

## # Experiment No.3 — Generative Adversarial Networks (GANs)

### **Group members :**

1. Kiran Shinde (202201040091)
2. Sanika (202201040092)
3. Shivanjali Jagtap (202201040070)
4. Yash Nair (202201040075)

**Batch:** GenAI-2

### **Title -**

Implementation and Comparative Analysis of DCGAN and cGAN for Image Generation

### **Experiment Details -**

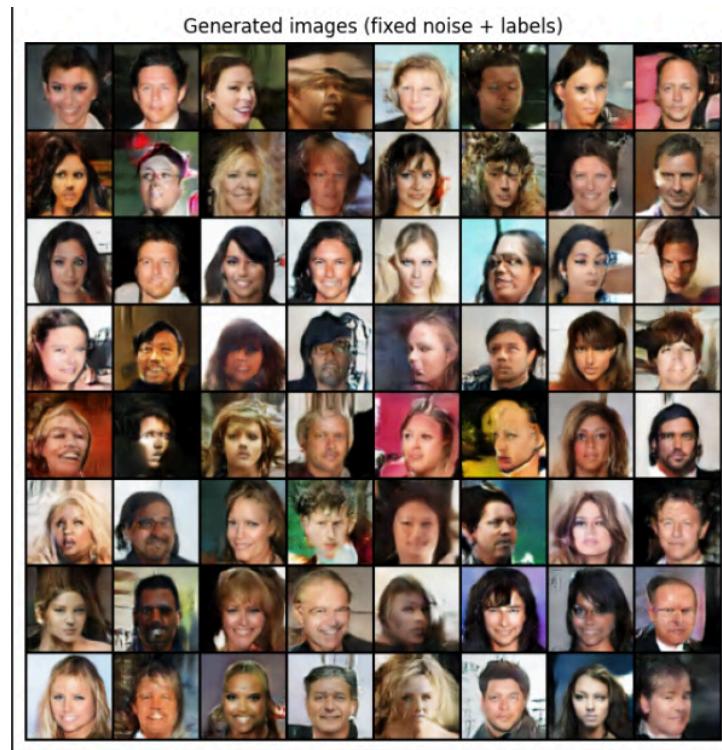
Implement a DCGAN and/or cGAN on a chosen dataset to generate original image data, providing a comprehensive summary of the GAN architectures, training processes, and generated results. Conduct a comparative analysis of the models, evaluating their effectiveness in image quality and consistency. Conclude with a presentation that includes a case study analysis, discussing ethical considerations and challenges in real-world GAN applications.

### **Colab Link -**

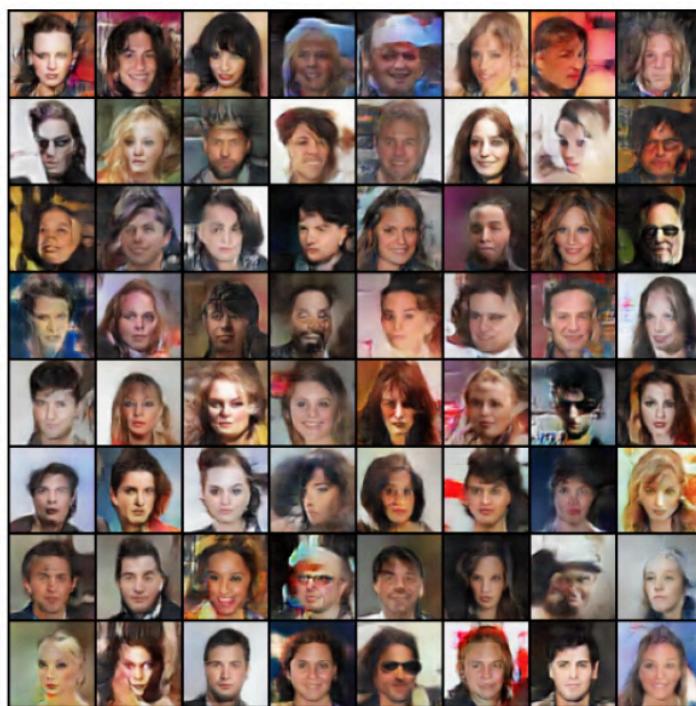
1. <https://colab.research.google.com/drive/1BIk7Tx4H8EyMZ0-V8LQnR0lNDMaz5wjx?usp=sharing>

### **Output -**

## CGAN -



## DCGAN -



## Experiment No.3 — Generative Adversarial Networks (GANs)

### Group members:

1. Kiran Shinde (202201040091)
2. Sanika (202201040092)
3. Shivanjali Jagtap (202201040070)
4. Yash Nair (202201040075)

**Batch:** GenAI-2

### Title

**Implementation and Comparative Analysis of DCGAN and cGAN for Image Generation**

### Experiment Details -

Implement a DCGAN and/or cGAN on a chosen dataset to generate original image data, providing a comprehensive summary of the GAN architectures, training processes, and generated results. Conduct a comparative analysis of the models, evaluating their effectiveness in image quality and consistency. Conclude with a presentation that includes a case study analysis, discussing ethical considerations and challenges in real-world GAN applications.

### ▼ CGAN

```

1 # 1. Install gdown
2 !pip install gdown --quiet
3
4 # 2. File ID from your Drive share link
5 file_id = "1jFjvLvjf8CJ0sMJeWtmXDnm16Fo9KFsy"
6
7 # 3. Set paths
8 zip_path = "/content/archive_4.zip"
9 extract_to = "/content/celeba"
10
11 # 4. Download using gdown
12 !gdown --id {file_id} -O {zip_path}
13
14 # 5. Create extraction folder
15 !mkdir -p {extract_to}
16
17 # 6. Unzip into folder
18 !unzip -q {zip_path} -d {extract_to}
19

```

```

/usr/local/lib/python3.12/dist-packages/gdown/__main__.py:140: FutureWarning: Option `--id` was deprecated in version 4.3.1 and
warnings.warn(
Downloading...
From (original): https://drive.google.com/uc?id=1jFjvLvjf8CJ0sMJeWtmXDnm16Fo9KFsy
From (redirected): https://drive.google.com/uc?id=1jFjvLvjf8CJ0sMJeWtmXDnm16Fo9KFsy&confirm=t&uuid=ede9d394-7e4f-4a2c-b179-0943
To: /content/archive_4.zip
100% 1.43G/1.43G [00:19<00:00, 74.4MB/s]

```

```

1 %matplotlib inline
2 import os
3 import random
4 from pathlib import Path
5 import math
6 import json
7 import time
8 from typing import Optional
9
10 import numpy as np
11 import pandas as pd

```

```

12 from PIL import Image
13 import matplotlib.pyplot as plt
14
15 import torch
16 import torch.nn as nn
17 import torch.optim as optim
18 from torch.utils.data import Dataset, DataLoader
19 import torchvision.transforms as transforms
20 import torchvision.utils as vutils
21 import torch.nn.functional as F

```

```

1 # -----
2 # User-editable parameters
3 #
4 dataroot = Path("data/celeba")      # root with 'img_align_celeba' and CSVs
5 image_folder = dataroot / "img_align_celeba" / "img_align_celeba"
6
7 # Training hyperparams
8 workers = 4
9 batch_size = 128
10 image_size = 64
11 nc = 3                         # number of image channels
12 nz = 100                        # noise vector size
13 label_dim = 1                   # if binary attribute: 1. For multi-class: set to number of cl
14 ngf = 64
15 ndf = 64
16 num_epochs = 10
17 lr = 0.0002
18 beta1 = 0.5
19 ngpu = 1
20
21 save_dir = Path("cgan_outputs")
22 save_dir.mkdir(exist_ok=True, parents=True)
23
24 # Random seed for reproducibility
25 manualSeed = 999
26 random.seed(manualSeed)
27 torch.manual_seed(manualSeed)
28 torch.use_deterministic_algorithms(True)
29
30 device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
31 print("Device:", device)

```

Device: cuda:0

```

1 from pathlib import Path
2
3 dataroot = Path("/content/celeba")
4
5 image_folder = dataroot / "img_align_celeba" / "img_align_celeba"
6
7 # CSV files
8 attr_csv = dataroot / "list_attr_celeba.csv"
9 landmarks_csv = dataroot / "list_landmarks_align_celeba.csv"
10 bbox_csv = dataroot / "list_bbox_celeba.csv"
11 partition_csv = dataroot / "list_eval_partition.csv"
12
13 use_celeba_attrs = False
14 if image_folder.exists() and attr_csv.exists() and landmarks_csv.exists() and bbox_c
15     print(" Found CelebA images + CSVs -> using CelebA attributes.")
16     use_celeba_attrs = True
17 else:

