Classical Algorithms Implementation

Anagha H.C. Sachin Prasanna Abhayjit Singh

October 2023

1 Denoising Diffusion Probabilistic Model

Diffusion Probabilistic Models, often abbreviated as DDPMs, are a noteworthy category within the realm of generative models. They do so by iteratively introducing controlled noise into an initial input signal. The underlying concept is to acquire a deep understanding of the noise removal process, enabling the generation of entirely fresh and coherent data samples.

Main File:

```

import torch
import logging
import os
import torch.nn as nn
from tqdm import tqdm
from torch import optim
from utils import *
from modules import UNet
from matplotlib import pyplot as plt
import logging
from torch.utils.tensorboard import SummaryWriter

logging.basicConfig(format="%(asctime)s␣-␣%(levelname)s:␣%(message)s", level=
    logging.INFO, datefmt="%I:%M:%S")


class Diffusion:
    def __init__(self, ns=10000, strt=1e-5, end=0.02, imsiz=256, device="cuda")
        :
        self.ns = ns
        self.strt = strt
        self.end = end
        self.imsiz = imsiz
        self.device = device

        self.beta = self.scheduler_noise().to(device)
        self.alpha = 1. - self.beta
        self.ahat = torch.cumprod(self.alpha, dim=0)

    def sample_timesteps(self, n):
        return torch.randint(low=1, high=self.ns, size=(n,))

    def scheduler_noise(self):
        return torch.linspace(self.strt, self.end, self.ns)

    def noise_images(self, x, t):
        alp1 = torch.sqrt(self.ahat[t])[:, None, None, None]
        alp2 = torch.sqrt(1 - self.ahat[t])[:, None, None, None]
          = torch.randn_like(x)
        return alp1 * x + alp2*   ,

    def sample(self, model, n):
```

```python
        logging.info(f"Sampling {n} new images....")
        model.eval()
        with torch.no_grad():
            x = torch.randn((n, 3, self.imsiz, self.imsiz)).to(self.device)
            for i in tqdm(reversed(range(1, self.ns)), position=0):
                t = (torch.ones(n) * i).long().to(self.device)
                pn = model(x, t)
                alpha = self.alpha[t][:, None, None, None]
                ahat = self.ahat[t][:, None, None, None]
                beta = self.beta[t][:, None, None, None]
                if i > 1:
                    noise = torch.randn_like(x)
                else:
                    noise = torch.zeros_like(x)
                x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 -
                    ahat))) * pn) + torch.sqrt(beta) * noise
        model.train()
        x = (x.clamp(-1, 1) + 1) / 2
        x = (x * 255).type(torch.uint8)
        return x


def train(args):
    setup_logging(args.run_name)
    device = args.device
    dataloader = get_data(args)
    model = UNet().to(device)
    optimizer = optim.AdamW(model.parameters(), lr=args.lr)
    mse = nn.MSELoss()
    diffusion = Diffusion(imsiz=args.image_size, device=device)
    logger = SummaryWriter(os.path.join("runs", args.run_name))
    l = len(dataloader)

    for epoch in range(args.epochs):
        logging.info(f"epoch: {epoch}:")
        pbar = tqdm(dataloader)
        for i, (images, _) in enumerate(pbar):
            images = images.to(device)
            t = diffusion.sample_timesteps(images.shape[0]).to(device)
            x_t, noise = diffusion.noise_images(images, t)
            pn = model(x_t, t)
            loss = mse(noise, pn)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            pbar.set_postfix(MSE=loss.item())
            logger.add_scalar("MSE", loss.item(), global_step=epoch * l + i)

        sampled_images = diffusion.sample(model, n=images.shape[0])
        save_images(sampled_images, os.path.join("results", args.run_name, f"{
            epoch}.jpg"))
        torch.save(model.state_dict(), os.path.join("models", args.run_name, f"
            ckpt.pt"))


def launch():
    import argparse
    parser = argparse.ArgumentParser()
    args = parser.parse_args()
    args.run_name = "DDPM_Uncondtional"
    args.epochs = 500
```

```
    args.batch_size = 12
    args.image_size = 64
    args.dataset_path = r"C:\Users\it303project\datasets\landscape_img_folder"
    args.device = "cuda"
    args.lr = 3e-4
    train(args)


if __name__ == '__main__':
    launch()
    device = "cuda"
    model = UNet().to(device)
    ckpt = torch.load("./working/orig/ckpt.pt")
    model.load_state_dict(ckpt)
    diffusion = Diffusion(imsiz=64, device=device)
    x = diffusion.sample(model, 8)
    print(x.shape)
    plt.figure(figsize=(32, 32))
    plt.imshow(torch.cat([
        torch.cat([i for i in x.cpu()], dim=-1),
    ], dim=-2).permute(1, 2, 0).cpu())
    plt.show()
```

