ДОДАТОК А

Програмний код

```
from __future__ import print_function
import matplotlib.pyplot as plt
# %matplotlib inline

import os
import sys
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.optim
import torch.nn as nn
import torchvision
from collections import namedtuple
from PIL import Image
from skimage.measure import compare_psnr
torch.backends.cudnn.enabled = True
torch.backends.cudnn.benchmark =True
dtype = torch.cuda.FloatTensor
imsize =-1
sigma = 50
sigma_ = sigma/255.
fname = 'gary.jpg'

"""#Utils"""

def add_module(self, module):
    self.add_module(str(len(self) + 1), module)

torch.nn.Module.add = add_module

def get_noisy_image(img_np, sigma):
img_noisy_np = np.clip(img_np + np.random.normal(scale=sigma, size=img_np.shape), 0,
1).astype(np.float32)
    img_noisy_pil = np_to_pil(img_noisy_np)
    return img_noisy_pil, img_noisy_np

def crop_image(img, d=32):
    new_size = (img.size[0] - img.size[0] % d,
                img.size[1] - img.size[1] % d)
    bbox = [
            int((img.size[0] - new_size[0])/2),
            int((img.size[1] - new_size[1])/2),
            int((img.size[0] + new_size[0])/2),
            int((img.size[1] + new_size[1])/2),
    ]
    img_cropped = img.crop(bbox)
    return img_cropped

def get_params(opt_over, net, net_input, downsampler=None):
    opt_over_list = opt_over.split(',')
    params = []
    for opt in opt_over_list:
```

```
            if opt == 'net':
                params += [x for x in net.parameters() ]
            elif  opt=='down':
                assert downsampler is not None
                params = [x for x in downsampler.parameters()]
            elif opt == 'input':
                net_input.requires_grad = True
                params += [net_input]
        return params

def get_image_grid(images_np, nrow=8):
    images_torch = [torch.from_numpy(x) for x in images_np]
    torch_grid = torchvision.utils.make_grid(images_torch, nrow)

    return torch_grid.numpy()
def plot_image_grid(images_np, nrow =8, factor=1, interpolation='lanczos'):
    n_channels = max(x.shape[0] for x in images_np)
    assert (n_channels == 3) or (n_channels == 1), "images should have 1 or 3 channels"
    images_np = [x if (x.shape[0] == n_channels) else np.concatenate([x, x, x], axis=0) for x in
images_np]
    grid = get_image_grid(images_np, nrow)
    plt.figure(figsize=(len(images_np) + factor, 12 + factor))
    if images_np[0].shape[0] == 1:
        plt.imshow(grid[0], cmap='gray', interpolation=interpolation)
    else:
        plt.imshow(grid.transpose(1, 2, 0), interpolation=interpolation)
    plt.show()
    return grid

def get_image(path, imsize=-1):
    img = Image.open(path)
    if isinstance(imsize, int):
        imsize = (imsize, imsize)
    if imsize[0]!= -1 and img.size != imsize:
        if imsize[0] > img.size[0]:
            img = img.resize(imsize, Image.BICUBIC)
        else:
            img = img.resize(imsize, Image.ANTIALIAS)
    img_np = pil_to_np(img)
    return img, img_np

def fill_noise(x, noise_type):
    if noise_type == 'u':
        x.uniform_()
    elif noise_type == 'n':
        x.normal_()
    else:
        assert False

def get_noise(input_depth, method, spatial_size, noise_type='u', var=1./10):
    if isinstance(spatial_size, int):
        spatial_size = (spatial_size, spatial_size)
```

```
    if method == 'noise':
        shape = [1, input_depth, spatial_size[0], spatial_size[1]]
        net_input = torch.zeros(shape)
        fill_noise(net_input, noise_type)
        net_input *= var
    elif method == 'meshgrid':
        assert input_depth == 2
        X, Y = np.meshgrid(np.arange(0, spatial_size[1])/float(spatial_size[1]-1), np.arange(0,
spatial_size[0])/float(spatial_size[0]-1))
        meshgrid = np.concatenate([X[None,:], Y[None,:]])
        net_input=  np_to_torch(meshgrid)
    else:
        assert False

    return net_input

def pil_to_np(img_PIL):
    ar = np.array(img_PIL)
    if len(ar.shape) == 3:
        ar = ar.transpose(2,0,1)
    else:
        ar = ar[None, ...]
    return ar.astype(np.float32) / 255.

