计算机视觉实验3

曾世鹏 _ U202115574 _CS2108

一、实验需求分析

1. 读取MNIST数据集。
2. 构建图片对数据集
3. 构建train_loader和test_loader。
4. 构建卷积神经网络，实现二分类。

二、实验环境介绍

1. 使用pytorch框架
2. 导入random

三、参数介绍

- 超参数设置

```
# setting
batch_size = 64
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
epochs = 45
```

四、实现思路

1. 获取MNIST数据集，开启download=True,从网上获取
2. 用Subset和random.sample通过取10%索引的方式取10%的样本作为训练集和测试集原集
3. 将10%的数据先分入0-9十个类别中，对每个类别中的图片，随机取n张同类别中的随机图片建立同类的图片对，再随机取m次，从剩下不同类别的图片集中取一张建立不同类的图片对，最终获得原图片量*（n+m）个图片对的数据集。
4. 分别用构建好后的数据集构建train_loader和test_loader。
5. 构建神经网络，使用了三个卷积模块和两层全连接层
    - 卷积模块：卷积层（卷积核为3×3）+BN层+ReLU激活+均值池化（卷积核为2×2）
    - 全连接层1：线性64×7×7（前面两个池化从28->14->7)->100(两两数字组合，认为有10*10种可能)+Sigmoid激活
    - 全连接层2：线性100->1
6. 实例化：损失函数因为二分类且前面用sigmoid选BCEWithLogitsLoss，优化器选用Adam
7. 训练模型并测试

五、具体实现

5.1 构建数据集

1. 加载MNIST数据集

```
# 加载mnist数据集
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)
```

2. 取10%的样本作为原始集，使用random.sample配合Subset实现从索引取10%作为数据集。

```
# 取10%的样本作为训练集和测试集 因为前面已经取了train为true和false，所以是隔离的，不会重复，直接取10%就可以,用效率高的方法随机取10%
sub_train_dataset = torch.utils.data.Subset(train_dataset,

 random.sample(range(len(train_dataset)), int(len(train_dataset) * 0.1)))
sub_test_dataset = torch.utils.data.Subset(test_dataset,

random.sample(range(len(test_dataset)), int(len(test_dataset) * 0.1)))
print("dataset created! ")
```

3. 转换为自定义数据集,先把原数据集进行分10类，再在每个类中每张图片选择n张同类别中的随机不同图片构建图片对，选择m张不同随机类别中的随机类别构建不同图片对。使用shuffle打乱顺序。

```
# 创建自定义数据集类，用于创建图片对
# 两两之间建立图片对，用百分之十的数据集就可以创建数据量和原来大小一样的数据集
class PairsDataset(Dataset):
    def __init__(self, mnist_dataset):
        self.data = mnist_dataset
        self.pairs = self.create_pairs()

    # 创建的数据集要相同和不相同的图片对比例为1:1，决定先把0-9的图片分开，对类别i里的图片k，随机取一张和他不同的类别i里的图片j，这样就创建了一对相同的图片对，然后再随机取一张和他不同的类别k中的图片p，这样就创建了一对不同的图片对
    def create_pairs(self):
        # 先把0-9的图片分开
        data = {}
        for i in range(10):
            data[i] = []
        for img, label in self.data:
            data[label].append(img)
        # 创建相同的图片对
        pairs = []
        for i in range(10):
            for times in range(5):
                for k in range(len(data[i])):
                    # 随机取一张和他不同的类别i里的图片j
                    j = random.randint(0, len(data[i]) - 1)
                    while j == k:
                        j = random.randint(0, len(data[i]) - 1)
```

```
                    # 创建一对相同的图片对
                    pairs.append((data[i][k], data[i][j], 1))
        # 创建不同的图片对
        for i in range(10):
            for times in range(7):
                for k in range(len(data[i])):
                    # 随机取一张和他不同的类别k中的图片p
                    p = random.randint(0, 9)
                    while p == i:
                        p = random.randint(0, 9)
                    j = random.randint(0, len(data[p]) - 1)
                    # 创建一对不同的图片对
                    pairs.append((data[i][k], data[p][j], 0))
        # 打乱顺序
        random.shuffle(pairs)
        # print一下个数
        print("pairs num:", len(pairs))
        return pairs

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

    def __getitem__(self, idx):
        img1, img2, label = self.pairs[idx]
        return img1, img2, label

# 转换为自定义数据集
sub_train_dataset = PairsDataset(sub_train_dataset)
sub_test_dataset = PairsDataset(sub_test_dataset)
# 创建数据加载器
train_loader = DataLoader(sub_train_dataset, batch_size=batch_size,
shuffle=True)
# 测试集的batch_size要大一点，因为测试集的数据量比较小，如果batch_size太小，会导致测试集
的准确率不稳定
test_loader = DataLoader(sub_test_dataset, batch_size=2000, shuffle=True)
print("all data loaded! ")
```

