148
Epoch [5/15], Step [400/1563], Loss: 0.4199
Epoch [5/15], Step [500/1563], Loss: 0.4380
Epoch [5/15], Step [600/1563], Loss: 0.4347
Epoch [5/15], Step [700/1563], Loss: 0.4176
Epoch [5/15], Step [800/1563], Loss: 0.4534
Epoch [5/15], Step [900/1563], Loss: 0.4561
Epoch [5/15], Step [1000/1563], Loss: 0.4683
Epoch [5/15], Step [1100/1563], Loss: 0.4392
Epoch [5/15], Step [1200/1563], Loss: 0.4646
Epoch [5/15], Step [1300/1563], Loss: 0.4893
Epoch [5/15], Step [1400/1563], Loss: 0.4597
Epoch [5/15], Step [1500/1563], Loss: 0.4715
Test Accuracy: 73.26%
Epoch [6/15], Step [100/1563], Loss: 0.3043
Epoch [6/15], Step [200/1563], Loss: 0.2832
Epoch [6/15], Step [300/1563], Loss: 0.3042
Epoch [6/15], Step [400/1563], Loss: 0.3205
Epoch [6/15], Step [500/1563], Loss: 0.3300
Epoch [6/15], Step [600/1563], Loss: 0.3100
Epoch [6/15], Step [700/1563], Loss: 0.3570
Epoch [6/15], Step [800/1563], Loss: 0.3424
Epoch [6/15], Step [900/1563], Loss: 0.3447
Epoch [6/15], Step [1000/1563], Loss: 0.3541
Epoch [6/15], Step [1100/1563], Loss: 0.3864
Epoch [6/15], Step [1200/1563], Loss: 0.3436
```

```
Epoch [6/15], Step [1300/1563], Loss: 0.3728
Epoch [6/15], Step [1400/1563], Loss: 0.3394
Epoch [6/15], Step [1500/1563], Loss: 0.3606
Test Accuracy: 72.76%
Epoch [7/15], Step [100/1563], Loss: 0.2155
Epoch [7/15], Step [200/1563], Loss: 0.2000
Epoch [7/15], Step [300/1563], Loss: 0.2334
Epoch [7/15], Step [400/1563], Loss: 0.2063
Epoch [7/15], Step [500/1563], Loss: 0.2537
Epoch [7/15], Step [600/1563], Loss: 0.2369
Epoch [7/15], Step [700/1563], Loss: 0.2655
Epoch [7/15], Step [800/1563], Loss: 0.2713
Epoch [7/15], Step [900/1563], Loss: 0.2534
Epoch [7/15], Step [1000/1563], Loss: 0.2882
Epoch [7/15], Step [1100/1563], Loss: 0.2558
Epoch [7/15], Step [1200/1563], Loss: 0.2492
Epoch [7/15], Step [1300/1563], Loss: 0.2783
Epoch [7/15], Step [1400/1563], Loss: 0.3303
Epoch [7/15], Step [1500/1563], Loss: 0.2834
Test Accuracy: 72.37%
Epoch [8/15], Step [100/1563], Loss: 0.1844
Epoch [8/15], Step [200/1563], Loss: 0.1580
Epoch [8/15], Step [300/1563], Loss: 0.1703
Epoch [8/15], Step [400/1563], Loss: 0.1754
Epoch [8/15], Step [500/1563], Loss: 0.1991
Epoch [8/15], Step [600/1563], Loss: 0.2047
Epoch [8/15], Step [700/1563], Loss: 0.2166
Epoch [8/15], Step [800/1563], Loss: 0.2229
Epoch [8/15], Step [900/1563], Loss: 0.2005
Epoch [8/15], Step [1000/1563], Loss: 0.2201
Epoch [8/15], Step [1100/1563], Loss: 0.2107
Epoch [8/15], Step [1200/1563], Loss: 0.2204
Epoch [8/15], Step [1300/1563], Loss: 0.2326
Epoch [8/15], Step [1400/1563], Loss: 0.2449
Epoch [8/15], Step [1500/1563], Loss: 0.2296
Test Accuracy: 73.50%
Epoch [9/15], Step [100/1563], Loss: 0.1352
Epoch [9/15], Step [200/1563], Loss: 0.1360
Epoch [9/15], Step [300/1563], Loss: 0.1631
Epoch [9/15], Step [400/1563], Loss: 0.1633
Epoch [9/15], Step [500/1563], Loss: 0.1663
Epoch [9/15], Step [600/1563], Loss: 0.1709
Epoch [9/15], Step [700/1563], Loss: 0.1710
Epoch [9/15], Step [800/1563], Loss: 0.1759
Epoch [9/15], Step [900/1563], Loss: 0.1610
Epoch [9/15], Step [1000/1563], Loss: 0.1981
Epoch [9/15], Step [1100/1563], Loss: 0.1882
Epoch [9/15], Step [1200/1563], Loss: 0.2003
Epoch [9/15], Step [1300/1563], Loss: 0.1812
Epoch [9/15], Step [1400/1563], Loss: 0.2106
Epoch [9/15], Step [1500/1563], Loss: 0.2108
Test Accuracy: 71.71%
Epoch [10/15], Step [100/1563], Loss: 0.1232
```

