# 8.2 基于线性层的自编码模型

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import hiddenlayer as hl

from sklearn.manifold import TSNE

from sklearn.svm import SVC

from sklearn.decomposition import PCA

from sklearn.metrics import classification\_report, accuracy\_score

import torch

from torch import nn

import torch.nn.functional as F

import torch.utils.data as Data

import torch.optim as optim

from torchvision import transforms

from torchvision.datasets import MNIST

from torchvision.utils import make\_grid

import os

import gzip

import logging

# 使用手写体数据

# 准备训练数据集

train\_data = MNIST(

root=f"/deep Learning/MNIST", # 数据的路径

train=True, # 只使用训练数据集

transform=transforms.ToTensor(),

download=True

)

train\_data\_x = train\_data.data.type(torch.FloatTensor) / 255.0

train\_data\_x = train\_data\_x.reshape(train\_data\_x.shape[0], -1)

train\_data\_y = train\_data.targets

# 定义一个数据加载器

train\_loader = Data.DataLoader(

dataset=train\_data\_x, # 使用的数据集

batch\_size=64, # 批处理样本大小

shuffle=True, # 每次迭代前打乱数据

num\_workers=0

)

test\_data = MNIST(

root="/deep Learning", # 数据的路径

train=False, # 只使用训练数据集

transform=transforms.ToTensor(),

download=True

)

# 为测试数据添加一个通道纬度,获取测试数据的X和Y

test\_data\_x = test\_data.data.type(torch.FloatTensor) / 255.0

test\_data\_x = test\_data\_x.reshape(test\_data\_x.shape[0], -1)

test\_data\_y = test\_data.targets

print("训练数据集:", train\_data\_x.shape)

print("测试数据集:", test\_data\_x.shape)

# 可视化一个batch的图像内容

# 获得一个batch的数据

for step, b\_x in enumerate(train\_loader):

if step > 0:

break

# 可视化一个batch的图像

im = make\_grid(b\_x.reshape((-1, 1, 28, 28)))

im = im.data.numpy().transpose((1, 2, 0))

plt.figure()

plt.imshow(im)

plt.axis("off")

plt.show()

class EnDecoder(nn.Module):

def \_\_init\_\_(self):

super(EnDecoder, self).\_\_init\_\_()

# 定义Encoder

self.Encoder = nn.Sequential(

nn.Linear(784, 512),

nn.Tanh(),

nn.Linear(512, 256),

nn.Tanh(),

nn.Linear(256, 128),

nn.Tanh(),

nn.Linear(128, 3),

nn.Tanh(),

)

# 定义Decoder

self.Decoder = nn.Sequential(

nn.Linear(3, 128),

nn.Tanh(),

nn.Linear(128, 256),

nn.Tanh(),

nn.Linear(256, 512),

nn.Tanh(),

nn.Linear(512, 784),

nn.Sigmoid(),

)

# 定义网络的向前传播路径

def forward(self, x):

encoder = self.Encoder(x)

decoder = self.Decoder(encoder)

return encoder, decoder

# 输出我们的网络结构

edmodel = EnDecoder()

print(edmodel)

# 定义优化器

optimizer = torch.optim.Adam(edmodel.parameters(), lr=0.003)

loss\_func = nn.MSELoss() # 损失函数

# 记录训练过程的指标

history1 = hl.History()

# 使用Canvas进行可视化

canvas1 = hl.Canvas()

train\_num = 0

val\_num = 0

## 对模型进行迭代训练,对所有的数据训练EPOCH轮

for epoch in range(10):

train\_loss\_epoch = 0

## 对训练数据的迭代器进行迭代计算

for step, b\_x in enumerate(train\_loader):

## 使用每个batch进行训练模型

\_, output = edmodel(b\_x) # 在训练batch上的输出

loss = loss\_func(output, b\_x) # 平方根误差

optimizer.zero\_grad() # 每个迭代步的梯度初始化为0

loss.backward() # 损失的后向传播，计算梯度

optimizer.step() # 使用梯度进行优化

train\_loss\_epoch += loss.item() \* b\_x.size(0)

train\_num = train\_num + b\_x.size(0)