UNet File:

```


import torch
import torch.nn as nn
import torch.nn.functional as F


class SelfAttention(nn.Module):
    def __init__(self, chs, size):
        super(SelfAttention, self).__init__()
        self.chs = chs
        self.size = size
        self.mha = nn.MultiheadAttention(chs, 4, batch_first=True)
        self.ln = nn.LayerNorm([chs])
        self.ff_self = nn.Sequential(
            nn.LayerNorm([chs]),
            nn.Linear(chs, chs),
            nn.GELU(),
            nn.Linear(chs, chs),
        )

    def forward(self, x):
        x = x.view(-1, self.chs, self.size * self.size).swapaxes(1, 2)
        x_ln = self.ln(x)
        atval, _ = self.mha(x_ln, x_ln, x_ln)
        atval = atval + x
        atval = self.ff_self(atval) + atval
        return atval.swapaxes(2, 1).view(-1, self.chs, self.size, self.size)


class DoubleConv(nn.Module):
    def __init__(self, inch, outch, midch=None, residual=False):
        super().__init__()
        self.residual = residual
        if not midch:
            midch = outch
        self.double_conv = nn.Sequential(
            nn.Conv2d(inch, midch, kernel_size=3, padding=1, bias=False),
            nn.GroupNorm(1, midch),
```

```python
            nn.GELU(),
            nn.Conv2d(midch, outch, kernel_size=3, padding=1, bias=False),
            nn.GroupNorm(1, outch),
        )

    def forward(self, x):
        if self.residual:
            return F.gelu(x + self.double_conv(x))
        else:
            return self.double_conv(x)


class Down(nn.Module):
    def __init__(self, inch, outch, emb_dim=256):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(inch, inch, residual=True),
            DoubleConv(inch, outch),
        )

        self.emb_layer = nn.Sequential(
            nn.SiLU(),
            nn.Linear(
                emb_dim,
                outch
            ),
        )

    def forward(self, x, t):
        x = self.maxpool_conv(x)
        emb = self.emb_layer(t)[:, :, None, None].repeat(1, 1, x.shape[-2], x.shape[-1])
        return x + emb


class Up(nn.Module):
    def __init__(self, inch, outch, emb_dim=256):
        super().__init__()

        self.up = nn.Upsample(scale_factor=2, mode="bilinear", align_corners=True)
        self.conv = nn.Sequential(
            DoubleConv(inch, inch, residual=True),
            DoubleConv(inch, outch, inch // 2),
        )

        self.emb_layer = nn.Sequential(
            nn.SiLU(),
            nn.Linear(
                emb_dim,
                outch
            ),
        )

    def forward(self, x, skip_x, t):
        x = self.up(x)
        x = torch.cat([skip_x, x], dim=1)
        x = self.conv(x)
        emb = self.emb_layer(t)[:, :, None, None].repeat(1, 1, x.shape[-2], x.shape[-1])
        return x + emb