```
Epoch [10/15], Step [200/1563], Loss: 0.1217
Epoch [10/15], Step [300/1563], Loss: 0.1338
Epoch [10/15], Step [400/1563], Loss: 0.1252
Epoch [10/15], Step [500/1563], Loss: 0.1213
Epoch [10/15], Step [600/1563], Loss: 0.1494
Epoch [10/15], Step [700/1563], Loss: 0.1774
Epoch [10/15], Step [800/1563], Loss: 0.1720
Epoch [10/15], Step [900/1563], Loss: 0.1519
Epoch [10/15], Step [1000/1563], Loss: 0.1444
Epoch [10/15], Step [1100/1563], Loss: 0.1690
Epoch [10/15], Step [1200/1563], Loss: 0.1495
Epoch [10/15], Step [1300/1563], Loss: 0.1467
Epoch [10/15], Step [1400/1563], Loss: 0.1620
Epoch [10/15], Step [1500/1563], Loss: 0.1624
Test Accuracy: 72.37%
Epoch [11/15], Step [100/1563], Loss: 0.1154
Epoch [11/15], Step [200/1563], Loss: 0.1304
Epoch [11/15], Step [300/1563], Loss: 0.0967
Epoch [11/15], Step [400/1563], Loss: 0.1071
Epoch [11/15], Step [500/1563], Loss: 0.1339
Epoch [11/15], Step [600/1563], Loss: 0.1371
Epoch [11/15], Step [700/1563], Loss: 0.1359
Epoch [11/15], Step [800/1563], Loss: 0.1394
Epoch [11/15], Step [900/1563], Loss: 0.1530
Epoch [11/15], Step [1000/1563], Loss: 0.1485
Epoch [11/15], Step [1100/1563], Loss: 0.1453
Epoch [11/15], Step [1200/1563], Loss: 0.1390
Epoch [11/15], Step [1300/1563], Loss: 0.1729
Epoch [11/15], Step [1400/1563], Loss: 0.1536
Epoch [11/15], Step [1500/1563], Loss: 0.1524
Test Accuracy: 73.24%
Epoch [12/15], Step [100/1563], Loss: 0.1082
Epoch [12/15], Step [200/1563], Loss: 0.1008
Epoch [12/15], Step [300/1563], Loss: 0.1025
Epoch [12/15], Step [400/1563], Loss: 0.1070
Epoch [12/15], Step [500/1563], Loss: 0.1183
Epoch [12/15], Step [600/1563], Loss: 0.0969
Epoch [12/15], Step [700/1563], Loss: 0.1066
Epoch [12/15], Step [800/1563], Loss: 0.1150
Epoch [12/15], Step [900/1563], Loss: 0.1408
Epoch [12/15], Step [1000/1563], Loss: 0.1523
Epoch [12/15], Step [1100/1563], Loss: 0.1441
Epoch [12/15], Step [1200/1563], Loss: 0.1164
Epoch [12/15], Step [1300/1563], Loss: 0.1283
Epoch [12/15], Step [1400/1563], Loss: 0.1230
Epoch [12/15], Step [1500/1563], Loss: 0.1463
Test Accuracy: 72.66%
Epoch [13/15], Step [100/1563], Loss: 0.1036
Epoch [13/15], Step [200/1563], Loss: 0.0979
Epoch [13/15], Step [300/1563], Loss: 0.0776
Epoch [13/15], Step [400/1563], Loss: 0.0869
Epoch [13/15], Step [500/1563], Loss: 0.1025
Epoch [13/15], Step [600/1563], Loss: 0.1063
```