```

```

18     print(" CelebA CSVs not fully present. Falling back to ImageFolder (images only)
19

Found CelebA images + CSVs -> using CelebA attributes.

1 # -----
2 # Load CSVs / build master dataframe (if CelebA)
3 #
4 if use_celebaAttrs:
5     attrs_df = pd.read_csv(attr_csv, index_col=0)
6     land_df = pd.read_csv(landmarks_csv, index_col=0)
7     bbox_df = pd.read_csv(bbox_csv, index_col=0)
8     part_df = pd.read_csv(partition_csv, index_col=0, header=None, names=["partition"])
9     # join
10    master_df = attrs_df.join([land_df, bbox_df, part_df], how="inner").reset_index()
11    master_df["filepath"] = master_df["filename"].apply(lambda fn: str(image_folder /
12        # filter to existing files
13        master_df = master_df[master_df["filepath"].apply(lambda p: Path(p).exists())].r
14        print("master_df shape:", master_df.shape)
15        # convert attributes -1/1 -> 0/1
16        attr_cols = attrs_df.columns.tolist()
17        master_df[attr_cols] = (master_df[attr_cols] == 1).astype(int)
18        # choose a conditioning attribute (binary). Default to 'Male' if present, else f
19        if "Male" in attr_cols:
20            chosen_attr = "Male"
21        else:
22            chosen_attr = attr_cols[0]
23        print("Conditioning attribute:", chosen_attr)
24        master_df["label"] = master_df[chosen_attr].astype(int)
25        # partition map: 0 train, 1 val, 2 test
26        part_map = {0:"train", 1:"val", 2:"test"}
27        master_df["partition_name"] = master_df["partition"].map(part_map)
28 else:
29     # fallback: create a very simple dataframe from ImageFolder
30     # assume structure: data/celeba/class_x/xxx.jpg (if classes present)
31     image_paths = sorted([str(p) for p in image_folder.glob("**/*") if p.suffix.lower() in
32     if len(image_paths) == 0:
33         raise FileNotFoundError(f"No images found in {image_folder}. Place images or
34     master_df = pd.DataFrame({"filename": [Path(p).name for p in image_paths],
35                               "filepath": image_paths})
36     # dummy labels (all zeros) unless folder classes exist - keep label as 0
37     master_df["label"] = 0
38     master_df["partition_name"] = "train"
39     print("ImageFolder fallback with", len(master_df), "images. Using dummy labels.")

master_df shape: (202599, 57)
Conditioning attribute: Male

```

```

1 # Diagnostic: why is train set empty?
2 from pathlib import Path
3
4 # Inspect master_df basic properties
5 print("master_df shape:", master_df.shape)
6 print("Columns:", master_df.columns.tolist()[:30])
7 print("Does 'partition_name' exist?:", "partition_name" in master_df.columns)
8 print("Sample rows (first 5):")
9 display(master_df.head())
10
11 # Count partition_name unique values and counts (if present)
12 if "partition_name" in master_df.columns:
13     print("\nPartition value counts:")
14     print(master_df["partition_name"].value_counts(dropna=False))
15

```

```

16 # Check how many filepaths exist
17 if "filepath" in master_df.columns:
18     exists_counts = master_df["filepath"].apply(lambda p: Path(p).exists()).value_counts()
19     print("\nFile existence counts:", exists_counts.to_dict())
20     # Show a few missing file examples (if any)
21     missing = master_df[~master_df["filepath"].apply(lambda p: Path(p).exists())]
22     if len(missing) > 0:
23         print("Examples of missing files (first 5):")
24         display(missing.head(5)[["filename", "filepath", "partition_name"]])
25
26 # Check chosen attribute presence
27 chosen_attr = "label" # or whichever you set earlier
28 print("\nIs chosen attr column present?:", chosen_attr in master_df.columns)
29 if chosen_attr in master_df.columns:
30     print("Label value counts (0/1 or classes):")
31     print(master_df[chosen_attr].value_counts(dropna=False).head(20))
32

```

master\_df shape: (202599, 59)  
 Columns: ['filename', '5\_o\_Clock\_Shadow', 'Arched\_Eyebrows', 'Attractive', 'Bags\_Under\_Eyes', 'Bald', 'Bangs', 'Big\_Lips', 'Big\_Nose', 'Black\_Hair', ...]  
 Does 'partition\_name' exist?: True  
 Sample rows (first 5):

	filename	5_o_Clock_Shadow	Arched_Eyebrows	Attractive	Bags_Under_Eyes	Bald	Bangs	Big_Lips	Big_Nose	Black_Hair	...
0	000001.jpg	0	1	1	0	0	0	0	0	0	...
1	000002.jpg	0	0	0	1	0	0	0	1	0	...
2	000003.jpg	0	0	0	0	0	0	1	0	0	...
3	000004.jpg	0	0	1	0	0	0	0	0	0	...
4	000005.jpg	0	1	1	0	0	0	1	0	0	...

5 rows × 59 columns

Partition value counts:

partition_name	Nan	202599
Name: count, dtype: int64		

File existence counts: {True: 202599}

Is chosen attr column present?: True

Label value counts (0/1 or classes):

label	0	118165
	1	84434
Name: count, dtype: int64		

```

1 # -----
2 # Build master_df (attributes & filepaths) and fix partition_name
3 # -----
4 if use_celebaAttrs:
5     attrs_df = pd.read_csv(attr_csv, index_col=0)
6     land_df = pd.read_csv(landmarks_csv, index_col=0)
7     bbox_df = pd.read_csv(bbox_csv, index_col=0)
8     part_df = pd.read_csv(partition_csv, index_col=0, header=None, names=["partition"])
9     master_df = attrs_df.join([land_df, bbox_df, part_df], how="inner").reset_index()
10    # build filepath using detected image_folder
11    master_df["filepath"] = master_df["filename"].apply(lambda fn: str(image_folder / fn))
12    # filter only existing files
13    master_df = master_df[master_df["filepath"].apply(lambda p: Path(p).exists())]
14    print("master_df shape after file-exists filter:", master_df.shape)
15    # convert attributes to 0/1
16    attr_cols = attrs_df.columns.tolist()
17    master_df[attr_cols] = (master_df[attr_cols] == 1).astype(int)
18    # choose conditioning attribute (default 'Male' if present)
19    if "Male" in attr_cols:
20        chosen_attr = "Male"
21    else:

```

```

22     chosen_attr = attr_cols[0]
23     master_df["label"] = master_df[chosen_attr].astype(int)
24     # Rebuild partition_name robustly from numeric partition column (some CSVs cause
25     def safe_int(x):
26         try:
27             return int(float(x))
28         except Exception:
29             return None
30     master_df["partition_int"] = master_df["partition"].apply(safe_int)
31     master_df["partition_name"] = master_df["partition_int"].map({0:"train", 1:"val"})
32     # If any partition_name still NaN, try to infer from original part values (print
33     if master_df["partition_name"].isna().all():
34         print("partition_name still NaN for all rows. Unique raw partition values (f"
35             print(master_df["partition"].unique()[:50]))
36     else:
37         print("Partition counts:\n", master_df["partition_name"].value_counts(dropna=True))
38     print(f"Conditioning attribute: {chosen_attr} (label distribution: {master_df['label'].value_counts()})")
39 else:
40     # fallback: find images under image_folder (any depth)
41     image_paths = sorted([str(p) for p in image_folder.glob("**/*") if p.suffix.lower() in allowed_extensions])
42     if len(image_paths) == 0:
43         raise FileNotFoundError(f"No images found under {image_folder}. Check datarc")
44     master_df = pd.DataFrame({"filename": [Path(p).name for p in image_paths],
45                               "filepath": image_paths})
46     master_df["label"] = 0
47     master_df["partition_name"] = "train"
48     print("ImageFolder fallback: found", len(master_df), "images. Using dummy labels")
49
50 # quick sanity
51 print("master_df sample:")
52 display(master_df.head())

```

master\_df shape after file-exists filter: (202599, 57)