5.2 构建神经网络

1. 网络初始化，第一个卷积模块，图片对会在前向过程在通道上拼接，所以输入通道为2，第一层输出通道数设为16；卷积核为5和为3时要注意padding记得改变，否则会出现两层图片尺寸不匹配的问题；因为是图像匹配类问题，需要提取共同特征以应对平移旋转等变化，加入池化层，尝试了最大值池化和均值池化，发现均值池化效果比较好，具体尝试见尝试记录。

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # 第一层卷积层
        self.conv1 = nn.Sequential(
            # 输入的有两张图片，所以输入通道为2
            # mat1 and mat2 shapes cannot be multiplied (64x2048 and
1568x100)
```

```
            # 这是因为卷积核的大小和步长的原因，导致卷积后的图片大小不是28*28，而是
26*26，所以要改变padding的大小,应该改为1，因为只吞掉了中间的左边一格，如果kernel为5，
padding应该为2

            nn.Conv2d(in_channels=2, out_channels=16, kernel_size=3,
stride=1, padding=1),
            # bn层
            nn.BatchNorm2d(16),
            nn.ReLU(),
            # 池化层使用均值池化，池化核的大小为2*2，步长为2，这样池化后的图片大小变为原
来的一半
            nn.AvgPool2d(kernel_size=2)
        )
```

2. 剩余卷积模块

```
 # 第二层卷积层
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2)
        )
 # 第三层卷积层
        self.conv3 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            # 使用均值池化
            # nn.AvgPool2d(kernel_size=7)
        )
```

3. 全连接层，根据前面池化情况调整最终神经元个数，并设定初始特征数，尝试了10,2,100,最终效果比较好为100，暂时可以理解为可以落入100种情况（10类拼10类），最终二分类直接输出单通道，根据靠近0还是1判断是否相同，而非双通道一个为0一个为1

```
    # 全连接层
        # 这里的全连接层的输入是32*7*7，因为经过两次池化，图片的大小变为原来的1/4，所以是
28/4=7
        self.fc1 = nn.Sequential(
            nn.Linear(64 * 7 * 7, 100),
            # 用sigmoid函数来将输出限制在0-1之间
            nn.Sigmoid()
        )
        # 最后的输出层,sigmoid函数将输出限制在0-1之间
        self.fc2 = nn.Linear(100, 1)
```

4. 前向过程，先使用cat拼接为双通道，再经过前向过程

```
    def forward(self, x):
        # 两张图片合并在一起
```

```
        # cat函数是将两个张量（tensor）拼接在一起，dim=1表示在第一个维度上拼接，即在通道上拼接
        x = torch.cat((x[0], x[1]), dim=1)
        # 卷积层
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        # 将卷积层的输出展平
        x = x.view(x.size(0), -1)
        # 全连接层
        x = self.fc1(x)
        # 输出层
        x = self.fc2(x)
        # sigmoid函数将输出限制在0-1之间
        # x = torch.softmax(x, dim=1)
        return x
```

5.3 实例化

构建模型，并放入GPU，构建损失函数和优化器

```
# 初始化模型和优化器、损失函数
model = CNN().to(device)
print("model loaded! ")
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
print("optimizer loaded! ")
# 二分类问题，所以用二分类的损失函数,这里用了sigmoid函数，所以用BCEWithLogitsLoss
criterion = nn.BCEWithLogitsLoss()
print("criterion loaded! ")
```

5.4 训练模型

训练的时候，将label转为和output一样的Float类型，并把张量shape对齐，用于计算loss

