1.用全连接神经网络实现线性回归

```
import torch
import torch.nn as nn
from sklearn import datasets
import numpy as np
import matplotlib.pyplot as plt
import torch.optim as optim


X_numpy,y_numpy=datasets.make_regression(n_samples=100,n_features=1,
                                         noise=20,random_state=4)  #生成数据


X=torch.tensor(X_numpy,dtype=torch.float32) #将numpy中的ndarray类型数据转化为
tensor(张量)类型数据的方法
y=torch.tensor(y_numpy,dtype=torch.float32) #将numpy中的ndarray类型数据转化为
tensor(张量)类型数据的方法

y=y.view(100,-1)   #将真实值y的维度由一维扩展为二维

_,n_features=X.shape
input_size=n_features  #输入特征数
output_size=1      #输出特征数
learning_rate=0.01  #学习率
num_epochs=100   #训练轮数

Loss=nn.MSELoss()   #选定损失函数
optimizer=optim.SGD(model.parameters(),lr=learning_rate)  #选定优化器（本质上就是一种
梯度下降法）

#训练模型
for epoch in range(num_epochs):
    y_predicted=model(X)  #把输入扔到模型里得到预测值
    loss=Loss(y_predicted,y)#根据上一部得到的预测值，利用已经选定的损失函数求损失
    loss.backward()  #通过反向传播调用梯度下降法得到新的模型参数
    optimizer.step()  #把上一部得到的模型参数更新到模型里去
    optimizer.zero_grad()

    if (epoch+1)%10==0:
        print(f'训练轮数:{epoch+1},模型损失:{loss.item()}')

predicted=y_predicted.detach()  #将预测值中的梯度剥离
plt.scatter(X_numpy,y_numpy)
plt.plot(X,predicted,c='r')
```

2.用全连接神经网络拟合数学曲线

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
```

```
import torch.nn.functional as F

x_data=torch.linspace(-3,5,50)
X_data=torch.unsqueeze(x_data,1)  #将数据的维度由一维扩展为二维，以适应全连接网络输入数据的维度要求
y_data=3*X_data**3+2*X_data

class MyNet(nn.Module):   #该模型除去输入层外，该模型包括一个隐藏层，一个输出层（预测层）
    def __init__(self,n_feature,n_hidden,n_output):   #初始化函数
        super(MyNet,self).__init__()
        self.hidden_layer=nn.Linear(n_feature,n_hidden)
        self.predict_layer=nn.Linear(n_hidden,n_output)

    def forward(self,input):  #前向传播函数
        hidden_result=self.hidden_layer(input)
        relu_result=F.relu(hidden_result)
        predict_result=self.predict_layer(relu_result)
        return predict_result

input_feature=1   #输入特征
output_feature=1  #输出特征
neutron_num=32    #第一个隐藏层上的神经元个数，其它隐藏层上的神经元个数也可以设置为超参数，课堂授课时为了节省时间采用了直接赋值的方法
learning_rate=0.01  #学习率
num_epochs=1600   #训练轮数

net=MyNet(n_feature=input_feature,n_hidden=neutron_num,n_output=output_feature)
net    #模型实例化

loss_func=nn.MSELoss()  #本案例选择MSE作为损失函数
optimizer=optim.Adam(net.parameters(),lr=learning_rate)  #本案例选择Adam作为优化器
losses=[]  #定义一个空列表，用来存放训练过程中每一轮得到的损失值

for epoch in range(num_epochs):
    train_loss=0
    y_predict=net(X_data)
    loss=loss_func(y_predict,y_data)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    train_loss+=loss.item()
    losses.append(train_loss)

    if (epoch+1)%50==0:
        print(f'Epoch:{epoch+1},Loss:{loss.item():.4f}')

plt.plot(range(num_epochs),losses)   #绘制损失值与训练轮次的走势曲线

#可视化训练结果
plt.scatter(X_data,y_data)
plt.plot(X_data,y_predict.detach(),c='r')
```

3.用全连接神经网络识别手写字体

```
#1  导入必要的模块
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torchvision import transforms,datasets
import torch.optim as optim
from torch.utils.data import DataLoader
import torchvision
import torch.nn.functional as F
import time


#2 定义预处理操作
pipline=transforms.Compose([transforms.ToTensor(),transforms.Normalize([0.5],
[0.5])])


#3 下载数据集，并把刚才定义好的数据预处理操作施加给下载的数据集,并进行分批操作
train_dataset=datasets.MNIST('data',train=True,transform=pipline,download=True)
test_dataset=datasets.MNIST('data',train=False,transform=pipline,download=True)

train_dataloader=DataLoader(train_dataset,batch_size=16,shuffle=True)
test_dataloader=DataLoader(test_dataset,batch_size=16,shuffle=False)


#4 对数据集中的数据进行可视化，做到心中有数
examples=iter(train_dataloader)
images,labels=next(examples)


for i in range(9):   #第一种可视化方法
    plt.subplot(3,3,i+1)
    plt.tight_layout()
    plt.axis('off')
    plt.imshow(images[i].squeeze(),cmap='gray')
    plt.title(labels[i].item())