Partition counts:

partition_name	count
train	162770
test	19962
val	19867

Name: count, dtype: int64

Conditioning attribute: Male (label distribution: {0: 118165, 1: 84434})

master\_df sample:

	filename	5_o_Clock_Shadow	Arched_Eyebrows	Attractive	Bags_Under_Eyes	Bald	Bangs	Big_Lips	Big_Nose	Black_Hair	Blond_Hair	Blurry	Brown_Hair	Dark_Hair	Double_Chin	Goatee	Hesitant	Male	Not_Posing	Oily	Smiling	Smooth_Skin	Wavy_Hair	Well_Groomed	Young
0	000001.jpg	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	000002.jpg	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	000003.jpg	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	000004.jpg	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	000005.jpg	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

5 rows × 60 columns

```

1 # -----
2 # Robust Dataset class (falls back to full dataset if partition filter empty)
3 # -----
4 class CelebADatasetRobust(Dataset):
5     def __init__(self, df, partition="train", image_size=64, transform=None,
6                  chosen_attr_col="label", use_partition=True, allow_fallback=True):
7         self.orig_df = df.copy()
8         self.chosen_attr_col = chosen_attr_col
9         self.transform = transform or transforms.Compose([
10             transforms.Resize(image_size),
11             transforms.CenterCrop(image_size),
12             transforms.ToTensor(),
13             transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
14         ])

```

```
15      # filter
16      if use_partition and "partition_name" in df.columns:
17          filtered = df[df["partition_name"] == partition].reset_index(drop=True)
18          if len(filtered) == 0:
19              print(f"WARNING: partition filter returned 0 rows for partition='{partition}'")
20              if allow_fallback:
21                  print("Falling back to entire dataset (use_partition=False to disable)")
22                  self.df = df.reset_index(drop=True)
23              else:
24                  self.df = filtered
25          else:
26              self.df = filtered
27      else:
28          # try numeric partition_int if partition is int
29          if use_partition and "partition_int" in df.columns and isinstance(partition, int):
30              filtered = df[df["partition_int"] == partition].reset_index(drop=True)
31              if len(filtered) == 0:
32                  print(f"WARNING: numeric partition filter produced 0 rows for partition={partition}")
33                  self.df = df.reset_index(drop=True)
34              else:
35                  self.df = filtered
36          else:
37              self.df = df.reset_index(drop=True)
38
39      if len(self.df) == 0:
40          print("ERROR: dataset is empty after filtering and fallback.")
41      else:
42          print(f"Dataset prepared with {len(self.df)} samples (partition_filter={partition})")
43
44  def __len__(self):
45      return len(self.df)
46
47  def __getitem__(self, idx):
48      row = self.df.iloc[idx]
49      img_path = row["filepath"]
50      try:
51          img = Image.open(img_path).convert("RGB")
52      except Exception as e:
53          raise FileNotFoundError(f"Cannot open image at {img_path} (index {idx}): {e}")
54      img_t = self.transform(img)
55      lab = 0
56      if self.chosen_attr_col in row.index:
57          try:
58              lab = int(row[self.chosen_attr_col])
59          except Exception:
60              lab = 0
61      label = torch.tensor(lab, dtype=torch.long)
62      return img_t, label
63
64 # create dataset + dataloader (use partition filter but fallback enabled)
65 train_ds = CelebADatasetRobust(master_df, partition="train", image_size=image_size,
66                                 chosen_attr_col="label", use_partition=True, allow_fallback=True)
67 print("Train dataset size:", len(train_ds))
68 if len(train_ds) == 0:
69     # force ignore partitions
70     train_ds = CelebADatasetRobust(master_df, partition="train", image_size=image_size,
71                                     chosen_attr_col="label", use_partition=False, all_partitions=True)
72     print("Train dataset size after forcing no partition:", len(train_ds))
73
74 train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,
75                           num_workers=min(workers, 4), pin_memory=True, drop_last=True)
76 print("DataLoader created; approx batches per epoch:", math.ceil(len(train_ds)/batch_size))
```

```
Dataset prepared with 162770 samples (partition_filter=True).
Train dataset size: 162770
DataLoader created; approx batches per epoch: 1272
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:627: UserWarning: This DataLoader will create 4 worker p
warnings.warn(
```

```
1 # -----
2 # Helper: weight init (DCGAN style)
3 # -----
4 def weights_init(m):
5     classname = m.__class__.__name__
6     if classname.find('Conv') != -1:
7         nn.init.normal_(m.weight.data, 0.0, 0.02)
8     elif classname.find('BatchNorm') != -1:
9         nn.init.normal_(m.weight.data, 1.0, 0.02)
10    nn.init.constant_(m.bias.data, 0)

1 # -----
2 # Generator & Discriminator (same as your CGAN)
3 # -----
4 class Generator(nn.Module):
5     def __init__(self, ngpu, nz, label_dim, ngf, nc):
6         super(Generator, self).__init__()
7         self.ngpu = ngpu
8         self.input_dim = nz + label_dim
9         self.main = nn.Sequential(
10             nn.ConvTranspose2d(self.input_dim, ngf*8, 4, 1, 0, bias=False),
11             nn.BatchNorm2d(ngf*8),
12             nn.ReLU(True),
13
14             nn.ConvTranspose2d(ngf*8, ngf*4, 4, 2, 1, bias=False),
15             nn.BatchNorm2d(ngf*4),
16             nn.ReLU(True),
17
18             nn.ConvTranspose2d(ngf*4, ngf*2, 4, 2, 1, bias=False),
19             nn.BatchNorm2d(ngf*2),
20             nn.ReLU(True),
21
22             nn.ConvTranspose2d(ngf*2, ngf, 4, 2, 1, bias=False),
23             nn.BatchNorm2d(ngf),
24             nn.ReLU(True),
25
26             nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
27             nn.Tanh()
28         )
29
30     def forward(self, noise, labels):
31         if noise.dim()==4:
32             noise = noise.view(noise.size(0), -1)
33         if labels.dim()==1 and label_dim==1:
34             lab = labels.float().unsqueeze(1)
35         elif labels.dim()==1 and label_dim>1:
36             lab = F.one_hot(labels, num_classes=label_dim).float()
37         else:
38             lab = labels.float()
39         x = torch.cat([noise, lab], dim=1)
40         x = x.unsqueeze(2).unsqueeze(3)
41         return self.main(x)
42
43 class Discriminator(nn.Module):
44     def __init__(self, ngpu, nc, ndf, label_dim):
45         super(Discriminator, self).__init__()
46         self.ngpu = ngpu
```

```

47         self.label_dim = label_dim
48         in_channels = nc + label_dim
49         self.main = nn.Sequential(
50             nn.Conv2d(in_channels, ndf, 4, 2, 1, bias=False),
51             nn.LeakyReLU(0.2, inplace=True),
52
53             nn.Conv2d(ndf, ndf*2, 4, 2, 1, bias=False),
54             nn.BatchNorm2d(ndf*2),
55             nn.LeakyReLU(0.2, inplace=True),
56
57             nn.Conv2d(ndf*2, ndf*4, 4, 2, 1, bias=False),
58             nn.BatchNorm2d(ndf*4),
59             nn.LeakyReLU(0.2, inplace=True),
60
61             nn.Conv2d(ndf*4, ndf*8, 4, 2, 1, bias=False),
62             nn.BatchNorm2d(ndf*8),
63             nn.LeakyReLU(0.2, inplace=True),
64
65             nn.Conv2d(ndf*8, 1, 4, 1, 0, bias=False),
66             nn.Sigmoid()
67     )
68     def forward(self, img, labels):
69         B,C,H,W = img.size()
70         if labels.dim()==1 and self.label_dim==1:
71             lab = labels.float().unsqueeze(1).unsqueeze(2).unsqueeze(3)
72             lab_map = lab.repeat(1,1,H,W)
73         elif labels.dim()==1 and self.label_dim>1:
74             lab = F.one_hot(labels, num_classes=self.label_dim).float()
75             lab_map = lab.unsqueeze(2).unsqueeze(3).repeat(1,1,H,W)
76         else:
77             lab_map = labels.unsqueeze(2).unsqueeze(3).repeat(1,1,H,W)
78         x = torch.cat([img, lab_map.to(img.device)], dim=1)
79         return self.main(x).view(-1)

```

```

1 # -----
2 # Instantiate nets, init weights
3 #
4 netG = Generator(ngpu, nz, label_dim, ngf, nc).to(device)
5 netD = Discriminator(ngpu, nc, ndf, label_dim).to(device)
6 if (device.type == 'cuda') and (ngpu > 1):
7     netG = nn.DataParallel(netG, list(range(ngpu)))
8     netD = nn.DataParallel(netD, list(range(ngpu)))
9 netG.apply(weights_init)
10 netD.apply(weights_init)
11
12 print(netG)
13 print(netD)