def np_to_pil(img_np):
    ar = np.clip(img_np*255,0,255).astype(np.uint8)
    if img_np.shape[0] == 1:
        ar = ar[0]
    else:
        ar = ar.transpose(1, 2, 0)
    return Image.fromarray(ar)

def np_to_torch(img_np):
    return torch.from_numpy(img_np)[None, :]

def torch_to_np(img_var):
    return img_var.detach().cpu().numpy()[0]

def optimize(optimizer_type, parameters, closure, LR, num_iter):
    if optimizer_type == 'L-BFGS':
        # Do several steps with adam first
        optimizer = torch.optim.Adam(parameters, lr=0.001)
        for j in range(100):
            optimizer.zero_grad()
            closure()
            optimizer.step()

        print('Starting optimization with L-BFGS')
        def closure2():
            optimizer.zero_grad()
            return closure()
```

```
        optimizer = torch.optim.L-BFGS(parameters, max_iter=num_iter, lr=LR,
tolerance_grad=-1, tolerance_change=-1)
        optimizer.step(closure2)

    elif optimizer_type == 'adam':
        print('Starting optimization with ADAM')
        optimizer = torch.optim.Adam(parameters, lr=LR)
        for j in range(num_iter):
            optimizer.zero_grad()
            closure()
            optimizer.step()

def get_net(input_depth, NET_TYPE, pad, upsample_mode, n_channels=3,
act_fun='LeakyReLU', skip_n33d=128, skip_n33u=128, skip_n11=4, num_scales=5,
downsample_mode='stride'):
    net = skip(input_depth, n_channels, num_channels_down = [skip_n33d]*num_scales if
isinstance(skip_n33d, int) else skip_n33d,
                                num_channels_up =   [skip_n33u]*num_scales if
isinstance(skip_n33u, int) else skip_n33u,
                                num_channels_skip = [skip_n11]*num_scales if
isinstance(skip_n11, int) else skip_n11,
                                upsample_mode=upsample_mode,
downsample_mode=downsample_mode,
                                need_sigmoid=True, need_bias=True, pad=pad, act_fun=act_fun)

    return net

def add_module(self, module):
    self.add_module(str(len(self) + 1), module)

torch.nn.Module.add = add_module

class Concat(nn.Module):
    def __init__(self, dim, *args):
        super(Concat, self).__init__()
        self.dim = dim

        for idx, module in enumerate(args):
            self.add_module(str(idx), module)

    def forward(self, input):
        inputs = []
        for module in self._modules.values():
            inputs.append(module(input))

        inputs_shapes2 = [x.shape[2] for x in inputs]
        inputs_shapes3 = [x.shape[3] for x in inputs]

        if np.all(np.array(inputs_shapes2) == min(inputs_shapes2)) and
np.all(np.array(inputs_shapes3) == min(inputs_shapes3)):
            inputs_ = inputs
        else:
```

```
            target_shape2 = min(inputs_shapes2)
            target_shape3 = min(inputs_shapes3)

            inputs_ = []
            for inp in inputs:
                diff2 = (inp.size(2) - target_shape2) // 2
                diff3 = (inp.size(3) - target_shape3) // 2
                inputs_.append(inp[:, :, diff2: diff2 + target_shape2, diff3:diff3 + target_shape3])
        return torch.cat(inputs_, dim=self.dim)

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

class GenNoise(nn.Module):
    def __init__(self, dim2):
        super(GenNoise, self).__init__()
        self.dim2 = dim2

    def forward(self, input):
        a = list(input.size())
        a[1] = self.dim2
        b = torch.zeros(a).type_as(input.data)
        b.normal_()
        x = torch.autograd.Variable(b)
        return x


class Swish(nn.Module):
    """
        https://arxiv.org/abs/1710.05941
        The hype was so huge that I could not help but try it
    """
    def __init__(self):
        super(Swish, self).__init__()
        self.s = nn.Sigmoid()

    def forward(self, x):
        return x * self.s(x)
def act(act_fun = 'LeakyReLU'):
    if isinstance(act_fun, str):
        if act_fun == 'LeakyReLU':
