计算机视觉实验2

曾世鹏 _ U202115574 _CS2108

一、实验需求分析

1. 读取MNIST数据集。
2. 构建train_loader和test_loader。
3. 构建带残差模块的卷积神经网络，实现10分类。

二、实验环境介绍

1. 使用pytorch框架

三、参数介绍

- 超参数设置

```
# 超参数
epochs = 10
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.01
# 这里的log_interval是指每隔多少个batch输出一次训练状态
log_interval = 10
random_seed = 1
# 设置种子，为了使得结果可复现
torch.manual_seed(random_seed)
```

四、实现思路

1. 创建train_loader和test_loader,开启download=True,从网上获取，归一化的均值和方差设置参考网上的针对MNIST数据集的值。
2. 构建残差模块
   - 网络设计：一个卷积模块为一个卷积层搭配一个bn层，激活函数使用relu。总共有两个卷积模块，第二个卷积模块如果输入输出维度不一样，则需要进行下采样再用relu。
   - 前向过程：保存identity，只用x预测残差，最后返回identity加由x预测出来的残差得出真实pred
3. 构建包含残差模块的卷积神经网络
   - 网络设计：先通过一个卷积层和bn，再通过四个残差模块通道数到128，最后用一个自适应均值池化把每个通道维度变为1*1，再用全连接将128通道转为10
4. 实例化：损失函数使用交叉熵，优化器选用Adam
5. 训练模型并测试

五、具体实现

5.1 构建数据集

1. 创建train_loader和test_loader

```
# 从torchvision.datasets中加载MNIST数据集，并对数据进行标准化处理,参考网上
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=True, download=True,
                               transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   # 这里设置均值和方差的值
                                   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.ToTensor(),
                                   torchvision.transforms.Normalize(
                                       (0.1307,), (0.3081,))
                               ])),
    batch_size=batch_size_test, shuffle=True)
```

5.2 构建残差模块

1. 卷积模块，使用卷积层，bn归一化和relu激活，用了bn后就可以不加偏置了好像会好点

```
# padding为1，保证输入输出的维度不变,bias=False,因为后面有BN层
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        # 这里的inplace=True是指将ReLU的输出直接覆盖到输入中，可以节省的显存，但是会影
响收敛性
        self.relu = nn.ReLU(inplace=True)
```

2. 下采样模块，如果步长不为1或需要输入输出通道数不同，则需要进行下采样，用卷积核为1，但输入输出通道符合输入需求的卷积层进行下采样，然后归一化

```
# 这里的downsample是指如果输入输出的通道不一致，就需要对输入进行下采样，使得维度一致
        # 原理是使用1*1的卷积核对输入进行卷积，同时步长为stride，这样就可以保证输入输出的
维度一致
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
```

3. 前向过程，先用identity保存x，用x预测出残差后加回identity作为返回的预测值

```python
    def forward(self,x):
        # 保存输入数据，采用恒等映射
        identity = x

        # 第一个卷积层
        out =self.conv1(x)
        out =self.bn1(out)
        out =self.relu(out)

        # 第二个卷积层
        out = self.conv2(out)
        out = self.bn2(out)

        # 下采样匹配卷积操作的输入输出维度
        identity = self.downsample(identity)

        # 还原结果
        out += identity
        out = self.relu(out)

        return out
```

5.3 构建卷积神经网络

因为尝试了不同层数的残差模块，最后使用自适应均值池化层将每个通道维数变为1*1,便于修改

最终效果比较好的为四层残差模块的网络

```python
class ResNet_CNN(nn.Module):
    def __init__(self,num_classes=10):
        super(ResNet_CNN,self).__init__()
        # mnist是灰度图，所以输入通道为1，输出通道为16,卷积核为3，步长为1，padding为1说明让
输入输出维度不变

        self.conv1 =nn.Conv2d(1,16,kernel_size=3,stride=1,padding=1,bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU()
        self.res1 = ResidualBlock(16,16)
        # 这里的stride=2,是因为输入输出维度不一致，需要下采样
        self.res2 = ResidualBlock(16,32,stride=2)
        self.res3 = ResidualBlock(32,64,stride=2)
        self.res4 = ResidualBlock(64,128,stride=2)
        # 用一个自适应均值池化层将每个通道维度变成1*1
        self.avg_pool = nn.AdaptiveAvgPool2d((1,1))
        # 感觉数字特征比较简单，一个全连接层就够了
        self.fc = nn.Linear(128,num_classes)
    def forward(self,x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.res1(x)
        x = self.res2(x)
        x = self.res3(x)
        x = self.res4(x)
        # 64个通道，每个通道1*1，输出64*1*1
```

```
        x = self.avg_pool(x)
        # 将数据拉成一维
        x = x.view(x.size(0),-1)
        x = self.fc(x)
        return x
```