```

```

Generator(
  (main): Sequential(
    (0): ConvTranspose2d(101, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(4, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)

```

```

(1): LeakyReLU(negative_slope=0.2, inplace=True)
(2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(4): LeakyReLU(negative_slope=0.2, inplace=True)
(5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(7): LeakyReLU(negative_slope=0.2, inplace=True)
(8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(10): LeakyReLU(negative_slope=0.2, inplace=True)
(11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
(12): Sigmoid()
)

```

```

1 # -----
2 # Loss, optimizers, fixed noise/labels for monitoring
3 # -----
4 criterion = nn.BCELoss()
5 fixed_noise = torch.randn(64, nz, device=device)
6 if label_dim == 1:
7     fixed_labels = torch.tensor([i%2 for i in range(64)], dtype=torch.long, device=c
8 else:
9     fixed_labels = torch.tensor([i%label_dim for i in range(64)], dtype=torch.long,
10 real_label = 1.0
11 fake_label = 0.0
12 optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
13 optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
14
15 G_losses = []
16 D_losses = []
17 img_list = []
18 iters = 0
19

```

```

1 # -----
2 # Training loop
3 # -----
4 print("Starting Training Loop...")
5 start_time = time.time()
6 for epoch in range(num_epochs):
7     epoch_start = time.time()
8     for i, (real_images, labels) in enumerate(train_loader):
9         b_size = real_images.size(0)
10        real_images = real_images.to(device)
11        labels = labels.to(device)
12
13        # (1) Update D
14        netD.zero_grad()
15        label_tensor = torch.full((b_size,), real_label, dtype=torch.float, device=c
16        output = netD(real_images, labels)
17        errD_real = criterion(output, label_tensor)
18        errD_real.backward()
19        D_x = output.mean().item()
20
21        noise = torch.randn(b_size, nz, device=device)
22        gen_labels = labels
23        fake = netG(noise, gen_labels)
24        label_tensor.fill_(fake_label)
25        output = netD(fake.detach(), gen_labels)
26        errD_fake = criterion(output, label_tensor)
27        errD_fake.backward()
28        D_G_z1 = output.mean().item()
29
30        errD = errD_real + errD_fake
31        optimizerD.step()

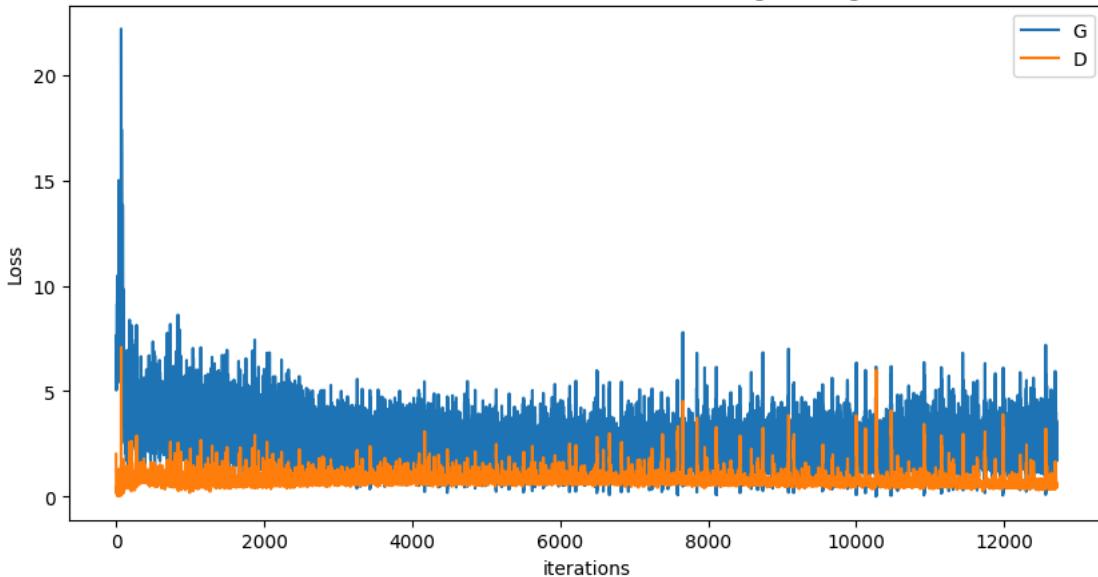
```

```
32
33     # (2) Update G
34     netG.zero_grad()
35     label_tensor.fill_(real_label)
36     output = netD(fake, gen_labels)
37     errG = criterion(output, label_tensor)
38     errG.backward()
39     D_G_z2 = output.mean().item()
40     optimizerG.step()
41
42     if i % 50 == 0:
43         print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G): %.4f
44             % (epoch, num_epochs, i, len(train_loader),
45             errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
46
47     G_losses.append(errG.item())
48     D_losses.append(errD.item())
49
50     if (iters % 500 == 0) or (epoch==num_epochs-1 and i==len(train_loader)-1):
51         with torch.no_grad():
52             fake_fixed = netG(fixed_noise, fixed_labels).detach().cpu()
53             img_list.append(vutils.make_grid(fake_fixed, padding=2, normalize=True))
54             vutils.save_image(fake_fixed, save_dir / f"fake_epoch{epoch}_iter{iters}")
55         iters += 1
56
57     epoch_time = time.time() - epoch_start
58     print(f"Epoch {epoch} finished in {epoch_time:.1f}s")
59     torch.save(netG.state_dict(), save_dir / f"netG_epoch{epoch}.pth")
60     torch.save(netD.state_dict(), save_dir / f"netD_epoch{epoch}.pth")
61
62 total_time = time.time() - start_time
63 print(f"Training finished in {total_time/60:.2f} min. Models & images saved to {save
```

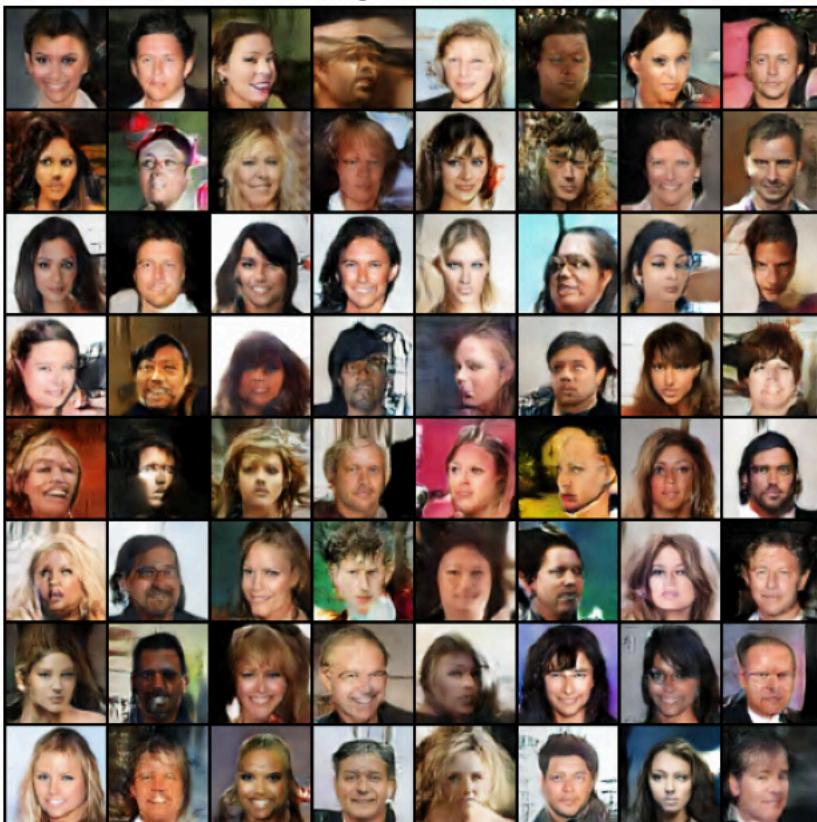
```
[9/10][0/1271] Loss_D: 2.2136 Loss_G: 0.4380 D(x): 0.1596 D(G): 0.0045/0.7053
[9/10][50/1271] Loss_D: 0.5848 Loss_G: 3.5796 D(x): 0.8928 D(G): 0.3359/0.0379
[9/10][100/1271] Loss_D: 0.4392 Loss_G: 2.6916 D(x): 0.8487 D(G): 0.2158/0.0876
[9/10][150/1271] Loss_D: 1.3206 Loss_G: 0.5160 D(x): 0.3419 D(G): 0.0370/0.6497
[9/10][200/1271] Loss_D: 0.8631 Loss_G: 2.1162 D(x): 0.6038 D(G): 0.1952/0.1602
[9/10][250/1271] Loss_D: 0.4703 Loss_G: 2.9048 D(x): 0.8730 D(G): 0.2667/0.0706
[9/10][300/1271] Loss_D: 2.4832 Loss_G: 6.3205 D(x): 0.9805 D(G): 0.8754/0.0047
[9/10][350/1271] Loss_D: 0.4665 Loss_G: 2.7251 D(x): 0.8059 D(G): 0.1929/0.0858
[9/10][400/1271] Loss_D: 0.4631 Loss_G: 2.3153 D(x): 0.7449 D(G): 0.1241/0.1265
[9/10][450/1271] Loss_D: 0.7089 Loss_G: 2.0696 D(x): 0.6667 D(G): 0.2007/0.1605
[9/10][500/1271] Loss_D: 0.6114 Loss_G: 4.2218 D(x): 0.9275 D(G): 0.3877/0.0196
[9/10][550/1271] Loss_D: 1.1620 Loss_G: 2.5808 D(x): 0.7686 D(G): 0.4960/0.1156
[9/10][600/1271] Loss_D: 0.4865 Loss_G: 1.7689 D(x): 0.7659 D(G): 0.1638/0.2148
[9/10][650/1271] Loss_D: 0.4823 Loss_G: 2.0401 D(x): 0.7333 D(G): 0.1227/0.1628
[9/10][700/1271] Loss_D: 0.7308 Loss_G: 3.6512 D(x): 0.9023 D(G): 0.4154/0.0378
```

```
1 # -----
2 # Plot losses & final visuals
3 # -----
4 plt.figure(figsize=(10,5))
5 plt.title("Generator and Discriminator Loss During Training")
6 plt.plot(G_losses, label="G")
7 plt.plot(D_losses, label="D")
8 plt.xlabel("iterations")
9 plt.ylabel("Loss")
10 plt.legend()
11 plt.show()
12
13 if len(img_list) > 0:
14     plt.figure(figsize=(8,8))
15     plt.axis("off")
16     plt.title("Generated images (fixed noise + labels)")
17     plt.imshow(np.transpose(img_list[-1], (1,2,0)))
18     plt.show()
19
```

