

#Real Image to Caricature Image

```
# extracting the compressed Dataset
from zipfile import ZipFile
dataset = '/content/drive/MyDrive/archive (1).zip'
from zipfile import ZipFile

# with ZipFile(dataset,'r') as zip:
#     zip.extractall()
!unzip '/content/drive/MyDrive/archive (1).zip' -d '/content/drive/MyDrive/SSP_Project'
print('The dataset is extracted')
```

 [Show hidden output](#)

```
!pip install torch torchvision matplotlib opencv-python tqdm
```

 [Show hidden output](#)

```
# Replace these paths with your real and caricature folder paths
real_path = "/content/drive/MyDrive/SSP_Project/CaVI_Dataset/CaVI_Dataset/Real"
caric_path = "/content/drive/MyDrive/SSP_Project/CaVI_Dataset/CaVI_Dataset/Caricature"

import cv2
import os

def crop_faces(input_folder, output_folder):
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
    os.makedirs(output_folder, exist_ok=True)
    for img_name in os.listdir(input_folder):
        img_path = os.path.join(input_folder, img_name)
        img = cv2.imread(img_path)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.3, 5)
        for (x, y, w, h) in faces:
            face = img[y:y+h, x:x+w]
            resized = cv2.resize(face, (256, 256))
            cv2.imwrite(os.path.join(output_folder, img_name), resized)
            break

import torch.nn as nn

class ResidualBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(dim, dim, kernel_size=3, padding=1),
            nn.InstanceNorm2d(dim),
            nn.ReLU(inplace=True),
            nn.Conv2d(dim, dim, kernel_size=3, padding=1),
            nn.InstanceNorm2d(dim),
        )

    def forward(self, x):
        return x + self.block(x)

class Generator(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, n_res_blocks=6):
        super().__init__()
        model = [
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=1, padding=3),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        ]

        # Downsampling
        model += [
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(256),
```

```

        nn.ReLU(inplace=True)
    ]

    # Residual blocks
    for _ in range(n_res_blocks):
        model += [ResidualBlock(256)]

    # Upsampling
    model += [
        nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1),
        nn.InstanceNorm2d(128),
        nn.ReLU(inplace=True),
        nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
        nn.InstanceNorm2d(64),
        nn.ReLU(inplace=True),
        nn.Conv2d(64, out_channels, kernel_size=7, stride=1, padding=3),
        nn.Tanh()
    ]

    self.model = nn.Sequential(*model)

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

class Discriminator(nn.Module):
    def __init__(self, in_channels=3):
        super().__init__()
        def discriminator_block(in_filters, out_filters, normalize=True):
            layers = [nn.Conv2d(in_filters, out_filters, kernel_size=4, stride=2, padding=1)]
            if normalize:
                layers.append(nn.InstanceNorm2d(out_filters))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            return layers

        self.model = nn.Sequential(
            *discriminator_block(in_channels, 64, normalize=False),
            *discriminator_block(64, 128),
            *discriminator_block(128, 256),
            *discriminator_block(256, 512),
            nn.Conv2d(512, 1, kernel_size=4, padding=1)
        )

    def forward(self, img):
        return self.model(img)

from torch.utils.data import Dataset, DataLoader
from PIL import Image
import glob
import torchvision.transforms as transforms
import os

class FaceCaricatureDataset(Dataset):
    def __init__(self, real_root, caric_root, transform=None):
        # Modified to include various image extensions and ensure paths are correct
        self.real_paths = sorted(glob.glob(os.path.join(real_root, '**/*.jpg'), recursive=True) +
                                glob.glob(os.path.join(real_root, '**/*.png'), recursive=True) +
                                glob.glob(os.path.join(real_root, '**/*.jpeg'), recursive=True))
        self.caric_paths = sorted(glob.glob(os.path.join(caric_root, '**/*.jpg'), recursive=True) +
                                glob.glob(os.path.join(caric_root, '**/*.png'), recursive=True) +
                                glob.glob(os.path.join(caric_root, '**/*.jpeg'), recursive=True))
        self.transform = transform

    def __len__(self):
        return min(len(self.real_paths), len(self.caric_paths)) # Ensure both paths have same number of images

    def __getitem__(self, index):
        real_img = Image.open(self.real_paths[index]).convert('RGB')
        caric_img = Image.open(self.caric_paths[index]).convert('RGB')
        if self.transform:
            real_img = self.transform(real_img)
            caric_img = self.transform(caric_img)
        return real_img, caric_img

transform = transforms.Compose([