## 计算一个epoch的损失

train\_loss = train\_loss\_epoch / train\_num

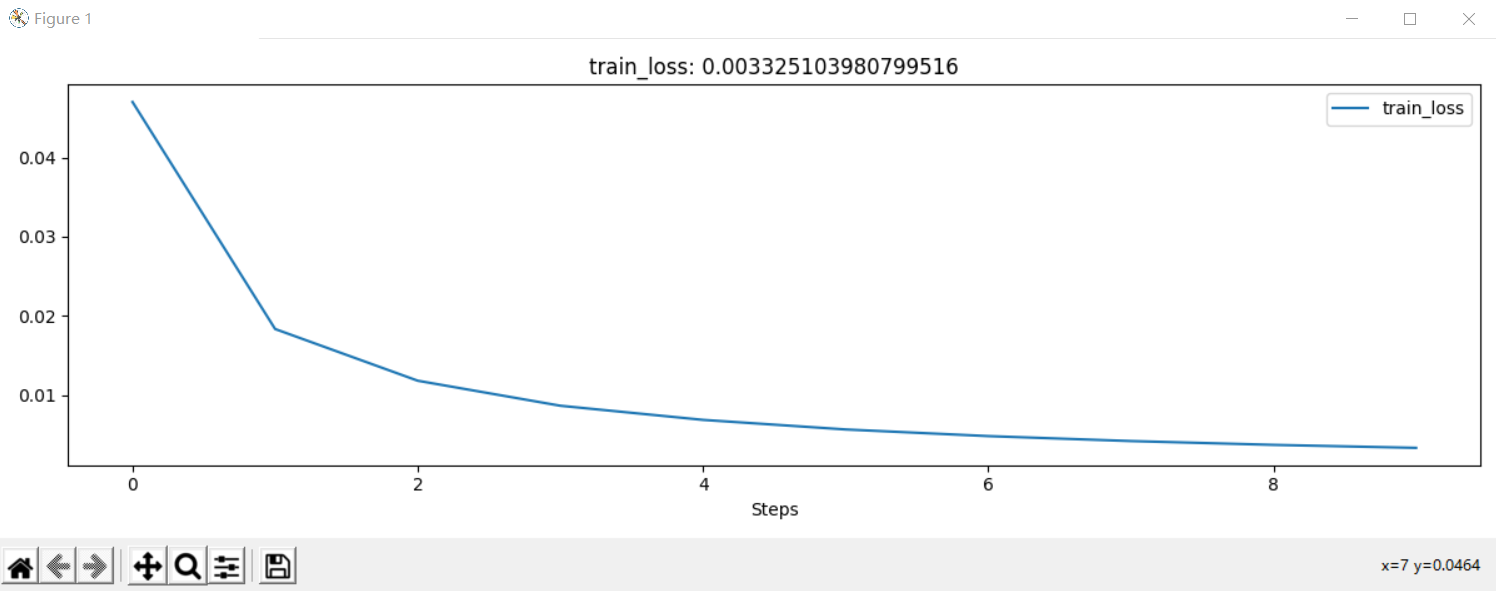
## 保存每个epoch上的输出loss

history1.log(epoch, train\_loss=train\_loss)

# 可视网络训练的过程

with canvas1:

canvas1.draw\_plot(history1["train\_loss"])



# 预测测试集前100张图像的输出

edmodel.eval()

\_, test\_decoder = edmodel(test\_data\_x[0:100, :])

# 可视化原始的图像

plt.figure(figsize=(6, 6))

for ii in range(test\_decoder.shape[0]):

plt.subplot(10, 10, ii + 1)

