```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import os
from PIL import Image
from tqdm import tqdm
import random
import time
import copy
import itertools
import shutil
from sklearn.model_selection import train_test_split

import torchvision
from torchvision import transforms, models
from torchvision.transforms import ToTensor, Normalize, ToPILImage
from torchvision.transforms.functional import hflip, vflip, rotate, adjust_hue

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
from torchvision.models.feature_extraction import create_feature_extractor
from torchsummary import summary

# pip install torchmetrics
import torchmetrics

cudnn.benchmark = True


# Key Parameters
download_original = 0 # download the slide image and masks from google bucket
generate_data = 1 # generate the downscale data
BATCH_SIZE = 8

# original width and height of image (standardized in all images)
width, height = 600,600
lr_width,·lr_height·=·150,·150·#·4x·reduction


# download the image the google bucket that I set up
# credit: http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
if download_original == 1:
    image_url = 'https://storage.googleapis.com/acv_project/landscape_img.zip'
    !curl -O $image_url
    !unzip landscape_img
    !rm landscape_img.zip
    # push all the images into a single place:
    os.mkdir('./original')
    for place in os.listdir('./landscape_img'):
        for img in os.listdir(os.path.join('./landscape_img', place)):
            shutil.copy(os.path.join(os.path.join('./landscape_img', place), img), './original')


if generate_data == 1:
    # train test split
    # split by id
    img_id = os.listdir('./original')
    train_id, test_id = train_test_split(img_id, test_size=0.1, random_state=1)

    # create test and train folder
    !rm -rf ./train
    os.mkdir('./train')
    os.mkdir('./train/original') # save all train data to this path

    !rm -rf ./test
    os.mkdir('./test')
    os.mkdir('./test/original') # save all train data to this path

    # save img into the respective folders as 1x downsampling
```

```
    for id in train_id:
        shutil.copy('./original/' + id, './train/original')

    for id in test_id:
        shutil.copy('./original/' + id, './test/original')

    # downscale images via Pillow
    # this will take quite some time to run
    downscale_factor_list = [1, 2, 4]
    for path in ['./train/', './test/']:
        img_id_list = os.listdir(path + 'original/')
        for img_id in tqdm(img_id_list):
            img = Image.open(path + 'original/' + img_id)
            img_arr_1x = np.array(img) # 1x downscale
            # downscale and upscale again (bicubic method)
            img_4x = img.resize((lr_width, lr_height))
            img_4x = img_4x.resize((width, height))
            img_arr_4x = np.array(img_4x) # 4x downscale
            np.save(os.path.join(path, img_id[:-4]), np.stack((img_arr_1x, img_arr_4x)))

    !rm -rf ./train/original
    !rm -rf ./test/original


# dataset and dataloader
class SatelliteDataset(torch.utils.data.Dataset):
    def __init__(self, img_dir):
        self.img_dir = img_dir
        self.img_files = os.listdir(img_dir)

    def __len__(self):
        return len(self.img_files)

    def __getitem__(self, idx):
        # load the image from disk
        img_arr = np.load(os.path.join(self.img_dir, self.img_files[idx]))
        img_1x = Image.fromarray(img_arr[0])
        img_4x = Image.fromarray(img_arr[1])
        img_4x = img_4x.resize((lr_width, lr_height))

        # apply flip, rotate, and convert to tensor
        # flipping

        c = np.random.randint(0,3)
        if c == 1:
            img_1x = hflip(img_1x)
            img_4x = hflip(img_4x)
        elif c == 2:
            img_1x = vflip(img_1x)
            img_4x = vflip(img_4x)
        elif c == 3:
            img_1x = vflip(img_1x)
            img_4x = vflip(img_4x)
            mask = vflip(mask)

        # rotation
        c = np.random.randint(0,3)
        img_1x  = rotate(img_1x, 90*c)
        img_4x  = rotate(img_4x, 90*c)

        # to tensor
        img_1x = ToTensor()(img_1x)
        img_4x = ToTensor()(img_4x)

        return img_4x, img_1x

def collate_fn(batch):
    # Filter failed images first
    batch = list(filter(lambda x: x is not None, batch))

    # Now collate into mini-batches
    lr = torch.stack([b[0] for b in batch])
    sr = torch.stack([b[1] for b in batch])
```

```
    return lr, sr

# implement custom image_dataset and wrap it with the dataloader
image_datasets = {x: SatelliteDataset(os.path.join('./', x)) for x in ['train', 'test']}

dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=BATCH_SIZE,
                                              shuffle=True, num_workers=0, collate_fn = collate_fn)
