# 8.3 基于卷积自编码的图像去噪

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.model\_selection import train\_test\_split

from skimage.util import random\_noise

from skimage.measure import compare\_psnr

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 STL10

from torchvision.utils import make\_grid

from tqdm import tqdm

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") # device的定义

print("使用设备：", device)

# 定义一个将bin文件处理为图像数据的函数

def read\_image(data\_path):

with open(data\_path, 'rb') as f:

data1 = np.fromfile(f, dtype=np.uint8)

# 图像[数量，通道，宽，高]

images = np.reshape(data1, (-1, 3, 96, 96))

# 图像转化为RGB的形式，方便使用matplotlib进行可视化

images = np.transpose(images, (0, 3, 2, 1))

# 输出的图像取值在0～1之间

return images / 255.0

# 读取训练数据集,5000张96\*96\*3的图像

data\_path = "deep Learning/STL10/train\_X.bin"

images = read\_image(data\_path)

print("images.shape:", images.shape)

# images.shape: (5000, 96, 96, 3)

# 为数据添加高斯噪声

def gaussian\_noise(images, sigma):

"""sigma:噪声标准差"""

sigma2 = sigma \*\* 2 / (255 \*\* 2) # 噪声方差

images\_noisy = np.zeros\_like(images)

for ii in range(images.shape[0]):

image = images[ii]

# 使用skimage库中的random\_noise函数添加噪声

noise\_im = random\_noise(image, mode="gaussian", var=sigma2, clip=True)

images\_noisy[ii] = noise\_im

return images\_noisy

images\_noise = gaussian\_noise(images, 30)

print("images\_noise:", images\_noise.min(), "~", images\_noise.max())

# images\_noise: 0.0 ~ 1.0

# 可视化其中的部分图像

# 不带噪声的图像

plt.figure(figsize=(8, 4))

for ii in np.arange(32):

plt.subplot(4, 8, ii + 1)

plt.imshow(images[ii, ...])

plt.axis("off")

plt.show()

# 带噪声的图像

plt.figure(figsize=(8, 4))

for ii in np.arange(32):

plt.subplot(4, 8, ii + 1)

plt.imshow(images\_noise[ii, ...])

plt.axis("off")

plt.show()

# 数据准备为Pytorch可用的形式

# 转化为［样本，通道，高，宽］的数据形式

data\_Y = np.transpose(images, (0, 3, 2, 1))

data\_X = np.transpose(images\_noise, (0, 3, 2, 1))

# 将数据集切分为训练集和验证集

X\_train, X\_val, y\_train, y\_val = train\_test\_split(

data\_X, data\_Y, test\_size=0.2, random\_state=123)

# 将图像数据转化为向量数据

X\_train = torch.tensor(X\_train, dtype=torch.float32)

y\_train = torch.tensor(y\_train, dtype=torch.float32)

X\_val = torch.tensor(X\_val, dtype=torch.float32)

y\_val = torch.tensor(y\_val, dtype=torch.float32)

# 将X和Y转化为数据集合

train\_data = Data.TensorDataset(X\_train, y\_train)

val\_data = Data.TensorDataset(X\_val, y\_val)

print("X\_train.shape:", X\_train.shape)

print("y\_train.shape:", y\_train.shape)

print("X\_val.shape:", X\_val.shape)

print("y\_val.shape:", y\_val.shape)

# X\_train.shape: torch.Size([4000, 3, 96, 96])

# y\_train.shape: torch.Size([4000, 3, 96, 96])

# X\_val.shape: torch.Size([1000, 3, 96, 96])

# y\_val.shape: torch.Size([1000, 3, 96, 96])

# 定义一个数据加载器

train\_loader = Data.DataLoader(

dataset=train\_data, # 使用的数据集

batch\_size=32, # 批处理样本大小

shuffle=True, # 每次迭代前打乱数据

num\_workers=0, # 使用4个进程

)

# 定义一个数据加载器

val\_loader = Data.DataLoader(

dataset=val\_data, # 使用的数据集

batch\_size=32, # 批处理样本大小

shuffle=True, # 每次迭代前打乱数据

num\_workers=0, # 使用4个进程

)

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

if step > 0:

break

# 输出训练图像的尺寸和标签的尺寸

print(b\_x.shape)

print(b\_y.shape)

