《计算机视觉》实验报告

姓名：严昕宇 学号：20121802

实验 10

一. 任务 1

a) 核心代码：

实验10 卷积神经网络

1. 完成深度学习环境配置，框架自行选择，推荐PyTorch，Tensorflow
2. 学习搭建一个卷积神经网络，画出网络结构示意图
3. 用CNN实现MNIST手写数字识别

1. 导入必要的库

[1]:
```
import os
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import torch
from torch import nn
from torchvision import datasets, transforms,utils
```

2. 准备数据

MNIST数据集包含60000个训练集和10000测试数据集。一个样本的格式为[data,label]，第一个存放数据（图片），第二个存放标签。图片是28*28的像素矩阵，标签为0~9共10个数字。
其参数中的root为数据集存放的路径，transform指定数据集导入的时候需要进行的变换，train设置为true表明导入的是训练集合，否则会测试集合。 Compose是把多种数据处理的方法集合在一起。

[2]:
```
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=[0.5],std=[0.5])])
train_data = datasets.MNIST(root = "./data/",
                            transform=transform,
                            train = True,
                            download = True)

test_data = datasets.MNIST(root="./data/",
                           transform = transform,
                           train = False)
print(len(train_data))
print(len(test_data))
```

```
60000
10000
```

3. 加载数据

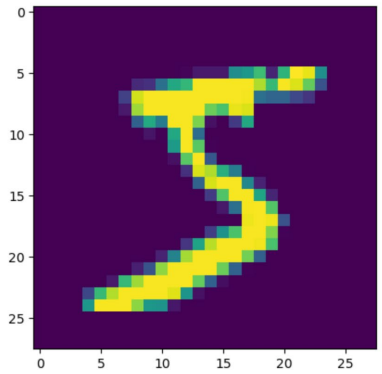
PyTorch提供的DataLoader类，负责向训练传递数据的任务。
其参数中的num_workers 表示用多少个子进程加载数据 shuffle 表示在装载过程中随机乱序。
设置batch_size=64后，加载器中的基本单为是一个batch的数据，所以train_loader 的长度是60000/64 = 938个batch，test_loader 的长度是10000/64= 157个batch。
加载到dataloader中后，一个dataloader是一个batch的数据

[3]:
```
train_loader = torch.utils.data.DataLoader(train_data,batch_size=64,
                                           shuffle=True,num_workers=2)
test_loader = torch.utils.data.DataLoader(test_data,batch_size=64,
                                           shuffle=True,num_workers=2)
print(len(train_loader))
print(len(test_loader))
```

```
938
157
```

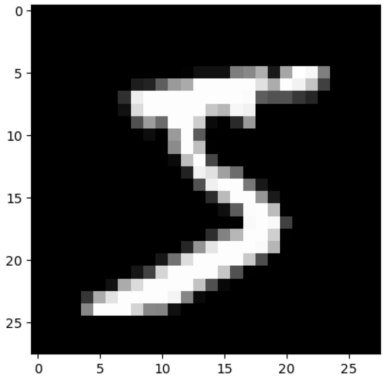
测试1：从二维数组生成一张图片

[4]:
```
oneimg,label = train_data[0]
oneimg = oneimg.numpy().transpose(1,2,0)
std = [0.5]
mean = [0.5]
oneimg = oneimg * std + mean
oneimg.resize(28,28)
plt.imshow(oneimg)
plt.show()
```



测试2：从三维生成一张黑白图片

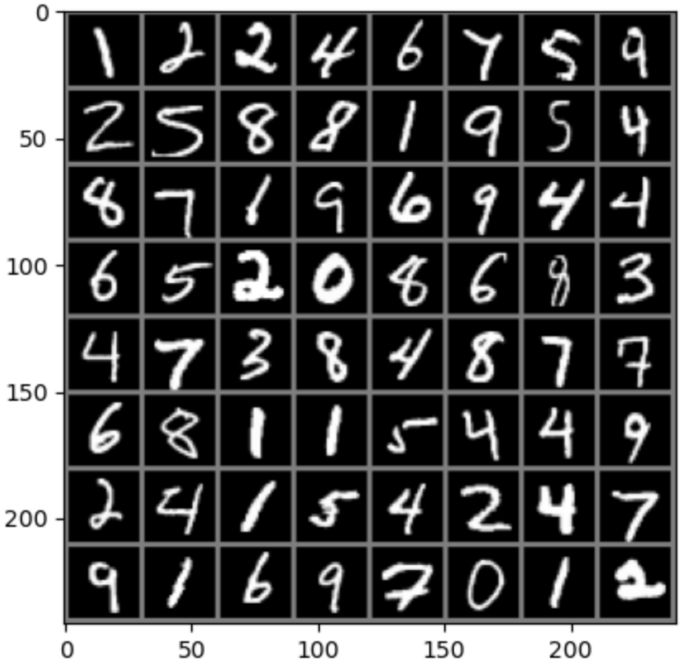
[5]:
```
oneimg,label = train_data[0]
grid = utils.make_grid(oneimg)
grid = grid.numpy().transpose(1,2,0)
std = [0.5]
mean = [0.5]
grid = grid * std + mean
plt.imshow(grid)
plt.show()
```