## Generator and Discriminator Loss During Training



Generated images (fixed noise + labels)



```

1 # compare real vs fake
2 real_batch = next(iter(train_loader))[0][:64].to(device)
3 with torch.no_grad():
4     fake_batch = netG(fixed_noise.to(device), fixed_labels.to(device)).detach().cpu()
5
6 plt.figure(figsize=(14,7))
7 plt.subplot(1,2,1)
8 plt.axis("off")
9 plt.title("Real Images")
10 plt.imshow(np.transpose(vutils.make_grid(real_batch.cpu(), padding=2, normalize=True
11 plt.subplot(1,2,2)
12 plt.axis("off")
13 plt.title("Fake Images")
14 plt.imshow(np.transpose(vutils.make_grid(fake_batch, padding=2, normalize=True), (1,
15 plt.show()

```



```

1 # Save final state_dicts (lightweight, most common)
2 torch.save(netG.state_dict(), save_dir / "netG_final.pth")
3 torch.save(netD.state_dict(), save_dir / "netD_final.pth")
4
5 # Save optimizer states + epoch + loss metrics in a checkpoint (recommended)
6 checkpoint = {
7     "epoch": num_epochs,
8     "netG_state": netG.state_dict(),
9     "netD_state": netD.state_dict(),
10    "optimG_state": optimizerG.state_dict(),
11    "optimD_state": optimizerD.state_dict(),
12    "G_losses": G_losses,
13    "D_losses": D_losses,
14    "manualSeed": manualSeed,
15 }
16 torch.save(checkpoint, save_dir / "cgan_checkpoint_final.pth")
17
18 print("Saved netG_final.pth, netD_final.pth and cgan_checkpoint_final.pth to", save_
19

```

Saved netG\_final.pth, netD\_final.pth and cgan\_checkpoint\_final.pth to cgan\_outputs

## ▼ DCGAN

```

1 %%matplotlib inline
2 import argparse
3 import os
4 import random
5 import torch
6 import torch.nn as nn
7 import torch.nn.parallel
8 import torch.optim as optim
9 import torch.utils.data
10 import torchvision.datasets as dset
11 import torchvision.transforms as transforms
12 import torchvision.utils as vutils
13 import numpy as np
14 import matplotlib.pyplot as plt

```

```
15 import matplotlib.animation as animation
16 from IPython.display import HTML
17
18 # Set random seed for reproducibility
19 manualSeed = 999
20 #manualSeed = random.randint(1, 10000) # use if you want new results
21 print("Random Seed: ", manualSeed)
22 random.seed(manualSeed)
23 torch.manual_seed(manualSeed)
24 torch.use_deterministic_algorithms(True) # Needed for reproducible results
```

Random Seed: 999

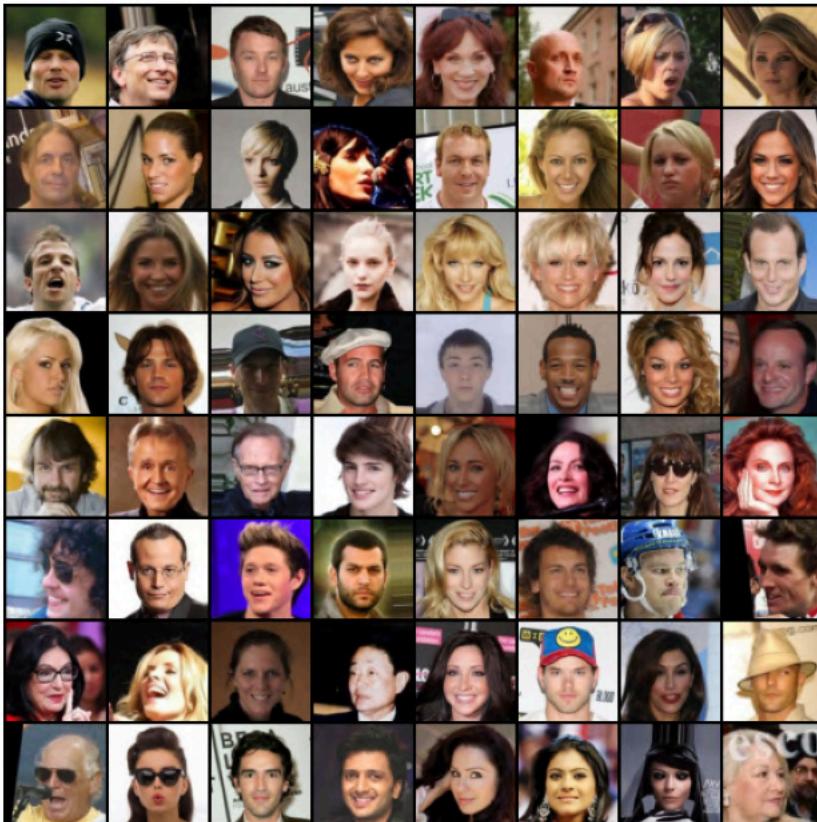
```
1 # Root directory for dataset
2 #
3 dataroot = Path("data/celeba")
4
5 # Number of workers for dataloader
6 workers = 2
7
8 # Batch size during training
9 batch_size = 128
10
11 # Spatial size of training images. All images will be resized to this
12 #   size using a transformer.
13 image_size = 64
14
15 # Number of channels in the training images. For color images this is 3
16 nc = 3
17
18 # Size of z latent vector (i.e. size of generator input)
19 nz = 100
20
21 # Size of feature maps in generator
22 ngf = 64
23
24 # Size of feature maps in discriminator
25 ndf = 64
26
27 # Number of training epochs
28 num_epochs = 10
29
30 # Learning rate for optimizers
31 lr = 0.0002
32
33 # Beta1 hyperparameter for Adam optimizers
34 beta1 = 0.5
35
36 # Number of GPUs available. Use 0 for CPU mode.
37 ngpu = 1
```

```
1 # Root directory for dataset
2 dataroot = "/content/celeba"
3
4 # Number of workers for dataloader
5 workers = 2
6
7 # Batch size during training
8 batch_size = 128
9
10 # Spatial size of training images. All images will be resized to this
11 #   size using a transformer.
```

```
12 image_size = 64
13
14 # Number of channels in the training images. For color images this is 3
15 nc = 3
16
17 # Size of z latent vector (i.e. size of generator input)
18 nz = 100
19
20 # Size of feature maps in generator
21 ngf = 64
22
23 # Size of feature maps in discriminator
24 ndf = 64
25
26 # Number of training epochs
27 num_epochs = 10
28
29 # Learning rate for optimizers
30 lr = 0.0002
31
32 # Beta1 hyperparameter for Adam optimizers
33 beta1 = 0.5
34
35 # Number of GPUs available. Use 0 for CPU mode.
36 ngpu = 1
```

```
1 # We can use an image folder dataset the way we have it setup.
2 # Create the dataset
3 dataset = dset.ImageFolder(root=dataroot,
4                             transform=transforms.Compose([
5                                 transforms.Resize(image_size),
6                                 transforms.CenterCrop(image_size),
7                                 transforms.ToTensor(),
8                                 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
9                             ]))
10 # Create the dataloader
11 dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
12                                         shuffle=True, num_workers=workers)
13
14 # Decide which device we want to run on
15 device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu"
16
17 # Plot some training images
18 real_batch = next(iter(dataloader))
19 plt.figure(figsize=(8,8))
20 plt.axis("off")
21 plt.title("Training Images")
22 plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:, :64], padding=2, r
23 plt.show()
```