```

```

        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize((0.5,), (0.5,))
    ])

```

```

dataset = FaceCaricatureDataset(real_path, caric_path, transform=transform)
print(len(dataset)) # Print dataset length to check for 0
print(dataset.real_paths[:5]) # Print a few real image paths for verification
print(dataset.caric_paths[:5]) # Print a few caricature image paths for verification
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

```

```

4816
['/content/drive/MyDrive/SSP_Project/CaVI_Dataset/CaVI_Dataset/Real/Aamir_Khan/Aamir_Khan_r_0.jpg', '/content/dri
['/content/drive/MyDrive/SSP_Project/CaVI_Dataset/CaVI_Dataset/Caricature/Aamir_Khan/Aamir_Khan_c_0.jpg', '/conte

```

#Train the Model

```

import torch
import torch.optim as optim
import torch.nn as nn
from tqdm import tqdm

```

```

# Check for GPU availability and fallback to CPU if not available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device) # Inform the user of the device being used

```

```

G = Generator().to(device)
D = Discriminator().to(device)

```

```

optimizer_G = optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_D = optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))

```

```

adversarial_loss = nn.MSELoss()
reconstruction_loss = nn.L1Loss()

```

```

for epoch in range(15):
    for i, (real, caric) in enumerate(tqdm(dataloader)):
        # Move data to the selected device
        real, caric = real.to(device), caric.to(device)
        valid = torch.ones((real.size(0), 1, 15, 15)).to(device)
        fake = torch.zeros((real.size(0), 1, 15, 15)).to(device)

        # -----
        # Train Generator
        # -----
        optimizer_G.zero_grad()
        fake_caric = G(real)
        g_adv = adversarial_loss(D(fake_caric), valid)
        g_recon = reconstruction_loss(fake_caric, caric)
        g_loss = g_adv + 10 * g_recon
        g_loss.backward()
        optimizer_G.step()

        # -----
        # Train Discriminator
        # -----
        optimizer_D.zero_grad()
        real_loss = adversarial_loss(D(caric), valid)
        fake_loss = adversarial_loss(D(fake_caric.detach()), fake)
        d_loss = 0.5 * (real_loss + fake_loss)
        d_loss.backward()
        optimizer_D.step()

```

```

print(f"Epoch {epoch} | Generator Loss: {g_loss.item():.4f} | Discriminator Loss: {d_loss.item():.4f}")

```

```

Using device: cuda
100%|██████████| 1204/1204 [05:43<00:00, 3.50it/s]
Epoch 0 | Generator Loss: 8.3174 | Discriminator Loss: 0.1212
100%|██████████| 1204/1204 [05:45<00:00, 3.48it/s]
Epoch 1 | Generator Loss: 6.9635 | Discriminator Loss: 0.0279
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 2 | Generator Loss: 6.1043 | Discriminator Loss: 0.1546
100%|██████████| 1204/1204 [05:45<00:00, 3.48it/s]
Epoch 3 | Generator Loss: 6.8456 | Discriminator Loss: 0.1113
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 4 | Generator Loss: 7.2564 | Discriminator Loss: 0.1043
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 5 | Generator Loss: 7.0108 | Discriminator Loss: 0.0650