5.4 实例化

构建模型，并放入GPU，构建损失函数和优化器

```
# 实例化
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = ResNet_CNN().to(device)
# 定义损失函数
loss_f = nn.CrossEntropyLoss()
# 定义优化器
optimizer = optim.Adam(model.parameters(),lr=learning_rate)
```

5.5 定义训练函数

注意用了bn后训练和测试的时候开model.train和model.eval,train的时候bn层会每个mini-batch都更新，eval的时候会累计数据进行归一化

```
# 训练模型
def train(epochs):
    for epoch in range(epochs):
        # 让BN层每一个mini-batch都要更新
        model.train()
        # 总损失
        # train_loss = 0
        # enumerate()函数用于将一个可遍历的数据对象组合为一个索引序列，同时列出数据和数据下标
        for batch_idx,(data,target) in enumerate(train_loader):
            data,target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = loss_f(output,target)
            loss.backward()
            optimizer.step()
            # train_loss += loss.item()
            # 每个mini-batch打印一次,loss.item()是一个mini-batch的平均损失
            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()
                ))
```

5.6 定义测试函数

单独统计每个类别正确的个数，用列表存储，用torch.max返回output向量最大值的索引即为预测值。

```
def test():
    # 让BN层累计数据进行归一化
    model.eval()
```

```python
    # 总正确数
    correct_all = 0
    # 单个类别的正确数,这里用列表存储
    correct_class = list(0. for i in range(10))
    # 单个类别的总数
    total_class = list(0. for i in range(10))
    # 总测试数
    total_all = 0
    with torch.no_grad():
        for images,labels in test_loader:
            images,labels = images.to(device),labels.to(device)
            output = model(images)
            # torch.max()返回最大值和最大值的索引,这里要不要data?
            _,predicted = torch.max(output,dim=1)
            # 增加总测试数和总正确数
            total_all += labels.size(0)
            correct_all += (predicted == labels).sum().item()
            # 增加单个类别的总数和正确数
            for i in range(batch_size_test):
                label = labels[i]
                correct_class[label] += (predicted[i] == label).item()
                total_class[label] += 1
    # 打印每个类别的准确率
    for i in range(10):
        print('Accuracy of number{}:{:.2f}%'.format(
            i,100*correct_class[i]/total_class[i]
        ))
    # 打印总体准确率
    print('Accuracy of all:{:.2f}%'.format(100*correct_all/total_all))
```

5.7 运行训练与测试

```python
# main
if __name__ == '__main__':
    epochs=10
    train(epochs)
    test()
```

六、实验记录

1.第一个卷积和残差模块都不用bn，开bias=True,epoch=10

Accuracy of number0:93.78%
Accuracy of number1:98.85%
Accuracy of number2:97.77%
Accuracy of number3:99.31%
Accuracy of number4:97.66%
Accuracy of number5:98.54%
Accuracy of number6:98.64%
Accuracy of number7:96.98%
Accuracy of number8:96.00%
Accuracy of number9:95.34%
Accuracy of all:**97.30%**

2.第一个卷积不用bn，开bias=True，epoch=10

Accuracy of all:**98.7%**

3.全用bn，禁用bias，三个残差块通道到64，epoch=10

```
# 构建包含ResidualBlock的网络，CNN
class ResNet_CNN(nn.Module):
    def __init__(self,num_classes=10):
        super(ResNet_CNN,self).__init__()
        # mnist是灰度图，所以输入通道为1，输出通道为16,卷积核为3，步长为1，padding为1说明让输入输出维度不变

        self.conv1 =nn.Conv2d(1,16,kernel_size=3,stride=1,padding=1,bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU()
        self.res1 = ResidualBlock(16,16)
        # 这里的stride=2,是因为输入输出维度不一致，需要下采样
        self.res2 = ResidualBlock(16,32,stride=2)
        self.res3 = ResidualBlock(32,64,stride=2)
        # 用一个自适应均值池化层将每个通道维度变成1*1
        self.avg_pool = nn.AdaptiveAvgPool2d((1,1))
        # 感觉数字特征比较简单，一个全连接层就够了
        self.fc = nn.Linear(64,num_classes)
    def forward(self,x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.res1(x)
        x = self.res2(x)
        x = self.res3(x)
        # 64个通道，每个通道1*1，输出64*1*1
        x = self.avg_pool(x)
        # 将数据拉成一维
        x = x.view(x.size(0),-1)
        x = self.fc(x)
        return x
```