```

```python

class UNet(nn.Module):
    def __init__(self, cinn=3, coutt=3, dimensiontime=256, device="cuda"):
        super().__init__()
        self.device = device
        self.dimensiontime = dimensiontime
        self.inc = DoubleConv(cinn, 64)
        self.dlayer1 = Down(64, 128)
        self.salayer1 = SelfAttention(128, 32)
        self.dlayer2 = Down(128, 256)
        self.salayer2 = SelfAttention(256, 16)
        self.dlayer3 = Down(256, 256)
        self.salayer3 = SelfAttention(256, 8)

        self.bottleneck1 = DoubleConv(256, 512)
        self.bottleneck2 = DoubleConv(512, 512)
        self.bottleneck3 = DoubleConv(512, 256)

        self.ulayer1 = Up(512, 128)
        self.salayer4 = SelfAttention(128, 16)
        self.ulayer2 = Up(256, 64)
        self.salayer5 = SelfAttention(64, 32)
        self.ulayer3 = Up(128, 64)
        self.salayer6 = SelfAttention(64, 64)
        self.outc = nn.Conv2d(64, coutt, kernel_size=1)

    def pos_encoding(self, t, chs):
        inv_freq = 1.0 / (
            10000
            ** (torch.arange(0, chs, 2, device=self.device).float() / chs)
        )
        posa = torch.sin(t.repeat(1, chs // 2) * inv_freq)
        posb = torch.cos(t.repeat(1, chs // 2) * inv_freq)
        posen = torch.cat([posa, posb], dim=-1)
        return posen

    def forward(self, x, t):
        t = t.unsqueeze(-1).type(torch.float)
        t = self.pos_encoding(t, self.dimensiontime)

        x1 = self.inc(x)
        x2 = self.dlayer1(x1, t)
        x2 = self.salayer1(x2)
        x3 = self.dlayer2(x2, t)
        x3 = self.salayer2(x3)
        x4 = self.dlayer3(x3, t)
        x4 = self.salayer3(x4)

        x4 = self.bottleneck1(x4)
        x4 = self.bottleneck2(x4)
        x4 = self.bottleneck3(x4)

        x = self.ulayer1(x4, x3, t)
        x = self.salayer4(x)
        x = self.ulayer2(x, x2, t)
        x = self.salayer5(x)
        x = self.ulayer3(x, x1, t)
        x = self.salayer6(x)
        output = self.outc(x)
        return output

if __name__ == '__main__':
    net = UNet(device="cpu")
```

```
    # net = UNet_conditional(num_classes=10, device="cpu")
    print(sum([p.numel() for p in net.parameters()]))
    x = torch.randn(3, 3, 64, 64)
    t = x.new_tensor([500] * x.shape[0]).long()
    y = x.new_tensor([1] * x.shape[0]).long()
    print(net(x, t, y).shape)
```

Utils File:

```
import os
import torch
import torchvision
from PIL import Image
from matplotlib import pyplot as plt
from torch.utils.data import DataLoader


def plot_images(images):
    plt.figure(figsize=(32, 32))
    plt.imshow(torch.cat([
        torch.cat([i for i in images.cpu()], dim=-1),
    ], dim=-2).permute(1, 2, 0).cpu())
    plt.show()


def save_images(images, path, **kwargs):
    grid = torchvision.utils.make_grid(images, **kwargs)
    ndarr = grid.permute(1, 2, 0).to('cpu').numpy()
    im = Image.fromarray(ndarr)
    im.save(path)


def get_data(args):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(80),  # args.image_size + 1/4 *args.
            image_size
        torchvision.transforms.RandomResizedCrop(args.image_size, scale=(0.8,
            1.0)),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    dataset = torchvision.datasets.ImageFolder(args.dataset_path, transform=
        transforms)
    dataloader = DataLoader(dataset, batch_size=args.batch_size, shuffle=True)
    return dataloader


def setup_logging(run_name):
    os.makedirs("models", exist_ok=True)
    os.makedirs("results", exist_ok=True)
    os.makedirs(os.path.join("models", run_name), exist_ok=True)
    os.makedirs(os.path.join("results", run_name), exist_ok=True)
```

2 Generative Models

Generative models are a class of artificial intelligence algorithms that focus on creating data rather than making predictions. They learn the underlying patterns and structures within datasets, enabling them to generate new, synthetic data samples. These models have diverse applications, including generating realistic images, natural language text and audio. The ability to generate data with high fidelity and diversity has significant implications for fields like computer vision, natural language processing, and data augmentation. DDPMs also have a similar aim of learning the distribution of a training data sample and then generating a new sample that closely resembles it.