Training Images



```

1 # custom weights initialization called on ``netG`` and ``netD``
2 def weights_init(m):
3     classname = m.__class__.__name__
4     if classname.find('Conv') != -1:
5         nn.init.normal_(m.weight.data, 0.0, 0.02)
6     elif classname.find('BatchNorm') != -1:
7         nn.init.normal_(m.weight.data, 1.0, 0.02)
8         nn.init.constant_(m.bias.data, 0)

```

```

1 # DCGAN Generator Code
2 class DCGAN_Generator(nn.Module):
3     def __init__(self, ngpu):
4         super(DCGAN_Generator, self).__init__()
5         self.ngpu = ngpu
6         self.main = nn.Sequential(
7             # input is Z, going into a convolution
8             nn.ConvTranspose2d(nz, ngf * 8, 4, 1, 0, bias=False),
9             nn.BatchNorm2d(ngf * 8),
10            nn.ReLU(True),
11            # state size. (ngf*8) x 4 x 4
12            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
13            nn.BatchNorm2d(ngf * 4),
14            nn.ReLU(True),
15            # state size. (ngf*4) x 8 x 8
16            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
17            nn.BatchNorm2d(ngf * 2),
18            nn.ReLU(True),
19            # state size. (ngf*2) x 16 x 16
20            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
21            nn.BatchNorm2d(ngf),
22            nn.ReLU(True),
23            # state size. (ngf) x 32 x 32
24            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
25            nn.Tanh()

```

```

26         # state size. (nc) x 64 x 64
27     )
28
29     def forward(self, input):
30         return self.main(input)
31

```

---

```

1 # Create the DCGAN generator
2 netG_dcgan = DCGAN_Generator(ngpu).to(device)
3
4 # Handle multi-GPU if desired
5 if (device.type == 'cuda') and (ngpu > 1):
6     netG_dcgan = nn.DataParallel(netG_dcgan, list(range(ngpu)))
7
8 # Apply the weights_init function to randomly initialize all weights
9 netG_dcgan.apply(weights_init)
10
11 # Print the model
12 print(netG_dcgan)
13

```

```

DCGAN_Generator(
    main: Sequential(
        (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): ReLU(inplace=True)
        (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (8): ReLU(inplace=True)
        (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (11): ReLU(inplace=True)
        (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (13): Tanh()
    )
)

```

```

1 # DCGAN Discriminator
2 class DCGAN_Discriminator(nn.Module):
3     def __init__(self, ngpu):
4         super(DCGAN_Discriminator, self).__init__()
5         self.ngpu = ngpu
6         self.main = nn.Sequential(
7             # input is (nc) x 64 x 64
8             nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
9             nn.LeakyReLU(0.2, inplace=True),
10            # state size. (ndf) x 32 x 32
11            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
12            nn.BatchNorm2d(ndf * 2),
13            nn.LeakyReLU(0.2, inplace=True),
14            # state size. (ndf*2) x 16 x 16
15            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
16            nn.BatchNorm2d(ndf * 4),
17            nn.LeakyReLU(0.2, inplace=True),
18            # state size. (ndf*4) x 8 x 8
19            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
20            nn.BatchNorm2d(ndf * 8),
21            nn.LeakyReLU(0.2, inplace=True),
22            # state size. (ndf*8) x 4 x 4
23            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
24            nn.Sigmoid()
25        )
26
27     def forward(self, input):

```

```
28     return self.main(input)
29
```

```
1 # Create the DCGAN Discriminator
2 netD_dcgan = DCGAN_Discriminator(ngpu).to(device)
3
4 # Handle multi-GPU if desired
5 if (device.type == 'cuda') and (ngpu > 1):
6     netD_dcgan = nn.DataParallel(netD_dcgan, list(range(ngpu)))
7
8 # Apply the weights_init function to randomly initialize all weights
9 netD_dcgan.apply(weights_init)
10
11 # Print the model
12 print(netD_dcgan)
13

DCGAN_Discriminator(
(main): Sequential(
(0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(1): LeakyReLU(negative_slope=0.2, inplace=True)
(2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(4): LeakyReLU(negative_slope=0.2, inplace=True)
(5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(7): LeakyReLU(negative_slope=0.2, inplace=True)
(8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(10): LeakyReLU(negative_slope=0.2, inplace=True)
(11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
(12): Sigmoid()
)
)
```

```
1 # Initialize the BCELoss function
2 criterion = nn.BCELoss()
3
4 # Create batch of latent vectors that we will use to visualize
5 # the progression of the generator
6 fixed_noise_dcgan = torch.randn(64, nz, 1, 1, device=device)
7
8 # Establish convention for real and fake labels during training
9 real_label = 1.0
10 fake_label = 0.0
11
12 # Setup Adam optimizers for both G and D (specific to DCGAN)
13 optimizerD_dcgan = optim.Adam(netD_dcgan.parameters(), lr=lr, betas=(beta1, 0.999))
14 optimizerG_dcgan = optim.Adam(netG_dcgan.parameters(), lr=lr, betas=(beta1, 0.999))
15
```

```
1 # -----
2 # DCGAN Training Loop
3 # -----
4
5 # Lists to keep track of progress (specific to DCGAN)
6 img_list_dcgan = []
7 G_losses_dcgan = []
8 D_losses_dcgan = []
9 iters_dcgan = 0
10
11 print("Starting DCGAN Training Loop...")
12 # For each epoch
13 for epoch in range(num_epochs):
14     for i, data in enumerate(train_loader, 0): # use same train_loader
15
16         #####
```

```

17 # (1) Update D network
18 #####
19 netD_dcgan.zero_grad()
20 real_cpu = data[0].to(device)
21 b_size = real_cpu.size(0)
22 label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
23
24 # Real batch
25 output = netD_dcgan(real_cpu).view(-1)
26 errD_real = criterion(output, label)
27 errD_real.backward()
28 D_x = output.mean().item()
29
30 # Fake batch
31 noise = torch.randn(b_size, nz, 1, 1, device=device)
32 fake = netG_dcgan(noise)
33 label.fill_(fake_label)
34 output = netD_dcgan(fake.detach()).view(-1)
35 errD_fake = criterion(output, label)
36 errD_fake.backward()
37 D_G_z1 = output.mean().item()
38 errD = errD_real + errD_fake
39 optimizerD_dcgan.step()
40
41 #####
42 # (2) Update G network
43 #####
44 netG_dcgan.zero_grad()
45 label.fill_(real_label) # fake labels = real
46 output = netD_dcgan(fake).view(-1)
47 errG = criterion(output, label)
48 errG.backward()
49 D_G_z2 = output.mean().item()
50 optimizerG_dcgan.step()
51
52 # Output training stats
53 if i % 50 == 0:
54     print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f\tD(G(z2)): %.4f'
55         % (epoch, num_epochs, i, len(train_loader),
56             errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
57
58 # Save losses
59 G_losses_dcgan.append(errG.item())
60 D_losses_dcgan.append(errD.item())
61
62 # Save images for visualization
63 if (iters_dcgan % 500 == 0) or ((epoch == num_epochs-1) and (i == len(train_loader)))
64     with torch.no_grad():
65         fake = netG_dcgan(fixed_noise_dcgan).detach().cpu()
66         img_list_dcgan.append(vutils.make_grid(fake, padding=2, normalize=True))
67
68 iters_dcgan += 1
69