```
# 训练模型
model.train()
for epoch in range(epochs):
    for i, (img1, img2, label) in enumerate(train_loader):
        img1 = img1.to(device)
        img2 = img2.to(device)

        # 前向传播

        output = model((img1, img2))
        # 将label的类型转换为Float，因为后面计算损失的时候需要
        label = label.type(torch.FloatTensor)
        # 将label的shape转换为和output一样的shape
        label = label.view(output.shape)
        label = label.to(device)
        # 计算损失
        loss = criterion(output, label)
        # 反向传播
        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()
        # 打印损失
        if i % 100 == 0:
            print('epoch: {}, step: {}, loss: {}'.format(epoch, i, loss.item()))
                ))
```

5.6 测试模型

将pred转为label一样的shape，并转为int比较

```
# 测试模型
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for i, (img1, img2, label) in enumerate(test_loader):
        img1 = img1.to(device)
        img2 = img2.to(device)

        # 前向传播
        output = model((img1, img2))
        # 计算准确率
        pred = torch.round(torch.sigmoid(output))
        # 将pred的shape转换为和label一样的shape
        pred = pred.view(label.shape)
        # 转为int类型
        pred = pred.type(torch.IntTensor)
        # 记得别丢gpu上了

        # debug 输出预测值和真实值
        # print("label:")
        # print(label)
        # print("pred:")
        # print(pred)
        correct += (pred == label).sum().item()
        total += label.size(0)
print('accuracy: {}'.format(correct / total))
```

六、实验记录

本次实验总体框架很快写完，但运行时遇到的问题较多，进行了很多调整才达到满意的准确率(97%)，所以尝试记录比较长，主要分为数据集构建和网络调整两方面的记录。

6.1 图片对创建方式记录

1.建立数据对

一开始使用九九乘法口诀那种，双重循环，自己以后的所有都建立数据对，但实在太慢了（理论有6000+5999+...+1对)

2.修正：打算一个相同对才能容纳一个不同对，限制不同对的数量和相同对1:1以减少不同对数量，但还是很慢。

3.限制单个图片获得了len*0.01(大约60对),就跳出，但还是非常慢，

所以打算用2的方法进行跑，直接在hpc上用screen挂着

4.最后不限制个数了，直接随便找一张和自己不同的匹配，相同和不同的差不多1:9

5.数据集严重不均衡，跑出来结果很烂，只有0.57，感觉数据集这样不是经典的得改加权损失函数，太麻烦，还是搞下数据集

6.花点时间重写了下快速匹配逻辑，先把0-9图片分开，建立索引，直接每个类别在本类中建立num个相同pair，随机和别的类建立num个不同pair，实现快速创建1:1数据集，最终选用num为5，acc达到97，速度也很快。

6.2 结果记录

1.epoch10 1:10（一张图片获得k对）创建数据集

```
epoch: 9, step: 900, loss: 0.24284107983112335
accuracy: 57.5008
```

2.epoch40 1:10创建数据集

结果也是57.多，说明已经拟合，不是epoch问题

3.epoch 20 1:50创建数据集

训练集

same pairs: 30107, diff pairs: 269893

测试集

same pairs: 4950, diff pairs: 45050

结果：

```
epoch: 19, step: 4600, loss: 0.2426588088274002
accuracy: 57.65056
```

感觉是网络问题了，超参和数据集调整变化都不大

4.数据集1:20 epoch20

将fc1从32*7 * 7 到10 再10到1 变为了到2再到1

结果还是这样，问了下老师，感觉还是要做一点控制样本比例

4.5 按照比例1:1创建数据集，先分开为10类再进行匹配这样快很多，果然建立index是有必要的，效果马上好很多

现在 训练集和数据集

pairs num: 12000
pairs num: 2000

epoch60的情况下，