3 Generative Adversarial Networks

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow and colleagues in 2014, comprise two crucial neural networks: a generator and a discriminator. The generator and discriminator (aka critic).The generator produces a sample, such as an image, from a latent code. Ideally, the distribution of these images should be indistinguishable from the training distribution, while the discriminator's role is to distinguish real data from generated data. They engage in a competitive game, with the generator refining its output to resemble genuine data and the discriminator enhancing its ability to differentiate. Typically, a GAN consists of two networks: GAN training strikes a balance with a dynamic feedback loop; as the generator improves, the discriminator adapts, fostering ongoing competition. GANs excel in producing highly realistic data for computer vision, art, and data augmentation. In a related context, adversarial nets establish a competitive framework, pitting a generative model against a discriminative model. The generative model aims to craft "counterfeit" samples indistinguishable from genuine data, while the discriminative model detects discrepancies. This approach, trainable using methods like backpropagation and dropout algorithms, holds promise for deep generative modeling, aligning with our exploration of Diffusion Models in this literature review.

```

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.datasets as datasets
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
from torch.utils.tensorboard import SummaryWriter  # to print to tensorboard


class Discriminator(nn.Module):
    def __init__(self, in_features):
        super().__init__()
        self.disc = nn.Sequential(
            nn.Linear(in_features, 128),
            nn.LeakyReLU(0.01),
            nn.Linear(128, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return self.disc(x)


class Generator(nn.Module):
    def __init__(self, z_dim, img_dim):
        super().__init__()
        self.gen = nn.Sequential(
            nn.Linear(z_dim, 256),
            nn.LeakyReLU(0.01),
            nn.Linear(256, img_dim),
            nn.Tanh(),  # normalize inputs to [-1, 1] so make outputs [-1, 1]
        )

    def forward(self, x):
        return self.gen(x)


# Hyperparameters etc.
device = "cuda" if torch.cuda.is_available() else "cpu"
lr = 3e-4
```

```python
z_dim = 64
image_dim = 28 * 28 * 1  # 784
batch_size = 32
num_epochs = 50

disc = Discriminator(image_dim).to(device)
gen = Generator(z_dim, image_dim).to(device)
fixed_noise = torch.randn((batch_size, z_dim)).to(device)
transforms = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,)),
    ]
)

dataset = datasets.MNIST(root="dataset/", transform=transforms, download=True)
loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
opt_disc = optim.Adam(disc.parameters(), lr=lr)
opt_gen = optim.Adam(gen.parameters(), lr=lr)
criterion = nn.BCELoss()
writer_fake = SummaryWriter(f"logs/fake")
writer_real = SummaryWriter(f"logs/real")
step = 0

for epoch in range(num_epochs):
    for batch_idx, (real, _) in enumerate(loader):
        real = real.view(-1, 784).to(device)
        batch_size = real.shape[0]

        noise = torch.randn(batch_size, z_dim).to(device)
        fake = gen(noise)
        disc_real = disc(real).view(-1)
        lossD_real = criterion(disc_real, torch.ones_like(disc_real))
        disc_fake = disc(fake).view(-1)
        lossD_fake = criterion(disc_fake, torch.zeros_like(disc_fake))
        lossD = (lossD_real + lossD_fake) / 2
        disc.zero_grad()
        lossD.backward(retain_graph=True)
        opt_disc.step()


        output = disc(fake).view(-1)
        lossG = criterion(output, torch.ones_like(output))
        gen.zero_grad()
        lossG.backward()
        opt_gen.step()

        if batch_idx == 0:
            print(
                f"Epoch␣[{epoch}/{num_epochs}]␣Batch␣{batch_idx}/{len(loader)}␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Loss␣D:␣{lossD:.4f},␣loss␣G:␣{lossG:.4f}"
            )

            with torch.no_grad():
                fake = gen(fixed_noise).reshape(-1, 1, 28, 28)
                data = real.reshape(-1, 1, 28, 28)
                img_grid_fake = torchvision.utils.make_grid(fake, normalize=True)
                img_grid_real = torchvision.utils.make_grid(data, normalize=True)

                writer_fake.add_image(
```

```
                    "Mnist Fake Images", img_grid_fake, global_step=step
                )
                writer_real.add_image(
                    "Mnist Real Images", img_grid_real, global_step=step
                )
                step += 1
```

4 Progressively Growing GAN

Progressive Growing is a methodology employed in the training of Generative Adversarial Networks (GANs) to enhance their proficiency in generating high-resolution images. The fundamental concept underpinning Progressive Growing involves the incremental training of a GAN on images of lower resolutions, subsequently advancing to higher resolutions as the training advances. This approach affords the neural network the opportunity to initially grasp fundamental and less intricate image features, which are progressively refined to produce more intricate and detailed visual content.