```

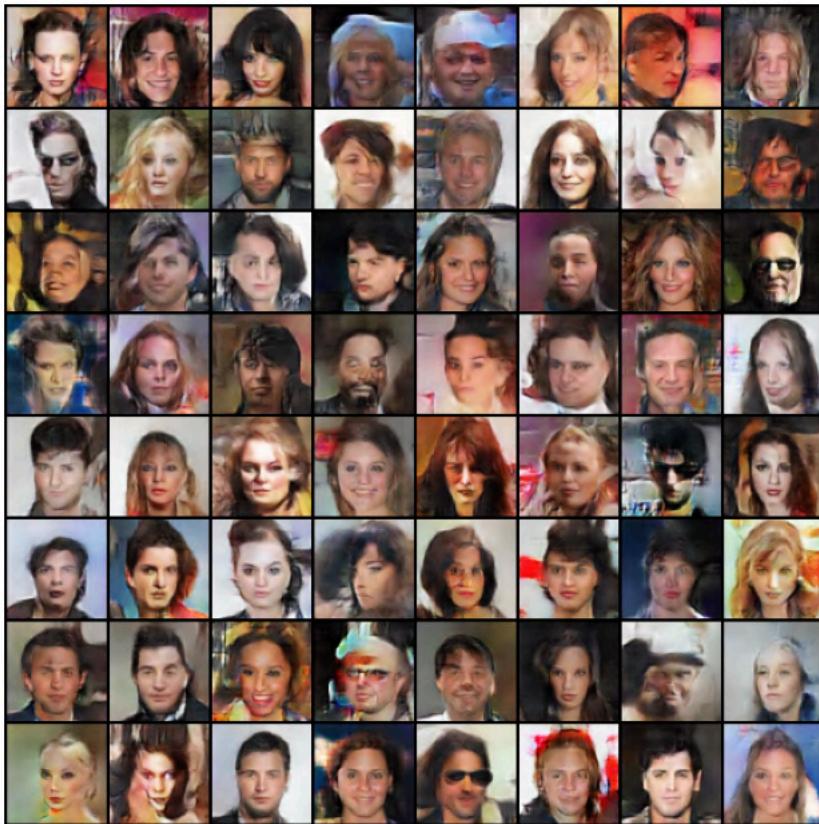
	Loss_D: 0.0109	Loss_G: 0.2279	D(x): 0.8974	D(G(z)): 0.2077 / 0.0473
[7/10][1050/1271]	Loss_D: 0.4692	Loss_G: 2.7304	D(x): 0.8192	D(G(z)): 0.2052 / 0.0812
[7/10][1100/1271]	Loss_D: 0.5472	Loss_G: 3.3892	D(x): 0.8773	D(G(z)): 0.3043 / 0.0450
[7/10][1150/1271]	Loss_D: 1.7285	Loss_G: 0.6643	D(x): 0.2522	D(G(z)): 0.0242 / 0.5807
[7/10][1200/1271]	Loss_D: 0.6862	Loss_G: 1.5404	D(x): 0.6032	D(G(z)): 0.0835 / 0.2547
[7/10][1250/1271]	Loss_D: 0.5049	Loss_G: 3.7651	D(x): 0.9105	D(G(z)): 0.3075 / 0.0309
[8/10][0/1271]	Loss_D: 0.8921	Loss_G: 0.9870	D(x): 0.4826	D(G(z)): 0.0504 / 0.4603
[8/10][50/1271]	Loss_D: 0.5408	Loss_G: 1.2358	D(x): 0.7005	D(G(z)): 0.1321 / 0.3327
[8/10][100/1271]	Loss_D: 0.6294	Loss_G: 3.7760	D(x): 0.8972	D(G(z)): 0.3643 / 0.0324
[8/10][150/1271]	Loss_D: 0.5532	Loss_G: 3.5244	D(x): 0.9392	D(G(z)): 0.3526 / 0.0417
[8/10][200/1271]	Loss_D: 0.3862	Loss_G: 4.0842	D(x): 0.9218	D(G(z)): 0.2388 / 0.0233
[8/10][250/1271]	Loss_D: 0.8500	Loss_G: 2.9132	D(x): 0.7782	D(G(z)): 0.3863 / 0.0726
[8/10][300/1271]	Loss_D: 1.0073	Loss_G: 3.4404	D(x): 0.8169	D(G(z)): 0.4825 / 0.0489
[8/10][350/1271]	Loss_D: 0.4928	Loss_G: 3.5403	D(x): 0.9064	D(G(z)): 0.2987 / 0.0373
[8/10][400/1271]	Loss_D: 0.5019	Loss_G: 2.7834	D(x): 0.8653	D(G(z)): 0.2737 / 0.0782
[8/10][450/1271]	Loss_D: 0.4516	Loss_G: 2.7544	D(x): 0.8058	D(G(z)): 0.1842 / 0.0853
[8/10][500/1271]	Loss_D: 0.4850	Loss_G: 2.2606	D(x): 0.7666	D(G(z)): 0.1647 / 0.1350
[8/10][550/1271]	Loss_D: 0.4123	Loss_G: 2.4139	D(x): 0.8180	D(G(z)): 0.1606 / 0.1217
[8/10][600/1271]	Loss_D: 0.5899	Loss_G: 3.9209	D(x): 0.9094	D(G(z)): 0.3518 / 0.0271
[8/10][650/1271]	Loss_D: 0.5183	Loss_G: 2.9969	D(x): 0.7322	D(G(z)): 0.1249 / 0.0768
[8/10][700/1271]	Loss_D: 0.5447	Loss_G: 2.2290	D(x): 0.7293	D(G(z)): 0.1637 / 0.1385
[8/10][750/1271]	Loss_D: 0.3586	Loss_G: 3.0303	D(x): 0.9269	D(G(z)): 0.2237 / 0.0643
[8/10][800/1271]	Loss_D: 0.4951	Loss_G: 2.2009	D(x): 0.8076	D(G(z)): 0.2123 / 0.1419
[8/10][850/1271]	Loss_D: 0.7298	Loss_G: 4.0777	D(x): 0.9214	D(G(z)): 0.4251 / 0.0267
[8/10][900/1271]	Loss_D: 0.4204	Loss_G: 3.0325	D(x): 0.7851	D(G(z)): 0.1145 / 0.0686
[8/10][950/1271]	Loss_D: 0.4483	Loss_G: 2.1519	D(x): 0.7755	D(G(z)): 0.1467 / 0.1457
[8/10][1000/1271]	Loss_D: 0.4150	Loss_G: 2.2518	D(x): 0.8076	D(G(z)): 0.1529 / 0.1401
[8/10][1050/1271]	Loss_D: 0.7876	Loss_G: 1.2701	D(x): 0.5345	D(G(z)): 0.0392 / 0.3434
[8/10][1100/1271]	Loss_D: 0.6545	Loss_G: 0.20169	D(x): 0.6185	D(G(z)): 0.0808 / 0.1926
[8/10][1150/1271]	Loss_D: 0.6062	Loss_G: 2.7706	D(x): 0.8506	D(G(z)): 0.3060 / 0.0857
[8/10][1200/1271]	Loss_D: 0.4881	Loss_G: 2.6078	D(x): 0.8573	D(G(z)): 0.2573 / 0.0919
[8/10][1250/1271]	Loss_D: 0.3351	Loss_G: 2.1291	D(x): 0.8317	D(G(z)): 0.1219 / 0.1576
[9/10][0/1271]	Loss_D: 0.4657	Loss_G: 3.5547	D(x): 0.8790	D(G(z)): 0.2597 / 0.0384
[9/10][50/1271]	Loss_D: 1.4900	Loss_G: 0.9234	D(x): 0.2887	D(G(z)): 0.0178 / 0.4761
[9/10][100/1271]	Loss_D: 0.3836	Loss_G: 2.9694	D(x): 0.8805	D(G(z)): 0.1944 / 0.0687
[9/10][150/1271]	Loss_D: 4.1948	Loss_G: 0.1820	D(x): 0.0346	D(G(z)): 0.0103 / 0.8515
[9/10][200/1271]	Loss_D: 0.3514	Loss_G: 2.9385	D(x): 0.8596	D(G(z)): 0.1610 / 0.0780
[9/10][250/1271]	Loss_D: 0.30493	Loss_G: 0.5413	D(x): 0.0855	D(G(z)): 0.0327 / 0.6725
[9/10][300/1271]	Loss_D: 0.6241	Loss_G: 1.8120	D(x): 0.6767	D(G(z)): 0.1474 / 0.2008
[9/10][350/1271]	Loss_D: 0.3673	Loss_G: 3.4450	D(x): 0.9175	D(G(z)): 0.2275 / 0.0416
[9/10][400/1271]	Loss_D: 0.3535	Loss_G: 2.1943	D(x): 0.8128	D(G(z)): 0.1184 / 0.1409
[9/10][450/1271]	Loss_D: 0.4648	Loss_G: 3.4404	D(x): 0.9167	D(G(z)): 0.2840 / 0.0414
[9/10][500/1271]	Loss_D: 0.5509	Loss_G: 4.4768	D(x): 0.9287	D(G(z)): 0.3419 / 0.0166
[9/10][550/1271]	Loss_D: 0.4345	Loss_G: 3.0673	D(x): 0.8236	D(G(z)): 0.1851 / 0.0703
[9/10][600/1271]	Loss_D: 0.3783	Loss_G: 2.3158	D(x): 0.7650	D(G(z)): 0.0787 / 0.1359
[9/10][650/1271]	Loss_D: 0.8112	Loss_G: 3.2468	D(x): 0.8535	D(G(z)): 0.4035 / 0.0632

```

1 fig = plt.figure(figsize=(8,8))
2 plt.axis("off")
3 ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list_dcgan]
4 ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True
5
6 HTML(ani.to_jshtml())
7