```
epoch: 57, step: 0, loss: 0.007227592170238495
epoch: 57, step: 100, loss: 0.0010115278419107199
epoch: 58, step: 0, loss: 0.003464236855506897
epoch: 58, step: 100, loss: 0.0029491642490029335
epoch: 59, step: 0, loss: 0.0019366780761629343
epoch: 59, step: 100, loss: 0.0013292803196236491
accuracy: 0.9
```

5.再试下把fc1改为到2，fc2改为2到1，按照二分类的思想

二分类但不能这么想，最后的二分类是体现到1个channel上，看偏向0还是1的，特征应该有更多，不该局限为2，最后为1已经是二分类的思想了

```
epoch: 68, step: 0, loss: 0.6933939456939697
epoch: 68, step: 100, loss: 0.6933144927024841
epoch: 69, step: 0, loss: 0.6932605504989624
epoch: 69, step: 100, loss: 0.6929800510406494
accuracy: 0.5
```

6.改为fc1为100，因为10*10种组合,最终的情况应该有100种，再连到1进行二分类

看训练效果非常好，loss到e-6级了，但最终结果还是没到预期

```
epoch: 67, step: 0, loss: 1.8619426555233076e-05
epoch: 67, step: 100, loss: 9.22539948078338e-06
epoch: 68, step: 0, loss: 6.1577702581416816e-06
epoch: 68, step: 100, loss: 1.3041415513725951e-05
epoch: 69, step: 0, loss: 1.5544039342785254e-05
epoch: 69, step: 100, loss: 8.946026355260983e-06
accuracy: 0.9205
```

7.试下增加点数据量,6是一张图片创建一个相同的图片对和不同的图片对,现在在创建数据集那加入times循环，先试下times设2，创建相同和不同分别为1:2

扩大为：

pairs num: 24000
pairs num: 4000

其实看epoch在30不到就loss到0.001级别了，感觉会有过拟合，也没设置早停策略，下次epoch 调小一点

```
epoch: 68, step: 300, loss: 1.8440111659856484e-07
epoch: 69, step: 0, loss: 2.0489094865183688e-08
epoch: 69, step: 100, loss: 3.352760558072987e-08
epoch: 69, step: 200, loss: 6.891780657269919e-08
epoch: 69, step: 300, loss: 8.940689610881236e-08
accuracy: 0.92075
```

8.epoch调为30

有一定效果，但还没达到预期，感觉还能早一点

```
epoch: 29, step: 0, loss: 0.00018792756600305438
epoch: 29, step: 100, loss: 0.00025915788137353957
epoch: 29, step: 200, loss: 0.00029104994609951973
epoch: 29, step: 300, loss: 0.00014653051039204001
accuracy: 0.93625
```

9.epoch调为20

```
epoch: 19, step: 100, loss: 0.007557747885584831
epoch: 19, step: 200, loss: 0.005784124135971069
epoch: 19, step: 300, loss: 0.01696370169520378
accuracy: 0.9255
```

感觉现在差不多就这样了epoch在30左右比较好，可能还是网络的问题

10.将卷积核改为3，把bn层加上了

效果还是没到预期

```
accuracy: 0.93325
```

11.再来一层卷积层吧，第二层不加maxpool（这里之前的一直用maxpool）

网络：

```
# 第一层卷积层
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=2, out_channels=16, kernel_size=3, stride=1,
padding=1),
            # bn层
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        # 第二层卷积层
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1,
padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            # nn.MaxPool2d(kernel_size=2)
        )
        # 第三层卷积层
        self.conv3 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1,
padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        # 全连接层
```

```
        # 这里的全连接层的输入是32*7*7，因为经过两次池化，图片的大小变为原来的1/4，所以是
28/4=7
        self.fc1 = nn.Sequential(
            nn.Linear(64 * 7 * 7, 100),
            # 用sigmoid函数来将输出限制在0-1之间
            nn.Sigmoid()
        )
        # 最后的输出层
        self.fc2 = nn.Linear(100, 1)
```

结果改变不大，0.93

12. 忘了加了网络层后得多几个epoch了，调了epochs=30后涨了