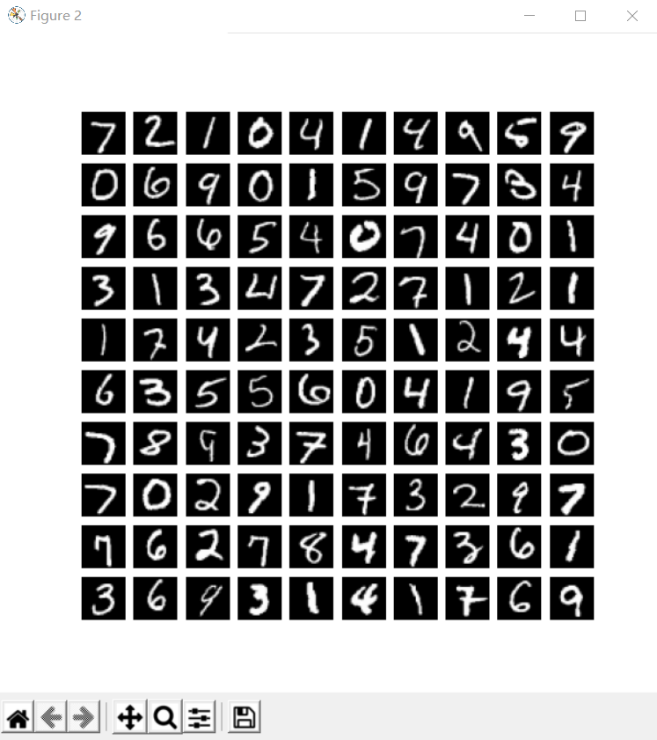
im = test\_data\_x[ii, :]

im = im.data.numpy().reshape(28, 28)

plt.imshow(im, cmap=plt.cm.gray)

plt.axis("off")

plt.show()



# 可视化编码后的图像

plt.figure(figsize=(6, 6))

for ii in range(test\_decoder.shape[0]):

plt.subplot(10, 10, ii + 1)

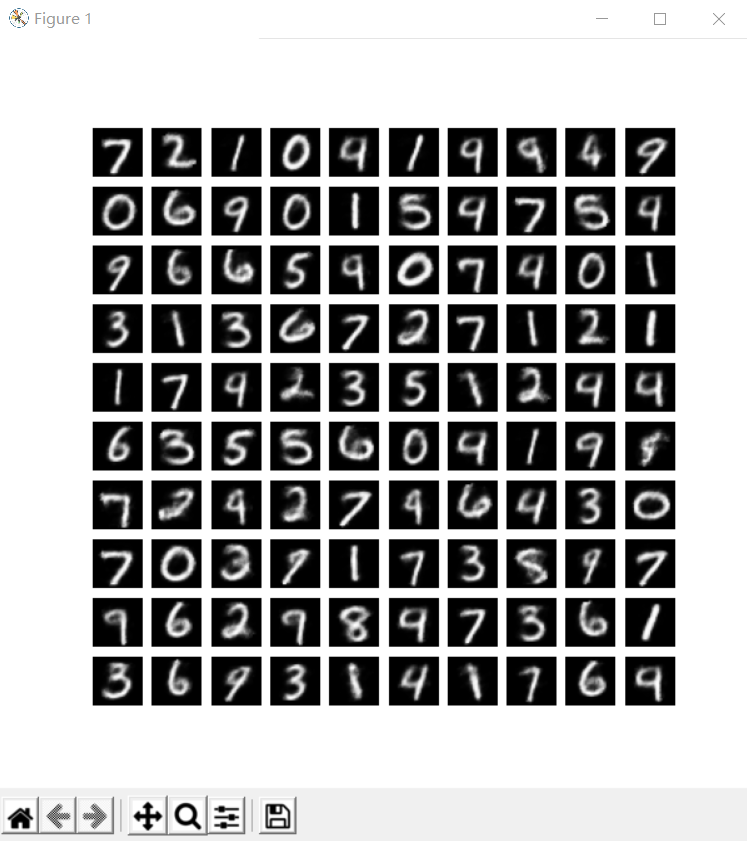
im = test\_decoder[ii, :]

im = im.data.numpy().reshape(28, 28)

plt.imshow(im, cmap=plt.cm.gray)

plt.axis("off")

plt.show()



# 获取前500个样本的自编码后的特征，并对数据进行可视化

edmodel.eval()