              for x in ['train', 'test']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'test']}
print('size of dataset', dataset_sizes)


sample = next(iter(dataloaders['train']))[0]
print(sample.shape)


# sample output
sample = next(iter(dataloaders['train']))
lr = sample[0][0]
sr = sample[1][0]
fig, axs = plt.subplots(1, 2, figsize=(15, 10))

# Display the LR and HR images using matplotlib
axs[0].imshow(ToPILImage()(sr))
axs[0].set_title('original image')
axs[1].imshow(ToPILImage()(lr))
axs[1].set_title('4x downscale')
plt.show()


# SRGAN Generator
# adapted from https://github.com/Lornatang/SRGAN-PyTorch/blob/main/model.py

def ResidualConvBlock(channels):
    return nn.Sequential(nn.Conv2d(in_channels = channels,
                                   out_channels = channels,
                                   kernel_size = (3, 3),
                                   stride = (1, 1),
                                   padding = (1, 1),
                                   bias=False),
                         nn.BatchNorm2d(channels),
                         nn.PReLU(), # great for mapping low-resolution images to high-resolution images
                         nn.Conv2d(channels, channels, (3, 3), (1, 1), (1, 1), bias=False),
                         nn.BatchNorm2d(channels),
                         nn.Dropout(p=0.05))

# only zoom in by 2x each time
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        in_channels = 3
        out_channels = 3
        channels = 64 # this is the intermediate channels in the network

        # low frequency information extraction layer
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, channels, (9, 9), (1, 1), (4, 4)),
            nn.PReLU())

        # 5 Residual Blocks (note that there will be an element wise sum)
        self.rcb1 = ResidualConvBlock(channels)
        self.rcb2 = ResidualConvBlock(channels)
        self.rcb3 = ResidualConvBlock(channels)
        self.rcb4 = ResidualConvBlock(channels)
        self.rcb5 = ResidualConvBlock(channels)

        # high-frequency information linear fusion layer
        self.conv2 = nn.Sequential(nn.Conv2d(in_channels = channels,
                                             out_channels = channels,
                                             kernel_size = (3, 3),
                                             stride = (1, 1),
                                             padding = (1, 1),
                                             bias=False),
```

```
                                  nn.BatchNorm2d(channels))

        # zoom block (we will only be zooming up by factor 2 each time)
        self.ub1 = nn.Sequential(nn.Conv2d(in_channels = channels,
                                           out_channels = channels * 4,
                                           kernel_size = (3, 3),
                                           stride = (1, 1),
                                           padding = (1, 1)),
                                 nn.PixelShuffle(2),
                                 nn.PReLU())
        self.ub2 = nn.Sequential(nn.Conv2d(in_channels = channels,
                                           out_channels = channels * 4,
                                           kernel_size = (3, 3),
                                           stride = (1, 1),
                                           padding = (1, 1)),
                                 nn.PixelShuffle(2),
                                 nn.PReLU())
        # reconstruction block
        self.conv3 = nn.Conv2d(in_channels = channels,
                               out_channels = out_channels,
                               kernel_size = (9, 9),
                               stride = (1, 1),
                               padding = (4, 4)) # retains the dimension of the output

    def forward(self, x):
        out1 = self.conv1(x)
        rcb1 = self.rcb1(out1)
        rcb2 = self.rcb2(torch.add(out1, rcb1)) # included the skip connections
        rcb3 = self.rcb3(torch.add(rcb1, rcb2))
        rcb4 = self.rcb4(torch.add(rcb2, rcb3))
        out2 = self.rcb5(torch.add(rcb3, rcb4))
        out2 = self.conv2(out2)
        out = torch.add(out1, out2)
        out = self.ub1(out)
        out = self.ub2(out)
        out = self.conv3(out)
        return out

# unit test
generator = Generator()
sample = next(iter(dataloaders['train']))
lr = sample[0]
sr = sample[1]
output = generator(lr)
print(output.shape)


def DiscriminatorConvBlock(in_channels, out_channels):
    return nn.Sequential(nn.Conv2d(in_channels = in_channels,
                                   out_channels = in_channels,
                                   kernel_size = (3, 3),
                                   stride = (2, 2),
                                   padding = (1, 1),
                                   bias=False),
