import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

import torch

import torchvision

import torch.nn as nn

import torch.optim as optim

import torchvision.transforms as transforms

import torchvision.datasets as datasets

from torchvision.utils import make\_grid

from torch.utils.data import Dataset, DataLoader

from tqdm import tqdm

class Discriminator(nn.Module):

def \_\_init\_\_(self, input\_c, num\_filters=64, n\_down=3):

super().\_\_init\_\_()

model = [self.get\_layers(input\_c, num\_filters, norm=False)]

model += [self.get\_layers(num\_filters \* 2 \*\* i, num\_filters \* 2 \*\* (i + 1), s=1 if i == (n\_down-1) else 2)

for i in range(n\_down)]

model += [self.get\_layers(num\_filters \* 2 \*\* n\_down, 1, s=1, norm=False, act=False)]

self.model = nn.Sequential(\*model)

def get\_layers(self, ni, nf, k=4, s=2, p=1, norm=True, act=True):

layers = [nn.Conv2d(ni, nf, k, s, p, bias=not norm)]

if norm: layers += [nn.BatchNorm2d(nf)]

if act: layers += [nn.LeakyReLU(0.2, True)]

return nn.Sequential(\*layers)

def forward(self, x):

return self.model(x)

def init\_weights(net, init='norm', gain=0.02):

def init\_func(m):

classname = m.\_\_class\_\_.\_\_name\_\_

if hasattr(m, 'weight') and 'Conv' in classname:

if init == 'norm':

nn.init.normal\_(m.weight.data, mean=0.0, std=gain)

elif init == 'xavier':

nn.init.xavier\_normal\_(m.weight.data, gain=gain)

elif init == 'kaiming':

nn.init.kaiming\_normal\_(m.weight.data, a=0, mode='fan\_in')

if hasattr(m, 'bias') and m.bias is not None:

nn.init.constant\_(m.bias.data, 0.0)

elif 'BatchNorm2d' in classname:

nn.init.normal\_(m.weight.data, 1., gain)

nn.init.constant\_(m.bias.data, 0.)

net.apply(init\_func)

print(f"Initializing the model with {init} initialization")

return net

def init\_model(model, device):

model = model.to(device)

model = init\_weights(model)

return model

class GANLoss(nn.Module):

def \_\_init\_\_(self, gan\_mode='vanilla', real\_label=0.9, fake\_label=0.1):

super().\_\_init\_\_()

self.register\_buffer('real\_label', torch.tensor(real\_label))

self.register\_buffer('fake\_label', torch.tensor(fake\_label))

if gan\_mode == 'vanilla':

self.loss = nn.BCEWithLogitsLoss()

elif gan\_mode == 'lsgan':

self.loss = nn.MSELoss()

def get\_labels(self, preds, target\_is\_real):

if target\_is\_real:

labels = self.real\_label

else:

labels = self.fake\_label

return labels.expand\_as(preds)

def \_\_call\_\_(self, preds, target\_is\_real):

labels = self.get\_labels(preds, target\_is\_real)

loss = self.loss(preds, labels)

return loss

class Model(nn.Module):

def \_\_init\_\_(self, G\_net, lr\_G=0.0004, lr\_D=0.0004, beta1=0.5, beta2=0.999, lamda=100.):

super().\_\_init\_\_()

self.device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

self.lamda = lamda

self.G\_net = G\_net.to(self.device)

self.D\_net = init\_model(Discriminator(input\_c=3, n\_down=3, num\_filters=64), self.device)

self.GANcriterion = GANLoss(gan\_mode='vanilla').to(self.device)

self.L1criterion = nn.L1Loss()

self.opt\_G = optim.Adam(self.G\_net.parameters(), lr=lr\_G, betas=(beta1, beta2))

self.opt\_D = optim.Adam(self.D\_net.parameters(), lr=lr\_D, betas=(beta1, beta2))

def set\_requires\_grad(self, model, requires\_grad=True):

for p in model.parameters():

p.requires\_grad = requires\_grad

def setup\_input(self, data):

self.L = data['L'].to(self.device)

self.ab = data['ab'].to(self.device)

def forward(self):

self.fake\_color = self.G\_net(self.L)

def backward\_D(self):

fake\_image = torch.cat([self.L, self.fake\_color], dim=1)

fake\_preds = self.D\_net(fake\_image.detach())

self.loss\_D\_fake = self.GANcriterion(fake\_preds, False)

real\_image = torch.cat([self.L, self.ab], dim=1)

real\_preds = self.D\_net(real\_image)

self.loss\_D\_real = self.GANcriterion(real\_preds, True)

self.loss\_D = (self.loss\_D\_fake + self.loss\_D\_real) \* 0.5

self.loss\_D.backward()

def backward\_G(self):

fake\_image = torch.cat([self.L, self.fake\_color], dim=1)

fake\_preds = self.D\_net(fake\_image)

self.loss\_G\_GAN = self.GANcriterion(fake\_preds, True)

self.loss\_G\_L1 = self.L1criterion(self.fake\_color, self.ab) \* self.lamda

self.loss\_G = self.loss\_G\_GAN + self.loss\_G\_L1

self.loss\_G.backward()

def optimize(self):

self.forward()

self.D\_net.train()

self.set\_requires\_grad(self.D\_net, True)

self.opt\_D.zero\_grad()

self.backward\_D()

self.opt\_D.step()

self.G\_net.train()

self.set\_requires\_grad(self.D\_net, False)

self.opt\_G.zero\_grad()

self.backward\_G()

self.opt\_G.step()