```
epoch: 29, step: 100, loss: 0.07466570287942886
epoch: 29, step: 200, loss: 0.017667517066001892
epoch: 29, step: 300, loss: 0.009270175360143185
label:
tensor([1, 0, 0,  ..., 0, 1, 1])
pred:
tensor([1, 0, 0,  ..., 0, 1, 1], dtype=torch.int32)
label:
tensor([1, 1, 1,  ..., 1, 1, 0])
pred:
tensor([1, 1, 1,  ..., 1, 1, 0], dtype=torch.int32)
accuracy: 0.9435
```

13. **试了下50，有点过拟合了，不如30**

14. **感觉又最大值池化可能会有点问题，先试下均值池化，不行就把池化去了**

    均值池化好像好一点

    ```
    epoch: 38, step: 200, loss: 9.107562800636515e-05
    epoch: 38, step: 300, loss: 6.987738015595824e-05
    epoch: 39, step: 0, loss: 2.6699543013819493e-05
    epoch: 39, step: 100, loss: 6.991713598836213e-05
    epoch: 39, step: 200, loss: 0.17281535267829895
    epoch: 39, step: 300, loss: 0.0674944669008255
    accuracy: 0.9545
    ```

15.把前面池化去了，只用倒数第二层全连接连回去64*28 *28

参数量有扩大四倍确实慢了很多，loss看着也挺大，可能epoch要多点才能收敛

```
epoch: 39, step: 0, loss: 0.016112878918647766
epoch: 39, step: 100, loss: 0.006402531173080206
epoch: 39, step: 200, loss: 0.01596333272755146
epoch: 39, step: 300, loss: 0.03451954573392868
accuracy: 0.944
```

16. 将三层卷积都用池化，前两层为2 * 2，最后一层为 7 * 7 ，fc1调为64 *1 *1 loss降不下去，最后直接7 * 7太狠了

好像不是这个的问题，是不小心最后return前多加了一层sigmoid

但结果也不好，直接7*7丢失太多信息了

