Lab 12: Implement a Deep Convolutional GAN to generate complex colour images

Aim:

To design and implement a Deep Convolutional GAN capable of generating complex colour images.

Description:

1. Generator (G):

Takes random noise (latent vector z) as input

Tries to produce realistic images similar to these in the training dataset.

2. Discriminator (D):

Receives both real and generated images

Tries to correctly distinguish between real and fake samples.

The Generator's objective is to fool the discriminator

$$\min_G \max_D V(D,G) = E_{x \sim p_{data}}[\log D(x)] + E_{z \sim p_z}[\log(1 - D(G(z)))]$$

where:

D(x): Probability that x is real

G(z): Generated image from noise z.

p_{data}: Distribution of real data

p_z: Distribution of latent noise.

Noise (100)

↓

Generator

Dense → Reshape → Conv 2D Transpose ×3

output (32× 32× 3) ←

↓

Fake image (32×32×3)

↓

Descriminator :

Conv 2D ×3 → Flatten → Dense → Sigmoid

↓

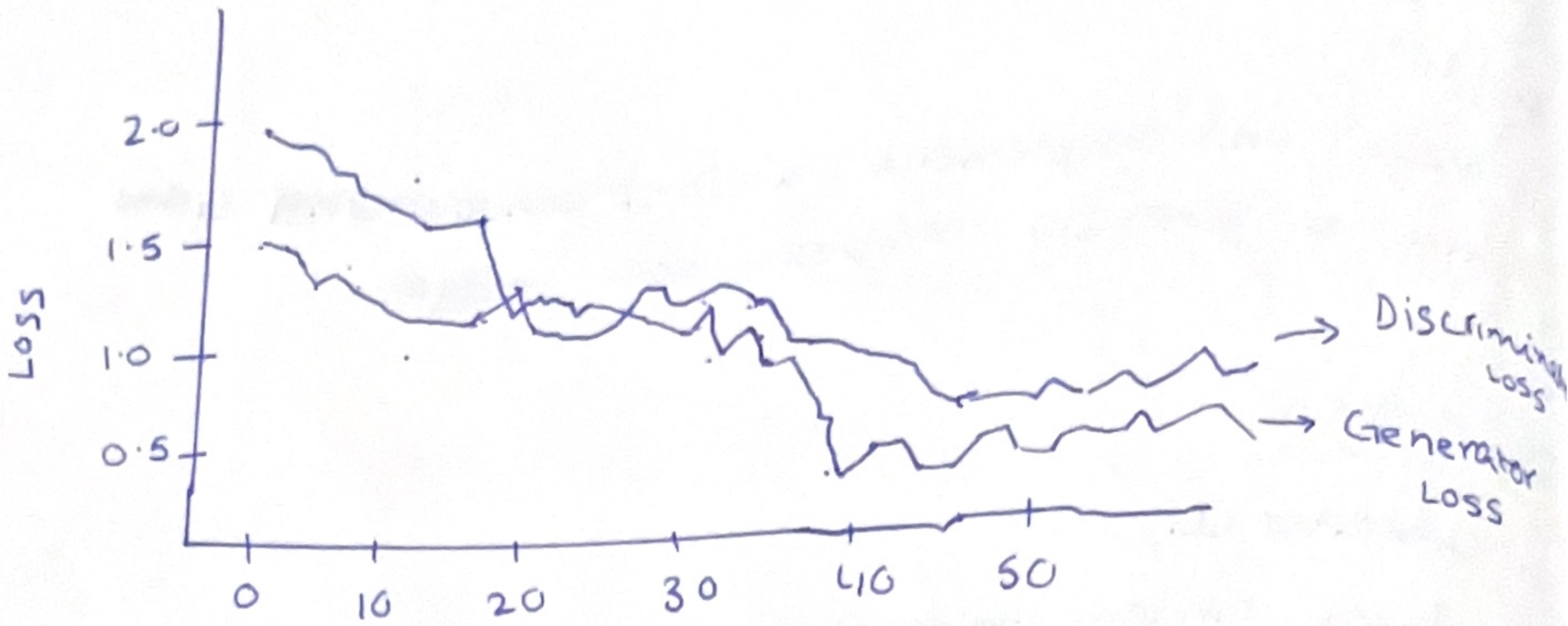
Output : Real / Fake

Adversarial Training

between Generator and Discriminator

Pseudocode:

1. Import PyTorch, torchvision, and etc.
2. load and preprocess dataset
3. Define Generator network
   Input: latent vector z (size 100)
   Layers: convTranspose2D + ReLU
4. Define Disclriminator network
   Input 3X64X64 image
5. Define loss function: Binary Cross Entropy (BCE).
6. Train
   For each epoch:
   a. Train Discriminator on real and fake images
   b. Train Generator to fool the discriminator
7. Plot loss curves for G and D.
8. Visualise generated images after training.



GAN Training LOSS

Epoch	G.LOSS	D.LOSS
1	2.10	1.80
10	1.20	1.16
20	0.75	0.65
30	0.45	0.55
40	0.32	0.47
50	0.22	0.40

```python
import os
import math
import random
from typing import Tuple
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, utils
import matplotlib.pyplot as plt

DATA_DIR = "./data"
RESULT_DIR = "./results"
os.makedirs(RESULT_DIR, exist_ok=True)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Data
IMAGE_SIZE = 32             # CIFAR-10 default size; for CelebA you might use 64
NC = 3                      # number of channels (3 = RGB)
BATCH_SIZE = 128
NUM_WORKERS = 4

# Model
NZ = 100                    # latent vector (z) size
NGF = 64                    # generator feature map size (base)
NDF = 64                    # discriminator feature map size (base)

# Training
NUM_EPOCHS = 50
LR = 2e-4
BETA1 = 0.5                 # Adam beta1
BETA2 = 0.999

# Misc
SEED = 999
PRINT_EVERY = 100
SAMPLE_FIXED_NOISE_COUNT = 64

random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)

transform = transforms.Compose([
    transforms.Resize(IMAGE_SIZE),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*NC, [0.5]*NC),  # (x - 0.5)/0.5 -> [-1,1]
])

train_dataset = datasets.CIFAR10(root=DATA_DIR, train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS, pin_memory=True)

def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
```

Run All | Restart | Interrupt | Clear All Outputs | Go To | Jupyter Variables | Outline | ...