config.py

```
import torch
from math import log2

START_TRAIN_AT_IMG_SIZE = 256
DATASET = 'celeba_hq'
CHECKPOINT_GEN = "generator.pth"
CHECKPOINT_CRITIC = "discriminator.pth"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
SAVE_MODEL = True
LOAD_MODEL = False
LEARNING_RATE = 1e-3
BATCH_SIZES = [32, 32, 32, 16, 16, 16, 16, 8, 4]
IMAGE_SIZE = 512
CHANNELS_IMG = 3
Z_DIM = 256
IN_CHANNELS = 256
CRITIC_ITERATIONS = 1
LAMBDA_GP = 10
NUM_STEPS = int(log2(IMAGE_SIZE / 4)) + 1

PROGRESSIVE_EPOCHS = [50] * len(BATCH_SIZES)
FIXED_NOISE = torch.randn(8, Z_DIM, 1, 1).to(DEVICE)
NUM_WORKERS = 4
```

progressiveGAN.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from math import log2

factors = [1, 1, 1, 1, 1/2, 1/4, 1/8, 1/16, 1/32]


class WSConv2d(nn.Module):
    def __init__(self, input_channel, out_channel, kernel_size=3 , stride=1,
       padding=1, gain=2):
        super(WSConv2d, self).__init__()
        self.conv = nn.Conv2d(input_channel, out_channel, kernel_size, stride,
           padding)
        self.scale = (gain / (input_channel * (kernel_size ** 2))) ** 0.5
        self.bias = self.conv.bias
        self.conv.bias = None
```

```python

        # conv layer
        nn.init.normal_(self.conv.weight)
        nn.init.zeros_(self.bias)

    def forward(self, x):
        return self.conv(x * self.scale) + self.bias.view(1, self.bias.shape[0], 1, 1)


class PixelNorm(nn.Module):
    def __init__(self):
        super(PixelNorm, self).__init__()
        self.epsilon = 1e-8

    def forward(self, x):
        return x / torch.sqrt(torch.mean(x ** 2, dim=1, keepdim=True) + self.epsilon)


class CNNBlock(nn.Module):
    def __init__(self, input_channel, out_channel, pixel_norm=True):
        super(CNNBlock, self).__init__()
        self.conv1 = WSConv2d(input_channel, out_channel)
        self.conv2 = WSConv2d(out_channel, out_channel)
        self.leaky = nn.LeakyReLU(0.2)
        self.pn = PixelNorm()
        self.use_pn = pixel_norm

    def forward(self, x):
        x = self.leaky(self.conv1(x))
        x = self.pn(x) if self.use_pn else x
        x = self.leaky(self.conv2(x))
        x = self.pn(x) if self.use_pn else x
        return x


class Generator(nn.Module):
    def __init__(self, z_dim, in_channels, img_channels=3):
        super(Generator, self).__init__()

        # initial takes 1x1 -> 4x4
        self.initial = nn.Sequential(
            PixelNorm(),
            nn.ConvTranspose2d(z_dim, in_channels, 4, 1, 0),
            nn.LeakyReLU(0.2),
            WSConv2d(in_channels, in_channels, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            PixelNorm(),
        )

        self.initial_rgb = WSConv2d(
            in_channels, img_channels, kernel_size=1, stride=1, padding=0
        )
        self.prog_blocks, self.rgb_layers = (
            nn.ModuleList([]),
            nn.ModuleList([self.initial_rgb]),
        )

        for i in range(
                len(factors) - 1
        ):  # -1 to prevent index error because of factors[i+1]
```