```
epoch: 39, step: 200, loss: 0.08969186991453171
epoch: 39, step: 300, loss: 0.047272320836782455
label:
tensor([1, 0, 0,  ..., 1, 0, 1])
pred:
tensor([1, 0, 1,  ..., 1, 0, 1], dtype=torch.int32)
label:
tensor([0, 0, 0,  ..., 1, 1, 1])
pred:
tensor([0, 0, 1,  ..., 1, 1, 1], dtype=torch.int32)
accuracy: 0.89225
```

17. **第二层用均值池化 最后一层不用，结果0.94**

18. **感觉网络好像没什么想改的了，再试下加数据集，一张图片创建五个相同对1:5和七个不同对1:7，达到预期0.97**

    呃，原来是数据集的问题，加多了点后达到预期了

    ```
    epoch: 39, step: 0, loss: 0.0010084829991683364
    epoch: 39, step: 100, loss: 8.389785944018513e-05
    epoch: 39, step: 200, loss: 0.000373548042261973
    epoch: 39, step: 300, loss: 0.0011921873083338141
    epoch: 39, step: 400, loss: 8.521764539182186e-05
    epoch: 39, step: 500, loss: 0.0004389956593513489
    epoch: 39, step: 600, loss: 8.498151873936877e-05
    epoch: 39, step: 700, loss: 0.00035886967089027166
    epoch: 39, step: 800, loss: 0.0007858683820813894
    epoch: 39, step: 900, loss: 0.00023107536253519356
    epoch: 39, step: 1000, loss: 0.006982605438679457
    epoch: 39, step: 1100, loss: 0.0033636679872870445
    accuracy: 0.973
    ```

    epoch调45

    ```
    epoch: 44, step: 0, loss: 0.001250970526598394
    epoch: 44, step: 100, loss: 0.002057656180113554
    epoch: 44, step: 200, loss: 0.0005242601037025452
    epoch: 44, step: 300, loss: 0.0005179892759770155
    epoch: 44, step: 400, loss: 0.00023276894353330135
    epoch: 44, step: 500, loss: 0.00036967589403502643
    epoch: 44, step: 600, loss: 0.00018837903917301446
    epoch: 44, step: 700, loss: 0.00027061329456046224
    epoch: 44, step: 800, loss: 0.0015647455584257841
    epoch: 44, step: 900, loss: 0.00021815230138599873
    epoch: 44, step: 1000, loss: 0.00011649538646452129
    epoch: 44, step: 1100, loss: 0.0015548078808933496
    accuracy: 0.9783333333333334
    ```

七、注意点与总结

本次实验原理比较简单，只要cat一下后从双通道的数据中最后得到二分类结果就行，但过程中遇到了包括创建数据集在内（其实这个是花费时间最久的板块）的许多问题。

1. 一开始创建数据集时没有分类建立索引，直接双重for循环，每张图片在全部图片中寻找label和自己相同的和不同的建立图片对，遍历寻找太浪费时间了；后面先把每张图片根据label放到10个类的列表中，再进行配对构建，就省去重复扫描的时间。
2. 数据集类别数量不均衡这个常见问题一定要注意，尽量拉平。
3. 二分类问题最终输出单通道，根据靠近0还是靠近1来进行二分类，前面小特征种类数根据分类任务特点进行调整
4. 图像匹配这种情况使用池化层提取共同特征，来适应旋转变形等情况。

八、附录

总体代码如下：

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
import random
import numpy as np

print("import success!")
# setting
batch_size = 64
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
epochs = 45


# 创建自定义数据集类，用于创建图片对
# 两两之间建立图片对，用百分之十的数据集就可以创建数据量和原来大小一样的数据集
class PairsDataset(Dataset):
    def __init__(self, mnist_dataset):
        self.data = mnist_dataset
        self.pairs = self.create_pairs()

    # 创建的数据集要相同和不相同的图片对比例为1:1，决定先把0-9的图片分开，对类别i里的图片k，随机取一张和他不同的类别i里的图片j，这样就创建了一对相同的图片对，然后再随机取一张和他不同的类别k中的图片p，这样就创建了一对不同的图片对
    def create_pairs(self):
        # 先把0-9的图片分开
        data = {}
        for i in range(10):
            data[i] = []
        for img, label in self.data:
            data[label].append(img)
        # 创建相同的图片对
        pairs = []
        for i in range(10):
```

```
            for times in range(5):
                for k in range(len(data[i])):
                    # 随机取一张和他不同的类别i里的图片j
                    j = random.randint(0, len(data[i]) - 1)
                    while j == k:
                        j = random.randint(0, len(data[i]) - 1)
                    # 创建一对相同的图片对
                    pairs.append((data[i][k], data[i][j], 1))
        # 创建不同的图片对
        for i in range(10):
            for times in range(7):
                for k in range(len(data[i])):
                    # 随机取一张和他不同的类别k中的图片p
                    p = random.randint(0, 9)
                    while p == i:
                        p = random.randint(0, 9)
                    j = random.randint(0, len(data[p]) - 1)
                    # 创建一对不同的图片对
                    pairs.append((data[i][k], data[p][j], 0))
        # 打乱顺序
        random.shuffle(pairs)
        # print一下个数
        print("pairs num:", len(pairs))
        return pairs

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

    def __getitem__(self, idx):
        img1, img2, label = self.pairs[idx]
        return img1, img2, label


# 数据预处理和加载
# transform是一个转换器，]Compose是将多个转换器组合起来
transform = transforms.Compose([
    # 将图片转换为tensor
    transforms.ToTensor(),