```
Epoch [13/15], Step [700/1563], Loss: 0.0906
Epoch [13/15], Step [800/1563], Loss: 0.1411
Epoch [13/15], Step [900/1563], Loss: 0.1185
Epoch [13/15], Step [1000/1563], Loss: 0.1258
Epoch [13/15], Step [1100/1563], Loss: 0.1184
Epoch [13/15], Step [1200/1563], Loss: 0.1461
Epoch [13/15], Step [1300/1563], Loss: 0.1415
Epoch [13/15], Step [1400/1563], Loss: 0.1326
Epoch [13/15], Step [1500/1563], Loss: 0.1090
Test Accuracy: 72.18%
Epoch [14/15], Step [100/1563], Loss: 0.1303
Epoch [14/15], Step [200/1563], Loss: 0.0885
Epoch [14/15], Step [300/1563], Loss: 0.0883
Epoch [14/15], Step [400/1563], Loss: 0.1026
Epoch [14/15], Step [500/1563], Loss: 0.0859
Epoch [14/15], Step [600/1563], Loss: 0.1161
Epoch [14/15], Step [700/1563], Loss: 0.1251
Epoch [14/15], Step [800/1563], Loss: 0.1029
Epoch [14/15], Step [900/1563], Loss: 0.1064
Epoch [14/15], Step [1000/1563], Loss: 0.1037
Epoch [14/15], Step [1100/1563], Loss: 0.1283
Epoch [14/15], Step [1200/1563], Loss: 0.1052
Epoch [14/15], Step [1300/1563], Loss: 0.1416
Epoch [14/15], Step [1400/1563], Loss: 0.1076
Epoch [14/15], Step [1500/1563], Loss: 0.1224
Test Accuracy: 72.60%
Epoch [15/15], Step [100/1563], Loss: 0.1017
Epoch [15/15], Step [200/1563], Loss: 0.0886
Epoch [15/15], Step [300/1563], Loss: 0.0759
Epoch [15/15], Step [400/1563], Loss: 0.0952
Epoch [15/15], Step [500/1563], Loss: 0.0846
Epoch [15/15], Step [600/1563], Loss: 0.0892
Epoch [15/15], Step [700/1563], Loss: 0.1045
Epoch [15/15], Step [800/1563], Loss: 0.1089
Epoch [15/15], Step [900/1563], Loss: 0.1196
Epoch [15/15], Step [1000/1563], Loss: 0.0986
Epoch [15/15], Step [1100/1563], Loss: 0.0982
Epoch [15/15], Step [1200/1563], Loss: 0.1151
Epoch [15/15], Step [1300/1563], Loss: 0.1265
Epoch [15/15], Step [1400/1563], Loss: 0.1047
Epoch [15/15], Step [1500/1563], Loss: 0.1203
Test Accuracy: 72.77%
```



2. 在MNIST数据集上实现CNN：

代码:

```
import torch
import numpy as np
from torchvision import datasets, transforms
import torchvision.transforms as transforms
from torchvision import datasets
import torchvision
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim

import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import os
trainset = datasets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
testset = datasets.MNIST(root='./data', train=False, download=True,
transform=transforms.ToTensor())
# batch_size=64表示每个批次包含64个样本。可以根据硬件（如内存/GPU显存）调整这个值。
# shuffle=True表示在每个epoch开始时都将训练数据集打乱顺序。
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
#对于测试集我们通常不需要打乱顺序
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

class MINSTCNN(nn.Module):
```

```
    def __init__(self):
        super(MINSTCNN, self).__init__()
        #TODO: 实现模型结构
        #TODO 实现self.conv1:卷积层
        self.conv1 = nn.Conv2d(1,16, kernel_size=3, padding=1)
        #TODO 实现self.conv2:卷积层
        self.conv2 = nn.Conv2d(16,16, kernel_size=3, padding=1)
        #TODO 实现self.pool: MaxPool2d
        self.pool = nn.MaxPool2d(kernel_size=2)
        #TODO 实现self.fc1: 线性层
        self.fc1 = nn.Linear(28*28,500)
        #TODO 实现self.fc2：线性层
        self.fc2 = nn.Linear(500,10)
        #TODO 实现 self.dropout: Dropout层
        self.dropout = nn.Dropout2d(0.2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x