# torch.Size([32, 3, 96, 96])

# torch.Size([32, 3, 96, 96])

class DenoiseAutoEncoder(nn.Module):

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

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

# 定义Encoder

self.Encoder = nn.Sequential(

nn.Conv2d(in\_channels=3, out\_channels=64,

kernel\_size=3, stride=1, padding=1), # [,64,96,96]

nn.ReLU(),

nn.BatchNorm2d(64),

nn.Conv2d(64, 64, 3, 1, 1), # [,64,96,96]

nn.ReLU(),

nn.BatchNorm2d(64),

nn.Conv2d(64, 64, 3, 1, 1), # [,64,96,96]

nn.ReLU(),

nn.MaxPool2d(2, 2), # [,64,48,48]

nn.BatchNorm2d(64),

nn.Conv2d(64, 128, 3, 1, 1), # [,128,48,48]

nn.ReLU(),

nn.BatchNorm2d(128),

nn.Conv2d(128, 128, 3, 1, 1), # [,128,48,48]

nn.ReLU(),

nn.BatchNorm2d(128),

nn.Conv2d(128, 256, 3, 1, 1), # [,256,48,48]

nn.ReLU(),

nn.MaxPool2d(2, 2), # [,256,24,24]

nn.BatchNorm2d(256),

)

# 定义Decoder

self.Decoder = nn.Sequential(

nn.ConvTranspose2d(256, 128, 3, 1, 1), # [,128,24,24]

nn.ReLU(),

nn.BatchNorm2d(128),

nn.ConvTranspose2d(128, 128, 3, 2, 1, 1), # [,128,48,48]

nn.ReLU(),

nn.BatchNorm2d(128),

nn.ConvTranspose2d(128, 64, 3, 1, 1), # [,64,48,48]

nn.ReLU(),

nn.BatchNorm2d(64),

nn.ConvTranspose2d(64, 32, 3, 1, 1), # [,32,48,48]

nn.ReLU(),

nn.BatchNorm2d(32),

nn.ConvTranspose2d(32, 32, 3, 1, 1), # [,32,48,48]

nn.ConvTranspose2d(32, 16, 3, 2, 1, 1), # [,16,96,96]

nn.ReLU(),

nn.BatchNorm2d(16),

nn.ConvTranspose2d(16, 3, 3, 1, 1), # [,3,96,96]

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

# 输出网络结构

DAEmodel = DenoiseAutoEncoder().to(device)

print(DAEmodel)

# DenoiseAutoEncoder(

# (Encoder): Sequential(

# (0): Conv2d(3, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (1): ReLU()

# (2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (3): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (4): ReLU()

# (5): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (6): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (7): ReLU()

# (8): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

# (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (10): Conv2d(64, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (11): ReLU()

# (12): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (13): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (14): ReLU()

# (15): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (16): Conv2d(128, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (17): ReLU()

# (18): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

# (19): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# )

# (Decoder): Sequential(

# (0): ConvTranspose2d(256, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (1): ReLU()

# (2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (3): ConvTranspose2d(128, 128, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), output\_padding=(1, 1))

# (4): ReLU()

# (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (6): ConvTranspose2d(128, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (7): ReLU()

# (8): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (9): ConvTranspose2d(64, 32, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (10): ReLU()

# (11): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (12): ConvTranspose2d(32, 32, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (13): ConvTranspose2d(32, 16, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), output\_padding=(1, 1))

# (14): ReLU()

# (15): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

# (16): ConvTranspose2d(16, 3, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1))

# (17): Sigmoid()

# )

# )

# 定义优化器

LR = 0.0003

optimizer = torch.optim.Adam(DAEmodel.parameters(), lr=LR)

loss\_func = nn.MSELoss().to(device) # 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

val\_loss\_epoch = 0

# 对训练数据的迭代器进行迭代计算

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

b\_x, b\_y = b\_x.to(device), b\_y.to(device)

DAEmodel.train()

# 使用每个batch进行训练模型

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

loss = loss\_func(output, b\_y) # 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)

# 使用每个batch进行验证模型

for step, (b\_x, b\_y) in enumerate(tqdm(val\_loader)):

b\_x, b\_y = b\_x.to(device), b\_y.to(device)