```

```

100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 6 | Generator Loss: 7.1445 | Discriminator Loss: 0.0920
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 7 | Generator Loss: 7.1613 | Discriminator Loss: 0.0739
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 8 | Generator Loss: 7.0254 | Discriminator Loss: 0.0801
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 9 | Generator Loss: 6.8316 | Discriminator Loss: 0.0137
100%|██████████| 1204/1204 [05:45<00:00, 3.48it/s]
Epoch 10 | Generator Loss: 6.6727 | Discriminator Loss: 0.1489
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 11 | Generator Loss: 5.6747 | Discriminator Loss: 0.2144
100%|██████████| 1204/1204 [05:45<00:00, 3.48it/s]
Epoch 12 | Generator Loss: 6.5118 | Discriminator Loss: 0.1131
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 13 | Generator Loss: 6.5336 | Discriminator Loss: 0.0787
100%|██████████| 1204/1204 [05:46<00:00, 3.48it/s]
Epoch 14 | Generator Loss: 6.6734 | Discriminator Loss: 0.0715

```

```

import os
from torchvision.utils import save_image

os.makedirs("output", exist_ok=True)

G.eval()
with torch.no_grad():
    for i, (real, _) in enumerate(data_loader):
        real = real.cuda()
        fake_caric = G(real)
        save_image(fake_caric, f'output/caric_{i}.png', normalize=True)
        save_image(real, f'output/real_{i}.png', normalize=True)
        if i == 2: break

# Save
torch.save(G.state_dict(), "/content/drive/MyDrive/SSP_Project/generator.pth") # Corrected path
torch.save(D.state_dict(), "/content/drive/MyDrive/SSP_Project/discriminator.pth") # Corrected path

```

↔ <All keys matched successfully>

#Load the Model

```

from google.colab import drive
drive.mount('/content/drive')

```

↔ Mounted at /content/drive

```
!pip install torch torchvision matplotlib opencv-python tqdm
```

↔

```

uninstalling nvidia-cufft-cu12-11.2.3.01:
Successfully uninstalled nvidia-cufft-cu12-11.2.3.01
Attempting uninstall: nvidia-cuda-runtime-cu12
Found existing installation: nvidia-cuda-runtime-cu12 12.5.82
Uninstalling nvidia-cuda-runtime-cu12-12.5.82:
Successfully uninstalled nvidia-cuda-runtime-cu12-12.5.82
Attempting uninstall: nvidia-cuda-nvrtc-cu12
Found existing installation: nvidia-cuda-nvrtc-cu12 12.5.82
Uninstalling nvidia-cuda-nvrtc-cu12-12.5.82:
Successfully uninstalled nvidia-cuda-nvrtc-cu12-12.5.82
Attempting uninstall: nvidia-cuda-cupti-cu12
Found existing installation: nvidia-cuda-cupti-cu12 12.5.82
Uninstalling nvidia-cuda-cupti-cu12-12.5.82:
Successfully uninstalled nvidia-cuda-cupti-cu12-12.5.82
Attempting uninstall: nvidia-cublas-cu12
Found existing installation: nvidia-cublas-cu12 12.5.3.2
Uninstalling nvidia-cublas-cu12-12.5.3.2:
Successfully uninstalled nvidia-cublas-cu12-12.5.3.2
Attempting uninstall: nvidia-cusparse-cu12
Found existing installation: nvidia-cusparse-cu12 12.5.1.3
Uninstalling nvidia-cusparse-cu12-12.5.1.3:
Successfully uninstalled nvidia-cusparse-cu12-12.5.1.3
Attempting uninstall: nvidia-cudnn-cu12
Found existing installation: nvidia-cudnn-cu12 9.3.0.75
Uninstalling nvidia-cudnn-cu12-9.3.0.75:
Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
Attempting uninstall: nvidia-cusolver-cu12
Found existing installation: nvidia-cusolver-cu12 11.6.3.83
Uninstalling nvidia-cusolver-cu12-11.6.3.83:
Successfully uninstalled nvidia-cusolver-cu12-11.6.3.83
Successfully installed nvidia-cublas-cu12-12.4.5.8 nvidia-cuda-cupti-cu12-12.4.127 nvidia-cuda-nvrtc-cu12-12.4.12

```

```

import torch.nn as nn
import torch
import torch.optim as optim
import torch.nn as nn
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import glob
import torchvision.transforms as transforms
import os
import cv2
import os