    # 并归一化像素值，这里的值是MNIST数据集的均值和标准差
    torchvision.transforms.Normalize(
        (0.1307,), (0.3081,))
])
print("transform loaded! ")
# 加载mnist数据集
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)
print("dataset loaded! ")
# 取10%的样本作为训练集和测试集 因为前面已经取了train为true和false，所以是隔离的，不会重复，直接取10%就可以,用效率高的方法随机取10%
sub_train_dataset = torch.utils.data.Subset(train_dataset,

 random.sample(range(len(train_dataset)), int(len(train_dataset) * 0.1)))
sub_test_dataset = torch.utils.data.Subset(test_dataset,
```

```
random.sample(range(len(test_dataset)), int(len(test_dataset) * 0.1)))
print("dataset created! ")
# 转换为自定义数据集
sub_train_dataset = PairsDataset(sub_train_dataset)
sub_test_dataset = PairsDataset(sub_test_dataset)
print("dataset transformed! ")
# 创建数据加载器
train_loader = DataLoader(sub_train_dataset, batch_size=batch_size, shuffle=True)
# 测试集的batch_size要大一点，因为测试集的数据量比较小，如果batch_size太小，会导致测试集的准确率不稳定
test_loader = DataLoader(sub_test_dataset, batch_size=2000, shuffle=True)
print("all data loaded! ")


class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # 第一层卷积层
        self.conv1 = nn.Sequential(
            # 输入的有两张图片，所以输入通道为2
            # mat1 and mat2 shapes cannot be multiplied (64x2048 and 1568x100)
            # 这是因为卷积核的大小和步长的原因，导致卷积后的图片大小不是28*28，而是26*26，所以要改变padding的大小,应该改为1，因为只吞掉了中间的左边一格，如果kernel为5，padding应该为2

            nn.Conv2d(in_channels=2, out_channels=16, kernel_size=3, stride=1, padding=1),
            # bn层
            nn.BatchNorm2d(16),
            nn.ReLU(),
            # 池化层使用均值池化，池化核的大小为2*2，步长为2，这样池化后的图片大小变为原来的一半
            nn.AvgPool2d(kernel_size=2)
        )
        # 第二层卷积层
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.AvgPool2d(kernel_size=2)
        )
        # 第三层卷积层
        self.conv3 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            # 使用均值池化
            # nn.AvgPool2d(kernel_size=7)
        )
        # 全连接层
        # 这里的全连接层的输入是32*7*7，因为经过两次池化，图片的大小变为原来的1/4，所以是28/4=7
        self.fc1 = nn.Sequential(
```

```
            nn.Linear(64 * 7 * 7, 100),
            # 用sigmoid函数来将输出限制在0-1之间
            nn.Sigmoid()
        )
        # 最后的输出层,sigmoid函数将输出限制在0-1之间
        self.fc2 = nn.Linear(100, 1)

    def forward(self, x):
        # 两张图片合并在一起
        # cat函数是将两个张量（tensor）拼接在一起，dim=1表示在第一个维度上拼接，即在通道上拼
接
        x = torch.cat((x[0], x[1]), dim=1)
        # 卷积层
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        # 将卷积层的输出展平
        x = x.view(x.size(0), -1)
        # 全连接层
        x = self.fc1(x)
        # 输出层
        x = self.fc2(x)
        # sigmoid函数将输出限制在0-1之间
        # x = torch.softmax(x, dim=1)
        return x


# 初始化模型和优化器、损失函数
model = CNN().to(device)
print("model loaded! ")
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
print("optimizer loaded! ")
# 二分类问题，所以用二分类的损失函数,这里用了sigmoid函数，所以用BCEWithLogitsLoss
criterion = nn.BCEWithLogitsLoss()
print("criterion loaded! ")
# 训练模型
model.train()
for epoch in range(epochs):
    for i, (img1, img2, label) in enumerate(train_loader):
        img1 = img1.to(device)
        img2 = img2.to(device)

        # 前向传播

        output = model((img1, img2))
        # 将label的类型转换为Float，因为后面计算损失的时候需要
        label = label.type(torch.FloatTensor)
        # 将label的shape转换为和output一样的shape
        label = label.view(output.shape)
        label = label.to(device)
        # 计算损失
        loss = criterion(output, label)
        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
        # 打印损失
        if i % 100 == 0:
            print('epoch: {}, step: {}, loss: {}'.format(epoch, i, loss.item()))
# 测试模型
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for i, (img1, img2, label) in enumerate(test_loader):
        img1 = img1.to(device)
        img2 = img2.to(device)

        # 前向传播
        output = model((img1, img2))
        # 计算准确率
        pred = torch.round(torch.sigmoid(output))
        # 将pred的shape转换为和label一样的shape
        pred = pred.view(label.shape)
        # 转为int类型
        pred = pred.type(torch.IntTensor)
        # 记得别丢gpu上了

        # debug 输出预测值和真实值
        # print("label:")
        # print(label)
        # print("pred:")
        # print(pred)
        correct += (pred == label).sum().item()
        total += label.size(0)
print('accuracy: {}'.format(correct / total))
```