测试3：输出一个batch的图片和标签

[6]:
```
images, lables = next(iter(train_loader))
img = utils.make_grid(images)
# transpose 转置函数(x=0,y=1,z=2),新的x是原来的y轴大小，新的y是原来的z轴大小，新的z是原来的x大小
#相当于把x=1这个一道最后面去。
img = img.numpy().transpose(1,2,0)
std = [0.5]
mean = [0.5]
img = img * std + mean
for i in range(64):
    print(lables[i], end=" ")
    i += 1
    if i%8 == 0:
        print(end='\n')
plt.imshow(img)
plt.show()
```

```
tensor(1) tensor(2) tensor(2) tensor(4) tensor(6) tensor(7) tensor(5) tensor(9)
tensor(2) tensor(5) tensor(8) tensor(8) tensor(1) tensor(9) tensor(5) tensor(4)
tensor(8) tensor(7) tensor(1) tensor(9) tensor(6) tensor(9) tensor(4) tensor(4)
tensor(6) tensor(5) tensor(2) tensor(0) tensor(8) tensor(6) tensor(8) tensor(3)
tensor(4) tensor(7) tensor(3) tensor(8) tensor(4) tensor(8) tensor(7) tensor(7)
tensor(6) tensor(8) tensor(1) tensor(1) tensor(5) tensor(4) tensor(4) tensor(9)
tensor(2) tensor(4) tensor(1) tensor(5) tensor(4) tensor(2) tensor(4) tensor(7)
tensor(9) tensor(1) tensor(6) tensor(9) tensor(7) tensor(0) tensor(1) tensor(2)
```



4. 搭建一个卷积神经网络

网络结构是2个卷积层，3个全连接层。 Conv2d参数

- in_channels(int) – 输入信号的通道数目
- out_channels(int) – 卷积产生的通道数目
- kerner_size(int or tuple) - 卷积核的尺寸
- stride(int or tuple, optional) - 卷积步长
- padding(int or tuple, optional) - 输入的每一条边补充0的层数

[7]:
```
import torch.nn.functional as F
class CNN(nn.Module):
    def __init__(self):
        super(CNN,self).__init__()
        self.conv1 = nn.Conv2d(1,32,kernel_size=3,stride=1,padding=1)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(32,64,kernel_size=3,stride=1,padding=1)
        self.fc1 = nn.Linear(64*7*7,1024)#两个池化，所以是7*7而不是14*14
        self.fc2 = nn.Linear(1024,512)
        self.fc3 = nn.Linear(512,10)
#       self.dp = nn.Dropout(p=0.5)
    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 * 7* 7)#将数据平整为一维的
        x = F.relu(self.fc1(x))
#       x = self.fc3(x)
#       self.dp(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
#       x = F.log_softmax(x,dim=1) NLLLoss()才需要，交叉熵不需要
        return x

net = CNN()
print(net)
```

5. 定义损失函数和优化函数

[8]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
#也可以选择Adam优化方法
# optimizer = torch.optim.Adam(net.parameters(),lr=1e-2)
```

6. 模型训练

[9]:
```
train_accs = []
train_loss = []
test_accs = []
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net = net.to(device)
for epoch in range(3):
    running_loss = 0.0
    for i,data in enumerate(train_loader,0):#0是下标起始位置默认为0
        # data 的格式[[inputs, labels]]
        # inputs,labels = data
        inputs,labels = data[0].to(device), data[1].to(device)
```

```
        #初始为0，清除上个batch的梯度信息
        optimizer.zero_grad()

        #前向+后向+优化
        outputs = net(inputs)
        loss = criterion(outputs,labels)
        loss.backward()
        optimizer.step()

        # loss 的输出，每个一百个batch输出，平均的loss
        running_loss += loss.item()
        if i%100 == 99:
            print('[%d,%5d] loss :%.3f' %
                  (epoch+1,i+1,running_loss/100))
            running_loss = 0.0
        train_loss.append(loss.item())