                         nn.BatchNorm2d(in_channels),
                         nn.LeakyReLU(0.2, True),
                         nn.Conv2d(in_channels = in_channels,
                                   out_channels = out_channels,
                                   kernel_size = (3, 3),
                                   stride = (2, 2),
                                   padding = (1, 1),
                                   bias=False),
                         nn.BatchNorm2d(out_channels),
                         nn.LeakyReLU(0.2, True),
                         nn.Dropout(p=0.05))

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        # input shape (3) x 600 x 600
        self.conv1 = nn.Sequential(nn.Conv2d(3, 64, (3, 3), (1, 1), (1, 1), bias=True),
                                   nn.LeakyReLU(0.2, True))
```

```
        # input shape (64) x 150 x 150
        self.conv2 = DiscriminatorConvBlock(64, 128)
        # input shape (131) x 38, 38
        self.conv3 = DiscriminatorConvBlock(128, 256)
        self.conv4 = DiscriminatorConvBlock(256, 512)
        self.conv5 = DiscriminatorConvBlock(512, 1024)
        self.conv6 = DiscriminatorConvBlock(1024, 2048)
        # input shape (2048) * 1, 1
        self.classifier = nn.Sequential(nn.Linear(2048, 1024),
                                        nn.LeakyReLU(0.2, True),
                                        nn.Dropout(p=0.05),
                                        nn.Linear(1024, 1),
                                        nn.Sigmoid()) # for BCE loss

    def forward(self, lr, sr):
        out = self.conv1(sr)
        out = self.conv2(out)
        x = torch.cat((lr, out), 1) # need both the input and the output to distinguish
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.conv5(out)
        out = self.conv6(out)
        out = torch.flatten(out, 1)
        out = self.classifier(out)
        return out

# unit test
discriminator = Discriminator()
generator = Generator()
sample = next(iter(dataloaders['train']))
lr = sample[0]
sr = sample[1]
output = discriminator(lr, sr)
print(output.shape)


# reference: https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-tra
# content loss (loss based on the perceptual quality of the generated SR image as compared to the perceptual q
class ContentLoss(nn.Module):
    def __init__(self):
        super(ContentLoss, self).__init__()

        # load the VGG19 model trained on the ImageNet dataset
        # vgg: features (36 nodes) -> avg pool -> classifier

        model = models.vgg19(weights=models.VGG19_Weights.IMAGENET1K_V1)

        # standard basic (this is hardcoded to prevent modifications)
        self.feature_model_extractor_node = "features.35" # extract the thirty-sixth layer output in the VGG19
        self.feature_model_normalize_mean = [0.485, 0.456, 0.406]
        self.feature_model_normalize_std = [0.229, 0.224, 0.225]

        # normalize input
        self.normalize = transforms.Normalize(self.feature_model_normalize_mean, self.feature_model_normalize_

        # feature extractor
        self.feature_extractor = create_feature_extractor(model, [self.feature_model_extractor_node])

        # set to validation mode
        self.feature_extractor.eval()

        # Freeze model parameters.
        for model_parameters in self.feature_extractor.parameters():
            model_parameters.requires_grad = False

        self.mse_loss = nn.MSELoss()

    def forward(self, out_image, target_image):
        # put feature extractor to the same device
        if out_image.is_cuda:
            self.feature_extractor.cuda()
        # standardized operations
        out_image = self.normalize(out_image)
```

```
        target_image = self.normalize(target_image)
        out_feature = self.feature_extractor(out_image)[self.feature_model_extractor_node]
        target_feature = self.feature_extractor(target_image)[self.feature_model_extractor_node]
        # find the feature map mse between the two images
        loss = self.mse_loss(target_feature, out_feature)
        return loss

# reference: https://towardsdatascience.com/super-resolution-a-basic-study-e01af1449e13
# total variation loss (supress the noise in the generated image)
class TVLoss(nn.Module):
    def __init__(self):
        super(TVLoss, self).__init__()

    def forward(self, x):
        batch_size = BATCH_SIZE
        h_x = height
        w_x = width
        h_tv = torch.pow((x[:, :, 1:, :] - x[:, :, :h_x - 1, :]), 2).sum()
        w_tv = torch.pow((x[:, :, :, 1:] - x[:, :, :, :w_x - 1]), 2).sum()
        return (h_tv + w_tv)/(w_x * h_x * 3 * batch_size)

# generator loss
# reference: https://github.com/leftthomas/SRGAN
class GeneratorLoss(nn.Module):
    def __init__(self):
        super(GeneratorLoss, self).__init__()