结果：
Accuracy of number0:99.69%
Accuracy of number1:99.91%
Accuracy of number2:96.12%
Accuracy of number3:99.11%
Accuracy of number4:98.88%
Accuracy of number5:99.22%
Accuracy of number6:98.75%
Accuracy of number7:99.12%
Accuracy of number8:99.79%
Accuracy of number9:99.11%
Accuracy of all:**98.97%**

4.全用bn，禁用bias，四个残差块通道到128,epoch=10

```
# 构建包含ResidualBlock的网络，CNN
class ResNet_CNN(nn.Module):
    def __init__(self,num_classes=10):
        super(ResNet_CNN,self).__init__()
        # mnist是灰度图，所以输入通道为1，输出通道为16,卷积核为3，步长为1，padding为1说明让输入输出维度不变

        self.conv1 =nn.Conv2d(1,16,kernel_size=3,stride=1,padding=1,bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU()
        self.res1 = ResidualBlock(16,16)
        # 这里的stride=2,是因为输入输出维度不一致，需要下采样
        self.res2 = ResidualBlock(16,32,stride=2)
        self.res3 = ResidualBlock(32,64,stride=2)
        self.res4 = ResidualBlock(64,128,stride=2)
        # 用一个自适应均值池化层将每个通道维度变成1*1
        self.avg_pool = nn.AdaptiveAvgPool2d((1,1))
        # 感觉数字特征比较简单，一个全连接层就够了
        self.fc = nn.Linear(128,num_classes)
    def forward(self,x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.res1(x)
        x = self.res2(x)
        x = self.res3(x)
        x = self.res4(x)
        # 64个通道，每个通道1*1，输出64*1*1
        x = self.avg_pool(x)
        # 将数据拉成一维
        x = x.view(x.size(0),-1)
        x = self.fc(x)
        return x
```

结果：
Accuracy of number0:99.80%
Accuracy of number1:99.21%
Accuracy of number2:99.42%
Accuracy of number3:99.41%
Accuracy of number4:98.57%
Accuracy of number5:99.33%
Accuracy of number6:99.27%
Accuracy of number7:98.25%
Accuracy of number8:99.49%
Accuracy of number9:99.01%
Accuracy of all:**99.17%**

七、注意点与总结

本次实验较为基础，没遇到特别大的问题，初始的acc也比较高，调整了网络深度和使用batchnorm后就达到了比较满意的精度。

1. 卷积完后使用batchnorm时，卷积层就不需要跟bias了，卷积核为3 * 3，步长为1时，最外圈没有卷积结果，padding为1，卷积核为5 * 5时，为2
2. 如果需要输入输出通道数不相同时，需要进行下采样，具体操作为直接用卷积核为1的conv卷一遍，但输入输出通道匹配外部需要，这样就能让通道数改变，每个通道维度不变
3. 如果使用残差模块，残差模块内预测出来的是残差，但网络中与残差模块相连的别的模块还是需要根据输入x预测出的y值的，所以返回时是identity+pred。
4. 可以使用自适应均值池化层 nn.AdaptiveAvgPool2d((1,1))最后把每个通道拉为（1*1），某些时候比用线性层全连接好。