TEST\_num = 500

test\_encoder, \_ = edmodel(test\_data\_x[0:TEST\_num, :])

print("test\_encoder.shape:", test\_encoder.shape)

test\_encoder\_arr = test\_encoder.data.numpy()

# 将前2个纬度的特征进行可视化

X = test\_encoder\_arr[:, 0]

Y = test\_encoder\_arr[:, 1]

plt.figure(figsize=(8, 6))

# 可视化前设置坐标系的取值范围

plt.xlim([min(X) - 0.1, max(X) + 0.1])

plt.ylim([min(Y) - 0.1, max(Y) + 0.1])

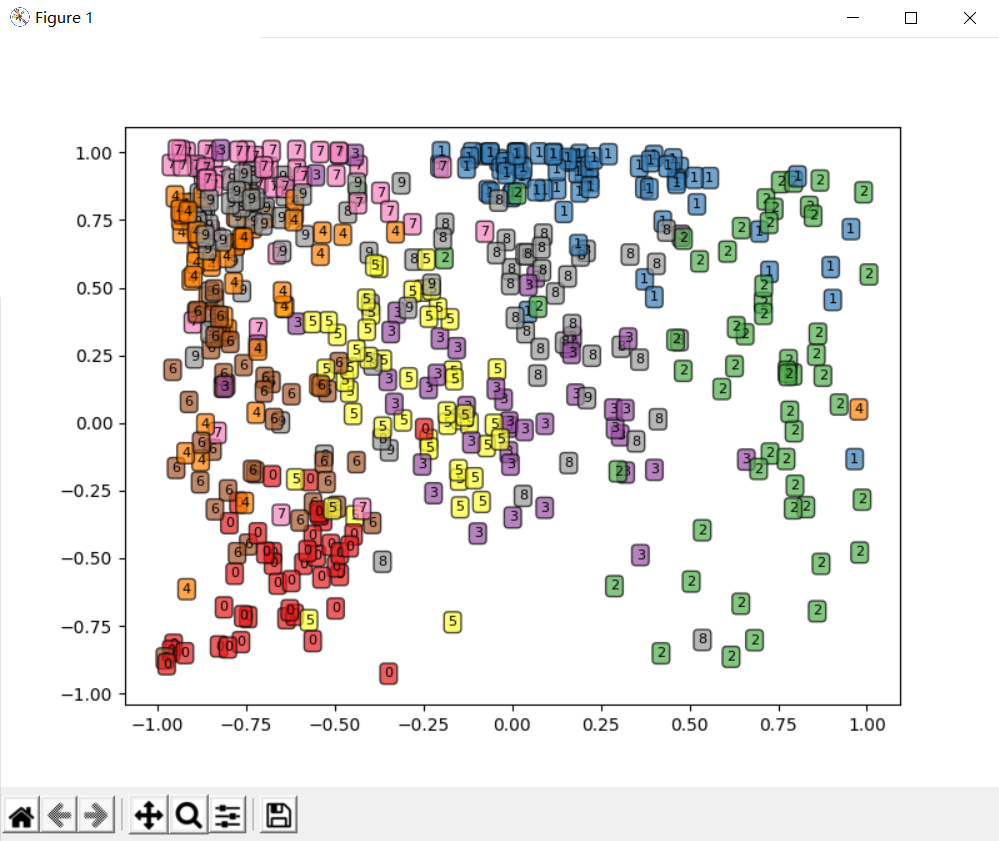
for ii in range(test\_encoder.shape[0]):

text = test\_data\_y.data.numpy()[ii]

plt.text(X[ii], Y[ii], str(text), fontsize=8,

bbox=dict(boxstyle="round", facecolor=plt.cm.Set1(text), alpha=0.7))

plt.show()



# 将3个纬度的特征进行可视化

test\_encoder\_arr = test\_encoder.data.numpy()

fig = plt.figure(figsize=(12, 8))

ax1 = Axes3D(fig)

X = test\_encoder\_arr[:, 0]

Y = test\_encoder\_arr[:, 1]