```python
            conv_in_c = int(in_channels * factors[i])
            conv_out_c = int(in_channels * factors[i + 1])
            self.prog_blocks.append(CNNBlock(conv_in_c, conv_out_c))
            self.rgb_layers.append(
                WSConv2d(conv_out_c, img_channels, kernel_size=1, stride=1, padding=0)
            )

    def fade_in(self, alpha, upscaled, generated):
        # alpha should be scalar within [0, 1], and upscale.shape == generated.shape
        return torch.tanh(alpha * generated + (1 - alpha) * upscaled)

    def forward(self, x, alpha, steps):
        out = self.initial(x)

        if steps == 0:
            return self.initial_rgb(out)

        for step in range(steps):
            upscaled = F.interpolate(out, scale_factor=2, mode="nearest")
            out = self.prog_blocks[step](upscaled)

        final_upscaled = self.rgb_layers[steps - 1](upscaled)
        final_out = self.rgb_layers[steps](out)
        return self.fade_in(alpha, final_upscaled, final_out)


class Discriminator(nn.Module):
    def __init__(self, z_dim, in_channels, img_channels=3):
        super(Discriminator, self).__init__()
        self.prog_blocks, self.rgb_layers = nn.ModuleList([]), nn.ModuleList([])
        self.leaky = nn.LeakyReLU(0.2)


        for i in range(len(factors) - 1, 0, -1):
            conv_in = int(in_channels * factors[i])
            conv_out = int(in_channels * factors[i - 1])
            self.prog_blocks.append(CNNBlock(conv_in, conv_out, pixel_norm=False))
            self.rgb_layers.append(
                WSConv2d(img_channels, conv_in, kernel_size=1, stride=1, padding=0)
            )

                self.initial_rgb = WSConv2d(
            img_channels, in_channels, kernel_size=1, stride=1, padding=0
        )
        self.rgb_layers.append(self.initial_rgb)
        self.avg_pool = nn.AvgPool2d(
            kernel_size=2, stride=2
        )  # down sampling using avg pool


        self.final_block = nn.Sequential(

            WSConv2d(in_channels + 1, in_channels, kernel_size=3, padding=1),
            nn.LeakyReLU(0.2),
            WSConv2d(in_channels, in_channels, kernel_size=4, padding=0, stride=1),
            nn.LeakyReLU(0.2),
            WSConv2d(
```

```python
                in_channels, 1, kernel_size=1, padding=0, stride=1
            ),
        )

    def fade_in(self, alpha, downscaled, out):
        return alpha * out + (1 - alpha) * downscaled

    def minibatch_std(self, x):
        batch_statistics = (
            torch.std(x, dim=0).mean().repeat(x.shape[0], 1, x.shape[2], x.shape[3]))
        return torch.cat([x, batch_statistics], dim=1)

    def forward(self, x, alpha, steps):
        cur_step = len(self.prog_blocks) - steps

        out = self.leaky(self.rgb_layers[cur_step](x))

        if steps == 0:
            out = self.minibatch_std(out)
            return self.final_block(out).view(out.shape[0], -1)

        downscaled = self.leaky(self.rgb_layers[cur_step + 1](self.avg_pool(x)))
        out = self.avg_pool(self.prog_blocks[cur_step](out))
        out = self.fade_in(alpha, downscaled, out)

        for step in range(cur_step + 1, len(self.prog_blocks)):
            out = self.prog_blocks[step](out)
            out = self.avg_pool(out)

        out = self.minibatch_std(out)
        return self.final_block(out).view(out.shape[0], -1)


if __name__ == "__main__":
    Z_DIM = 100
    IN_CHANNELS = 256
    gen = Generator(Z_DIM, IN_CHANNELS, img_channels=3)
    critic = Discriminator(Z_DIM, IN_CHANNELS, img_channels=3)

    for img_size in [4, 8, 16, 32, 64, 128, 256, 512, 1024]:
        num_steps = int(log2(img_size / 4))
        x = torch.randn((1, Z_DIM, 1, 1))
        z = gen(x, 0.5, steps=num_steps)
        assert z.shape == (1, 3, img_size, img_size)
        out = critic(z, alpha=0.5, steps=num_steps)
        assert out.shape == (1, 1)
        print(f"Success! At img size: {img_size}")
```

train.py

```python
import torch
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
from utils import (
    gradient_penalty,
    plot_to_tensorboard,
    save_checkpoint,
    load_checkpoint,
    generate_examples
```