        # Load the VGG19 model trained on the ImageNet dataset.
        self.max_pool = nn.MaxPool2d(4)
        self.content_loss = ContentLoss()
        self.tv_loss = TVLoss()
        self.pixel_loss = nn.MSELoss()

    def forward(self, out_labels, out_images, target_images):
        adversarial_loss = torch.mean(1 - out_labels)
        content_loss = self.content_loss(self.max_pool(out_images), self.max_pool(target_images))
        tv_loss = self.tv_loss(out_images)
        pixel_loss = self.pixel_loss(out_images, target_images)
        # print('adversarial_loss', adversarial_loss)
        # print('content loss:', content_loss)
        # print('total variation loss:', tv_loss)
        # print('pixel_loss:', pixel_loss)

        return 0.01 * adversarial_loss + content_loss + 0.1 * tv_loss + pixel_loss

# unit test
sample = next(iter(dataloaders['train']))
sr1 = sample[1]
sample = next(iter(dataloaders['train']))
sr2 = sample[1]
gl = GeneratorLoss()
print('generator loss:', gl(torch.ones(BATCH_SIZE), sr1, sr2))



# model training
# create a place to save memory
model_path = './model'
if not os.path.exists(model_path):
    os.mkdir(model_path) # save all models to this path


#train generator network first (warm start)

# parameters for training the generator network (round 1)
device = 'cuda:0'
model_g = Generator().to(device)
optimizer_g = optim.Adam(model_g.parameters(), lr=0.001)
scheduler_g = lr_scheduler.StepLR(optimizer_g, step_size = 8, gamma = 0.5)
criterion_g = GeneratorLoss()
num_epochs = 5 # we just want to warm start the generator here


# train the generator Model
train_loss_list = []
```

```
val_loss_list = []
best_loss = 100.0

for epoch in range(num_epochs):
    # training step
    model_g.train()
    torch.set_grad_enabled(True)
    train_running_loss = 0
    for lr, sr in tqdm(dataloaders['train']):
        lr = lr.to(device)
        sr = sr.to(device)
        optimizer_g.zero_grad()
        outputs = model_g(lr)
        loss = criterion_g(torch.ones(output.shape, device = device), outputs, sr) # adverserial loss is 0
        loss.backward()
        optimizer_g.step()
        train_running_loss += loss.item()
    scheduler_g.step()
    train_loss = train_running_loss * BATCH_SIZE/dataset_sizes['train']
    train_loss_list.append(train_loss)

    # validation step
    model_g.eval()
    torch.set_grad_enabled(False)
    # visualize how the mask prediction changes over time

    val_running_loss = 0
    for lr, sr in dataloaders['test']:
        lr = lr.to(device)
        sr = sr.to(device)
        outputs = model_g(lr)
        loss = criterion_g(torch.ones(output.shape, device = device), outputs, sr) # adverserial loss is 0
        val_running_loss += loss.item()
    val_loss = val_running_loss * BATCH_SIZE/dataset_sizes['test']
    val_loss_list.append(val_loss)

    # update the best model
    if val_loss < best_loss:
        best_loss = val_loss
        torch.save(model_g.state_dict(), './model/generator')

    print(f'epoch: {epoch + 1}/{num_epochs}, Train Loss: {train_loss:.8f}, Test Loss: {val_loss:.8f}')


# unit test
# sample output from pretrained generator
model_g.to('cpu')
img = next(iter(dataloaders['test']))
lr = img[0]
gen_sr = model_g(lr)[0]
lr = lr[0]
sr = img[1][0]

fig, axs = plt.subplots(1, 3, figsize=(15, 10))

# Display the LR and HR images using matplotlib
axs[0].imshow(ToPILImage()(sr))
axs[0].set_title('sr image 1x')
axs[1].imshow(ToPILImage()(gen_sr))
axs[1].set_title('gen image 1x')
axs[2].imshow(ToPILImage()(lr))
axs[2].set_title('lr image 4x')
plt.show()