Z = test\_encoder\_arr[:, 2]

ax1.set\_xlim([min(X), max(X)])

ax1.set\_ylim([min(Y), max(Y)])

ax1.set\_zlim([min(Z), max(Z)])

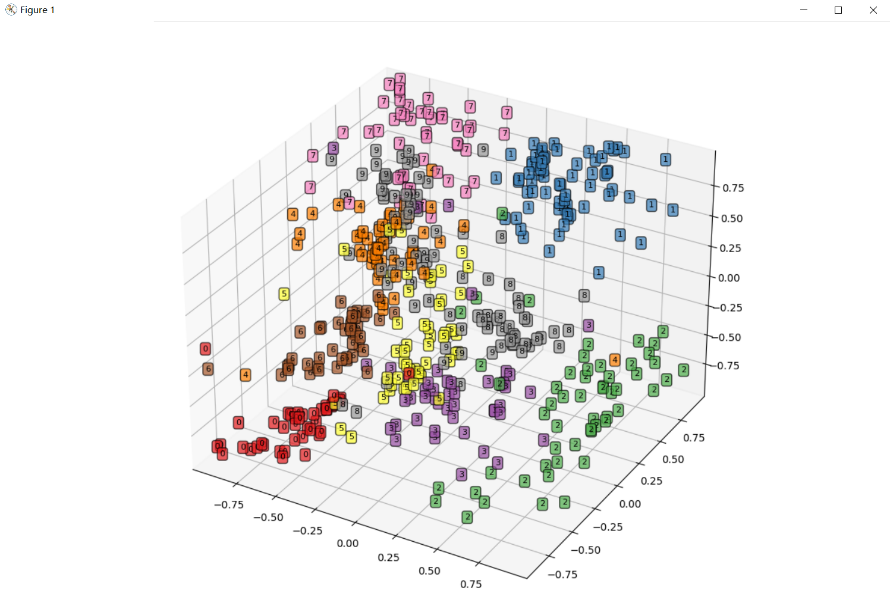
for ii in range(test\_encoder.shape[0]):

text = test\_data\_y.data.numpy()[ii]

ax1.text(X[ii], Y[ii,], Z[ii], str(text), fontsize=8,

bbox=dict(boxstyle="round", facecolor=plt.cm.Set1(text), alpha=0.7))

plt.show()



# 自编码后的特征训练集和测试集

train\_ed\_x, \_ = edmodel(train\_data\_x)

train\_ed\_x = train\_ed\_x.data.numpy()

train\_y = train\_data\_y.data.numpy()

test\_ed\_x, \_ = edmodel(test\_data\_x)

test\_ed\_x = test\_ed\_x.data.numpy()

test\_y = test\_data\_y.data.numpy()

print(train\_ed\_x.shape)

print(train\_y.shape)

# (60000, 3)

# (60000,)

# PCA降维获得的训练集和测试集前3个主成分

pcamodel = PCA(n\_components=3, random\_state=10)

train\_pca\_x = pcamodel.fit\_transform(train\_data\_x.data.numpy())

test\_pca\_x = pcamodel.transform(test\_data\_x.data.numpy())

print(train\_pca\_x.shape)

# (60000, 3)

# 使用自编码数据建立分类器,训练和预测

encodersvc = SVC(kernel="rbf", random\_state=123)

encodersvc.fit(train\_ed\_x, train\_y)

edsvc\_pre = encodersvc.predict(test\_ed\_x)

print(classification\_report(test\_y, edsvc\_pre))

print("模型精度", accuracy\_score(test\_y, edsvc\_pre))

# 模型精度 0.8752

# 使用PCA降维数据建立分类器,训练和预测

pcasvc = SVC(kernel="rbf", random\_state=123)

pcasvc.fit(train\_pca\_x, train\_y)

pcasvc\_pre = pcasvc.predict(test\_pca\_x)

print(classification\_report(test\_y, pcasvc\_pre))

print("模型精度", accuracy\_score(test\_y, pcasvc\_pre))

# 模型精度 0.5426