```

Define the Generator and Discriminator classes first

```

class ResidualBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(dim, dim, kernel_size=3, padding=1),
            nn.InstanceNorm2d(dim),
            nn.ReLU(inplace=True),
            nn.Conv2d(dim, dim, kernel_size=3, padding=1),
            nn.InstanceNorm2d(dim),
        )

    def forward(self, x):
        return x + self.block(x)

class Generator(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, n_res_blocks=6):
        super().__init__()
        model = [
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=1, padding=3),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True)
        ]

        # Downsampling
        model += [
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(256),
            nn.ReLU(inplace=True)
        ]

        # Residual blocks
        for i in range(n_res_blocks):

```

```

        for i in range(n_res_blocks):
            model += [ResidualBlock(256)]

    # Upsampling
    model += [
        nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1),
        nn.InstanceNorm2d(128),
        nn.ReLU(inplace=True),
        nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
        nn.InstanceNorm2d(64),
        nn.ReLU(inplace=True),
        nn.Conv2d(64, out_channels, kernel_size=7, stride=1, padding=3),
        nn.Tanh()
    ]

    self.model = nn.Sequential(*model)

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

class Discriminator(nn.Module):
    def __init__(self, in_channels=3):
        super().__init__()
    def discriminator_block(in_filters, out_filters, normalize=True):
        layers = [nn.Conv2d(in_filters, out_filters, kernel_size=4, stride=2, padding=1)]
        if normalize:
            layers.append(nn.InstanceNorm2d(out_filters))
        layers.append(nn.LeakyReLU(0.2, inplace=True))
        return layers

    self.model = nn.Sequential(
        *discriminator_block(in_channels, 64, normalize=False),
        *discriminator_block(64, 128),
        *discriminator_block(128, 256),
        *discriminator_block(256, 512),
        nn.Conv2d(512, 1, kernel_size=4, padding=1)
    )

    def forward(self, img):
        return self.model(img)

# Create instances of Generator and Discriminator
G = Generator()
D = Discriminator()

```

#Import the Model

```

# Load the state dictionaries with map_location to CPU
G.load_state_dict(torch.load("/content/drive/MyDrive/SSP_Project/generator.pth", map_location=torch.device('cpu'))) #
D.load_state_dict(torch.load("/content/drive/MyDrive/SSP_Project/discriminator.pth", map_location=torch.device('cpu')))

```

 <All keys matched successfully>

#Test the Model

```

import os
from torchvision.utils import save_image
from PIL import Image

# Set this path to a folder of test face images (either real or caricature) or a single image
test_path = "/content/drive/MyDrive/SSP_Project/Real/Albert_Einstein/Albert_Einstein_0.jpg"

# Output folder
output_path = "/content/output"
os.makedirs(output_path, exist_ok=True)

# Transform for test images
transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, ), (0.5, ))
])

# Inference

```

```

# Check if CUDA is available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
G.to(device).eval() # Move the model to the correct device and set to evaluation mode

with torch.no_grad():
    # If test_path is a directory, loop through images
    if os.path.isdir(test_path):
        for img_name in os.listdir(test_path):
            img_path = os.path.join(test_path, img_name)
            image = Image.open(img_path).convert("RGB")
            input_tensor = transform(image).unsqueeze(0).to(device) # Move input tensor to the correct device
            output = G(input_tensor)
            save_path = os.path.join(output_path, f"out_{img_name}")
            save_image(output, save_path, normalize=True)
    # If test_path is a single image file, process it directly
    else:
        image = Image.open(test_path).convert("RGB")
        input_tensor = transform(image).unsqueeze(0).to(device) # Move input tensor to the correct device
        output = G(input_tensor)
        # Get the filename from the path
        img_name = os.path.basename(test_path)
        save_path = os.path.join(output_path, f"out_{img_name}")
        save_image(output, save_path, normalize=True)

print("output: Real Image to Caricature Image\n\n")
from IPython.display import Image
orig_img = Image(filename=test_path)
display(orig_img)
pil_img = Image(filename=save_path)
display(pil_img)

```

↗ output images Real to Caricature