```
)
from progressive_GAN import Discriminator, Generator
from math import log2
from tqdm import tqdm
import config

torch.backends.cudnn.benchmarks = True


def get_loader(image_size):
    transform = transforms.Compose(
        [
            transforms.Resize((image_size, image_size)),
            transforms.ToTensor(),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.Normalize(
                [0.5 for _ in range(config.CHANNELS_IMG)],
                [0.5 for _ in range(config.CHANNELS_IMG)],
            ),
        ]
    )
    batch_size = config.BATCH_SIZES[int(log2(image_size / 4))]
    dataset = datasets.ImageFolder(root=config.DATASET, transform=transform)
    loader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=config.NUM_WORKERS,
        pin_memory=True,
    )
    return loader, dataset


def train_fn(
    critic,
    gen,
    loader,
    dataset,
    step,
    alpha,
    opt_critic,
    opt_gen,
    tensorboard_step,
    writer,
    scaler_gen,
    scaler_critic,
):
    loop = tqdm(loader, leave=True)
    for batch_idx, (real, _) in enumerate(loop):
        real = real.to(config.DEVICE)
        cur_batch_size = real.shape[0]


        noise = torch.randn(cur_batch_size, config.Z_DIM, 1, 1).to(config.DEVICE)

        with torch.cuda.amp.autocast():
            fake = gen(noise, alpha, step)
            critic_real = critic(real, alpha, step)
            critic_fake = critic(fake.detach(), alpha, step)
            gp = gradient_penalty(critic, real, fake, alpha, step, device=config.DEVICE)
            loss_critic = (
```

```
                -(torch.mean(critic_real) - torch.mean(critic_fake))
                + config.LAMBDA_GP * gp
                + (0.001 * torch.mean(critic_real ** 2))
            )

        opt_critic.zero_grad()
        scaler_critic.scale(loss_critic).backward()
        scaler_critic.step(opt_critic)
        scaler_critic.update()


        with torch.cuda.amp.autocast():
            gen_fake = critic(fake, alpha, step)
            loss_gen = -torch.mean(gen_fake)

        opt_gen.zero_grad()
        scaler_gen.scale(loss_gen).backward()
        scaler_gen.step(opt_gen)
        scaler_gen.update()

        # Update alpha and ensure less than 1
        alpha += cur_batch_size / (
            (config.PROGRESSIVE_EPOCHS[step] * 0.5) * len(dataset)
        )
        alpha = min(alpha, 1)

        if batch_idx % 500 == 0:
            with torch.no_grad():
                fixed_fakes = gen(config.FIXED_NOISE, alpha, step) * 0.5 + 0.5
            plot_to_tensorboard(
                writer,
                loss_critic.item(),
                loss_gen.item(),
                real.detach(),
                fixed_fakes.detach(),
                tensorboard_step,
            )
            tensorboard_step += 1

        loop.set_postfix(
            gp=gp.item(),
            loss_critic=loss_critic.item(),
        )

    return tensorboard_step, alpha


def main():
    gen = Generator(
        config.Z_DIM, config.IN_CHANNELS, img_channels=config.CHANNELS_IMG
    ).to(config.DEVICE)
    critic = Discriminator(
        config.Z_DIM, config.IN_CHANNELS, img_channels=config.CHANNELS_IMG
    ).to(config.DEVICE)

    # initialize optimizers and scalers for FP16 training
    opt_gen = optim.Adam(gen.parameters(), lr=config.LEARNING_RATE, betas=(0.0,
        0.99))
    opt_critic = optim.Adam(
        critic.parameters(), lr=config.LEARNING_RATE, betas=(0.0, 0.99)
    )
    scaler_critic = torch.cuda.amp.GradScaler()
    scaler_gen = torch.cuda.amp.GradScaler()
```

```python

    # for tensorboard plotting
    writer = SummaryWriter(f"logs/gan1")

    if config.LOAD_MODEL:
        load_checkpoint(
            config.CHECKPOINT_GEN, gen, opt_gen, config.LEARNING_RATE,
        )
        load_checkpoint(
            config.CHECKPOINT_CRITIC, critic, opt_critic, config.LEARNING_RATE,
        )

    gen.train()
    critic.train()

    tensorboard_step = 0

    step = int(log2(config.START_TRAIN_AT_IMG_SIZE / 4))
    for num_epochs in config.PROGRESSIVE_EPOCHS[step:]:
        alpha = 1e-5
        loader, dataset = get_loader(4 * 2 ** step)
        print(f"Current image size: {4 * 2 ** step}")