        # 训练曲线的绘制 一个batch中的准确率
        correct = 0
        total = 0
        _, predicted = torch.max(outputs.data, 1)
        total = labels.size(0)# labels 的长度
        correct = (predicted == labels).sum().item() # 预测正确的数目
        train_accs.append(100*correct/total)

print('Finished Training')
```

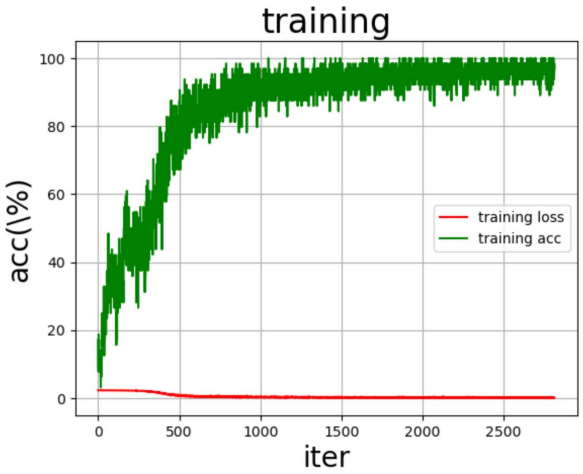
```
[1,  100] loss :2.288
[1,  200] loss :2.245
[1,  300] loss :2.136
[1,  400] loss :1.753
[1,  500] loss :1.054
[1,  600] loss :0.630
[1,  700] loss :0.480
[1,  800] loss :0.410
[1,  900] loss :0.361
[2,  100] loss :0.298
[2,  200] loss :0.280
[2,  300] loss :0.265
[2,  400] loss :0.248
[2,  500] loss :0.221
[2,  600] loss :0.216
[2,  700] loss :0.191
[2,  800] loss :0.180
[2,  900] loss :0.168
[3,  100] loss :0.148
[3,  200] loss :0.154
[3,  300] loss :0.148
[3,  400] loss :0.145
[3,  500] loss :0.137
[3,  600] loss :0.130
[3,  700] loss :0.120
[3,  800] loss :0.129
[3,  900] loss :0.111
Finished Training
```

7. 模型保存

[10]:
```
PATH='./model.pth'
torch.save(net.state_dict(), PATH)
```

8. 模型评估

[11]:
```
def draw_train_process(title,iters,costs,accs,label_cost,lable_acc):
    plt.title(title, fontsize=24)
    plt.xlabel("iter", fontsize=20)
    plt.ylabel("acc(\%)", fontsize=20)
    plt.plot(iters, costs,color='red',label=label_cost)
    plt.plot(iters, accs,color='green',label=lable_acc)
    plt.legend()
    plt.grid()
    plt.show()
train_iters = range(len(train_accs))
draw_train_process('training',train_iters,train_loss,train_accs,'training loss','training acc')
```



9. 测试集上面整体的准确率

[12]:
```
test_net = CNN()
test_net.load_state_dict(torch.load(PATH))
test_out = test_net(images)
```

[13]:
```
correct = 0
total = 0
with torch.no_grad():# 进行评测的时候网络不更新梯度
    for data in test_loader:
        images, labels = data
        outputs = test_net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)# labels 的长度
        correct += (predicted == labels).sum().item() # 预测正确的数目

print('Accuracy of the network on the  test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the  test images: 97 %
```

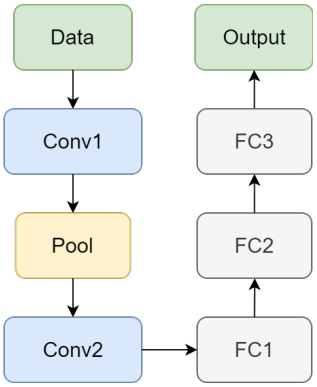
10. 10个类别的准确率

[14]:
```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = test_net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels)
        # print(predicted == labels)
        for i in range(10):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %d : %2d %%' % (
        i, 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of 0 : 99 %
Accuracy of 1 : 98 %
Accuracy of 2 : 94 %
Accuracy of 3 : 95 %
Accuracy of 4 : 99 %
Accuracy of 5 : 95 %
Accuracy of 6 : 97 %
Accuracy of 7 : 97 %
Accuracy of 8 : 96 %
Accuracy of 9 : 90 %
```

b) 实验结果




```
CNN(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=3136, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=10, bias=True)
)
```

c) 实验小结

MNIST 手写数字识别是一个常用来测试图像分类模型的基础数据集。MNIST 数据集包含了大约 7 万张手写数字图像，每张图像都是 28x28 像素的灰度图。每张图像都被标记为 0 到 9 之间的数字。

在 MNIST 手写数字识别实验中，通过训练一个 CNN 模型，让它学会识别 MNIST 数据集中的手写数字。这个模型需要输入一张图像，然后输出图像上的数字。在训练过程中，使用了反向传播算法来更新模型的参数，以使模型在预测数字时更加准确。

MNIST 手写数字识别是一个经典的机器学习问题，能够帮助我了解图像分类的基本流程，并且是一个很好的入门数据集。通过这个实验，我也更好地学习如何使用神经网络模型。