# 设计网络模型
class Net(nn.Module):
    def __init__(self,in_dim,n_hidden_1,out_dim):
        super(Net,self).__init__()

 self.layer01=nn.Sequential(nn.Linear(in_dim,n_hidden_1),nn.BatchNorm1d(n_hidden_
1))
        self.layer02=nn.Sequential(nn.Linear(n_hidden_1,out_dim))

    def forward(self,x):
        x=F.relu(self.layer01(x))
        x=torch.sigmoid(self.layer02(x))
        return x
```

```
net=Net(28*28,300,10)


device=torch.device('cuda：0' if torch.cuda.is_available() else 'cpu') #如果有GPU则把模型写到GPU里，否则写到cpu里
net.to(device)


#选择优化器和损失函数
loss_fun=nn.CrossEntropyLoss()  #交叉熵损失函数
optimizer=torch.optim.SGD(net.parameters(),lr=0.01)


losses=[]  #用来存放每一轮训练产生的损失值
acces=[]   #用来存放每一轮训练产生的准确率


#训练网络模型
start=time.time()
for epoch in range(20):
    train_loss=0   #用来存放当前轮次训练产生的损失值
    train_acc=0    #用来存放当前轮次训练产生的准确率
    net.train()    #该语句用来启动网络模型中的BN操作
    if epoch%5==0:
        optimizer.param_groups[0]['lr']*=0.9  #使用逐步衰减的学习率
    for img,label in train_dataloader:
        img=img.to(device)
        label=label.to(device)
        img=img.view(img.size(0),-1)
        out=net(img)
        loss=loss_fun(out,label)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss+=loss.item()
        _,pred=out.max(1)
        num_correct=(pred==label).sum().item()
        acc=num_correct/img.shape[0]  #用来存放当前批次训练产生的损失值
        train_acc+=acc  #当前轮次的损失值为所有批次的损失值之和
    print(f'epoch:{epoch+1},Train_loss:
{train_loss/len(train_dataloader):.4f},Train_acc:
{train_acc/len(train_dataloader):.4}')
    losses.append(train_loss/len(train_dataloader)) #当前轮次的损失值除以批次数得该轮次的平均损失，然后添加到列表中供以后可视化输出
    acces.append(train_acc/len(train_dataloader)) #当前轮次的损失值除以批次数得该轮次的平均损失，然后添加到列表中供以后可视化输出

end=time.time()


#可视化训练结果
print(f'训练时间:{end-start:.2f}s')
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
```

```
plt.ylabel('Train_Loss',fontsize=14,color='r')
plt.xlabel('Epoch',fontsize=14,color='b')
plt.plot(np.arange(len(losses)),losses)
plt.subplot(1,2,2)
plt.ylabel('Train_Acc',fontsize=14,color='r')
plt.xlabel('Epoch',fontsize=14,color='b')
plt.plot(np.arange(len(losses)),acces)


torch.save(net,'net_11-29_01.pth')  #保存网络模型，方便以后不需要再训练随时使用


net1=torch.load('net_11-29_01.pth')  #加载保存的训练模型


#测试网络模型【可利用加载来的额模型测试】
eval_losses=[]
eval_acces=[]
eval_loss=0
eval_acc=0
net1.eval()  #该语句用来关闭网络模型在训练时启动的BN操作
with torch.no_grad():
    for img,label in test_dataloader:
        img=img.to(device)
        label=label.to(device)
        img=img.view(img.size(0),-1)
        out=net1(img)
        loss=loss_fun(out,label)

        eval_loss+=loss.item()
        _,pred=out.max(1)
        num_correct=(pred==label).sum().item()
        acc=num_correct/img.shape[0]
        eval_acc+=acc
    eval_losses.append(eval_loss/len(test_dataloader))
    eval_acces.append(eval_acc/len(test_dataloader))
    print('Test Loss:{:.4f},Test Acc:
{:.4f}'.format(eval_loss/len(test_dataloader),eval_acc/len(test_dataloader)))
```

4.卷积提取特征示例

```
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt


img=Image.open(r'C:\Users\Lenovo\Desktop\lena2.jpg')  #注意换成你的图像路径或使用相对
路径

img=img.convert('L')  #转换为灰度图

to_tensor=transforms.ToTensor()
input=to_tensor(img)  #将导入的额图像转换为tensoor
```

```
to_pil=transforms.ToPILImage()
to_pil(input)  #显示转换为tensor的图像

kernel=torch.ones(3,3)/-9   #构建3*3的卷积核，并赋值
kernel[1,1]=1

conv.weight.data=kernel.view(1,1,3,3)   #将kernel升维为四维张量，并将其值赋给刚才构建的卷积网络的待训练参数

out=conv(input.unsqueeze(0))  #将input扩张为四维tensor后输入到卷积网络中农得到输出值

to_pil(out.squeeze(0))  #显示输出结果

pool=nn.MaxPool2d(2,2)  #构建一个2*2的池化层

list(pool.parameters())   #列出池化层的参数，以证明池化层不包含任何待训练参数

relu=nn.ReLU()  #构建一个relu激活函数层
list(relu.parameters())  #列出激活函数的参数，以证明改也不含包含任何待训练参数

to_pil(pool(out.squeeze(0)))  #显示池化后的图像，以证明其经过池化后图像大小确实减小啦
```