            return nn.LeakyReLU(0.2, inplace=True)
        elif act_fun == 'Swish':
            return Swish()
        elif act_fun == 'ELU':
            return nn.ELU()
        elif act_fun == 'none':
            return nn.Sequential()
    else:
        return act_fun()

def bn(num_features):
```

```
    return nn.BatchNorm2d(num_features)

def conv(in_f, out_f, kernel_size, stride=1, bias=True, pad='zero',
downsample_mode='stride'):
    downsampler = None
    if stride != 1 and downsample_mode != 'stride':
        if downsample_mode == 'avg':
            downsampler = nn.AvgPool2d(stride, stride)
        elif downsample_mode == 'max':
            downsampler = nn.MaxPool2d(stride, stride)
        elif downsample_mode  in ['lanczos2', 'lanczos3']:
            downsampler = Downsampler(n_planes=out_f, factor=stride,
kernel_type=downsample_mode, phase=0.5, preserve_size=True)
        stride = 1
    padder = None
    to_pad = int((kernel_size - 1) / 2)
    if pad == 'reflection':
        padder = nn.ReflectionPad2d(to_pad)
        to_pad = 0
   convolver = nn.Conv2d(in_f, out_f, kernel_size, stride, padding=to_pad, bias=bias)
    layers = filter(lambda x: x is not None, [padder, convolver, downsampler])
    return nn.Sequential(*layers)

def skip(
        num_input_channels=2, num_output_channels=3,
        num_channels_down=[16, 32, 64, 128, 128], num_channels_up=[16, 32, 64, 128, 128],
num_channels_skip=[4, 4, 4, 4, 4],
        filter_size_down=3, filter_size_up=3, filter_skip_size=1,
        need_sigmoid=True, need_bias=True,
        pad='zero', upsample_mode='nearest', downsample_mode='stride',
act_fun='LeakyReLU',
        need1x1_up=True):
    assert len(num_channels_down) == len(num_channels_up) == len(num_channels_skip)
    n_scales = len(num_channels_down)
    if not (isinstance(upsample_mode, list) or isinstance(upsample_mode, tuple)) :
        upsample_mode   = [upsample_mode]*n_scales
    if not (isinstance(downsample_mode, list)or isinstance(downsample_mode, tuple)):
        downsample_mode   = [downsample_mode]*n_scales
    if not (isinstance(filter_size_down, list) or isinstance(filter_size_down, tuple)) :
        filter_size_down   = [filter_size_down]*n_scales
    if not (isinstance(filter_size_up, list) or isinstance(filter_size_up, tuple)) :
        filter_size_up   = [filter_size_up]*n_scales
    last_scale = n_scales - 1


    cur_depth = None
    model = nn.Sequential()
    model_tmp = model
    input_depth = num_input_channels

    for i in range(len(num_channels_down)):
        deeper = nn.Sequential()
```

```
        skip = nn.Sequential()

        if num_channels_skip[i] != 0:
            model_tmp.add(Concat(1, skip, deeper))
        else:
            model_tmp.add(deeper)
        model_tmp.add(bn(num_channels_skip[i] + (num_channels_up[i + 1] if i < last_scale
else num_channels_down[i])))

        if num_channels_skip[i] != 0:
            skip.add(conv(input_depth, num_channels_skip[i], filter_skip_size, bias=need_bias,
pad=pad))
            skip.add(bn(num_channels_skip[i]))
            skip.add(act(act_fun))

        deeper.add(conv(input_depth, num_channels_down[i], filter_size_down[i], 2,
bias=need_bias, pad=pad, downsample_mode=downsample_mode[i]))
        deeper.add(bn(num_channels_down[i]))
        deeper.add(act(act_fun))
        deeper.add(conv(num_channels_down[i], num_channels_down[i], filter_size_down[i],
bias=need_bias, pad=pad))
        deeper.add(bn(num_channels_down[i]))
        deeper.add(act(act_fun))
        deeper_main = nn.Sequential()

        if i == len(num_channels_down) - 1:
            # The deepest
            k = num_channels_down[i]
        else:
            deeper.add(deeper_main)
            k = num_channels_up[i + 1]

        deeper.add(nn.Upsample(scale_factor=2, mode=upsample_mode[i]))

        model_tmp.add(conv(num_channels_skip[i] + k, num_channels_up[i], filter_size_up[i],
1, bias=need_bias, pad=pad))