def trainMINST(model, train_loader, test_loader, device):
    num_epochs = 3
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)
            #TODO:实现训练部分，完成反向传播过程
            #TODO: optimizer梯度清除
            optimizer.zero_grad()
            #TODO: 模型输入
            outputs = model(inputs)
            #TODO: 计算损失
            loss = criterion(outputs, labels).sum()
            #TODO: 反向传播
            loss.backward()
            #TODO: 更新参数
            optimizer.step()

            running_loss += loss.item()
            if i % 100 == 99:  # 每100个batch打印一次损失
                print(
```

```
                  f'Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1}/{len(train_loader)}],
Loss: {running_loss / 100:.4f}')
                running_loss = 0.0

        #每个epoch结束后在测试集上评估模型
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        print(f'Test Accuracy: {100 * correct / total:.2f}%')

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#创建模型
model = MINSTCNN().to(device)
trainMINST(model, trainloader, testloader, device)
```

实验结果:

```
Epoch [1/3], Step [100/938], Loss: 0.9629
Epoch [1/3], Step [200/938], Loss: 0.3137
Epoch [1/3], Step [300/938], Loss: 0.2045
Epoch [1/3], Step [400/938], Loss: 0.1725
Epoch [1/3], Step [500/938], Loss: 0.1320
Epoch [1/3], Step [600/938], Loss: 0.1035
Epoch [1/3], Step [700/938], Loss: 0.1083
Epoch [1/3], Step [800/938], Loss: 0.0970
Epoch [1/3], Step [900/938], Loss: 0.0921
Test Accuracy: 97.99%
Epoch [2/3], Step [100/938], Loss: 0.0747
Epoch [2/3], Step [200/938], Loss: 0.0665
Epoch [2/3], Step [300/938], Loss: 0.0677
Epoch [2/3], Step [400/938], Loss: 0.0702
Epoch [2/3], Step [500/938], Loss: 0.0569
Epoch [2/3], Step [600/938], Loss: 0.0547
Epoch [2/3], Step [700/938], Loss: 0.0527
Epoch [2/3], Step [800/938], Loss: 0.0480
Epoch [2/3], Step [900/938], Loss: 0.0642
Test Accuracy: 98.31%
Epoch [3/3], Step [100/938], Loss: 0.0420
Epoch [3/3], Step [200/938], Loss: 0.0396
Epoch [3/3], Step [300/938], Loss: 0.0420
Epoch [3/3], Step [400/938], Loss: 0.0469
Epoch [3/3], Step [500/938], Loss: 0.0379
Epoch [3/3], Step [600/938], Loss: 0.0363
Epoch [3/3], Step [700/938], Loss: 0.0483
Epoch [3/3], Step [800/938], Loss: 0.0473
```

```
Epoch [3/3], Step [900/938], Loss: 0.0443
Test Accuracy: 98.53%
```

3. 卷积神经网络（LeNet）

代码:

```
class Mynet(nn.Module):
    def __init__(self):
        super(Mynet, self).__init__()
        self.conv1 = nn.Conv2d(1,6, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(6,16, kernel_size=5)
        self.pool = nn.AvgPool2d(kernel_size=2, stride=2)
        self.sigmoid = nn.Sigmoid()
        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.conv1(x)
        x = self.sigmoid(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.sigmoid(x)
        x = self.pool(x)
        x = x.view(-1,400)
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#创建模型
model = Mynet().to(device)
trainMINST(model, trainloader, testloader, device)
```

实验结果:

```
Epoch [1/3], Step [100/938], Loss: 2.3094
Epoch [1/3], Step [200/938], Loss: 2.3012
Epoch [1/3], Step [300/938], Loss: 1.9088
Epoch [1/3], Step [400/938], Loss: 1.0515
Epoch [1/3], Step [500/938], Loss: 0.7158
Epoch [1/3], Step [600/938], Loss: 0.5331
Epoch [1/3], Step [700/938], Loss: 0.4206
Epoch [1/3], Step [800/938], Loss: 0.3769
Epoch [1/3], Step [900/938], Loss: 0.3328
Test Accuracy: 91.50%
Epoch [2/3], Step [100/938], Loss: 0.3011
Epoch [2/3], Step [200/938], Loss: 0.2827
Epoch [2/3], Step [300/938], Loss: 0.2506
Epoch [2/3], Step [400/938], Loss: 0.2378
```