八、附录

总体代码如下：

```
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader

# 设计一个卷积神经网络，并在其中使用ResNet模块，在MNIST数据集上实现10分类手写体数字识别。
# 算一下每个数字的准确率
# 超参数
epochs = 10
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.01
# 这里的log_interval是指每隔多少个batch输出一次训练状态
log_interval = 10
random_seed = 1
# 设置种子，为了使得结果可复现
torch.manual_seed(random_seed)

# 从torchvision.datasets中加载MNIST数据集，并对数据进行标准化处理，参考网上
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./data/', train=True, download=True,
                               transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   # 这里设置均值和方差的值
                                   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.ToTensor(),
                                   torchvision.transforms.Normalize(
                                       (0.1307,), (0.3081,))
                               ])),
    batch_size=batch_size_test, shuffle=True)


# 定义残差模块
```

```
class ResidualBlock(torch.nn.Module):
    # 这里stride是指卷积的步长,保持输入输出的维度不变
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        # padding为1, 保证输入输出的维度不变,bias=False,因为后面有BN层
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        # 这里的inplace=True是指将ReLU的输出直接覆盖到输入中, 可以节省的显存, 但是会影响收敛
性
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = nn.Sequential()
        # 这里的downsample是指如果输入输出的维度不一致, 就需要对输入进行下采样, 使得维度一致
        # 原理是使用1*1的卷积核对输入进行卷积, 同时步长为stride, 这样就可以保证输入输出的维度
一致
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
    def forward(self,x):
        # 保存输入数据, 采用恒等映射
        identity = x

        # 第一个卷积层
        out =self.conv1(x)
        out =self.bn1(out)
        out =self.relu(out)

        # 第二个卷积层
        out = self.conv2(out)
        out = self.bn2(out)

        # 下采样匹配卷积操作的输入输出维度
        identity = self.downsample(identity)

        # 还原结果
        out += identity
        out = self.relu(out)

        return out

# 构建包含ResidualBlock的网络, CNN
class ResNet_CNN(nn.Module):
    def __init__(self,num_classes=10):
        super(ResNet_CNN,self).__init__()
        # mnist是灰度图, 所以输入通道为1, 输出通道为16,卷积核为3, 步长为1, padding为1说明让
输入输出维度不变

        self.conv1 =nn.Conv2d(1,16,kernel_size=3,stride=1,padding=1,bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU()
```

```
        self.res1 = ResidualBlock(16,16)
        # 这里的stride=2,是因为输入输出维度不一致，需要下采样
        self.res2 = ResidualBlock(16,32,stride=2)
        self.res3 = ResidualBlock(32,64,stride=2)
        self.res4 = ResidualBlock(64,128,stride=2)
        # 用一个自适应均值池化层将每个通道维度变成1*1
        self.avg_pool = nn.AdaptiveAvgPool2d((1,1))
        # 感觉数字特征比较简单，一个全连接层就够了
        self.fc = nn.Linear(128,num_classes)
    def forward(self,x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.res1(x)
        x = self.res2(x)
        x = self.res3(x)
        x = self.res4(x)
        # 64个通道，每个通道1*1，输出64*1*1
        x = self.avg_pool(x)
        # 将数据拉成一维
        x = x.view(x.size(0),-1)
        x = self.fc(x)
        return x

# 实例化
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = ResNet_CNN().to(device)
# 定义损失函数
loss_f = nn.CrossEntropyLoss()
# 定义优化器
optimizer = optim.Adam(model.parameters(),lr=learning_rate)

# 训练模型
def train(epochs):
    for epoch in range(epochs):
        # 让BN层每一个mini-batch都要更新
        model.train()
        # 总损失
        # train_loss = 0
        # enumerate()函数用于将一个可遍历的数据对象组合为一个索引序列，同时列出数据和数据下标
        for batch_idx,(data,target) in enumerate(train_loader):
            data,target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = loss_f(output,target)
            loss.backward()
            optimizer.step()
            # train_loss += loss.item()
            # 每个mini-batch打印一次,loss.item()是一个mini-batch的平均损失
            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()
                ))

# 测试模型,每一个类别都要统计准确率，并统计总体准确率
```

```python
def test():
    # 让BN层累计数据进行归一化
    model.eval()
    # 总正确数
    correct_all = 0
    # 单个类别的正确数,这里用列表存储
    correct_class = list(0. for i in range(10))
    # 单个类别的总数
    total_class = list(0. for i in range(10))
    # 总测试数
    total_all = 0
    with torch.no_grad():
        for images,labels in test_loader:
            images,labels = images.to(device),labels.to(device)
            output = model(images)
            # torch.max()返回最大值和最大值的索引,这里要不要data?
            _,predicted = torch.max(output,dim=1)
            # 增加总测试数和总正确数
            total_all += labels.size(0)
            correct_all += (predicted == labels).sum().item()
            # 增加单个类别的总数和正确数
            for i in range(batch_size_test):
                label = labels[i]
                correct_class[label] += (predicted[i] == label).item()
                total_class[label] += 1
    # 打印每个类别的准确率
    for i in range(10):
        print('Accuracy of number{}:{:.2f}%'.format(
            i,100*correct_class[i]/total_class[i]
        ))
    # 打印总体准确率
    print('Accuracy of all:{:.2f}%'.format(100*correct_all/total_all))

# main
if __name__ == '__main__':
    epochs=10
    train(epochs)
    test()
```