Python 3.13.1

```
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

class Generator(nn.Module):
    def __init__(self, nz=NZ, ngf=NGF, nc=NC):
        super().__init__()
        self.net = nn.Sequential(
            # Input Z: (nz, 1, 1)
            nn.ConvTranspose2d(nz, ngf * 8, kernel_size=4, stride=1, padding=0, bias=False),  # -> (ngf*8) x 4 x 4
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 8, ngf * 4, kernel_size=4, stride=2, padding=1, bias=False),  # -> (ngf*4) x 8 x 8
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 4, ngf * 2, kernel_size=4, stride=2, padding=1, bias=False),  # -> (ngf*2) x 16 x 16
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 2, ngf, kernel_size=4, stride=2, padding=1, bias=False),      # -> ngf x 32 x 32
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf, nc, kernel_size=3, stride=1, padding=1, bias=False),          # -> nc x 32 x 32
            nn.Tanh()  # output in [-1,1]
        )

    def forward(self, z):
        z = z.view(z.size(0), z.size(1), 1, 1)
        return self.net(z)

class Discriminator(nn.Module):
    def __init__(self, nc=NC, ndf=NDF):
        super().__init__()
        # For 32x32 input, go down: 32 -> 16 -> 8 -> 4
        self.net = nn.Sequential(
            # input nc x 32 x 32
            nn.Conv2d(nc, ndf, kernel_size=4, stride=2, padding=1, bias=False),  # -> ndf x 16 x 16
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf, ndf * 2, kernel_size=4, stride=2, padding=1, bias=False),  # -> ndf*2 x 8 x 8
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 2, ndf * 4, kernel_size=4, stride=2, padding=1, bias=False),  # -> ndf*4 x 4 x 4
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 4, 1, kernel_size=4, stride=1, padding=0, bias=False),  # -> 1 x 1 x 1
            nn.Sigmoid()
        )
```

```python
netG = Generator().to(DEVICE)
netD = Discriminator().to(DEVICE)
netG.apply(weights_init)
netD.apply(weights_init)

criterion = nn.BCELoss()  # binary cross-entropy

optimizerD = optim.Adam(netD.parameters(), lr=LR, betas=(BETA1, BETA2))
optimizerG = optim.Adam(netG.parameters(), lr=LR, betas=(BETA1, BETA2))

fixed_noise = torch.randn(SAMPLE_FIXED_NOISE_COUNT, NZ, device=DEVICE)

# Create labels
real_label_value = 1.0
fake_label_value = 0.0

G_losses = []
D_losses = []
iters = 0

print("Starting Training on device:", DEVICE)
for epoch in range(1, NUM_EPOCHS + 1):
    for i, (data, _) in enumerate(train_loader, 0):

        netD.zero_grad()

        # Train with real batch
        real_images = data.to(DEVICE)
        b_size = real_images.size(0)
        label = torch.full((b_size,), real_label_value, device=DEVICE)

        output = netD(real_images)  # (B,)
        errD_real = criterion(output, label)
        errD_real.backward()
        D_x = output.mean().item()

        # Train with fake batch
        noise = torch.randn(b_size, NZ, device=DEVICE)
        fake_images = netG(noise)
        label.fill_(fake_label_value)

        output = netD(fake_images.detach())  # detach so G not updated on these gradients
        errD_fake = criterion(output, label)
        errD_fake.backward()
        D_G_z1 = output.mean().item()

        errD = errD_real + errD_fake
        optimizerD.step()

        netG.zero_grad()
        label.fill_(real_label_value)  # want generator to fool the discriminator
        output = netD(fake_images)     # now with gradients flowing to G
        errG = criterion(output, label)
        errG.backward()
        D_G_z2 = output.mean().item()
        optimizerG.step()

        # Save losses for plotting later
```

```
        optimizerD.step()

        netG.zero_grad()
        label.fill_(real_label_value)  # want generator to fool the discriminator
        output = netD(fake_images)     # now with gradients flowing to G
        errG = criterion(output, label)
        errG.backward()
        D_G_z2 = output.mean().item()
        optimizerG.step()

        # Save losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        if i % PRINT_EVERY == 0:
            print(f"Epoch [{epoch}/{NUM_EPOCHS}] Batch [{i}/{len(train_loader)}] "
                  f"Loss_D: {errD.item():.4f} Loss_G: {errG.item():.4f} D(X): {D_x:.4f} D(G(z)): {D_G_z1:.4f}/{D_G_z2:.4f}")

        # Save images at iteration boundaries: generate fixed_noise samples
        if (iters % 500 == 0) or (i == len(train_loader)-1):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            # denormalize from [-1,1] to [0,1]
            utils.save_image((fake + 1.0) / 2.0, os.path.join(RESULT_DIR, f"epoch_{epoch:03d}_iter_{iters:06d}.png"), nrow=8)

        iters += 1

    # Save checkpoint at end of each epoch
    torch.save({
        'epoch': epoch,
        'netG_state_dict': netG.state_dict(),
        'netD_state_dict': netD.state_dict(),
        'optimizerG_state_dict': optimizerG.state_dict(),
        'optimizerD_state_dict': optimizerD.state_dict(),
    }, os.path.join(RESULT_DIR, f"dcgan_checkpoint_epoch_{epoch:03d}.pth"))

with torch.no_grad():
    sample_z = torch.randn(64, NZ, device=DEVICE)
    fake_samples = netG(sample_z).detach().cpu()
    utils.save_image((fake_samples + 1.0) / 2.0, os.path.join(RESULT_DIR, "final_samples.png"), nrow=8)

# Plot losses
plt.figure(figsize=(8,5))
plt.plot(G_losses, label="G_loss")
plt.plot(D_losses, label="D_loss")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend()
plt.title("DCGAN Losses")
plt.grid(True)
plt.savefig(os.path.join(RESULT_DIR, "loss_curve.png"))
plt.close()

print("Training finished. Results saved to", RESULT_DIR)
```