        model_tmp.add(bn(num_channels_up[i]))
        model_tmp.add(act(act_fun))

        if need1x1_up:
            model_tmp.add(conv(num_channels_up[i], num_channels_up[i], 1, bias=need_bias,
pad=pad))
            model_tmp.add(bn(num_channels_up[i]))
            model_tmp.add(act(act_fun))
        input_depth = num_channels_down[i]
        model_tmp = deeper_main

    model.add(conv(num_channels_up[0], num_output_channels, 1, bias=need_bias,
pad=pad))
    if need_sigmoid:
        model.add(nn.Sigmoid())
    return model
```

```
class Downsampler(nn.Module):
    ''' http://www.realitypixels.com/turk/computergraphics/ResamplingFilters.pdf'''
    def __init__(self, n_planes, factor, kernel_type, phase=0, kernel_width=None,
support=None, sigma=None, preserve_size=False):
        super(Downsampler, self).__init__()
        if kernel_type == 'lanczos2':
            support = 2
            kernel_width = 4 * factor + 1
            kernel_type_ = 'lanczos'
        elif kernel_type == 'lanczos3':
            support = 3
            kernel_width = 6 * factor + 1
            kernel_type_ = 'lanczos'
        elif kernel_type in ['lanczos', 'gauss', 'box']:
            kernel_type_ = kernel_type

self.kernel = get_kernel(factor, kernel_type_, phase, kernel_width, support=support,
sigma=sigma)

        downsampler = nn.Conv2d(n_planes, n_planes, kernel_size=self.kernel.shape,
stride=factor, padding=0)
        downsampler.weight.data[:] = 0
        downsampler.bias.data[:] = 0

        kernel_torch = torch.from_numpy(self.kernel)
        for i in range(n_planes):
            downsampler.weight.data[i, i] = kernel_torch
        self.downsampler_ = downsampler

        if preserve_size:
            if  self.kernel.shape[0] % 2 == 1:
                pad = int((self.kernel.shape[0] - 1) / 2.)
            else:
                pad = int((self.kernel.shape[0] - factor) / 2.)
            self.padding = nn.ReplicationPad2d(pad)
        self.preserve_size = preserve_size

    def forward(self, input):
        if self.preserve_size:
            x = self.padding(input)
        else:
            x= input
        self.x = x
        return self.downsampler_(x)

def get_kernel(factor, kernel_type, phase, kernel_width, support=None, sigma=None):
    assert kernel_type in ['lanczos', 'gauss', 'box']

    # factor  = float(factor)
    if phase == 0.5 and kernel_type != 'box':
        kernel = np.zeros([kernel_width - 1, kernel_width - 1])
```

```
    else:
        kernel = np.zeros([kernel_width, kernel_width])

    if kernel_type == 'box':
        assert phase == 0.5, 'Box filter is always half-phased'
        kernel[:] = 1./(kernel_width * kernel_width)
    elif kernel_type == 'lanczos':
        assert support, 'support is not specified'
        center = (kernel_width + 1) / 2.

        for i in range(1, kernel.shape[0] + 1):
            for j in range(1, kernel.shape[1] + 1):
                if phase == 0.5:
                    di = abs(i + 0.5 - center) / factor
                    dj = abs(j + 0.5 - center) / factor
                else:
                    di = abs(i - center) / factor
                    dj = abs(j - center) / factor
                pi_sq = np.pi * np.pi

                val = 1
                if di != 0:
                    val = val * support * np.sin(np.pi * di) * np.sin(np.pi * di / support)
                    val = val / (np.pi * np.pi * di * di)

                if dj != 0:
                    val = val * support * np.sin(np.pi * dj) * np.sin(np.pi * dj / support)
                    val = val / (np.pi * np.pi * dj * dj)
                kernel[i - 1][j - 1] = val

    kernel /= kernel.sum()
    return kernel

img_pil = crop_image(get_image(fname, imsize)[0], d=32)
img_np = pil_to_np(img_pil)
img_noisy_pil, img_noisy_np = get_noisy_image(img_np, sigma_)
plot_image_grid([img_np, img_noisy_np], 4, 5);

INPUT = 'noise'
pad = 'reflection'
OPT_OVER = 'net,input' #'net'