```

WARNING:matplotlib.animation:Animation size has reached 21173940 bytes, exceeding the limit of 20971520.0. If you're sure you want to continue, use plt.ion()



[ - ] [ < ] [ < ] [ < ] [ II ] [ > ] [ > ] [ + ]  
 Once  Loop  Reflect

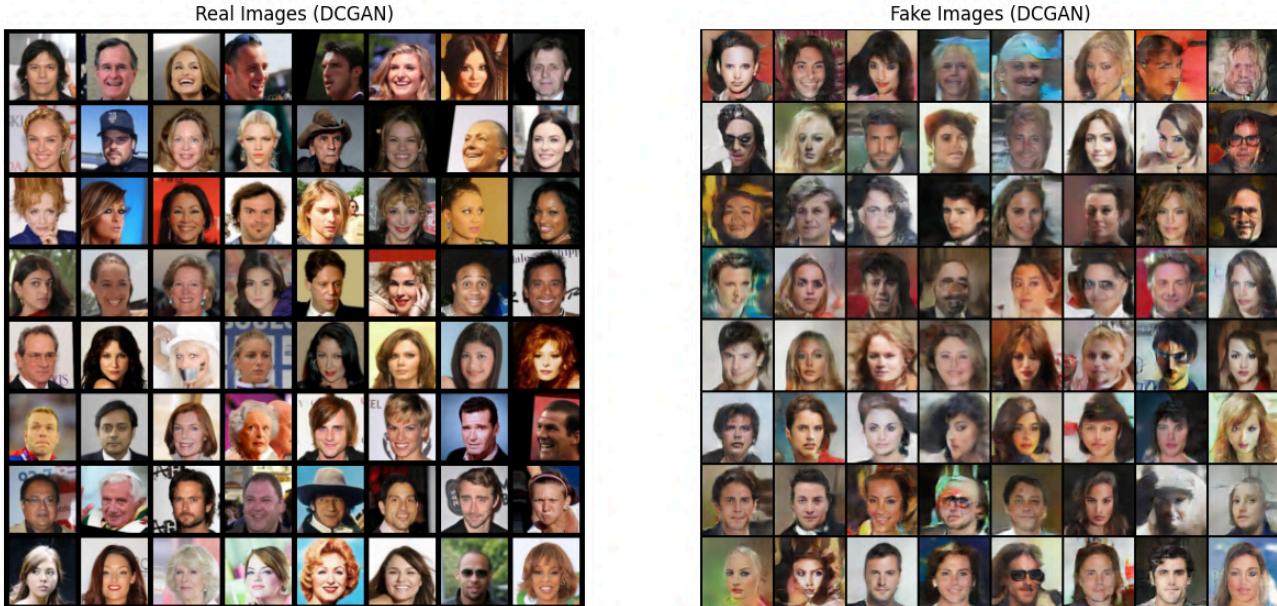


```

1 # Grab a batch of real images from the dataloader
2 real_batch = next(iter(dataloader))
3
4 # Plot the real images
5 plt.figure(figsize=(15,15))
6 plt.subplot(1,2,1)
7 plt.axis("off")
8 plt.title("Real Images (DCGAN)")
9 plt.imshow(np.transpose(
10     vutils.make_grid(real_batch[0].to(device)[:64], padding=5, normalize=True).cpu()
11     (1,2,0)
12 ))
13
14 # Plot the fake images from the last epoch of DCGAN
15 plt.subplot(1,2,2)
16 plt.axis("off")
17 plt.title("Fake Images (DCGAN)")
18 plt.imshow(np.transpose(img_list_dcgan[-1], (1,2,0)))
19 plt.show()

```





```

1 # Save final state_dicts (lightweight, most common)
2 torch.save(netG_dcgan.state_dict(), save_dir / "netG_dcgan_final.pth")
3 torch.save(netD_dcgan.state_dict(), save_dir / "netD_dcgan_final.pth")
4
5 # Save optimizer states + epoch + loss metrics in a checkpoint (recommended)
6 checkpoint_dcgan = {
7     "epoch": num_epochs,
8     "netG_state": netG_dcgan.state_dict(),
9     "netD_state": netD_dcgan.state_dict(),
10    "optimG_state": optimizerG_dcgan.state_dict(),
11    "optimD_state": optimizerD_dcgan.state_dict(),
12    "G_losses": G_losses_dcgan,
13    "D_losses": D_losses_dcgan,
14    "manualSeed": manualSeed,
15 }
16 torch.save(checkpoint_dcgan, save_dir / "dcgan_checkpoint_final.pth")
17
18 print("Saved netG_dcgan_final.pth, netD_dcgan_final.pth and dcgan_checkpoint_final.pth")
19

```

Saved netG\_dcgan\_final.pth, netD\_dcgan\_final.pth and dcgan\_checkpoint\_final.pth to cgan\_outputs

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import torchvision.utils as vutils
4
5 # -----
6 # Get a real batch (common for both)
7 # -----
8 real_batch = next(iter(train_loader)) # reuse same dataloader for CelebA
9 real_images = real_batch[0].to(device)[:64]
10
11 # -----
12 # Generate fake images from CGAN
13 # -----
14 with torch.no_grad():
15     fake_cgan = netG(fixed_noise.to(device), fixed_labels.to(device)).detach().cpu()
16

```

```

17 # -----
18 # Generate fake images from DCGAN
19 #
20 with torch.no_grad():
21     fake_dcgan = netG_dcgan(fixed_noise_dcgan.to(device)).detach().cpu()
22
23 #
24 # Plotting comparison
25 #
26 plt.figure(figsize=(18,9))
27
28 # Real images
29 plt.subplot(1,3,1)
30 plt.axis("off")
31 plt.title("Real Images (CelebA)")
32 plt.imshow(np.transpose(
33     vutils.make_grid(real_images.cpu(), padding=2, normalize=True),
34     (1,2,0)
35 ))
36
37 # CGAN fake images
38 plt.subplot(1,3,2)
39 plt.axis("off")
40 plt.title("Fake Images (CGAN)")
41 plt.imshow(np.transpose(
42     vutils.make_grid(fake_cgan, padding=2, normalize=True),
43     (1,2,0)
44 ))
45
46 # DCGAN fake images
47 plt.subplot(1,3,3)
48 plt.axis("off")
49 plt.title("Fake Images (DCGAN)")
50 plt.imshow(np.transpose(
51     vutils.make_grid(fake_dcgan, padding=2, normalize=True),
52     (1,2,0)
53 ))
54
55 plt.show()

```

/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:627: UserWarning: This DataLoader will create 4 worker processes. Consider specifying num\_workers=0 or increasing max\_queue\_size to prevent memory issues.

