lab3

网络架构

实验3的网络架构在卷积部分与lab2中的网络时相似的，目的是特征抽取，但是在MLP的部分，我将两张图片卷积后的结果展开，利用 `torch.cat` 将他们拼接在一起，然后送入全连接层进行判断。网络结构如下。

```
# 定义卷积神经网络
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5)
        self.fc1 = nn.Linear(2048, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, 1)
        self.sigmoid = nn.Sigmoid()


    def forward(self, x1, x2):
        x1 = self.conv1(x1)
        x1 = nn.functional.relu(x1)
        x1 = nn.functional.max_pool2d(x1, 2)
        x1 = self.conv2(x1)
        x1 = nn.functional.relu(x1)
        x1 = nn.functional.max_pool2d(x1, 2)
        x1 = x1.view(-1, 1024)

        x2 = self.conv1(x2)
        x2 = nn.functional.relu(x2)
        x2 = nn.functional.max_pool2d(x2, 2)
        x2 = self.conv2(x2)
        x2 = nn.functional.relu(x2)
        x2 = nn.functional.max_pool2d(x2, 2)
        x2 = x2.view(-1, 1024)

        x = torch.cat((x1, x2), dim=1)
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.fc2(x)
        x = nn.functional.relu(x)
        x = self.fc3(x)
        x = self.sigmoid(x)
        return x

# 实例化模型和定义优化器
model = CNN()
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), momentum=0.9,lr=0.002)
```

数据集

这次的数据集有所不同，因为一次要抽取两张图片，所以我们使用 `from torch.utils.data import Dataset, DataLoader` 来自定义我们的数据集。

```
# 定义用于加载两张图片的自定义数据集类
class TwoMNISTDataset(Dataset):
    def __init__(self, mnist_dataset):
        self.mnist_dataset = mnist_dataset

    def __len__(self):
        return len(self.mnist_dataset)

    def __getitem__(self, idx):
        if(idx%2):
            img1, label1 = self.mnist_dataset[idx]
            img2, label2 = self.mnist_dataset[idx]
        else:
            img1, label1 = self.mnist_dataset[abs(len(self.mnist_dataset)-idx-1)]
            img2, label2 = self.mnist_dataset[idx]
        return img1, label1, img2, label2, torch.tensor((label1==label2), dtype=torch.float32)


mnist_train = MNIST('./data', train=True, download=True, transform=torchvision.transforms.ToTensor())
mnist_test = MNIST('./data', train=False, download=True, transform=torchvision.transforms.ToTensor())

train_dataset = TwoMNISTDataset(mnist_train)
test_dataset = TwoMNISTDataset(mnist_test)
```

这里有一个很有意思的问题，就是假如我每一次都选取数据集中相邻的两张图片，然后将他们送入神经网络，这样的话，按照概率学上， `label` 为1的概率为10%，为0的概率为90%，那么对于神经网络来说，他只要重复的输出0，无论他的输入是什么，就能获得百分之90的准确率，这样会字节导致我们的训练出现大问题，所以我人为的改变了抽取样本的方式，在数据集中混入了百分之五十的两张相同的图片，也就是混入了百分之50的 `label` 为1的数据，这使得我们的训练得以成功的进行下去。

而且为什么使用两张相同的图片呢？我认为这个是有一定好处的，因为这里理论上根据我们网络的架构来说，根据对称性，两张图片经过的卷积层应该是相同的，如果我们在数据集中混入了两张相同的图片，会有利于网络的对称性。（只是臆测）