DAEmodel.eval()

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

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

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

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

# 计算一个epoch的损失

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

val\_loss = val\_loss\_epoch / val\_num

# 保存每个epoch上的输出loss

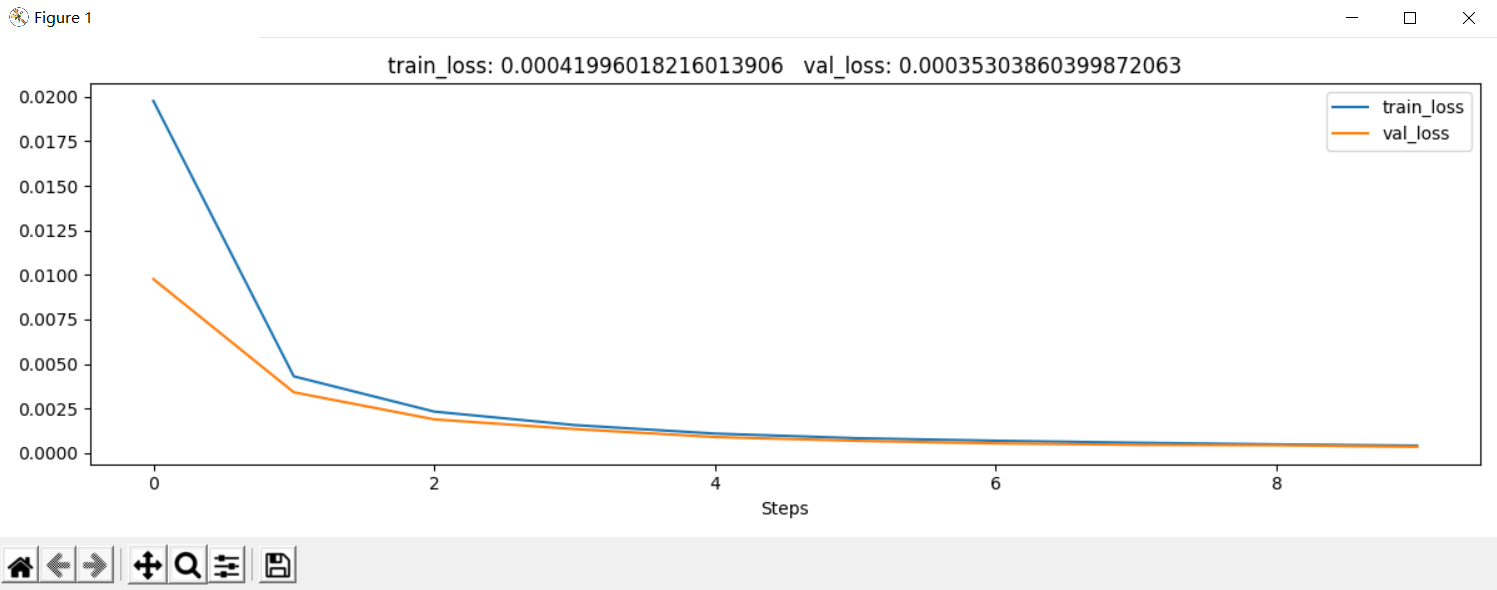
history1.log(epoch, train\_loss=train\_loss,

val\_loss=val\_loss)

# 可视网络训练的过程

with canvas1:

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



# 输入

imageindex = 1

im = X\_val[imageindex, ...]

im = im.unsqueeze(0)

im = im.cpu()

imnose = np.transpose(im.data.numpy(), (0, 3, 2, 1))

imnose = imnose[0, ...]

# 去噪

DAEmodel.eval()

DAEmodel.cpu()

with torch.no\_grad():

\_, output = DAEmodel(im)

im\_cpu = im.cpu()

output\_cpu = output.cpu()

# 将张量转换为 NumPy 数组

imnose = np.transpose(im\_cpu.data.numpy(), (0, 3, 2, 1))

imnose = imnose[0, ...]

imde = np.transpose(output\_cpu.data.numpy(), (0, 3, 2, 1))

imde = imde[0, ...]