reg_noise_std = 1./30. # set to 1./20. for sigma=50
# reg_noise_std = 1./20.

LR = 0.01
# OPTIMIZER= 'adam'
OPTIMIZER='L-BFGS'
act_fun='LeakyReLU'
# act_fun='ELU'
# act_fun='Swish'

show_every = 100
```

```
exp_weight=0.99

num_iter = 3500
input_depth = 2
figsize = 5

net = skip(
            input_depth, 3,
            num_channels_down = [8, 16, 32, 64, 128],
            num_channels_up   = [8, 16, 32, 64, 128],
            num_channels_skip = [0, 0, 0, 4, 4],
            upsample_mode='bilinear',
            need_sigmoid=True, need_bias=True, pad=pad, act_fun=act_fun,
            ).type(dtype)

net_input = get_noise(input_depth, INPUT, (img_pil.size[1],
img_pil.size[0])).type(dtype).detach()

# Compute number of parameters
s  = sum([np.prod(list(p.size())) for p in net.parameters()]);
print ('Number of params: %d' % s)

# Loss
mse = torch.nn.MSELoss().type(dtype)
img_noisy_torch = np_to_torch(img_noisy_np).type(dtype)

"""# Optimize"""
net_input_saved = net_input.detach().clone()
noise = net_input.detach().clone()
out_avg = None
last_net = None
psrn_noisy_last = 0

i = 0
psrn = []
Results = namedtuple('Results', 'iteration loss PSNR_noisy PSNR_gt delta')

def closure():
    global i, out_avg, psrn_noisy_last, last_net, net_input

    if reg_noise_std > 0:
        net_input = net_input_saved + (noise.normal_() * reg_noise_std)

    out = net(net_input)
    # Smoothing
    if out_avg is None:
        out_avg = out.detach()
    else:
        out_avg = out_avg * exp_weight + out.detach() * (1 - exp_weight)

    total_loss = mse(out, img_noisy_torch)
    total_loss.backward()
```

```
    psrn_noisy = compare_psnr(img_noisy_np, out.detach().cpu().numpy()[0])
    psrn_gt    = compare_psnr(img_np, out.detach().cpu().numpy()[0])

    psrn.append(Results(i, total_loss.item(), psrn_noisy, psrn_gt, psrn_gt-psrn_noisy),)
    if  i % show_every == 0:
        out_np = torch_to_np(out)
        plot_image_grid([np.clip(out_np, 0, 1),
                         np.clip(torch_to_np(out_avg), 0, 1)], factor=figsize, nrow=1)

    # Backtracking
    if i % show_every:
        if psrn_noisy - psrn_noisy_last < -5:
            for new_param, net_param in zip(last_net, net.parameters()):
                net_param.data.copy_(new_param.cuda())
            return total_loss*0
        else:
            last_net = [x.detach().cpu() for x in net.parameters()]
            psrn_noisy_last = psrn_noisy

    i += 1
    return total_loss

p = get_params(OPT_OVER, net, net_input)
optimize(OPTIMIZER, p, closure, LR, num_iter)

out_np = torch_to_np(net(net_input))
q = plot_image_grid([np.clip(out_np, 0, 1)], factor=13);
```

ДОДАТОК Б

Відомість атестаційної роботи магістра

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	85 с.
	Інші документи	
2.	Презентаційні матеріали	20 плакатів

.	.	Прізвище та ініціали.	Підп.	Дата		
Розробив		Омельченко С.О.			Відновлення пошкоджених зображень за допомогою глибинної нейронної мережі	Шифр групи: СШІм-19-1; Код напр./спец.: 122
Перевірив		Кулішова Н.Є.				
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ
Затв.		Філатов В.О.				