训练

采用标准的训练方式，代码如下，测试代码与训练相似

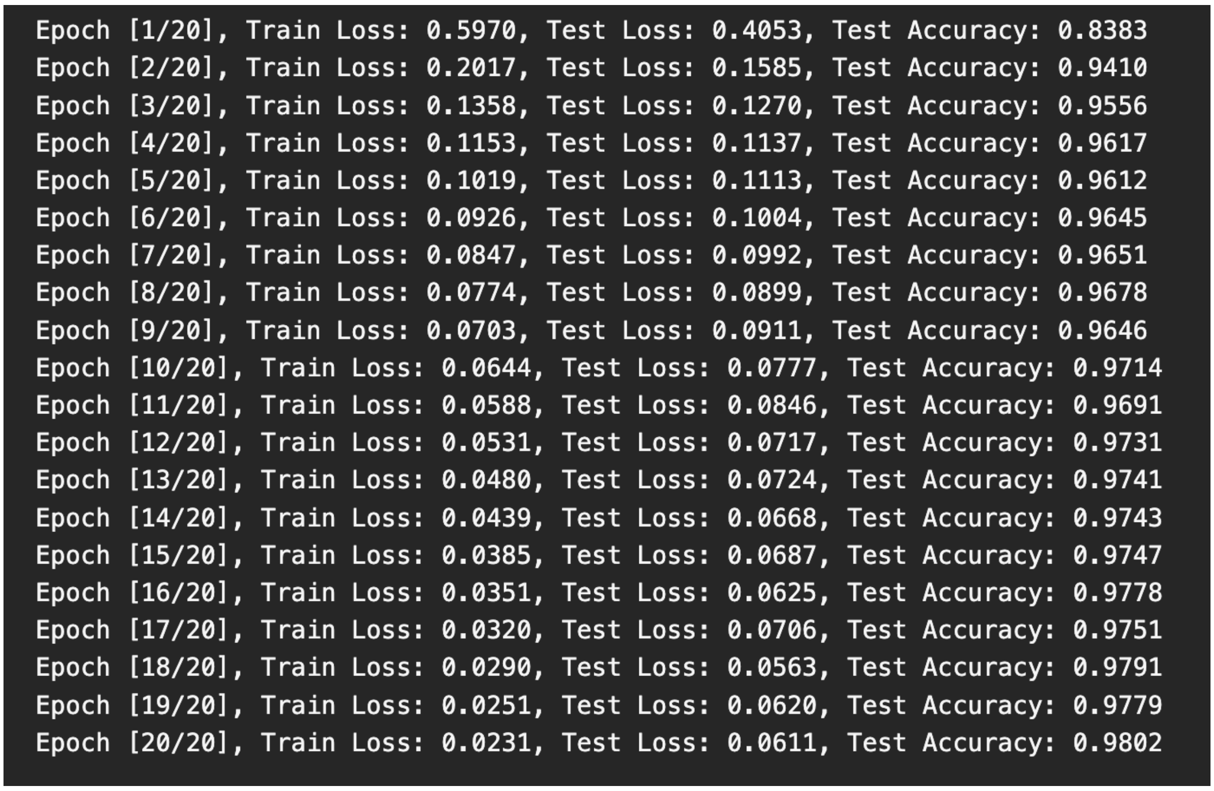
```
# 定义训练和测试函数
def train(model, criterion, optimizer, train_loader):
    model.train()
    train_loss = 0.0
    for data in train_loader:
        img1, _, img2, _, label = data
        optimizer.zero_grad()
        output = model(img1, img2)
        loss = criterion(output,label.unsqueeze(1))
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
    return train_loss / len(train_loader)

def test(model, criterion, test_loader):
    model.eval()
    correct = 0
    total = 0
    test_loss = 0
    with torch.no_grad():
        for data in test_loader:
            img1, _, img2, _, label = data
            outputs = model(img1, img2)
            loss = criterion(outputs,label.unsqueeze(1))
            predicted = torch.round(outputs)
            total += label.size(0)
            correct += (predicted == label.unsqueeze(1)).sum().item()
            test_loss += loss.item()
    return test_loss/len(test_loader), correct / total
```

实验

实验环境为MacOS Apple Silicon M1 Pro，Python版本3.9，Pytorch版本2.0.0(MacOS版)

训练了10个Epoch实验结果如下



```
Epoch [1/20], Train Loss: 0.5970, Test Loss: 0.4053, Test Accuracy: 0.8383
Epoch [2/20], Train Loss: 0.2017, Test Loss: 0.1585, Test Accuracy: 0.9410
Epoch [3/20], Train Loss: 0.1358, Test Loss: 0.1270, Test Accuracy: 0.9556
Epoch [4/20], Train Loss: 0.1153, Test Loss: 0.1137, Test Accuracy: 0.9617
Epoch [5/20], Train Loss: 0.1019, Test Loss: 0.1113, Test Accuracy: 0.9612
Epoch [6/20], Train Loss: 0.0926, Test Loss: 0.1004, Test Accuracy: 0.9645
Epoch [7/20], Train Loss: 0.0847, Test Loss: 0.0992, Test Accuracy: 0.9651
Epoch [8/20], Train Loss: 0.0774, Test Loss: 0.0899, Test Accuracy: 0.9678
Epoch [9/20], Train Loss: 0.0703, Test Loss: 0.0911, Test Accuracy: 0.9646
Epoch [10/20], Train Loss: 0.0644, Test Loss: 0.0777, Test Accuracy: 0.9714
Epoch [11/20], Train Loss: 0.0588, Test Loss: 0.0846, Test Accuracy: 0.9691
Epoch [12/20], Train Loss: 0.0531, Test Loss: 0.0717, Test Accuracy: 0.9731
Epoch [13/20], Train Loss: 0.0480, Test Loss: 0.0724, Test Accuracy: 0.9741
Epoch [14/20], Train Loss: 0.0439, Test Loss: 0.0668, Test Accuracy: 0.9743
Epoch [15/20], Train Loss: 0.0385, Test Loss: 0.0687, Test Accuracy: 0.9747
Epoch [16/20], Train Loss: 0.0351, Test Loss: 0.0625, Test Accuracy: 0.9778
Epoch [17/20], Train Loss: 0.0320, Test Loss: 0.0706, Test Accuracy: 0.9751
Epoch [18/20], Train Loss: 0.0290, Test Loss: 0.0563, Test Accuracy: 0.9791
Epoch [19/20], Train Loss: 0.0251, Test Loss: 0.0620, Test Accuracy: 0.9779
Epoch [20/20], Train Loss: 0.0231, Test Loss: 0.0611, Test Accuracy: 0.9802
```