# parameters for training the discriminator network
model_g.load_state_dict(torch.load('./model/generator'))
model_g.to(device)
model_d = Discriminator().to(device)
optimizer_d = optim.Adam(model_d.parameters(), lr=0.01)
criterion_d = nn.BCELoss()
num_epochs = 2 # warm start so does not need that many
```

```
# warm start the discriminator model (4x -> 1x)
# note that we are simplifying the loss function for the discriminator
discriminator_loss_list = []
discriminator_val_loss_list = []
best_loss = 100.0

for epoch in range(num_epochs):
    # train step
    running_loss_d = 0
    running_loss_g = 0
    model_d.train()
    model_g.train()
    torch.set_grad_enabled(True)
    for lr, sr in tqdm(dataloaders['train']):
        lr = lr.to(device)
        sr = sr.to(device)
        # log(D(x))
        optimizer_d.zero_grad()
        output = model_d(lr, sr)
        loss_d_real = criterion_d(output, torch.ones(output.shape, device = device))
        loss_d_real.backward()
        # log(1 - D(G(z)))
        fake_sr = model_g(lr)
        output = model_d(lr, fake_sr)
        loss_d_fake = criterion_d(output, torch.zeros(output.shape, device = device))
        loss_d_fake.backward(retain_graph=True)
        loss_d = loss_d_real + loss_d_fake
        optimizer_d.step()
        running_loss_d += loss_d.item()
    discriminator_loss = running_loss_d/dataset_sizes['train']
    discriminator_loss_list.append(discriminator_loss)

    # validation step
    model_d.eval()
    val_running_loss_d = 0
    torch.set_grad_enabled(False)
    for lr, sr in dataloaders['test']:
        lr = lr.to(device)
        sr = sr.to(device)
        # log(D(x))
        output = model_d(lr, sr)
        loss_d_real = criterion_d(output, torch.ones(output.shape, device = device))
        # log(1 - D(G(z)))
        fake_sr = model_g(lr)
        output = model_d(lr, fake_sr)
        loss_d_fake = criterion_d(output, torch.zeros(output.shape, device = device))
        loss_d = loss_d_real + loss_d_fake
        val_running_loss_d += loss_d.item()
    val_loss = val_running_loss_d/dataset_sizes['test']
    discriminator_val_loss_list.append(val_loss)
    # update the best model
    if val_loss < best_loss:
        # save the weights
        best_loss = val_loss
        torch.save(model_d.state_dict(), './model/discriminator')


# load the pretrained generator and discriminator
model_g.load_state_dict(torch.load('./model/generator'))
model_g.to(device)
model_d.load_state_dict(torch.load('./model/discriminator'))
model_d.to(device)

# set the number of epochs
num_epochs = 15

# train the SRGAN model (4x -> 1x)
discriminator_loss_list = []
generator_loss_list = []
generator_val_loss_list = []
best_loss = 100.0
```

```
for epoch in range(num_epochs):
    # train step
    running_loss_d = 0
    running_loss_g = 0
    model_d.train()
    model_g.train()
    torch.set_grad_enabled(True)
    for lr, sr in tqdm(dataloaders['train']):
        lr = lr.to(device)
        sr = sr.to(device)
        ############################
        # (1) update D network: maximize log(D(x))+log(1-D(G(z)))
        ###########################
        optimizer_d.zero_grad()
        output = model_d(lr, sr)
        loss_d_real = criterion_d(output, torch.ones(output.shape, device = device))
        loss_d_real.backward()
        # log(1 - D(G(z)))
        fake_sr = model_g(lr)
        output = model_d(lr, fake_sr)
        loss_d_fake = criterion_d(output, torch.zeros(output.shape, device = device))
        loss_d_fake.backward(retain_graph=True)
        loss_d = loss_d_real + loss_d_fake
        optimizer_d.step()
        running_loss_d += loss_d.item()
        ############################