# 获取原始图像

im = y\_val[imageindex, ...]

imor = im.unsqueeze(0)

imor = imor.to(device) # 确保原始图像也在 GPU 上

imor\_cpu = imor.cpu()

imor = np.transpose(imor\_cpu.data.numpy(), (0, 3, 2, 1))

imor = imor[0, ...]

# 计算去噪后的 PSNR

print("加噪后的 PSNR:", compare\_psnr(imor, imnose), "dB")

print("去噪后的 PSNR:", compare\_psnr(imor, imde), "dB")

# 加噪后的 PSNR: 19.467736211368948 dB

# 去噪后的 PSNR: 25.285308385089348 dB

# 将图像可视化

plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)

plt.imshow(imor)

plt.axis("off")

plt.title("Origin image")

plt.subplot(1, 3, 2)

plt.imshow(imnose)

plt.axis("off")

plt.title("Noise image $\sigma$=30")

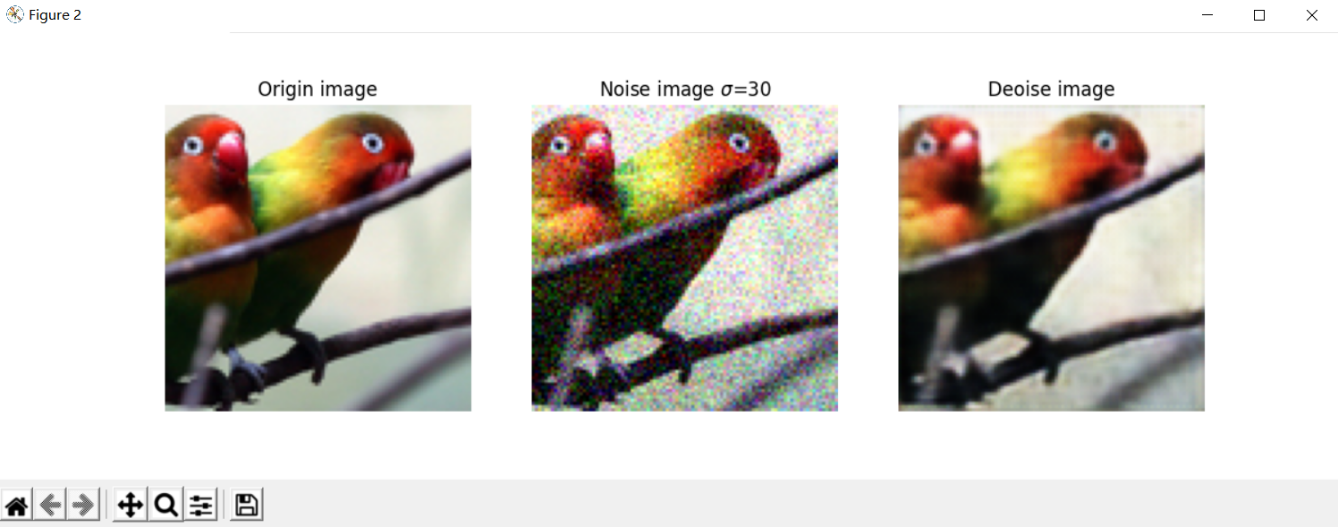
plt.subplot(1, 3, 3)

plt.imshow(imde)

plt.axis("off")

plt.title("Deoise image")

plt.show()



# 计算模型对整个验证集去噪后的PSNR提升量的均值

PSNR\_val = []

DAEmodel.eval()

for ii in range(X\_val.shape[0]):

imageindex = ii

# 输入

im = X\_val[imageindex, ...]

im = im.unsqueeze(0)

imnose = np.transpose(im.data.numpy(), (0, 3, 2, 1))

imnose = imnose[0, ...]

# 去噪

\_, output = DAEmodel(im)

imde = np.transpose(output.data.numpy(), (0, 3, 2, 1))

imde = imde[0, ...]

# 输出

im = y\_val[imageindex, ...]

imor = im.unsqueeze(0)

imor = np.transpose(imor.data.numpy(), (0, 3, 2, 1))

imor = imor[0, ...]

# 计算去噪后的PSNR

PSNR\_val.append(compare\_psnr(imor, imde) - compare\_psnr(imor, imnose))

print("PSNR的平均提升量为:", np.mean(PSNR\_val), "dB")

# PSNR的平均提升量为: 5.824640007434276 dB