import torch

from tqdm import tqdm

from diffusion import GaussianDiffusion

from utils import lab\_to\_pil, lab\_to\_rgb, show\_lab\_image, split\_lab\_channels

import torchvision

import wandb

import pytorch\_lightning as pl

from matplotlib import pyplot as plt

from torch\_ema import ExponentialMovingAverage

class ColorDiffusion(pl.LightningModule):

def \_\_init\_\_(self,

unet,

train\_dl,

val\_dl,

encoder,

loss\_fn="l2",

T=300,

lr=1e-4,

batch\_size=1,

sample=True,

should\_log=True,

using\_cond=False,

display\_every=None,

dynamic\_threshold=False,

use\_ema=True,

\*\*kwargs):

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

self.unet = unet.to(self.device)

self.T = T

self.lr = lr

self.using\_cond = using\_cond

self.sample = sample

self.should\_log = should\_log

self.encoder = encoder

self.display\_every = display\_every

self.val\_dl = val\_dl

self.train\_dl = train\_dl

if loss\_fn == "l1":

self.loss\_fn = torch.nn.functional.l1\_loss

else:

self.loss\_fn = torch.nn.functional.mse\_loss

self.ema = ExponentialMovingAverage(self.unet.parameters(),

decay=0.9999)

self.ema.to(self.device)

self.diffusion = GaussianDiffusion(T,

dynamic\_threshold=dynamic\_threshold)

if sample is True and display\_every is None:

display\_every = 1000

self.save\_hyperparameters(ignore=['unet'])

def forward(self, x\_noised, t, x\_l):

"""

Performs one denoising step on batch of noised inputs

Unet is conditioned on timestep and features extracted from greyscale channel

"""

cond = self.encoder(x\_l)

noise\_pred = self.unet(x\_noised, t, greyscale\_embs=cond)

return noise\_pred

def get\_batch\_pred(self, x\_0, x\_l):

"""

Samples a timestep from range [0, T]

Adds noise to images x\_0 to get x\_t (x\_0 with color channels noised)

Returns:

- The model's prediction of the noise,

- The real noise applied to the color channels by the forward diffusion process

"""

t = torch.randint(0, self.T, (x\_0.shape[0],)).to(x\_0)

x\_noised, noise = self.diffusion.forward\_diff(x\_0, t, T=self.T)

return (self(x\_noised, t, x\_l), noise)

def get\_losses(self, noise\_pred, noise, x\_l):

diff\_loss = self.loss\_fn(noise\_pred, noise)

return {"total loss": diff\_loss}

def training\_step(self, x\_0, batch\_idx):

x\_l, \_ = split\_lab\_channels(x\_0)

noise\_pred, noise = self.get\_batch\_pred(x\_0, x\_l)

losses = self.get\_losses(noise\_pred, noise, x\_l)

self.log\_dict(losses, on\_step=True)

if self.sample and batch\_idx and batch\_idx % self.display\_every == 0 and self.global\_step > 1:

self.test\_step(x\_0)

return losses["total loss"]

def validation\_step(self, batch, batch\_idx):

x\_l, \_ = split\_lab\_channels(batch)

noise\_pred, noise = self.get\_batch\_pred(batch, x\_l)

losses = self.get\_losses(noise\_pred, noise, x\_l)

if self.should\_log:

self.log("val\_loss", losses["total loss"])

if self.sample and batch\_idx and batch\_idx % self.display\_every == 0:

self.sample\_plot\_image(batch)

return losses["total loss"]

@torch.inference\_mode()

def test\_step(self, batch, \*args, \*\*kwargs):

x = next(iter(self.val\_dl)).to(batch)

self.sample\_plot\_image(x)

self.sample\_plot\_image(x, use\_ema=True)

def configure\_optimizers(self):

learnable\_params = list(self.unet.parameters()) \

+ list(self.encoder.parameters())

global\_optim = torch.optim.AdamW(learnable\_params,

lr=self.lr,

weight\_decay=28e-3)

return global\_optim

def log\_img(self, image, caption="diff samples", use\_ema=False):

rgb\_imgs = lab\_to\_rgb(\*split\_lab\_channels(image))

if use\_ema:

self.logger.log\_image("EMA samples", [rgb\_imgs])

else:

self.logger.log\_image("samples", [rgb\_imgs])

def on\_before\_zero\_grad(self, \*args, \*\*kwargs):

self.ema.update()

@torch.inference\_mode()

def sample\_loop(self, x\_l, prog=False, use\_ema=False, save\_all=False):

"""

Noises color channels to timestep T, then denoises the color channels

to t=0 to get the colorized image.

Returns an array containing the noised image,

intermediate images in the denoising process, and the final image

"""

ema = self.ema if use\_ema else None

images = []

num\_images = 13

img\_size = x\_l.shape[-1]

stepsize = self.T // num\_images

# Initialize image with random noise in color channels

x\_ab = torch.randn((x\_l.shape[0], 2, img\_size, img\_size)).to(x\_l)

img = torch.cat((x\_l, x\_ab), dim=1)

counter = range(0, self.T)[::-1]

if prog:

counter = tqdm(counter)

for i in counter:

t = torch.full((1,), i, dtype=torch.long).to(img)

img = self.diffusion.sample\_timestep(self.unet,

self.encoder,

img,

t,

T=self.T,

cond=x\_l,

ema=ema)

if i % stepsize == 0:

images += img.unsqueeze(0)

if save\_all and i % 2 == 0:

pil\_img = lab\_to\_pil(img)

pil\_img.save(f"./visualization/denoising/{i:04d}.png")

return images

@torch.inference\_mode()

def sample\_plot\_image(self, x\_0, show=True, prog=False,

use\_ema=False, log=True, save\_all=False):

"""

Denoises a single image and displays a grid showing:

- ground truth image

- intermediate denoised outputs

- the final denoised image

"""

print("Sampling image")

ground\_truth\_images = []

if x\_0.shape[1] == 3:

x\_l, \_ = split\_lab\_channels(x\_0)

ground\_truth\_images.append(x\_0[:1])

else:

x\_l = x\_0

x\_l = x\_l[:1]

greyscale = torch.cat((x\_l, \*[torch.zeros\_like(x\_l)] \* 2), dim=1)

ground\_truth\_images += greyscale.unsqueeze(0)

if len(x\_l.shape) == 3:

x\_l = x\_l.unsqueeze(0)

images = ground\_truth\_images + self.sample\_loop(x\_l,

prog=prog,

use\_ema=use\_ema,

save\_all=save\_all)

grid = torchvision.utils.make\_grid(torch.cat(images), dim=0).to(x\_l)

if show:

show\_lab\_image(grid.unsqueeze(0), log=self.should\_log)

plt.show()

if self.should\_log and log:

self.log\_img(grid.unsqueeze(0), use\_ema=use\_ema)

return lab\_to\_rgb(\*split\_lab\_channels(images[-1]))