```
Epoch [2/3], Step [500/938], Loss: 0.2150
Epoch [2/3], Step [600/938], Loss: 0.2167
Epoch [2/3], Step [700/938], Loss: 0.2099
Epoch [2/3], Step [800/938], Loss: 0.1890
Epoch [2/3], Step [900/938], Loss: 0.1884
Test Accuracy: 94.94%
Epoch [3/3], Step [100/938], Loss: 0.1641
Epoch [3/3], Step [200/938], Loss: 0.1644
Epoch [3/3], Step [300/938], Loss: 0.1693
Epoch [3/3], Step [400/938], Loss: 0.1636
Epoch [3/3], Step [500/938], Loss: 0.1687
Epoch [3/3], Step [600/938], Loss: 0.1377
Epoch [3/3], Step [700/938], Loss: 0.1431
Epoch [3/3], Step [800/938], Loss: 0.1453
Epoch [3/3], Step [900/938], Loss: 0.1265
Test Accuracy: 95.98%
```

4. 批量规范化

代码:

```
def nettrainMINST(net, train_loader, test_loader, device):
    num_epochs = 3
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = inputs
            optimizer.zero_grad()
            for layer in net:
                outputs = layer(outputs)
            loss = criterion(outputs, labels).sum()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if i % 100 == 99:  # 每100个batch打印一次损失
                print(
                  f'Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1}/{len(train_loader)}],
Loss: {running_loss / 100:.4f}')
                running_loss = 0.0

        #每个epoch结束后在测试集上评估模型
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
```

```
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        print(f'Test Accuracy: {100 * correct / total:.2f}%')

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
trainMINST(net, trainloader, testloader, device)
```

实验结果;

```
Epoch [1/3], Step [100/938], Loss: 2.3091
Epoch [1/3], Step [200/938], Loss: 2.3001
Epoch [1/3], Step [300/938], Loss: 1.8815
Epoch [1/3], Step [400/938], Loss: 1.1086
Epoch [1/3], Step [500/938], Loss: 0.8073
Epoch [1/3], Step [600/938], Loss: 0.6311
Epoch [1/3], Step [700/938], Loss: 0.5443
Epoch [1/3], Step [800/938], Loss: 0.4333
Epoch [1/3], Step [900/938], Loss: 0.3916
Test Accuracy: 90.58%
Epoch [2/3], Step [100/938], Loss: 0.3173
Epoch [2/3], Step [200/938], Loss: 0.2990
Epoch [2/3], Step [300/938], Loss: 0.2625
Epoch [2/3], Step [400/938], Loss: 0.2399
Epoch [2/3], Step [500/938], Loss: 0.2385
Epoch [2/3], Step [600/938], Loss: 0.2330
Epoch [2/3], Step [700/938], Loss: 0.2217
Epoch [2/3], Step [800/938], Loss: 0.2032
Epoch [2/3], Step [900/938], Loss: 0.1988
Test Accuracy: 94.57%
Epoch [3/3], Step [100/938], Loss: 0.1789
Epoch [3/3], Step [200/938], Loss: 0.1766
Epoch [3/3], Step [300/938], Loss: 0.1638
Epoch [3/3], Step [400/938], Loss: 0.1631
Epoch [3/3], Step [500/938], Loss: 0.1511
Epoch [3/3], Step [600/938], Loss: 0.1443
Epoch [3/3], Step [700/938], Loss: 0.1306
Epoch [3/3], Step [800/938], Loss: 0.1370
Epoch [3/3], Step [900/938], Loss: 0.1347
Test Accuracy: 96.05%
```

实验感想及收获

通过本次实验，我对卷积神经网络（CNN）在图像处理任务中的应用有了更深入的理解，手动实现了神经网络的学习过程，学会了如何通过调整网络结构、优化器和正则化技术来提高模型的性能。这些知识和技能对我今后在深度学习领域的学习和研究具有重要的指导意义。同时，我也意识到在实际应用中，数据预处理、模型调参和结果分析同样重要，这些都是未来需要进一步学习和实践的方向。