        # (2) update G network: minimize 1-D(G(z)) + Content Loss + TV Loss
        ###########################
        optimizer_g.zero_grad()
        # note that fake labels are real for generator cost
        loss_g = criterion_g(out_labels = output, out_images = fake_sr, target_images = sr)
        loss_g.backward()
        optimizer_g.step()
        running_loss_g += loss_g.item()
    discriminator_loss = running_loss_d/dataset_sizes['train']
    discriminator_loss_list.append(discriminator_loss)
    generator_loss = running_loss_g/dataset_sizes['train']
    generator_loss_list.append(generator_loss)

    # validation step
    model_g.eval()
    val_running_loss = 0
    torch.set_grad_enabled(False)
    for lr, sr in dataloaders['test']:
        lr = lr.to(device)
        sr = sr.to(device)
        fake_sr = model_g(lr)
        output = model_d(lr, fake_sr)
        loss = criterion_g(out_labels = output, out_images = fake_sr, target_images = sr)
        val_running_loss += loss.item()
    val_loss = val_running_loss/dataset_sizes['test']
    generator_val_loss_list.append(val_loss)
    # update the best model
    if val_loss < best_loss:
        # save the weights
        best_loss = val_loss
        torch.save(model_g.state_dict(), './model/gan_generator')
        torch.save(model_d.state_dict(), './model/gan_discriminator')

    # print the progress to determine if the progress has stagnated
    print(f'epoch: {epoch+1}/{num_epochs}, Generator Loss: {generator_loss:.4f}, Discriminator Loss: {discrimi

# unit test
# sample output from pretrained generator
model_g.to('cpu')
img = next(iter(dataloaders['test']))
lr = img[0]
gen_sr = model_g(lr)[0]
lr = lr[0]
sr = img[1][0]

fig, axs = plt.subplots(1, 3, figsize=(15, 10))
```

```
# Display the LR and HR images using matplotlib
axs[0].imshow(ToPILImage()(sr))
axs[0].set_title('sr image 1x')
axs[1].imshow(ToPILImage()(gen_sr))
axs[1].set_title('gen sr image 1x')
axs[2].imshow(ToPILImage()(lr))
axs[2].set_title('lr image 4x')
plt.show()


# adapted from xingyue model

# !pip install torchmetrics
# !pip install lpips
from torchmetrics import PeakSignalNoiseRatio
from torchmetrics import StructuralSimilarityIndexMeasure
import lpips
psnr = PeakSignalNoiseRatio().to(device)
ssim = StructuralSimilarityIndexMeasure(data_range=1.0).to(device)
loss_fn = lpips.LPIPS(net='alex').to(device)

# Evaluate model performance on test dataset (modified from xinyue model)
l1_loss = []
mse_loss = []
psnr_list = []
ssim_list = []
output_list = []
lpips_list = []
criterion1 = nn.L1Loss()
criterion2 = nn.MSELoss()
model_g.to(device)
model_g.eval()
with torch.no_grad():
    for lr, sr in dataloaders['test']:
        lr = lr.to(device)
        sr = sr.to(device)
        fake_sr = model_g(lr)
        # Compute L1 loss
        loss = criterion1(fake_sr, sr)
        l1_loss.append(loss.item())

        # Compute MSE loss
        loss2 = criterion2(fake_sr, sr)
        mse_loss.append(loss2.item())

        # Compute PSNR
        psnr_value = psnr(fake_sr, sr)
        psnr_value = psnr_value.clone().cpu().detach().numpy()
        psnr_list.append(psnr_value)

        # Compute SSIM
        ssim_value = ssim(fake_sr, sr)
        ssim_value = ssim_value.clone().cpu().detach().numpy()
        ssim_list.append(ssim_value)

        # Compute LPIPS
        d = loss_fn.forward(fake_sr, sr).clone().cpu().detach().numpy()
        lpips_list.append(d)

print('l1_loss:', np.mean(l1_loss))
print('mse_loss', np.mean(mse_loss))
print('psnr:', np.mean(psnr_list))
print('lpips:', np.mean(lpips_list))
print('ssim:', np.mean(ssim_list))
```