File Edit Selection View Go Run ··· CODE

lab8.ipynb | lab11.ipynb | lab12.ipynb ● | lab14.ipynb ● | lab15.ipynb

.vscode > PPS > lab12.ipynb > import os

Generate + Code + Markdown | Interrupt Restart Clear All Outputs Go To | Jupyter Variables Outline ··· Python 3.13.1

```
                with torch.no_grad():
                    fake = netG(fixed_noise).detach().cpu()
                # denormalize from [-1,1] to [0,1]
                utils.save_image((fake + 1.0) / 2.0, os.path.join(RESULT_pps, f"epoch_{epoch:03d}_step_{steps:06d}.png"), nrow=8)

            steps += 1

        # Save checkpoint at end of each epoch
        torch.save({
            'epoch': epoch,
            'netG_state_dict': netG.state_dict(),
            'netD_state_dict': netD.state_dict(),
            'optimizerG_state_dict': optimizerG.state_dict(),
            'optimizerD_state_dict': optimizerD.state_dict(),
        }, os.path.join(RESULT_pps, f'dcgan_checkpoint_epoch_{epoch:03d}.pth'))

    with torch.no_grad():
        sample_z = torch.randn(64, NZ, device=DEVICE)
        fake_samples = netG(sample_z).detach().cpu()
        utils.save_image((fake_samples + 1.0) / 2.0, os.path.join(RESULT_pps, "final_samples.png"), nrow=8)

    # Plot losses
    plt.figure(figsize=(8,5))
    plt.plot(G_losses, label="G_loss")
    plt.plot(D_losses, label="D_loss")
    plt.xlabel("Iteration")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("DCGAN Losses")
    plt.grid(True)
    plt.savefig(os.path.join(RESULT_pps, "loss_curve.png"))
    plt.close()

    print("Training finished. Results saved to", RESULT_pps)
```

17m 18.0s Python

```
...  Starting Training on device: cpu
Epoch [1/50] Batch [0/391] Loss_D: 1.9106 Loss_G: 2.3241 D(x): 0.4081 D(G(z)): 0.5482/0.1183
Epoch [1/50] Batch [100/391] Loss_D: 0.4294 Loss_G: 5.1791 D(x): 0.8048 D(G(z)): 0.1317/0.0064
Epoch [1/50] Batch [200/391] Loss_D: 0.9949 Loss_G: 1.9705 D(x): 0.6952 D(G(z)): 0.4033/0.1732
Epoch [1/50] Batch [300/391] Loss_D: 0.3843 Loss_G: 3.3106 D(x): 0.8460 D(G(z)): 0.1699/0.0440
Epoch [2/50] Batch [0/391] Loss_D: 0.4982 Loss_G: 3.9430 D(x): 0.8053 D(G(z)): 0.1651/0.0287
Epoch [2/50] Batch [100/391] Loss_D: 0.9398 Loss_G: 2.5019 D(x): 0.6999 D(G(z)): 0.3868/0.1031
Epoch [2/50] Batch [200/391] Loss_D: 0.7287 Loss_G: 2.4360 D(x): 0.7236 D(G(z)): 0.2433/0.1527
Epoch [2/50] Batch [300/391] Loss_D: 0.8719 Loss_G: 2.1861 D(x): 0.6271 D(G(z)): 0.2522/0.1497
Epoch [3/50] Batch [0/391] Loss_D: 0.5973 Loss_G: 3.0438 D(x): 0.7496 D(G(z)): 0.2037/0.0628
Epoch [3/50] Batch [100/391] Loss_D: 0.4737 Loss_G: 2.9807 D(x): 0.7592 D(G(z)): 0.1260/0.0681
```

⊗0 ⚠4 Spaces:4 CRLF {} Cell 1 of 1 Go Live

27°C Partly cloudy | Search | ENG IN | 04-11-2025