        for epoch in range(num_epochs):
            print(f"Epoch [{epoch+1}/{num_epochs}]")
            tensorboard_step, alpha = train_fn(
                critic,
                gen,
                loader,
                dataset,
                step,
                alpha,
                opt_critic,
                opt_gen,
                tensorboard_step,
                writer,
                scaler_gen,
                scaler_critic,
            )

            if config.SAVE_MODEL:
                save_checkpoint(gen, opt_gen, filename=config.CHECKPOINT_GEN)
                save_checkpoint(critic, opt_critic, filename=config.CHECKPOINT_CRITIC)

        step += 1


if __name__ == "__main__":
    main()
```

utils.py

```python

import torch
import random
import numpy as np
import os
import torchvision
import config
from torchvision.utils import save_image
from scipy.stats import truncnorm


```

```python

def plot_to_tensorboard(
    writer, loss_critic, loss_gen, real, fake, tensorboard_step
):
    writer.add_scalar("Loss Critic", loss_critic, global_step=tensorboard_step)

    with torch.no_grad():
                img_grid_real = torchvision.utils.make_grid(real[:8], normalize=True)
        img_grid_fake = torchvision.utils.make_grid(fake[:8], normalize=True)
        writer.add_image("Real", img_grid_real, global_step=tensorboard_step)
        writer.add_image("Fake", img_grid_fake, global_step=tensorboard_step)


def gradient_penalty(critic, real, fake, alpha, train_step, device="cpu"):
    BATCH_SIZE, C, H, W = real.shape
    beta = torch.rand((BATCH_SIZE, 1, 1, 1)).repeat(1, C, H, W).to(device)
    interpolated_images = real * beta + fake.detach() * (1 - beta)
    interpolated_images.requires_grad_(True)

    # Calculate critic scores
    mixed_scores = critic(interpolated_images, alpha, train_step)


    gradient = torch.autograd.grad(
        inputs=interpolated_images,
        outputs=mixed_scores,
        grad_outputs=torch.ones_like(mixed_scores),
        create_graph=True,
        retain_graph=True,
    )[0]
    gradient = gradient.view(gradient.shape[0], -1)
    gradient_norm = gradient.norm(2, dim=1)
    gradient_penalty = torch.mean((gradient_norm - 1) ** 2)
    return gradient_penalty


def save_checkpoint(model, optimizer, filename="my_checkpoint.pth.tar"):
    print("=> Saving checkpoint")
    checkpoint = {
        "state_dict": model.state_dict(),
        "optimizer": optimizer.state_dict(),
    }
    torch.save(checkpoint, filename)


def load_checkpoint(checkpoint_file, model, optimizer, lr):
    print("=> Loading checkpoint")
    checkpoint = torch.load(checkpoint_file, map_location="cuda")
    model.load_state_dict(checkpoint["state_dict"])
    optimizer.load_state_dict(checkpoint["optimizer"])


    for param_group in optimizer.param_groups:
        param_group["lr"] = lr

def generate_examples(gen, steps, truncation=0.7, n=100):
     gen.eval()
    alpha = 1.0
    for i in range(n):
        with torch.no_grad():
            noise = torch.tensor(truncnorm.rvs(-truncation, truncation, size=(1, config.Z_DIM, 1, 1)), device=config.DEVICE, dtype=torch.
```

```
                float32)
            img = gen(noise, alpha, steps)
            save_image(img*0.5+0.5, f"saved_examples/img_{i}.png")

    gen.train()
```

5 Individual Contribution

Student Name	Paper Implemented
Sachin Prasanna	Denoising Diffusion Probabilistic Models
Anagha H C	Progressively Growing GANS
Abhayjit Singh Gulati	Generative Adversarial Nets

References

1. https://arxiv.org/pdf/2006.11239.pdf - Denoising Diffusion Probabilistic Models

2. https://arxiv.org/pdf/2102.09672.pdf - Improved Denoising Diffusion Probabilistic Models

3. https://arxiv.org/pdf/2105.05233.pdf - Diffusion Models Beat GANs on Image Synthesis

4. https://arxiv.org/pdf/1406.2661.pdf - Generative Adversarial Nets

5. https://arxiv.org/pdf/1710.10196.pdf - Progressive Growing of GANS for Improved Quality, Stability and VariationN