Term Project

CPS 584

Introduction

In this project, for the task of generating additional human face images, a new diffusion based generative model was created. My dataset consisted of facial images categorized into three demographic groups: Now let me list the following groups of people: the Orientals, Indians and White Europeans. The task involved creating images from different categories that are demographic, including Orientals/Indians, Orientals/Europeans, Indian/Europeans and Orientals/Indian/Europeans. To this end, we described a novel architecture based on the UNet model for noise prediction.

Project Description

This project specifically aims at developing a generative model for generating fake face images using a diffusion-based framework. The dataset used for this project consists of facial images categorized on three racial groups: White Europeans; Orientals; and Indians. The goal of the work is to reconstruct realistic face images based on elements of different races using a diffusion model.

The training of a diffusion-based generative model is at the center of the project; generating new faces is done through gradually introducing noise into actual images and then learning how to invert the process. In this project the training data is managed and prepared in such a way that it's easy handling and managing of the data. The basic progressing architecture incorporates the UNet model which is popular in applications of image generation. The use of loss values and visualization of images generated at the different stages of the model also forms part of the project. The final outputs will be these generated images, model evaluations and a detailed report on threads, methodology and findings.

Summary of Tasks

This work was divided into several major steps toward the construction of synthetic face images using a diffusion-based model. Below is a summary of the tasks completed during the project:

Data Collection and Preprocessing: The first activity was a data gathering process where facial images were obtained from an online database based on race of origin (White European, Oriental, Indian). They were then sorted into categories according to race of the subject. The dataset was preprocessed by taking the images and resizing them to maintain equal size before applying normalization to the images.

Dataset Organization: The images were arranged according to subcategories: Orientals Indians White Europeans. The three attributes of race were taken from the file names of the images and then grouped into these

categories. This step made certain that the model would feed on clean, well formatted data during training in order to learn efficiently.

Model Architecture Development: To drive the diffusion process UNet-based model was designed for the model. Since UNet has an encoder-decoder approach, it can be used effectively in image generation because the high-level features are obtained in the encoder, and the precise pixel precise forecast is enrolled in the decoder. A reasonable scheduler for the alteration of the diffusion process was also developed. This scheduler allows to manage the process of noise adding and removing at different timesteps for the further generation of new faces.

Model Training: For the model training purpose, the dataset was prepared where images included noise in them during the training process in order to mimic the process of image diffusion. A loss function was established in order to compute the difference between the predicted noise and the actual noise, and then an optimizer to minimize this loss. The model was trained through more than one epoch while tracking the loss and displaying it.

Generation of Synthetic Images: Once training was complete, images are synthesized for the racial mix of Orientals/Indians, Orientals/Europeans, Indian/Europeans, and Orientals/Indian/Europeans. The images that generated by this program were assessed based on the quality and the level of diversification and realism involved.

Evaluation and Reporting: The last activity involved the assessment of the operational worth of the model under development and data archiving. It gave characteristics of loss trends after being trained and tested, and a report was written to explain the methodology and results of the project followed by the plan of future work.

Image Collection

For this project, I collected images using Google Image Search, where I grouped images based on specific racial categories: Orientals, Indians and White Europeans. The search terms used accorded with these demographic groups, and every effort was made to choose faces that could represent each category of the search terms adequately. For this data preprocessing, the images were downloaded individually, and they contained labels in their filenames relative to the age, gender, and race of the subjects. It was possible to gather enough images for each group: the final database for each group contains 500 images.

Although Google Image Search was widely used for image collection, several publicly available datasets could also have been utilized for face image collection as well. Some of these include:

UTKFace Dataset: The UTK face database – a large-scale dataset that consists of face images compliant with age, gender, ethnicity put in here: https://susanqq.github.io/UTKFace/.

CelebA Dataset: Another dataset that shows celebrity images along with 40 labels such as gender and skin color is available on this link http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html.

LFW (Labeled Faces in the Wild): A face dataset in the wild obtained from the web especially for the lower face recognition tasks (http://vis-www.cs.umass.edu/lfw/).

Such datasets are employed for face recognition and demographic classification; however, for this task, images were collected from the internet, and then identified by the author, and employed for the generative model training.

Proposed Method

The method suggested for generating synthetic face images is a diffusion-based generative model. The input face images in the model are preprocessed, and are of size 64 x 64 RGB, labeled with race (White European, Oriental, and Indian) and gender, and a training dataset of faces is used, normalized for the model. The key component of the method is a UNet architecture adopted for image generation tasks because of its encoder-decoder structure.

The training phase entails applying noise to the images several times in order to indicate the diffusion steps. The model learns to solve this adding of noise, gradually moving back to the original image distribution. In this case, a custom diffusion scheduler has been used to manage both the addition and removal of noise for proper training.

The model generates synthetically images which are derived from the interpolation of the mentioned groups of demographics. The generated faces involve Oriental and Indians and Europeans, Indian and Europeans and Oriental, Indians and Europeans and Oriental, and Orientals and Indian and Europeans respectively to diversify the faces being generated. The quality of the generated images is assessed qualitatively with reference to the distribution of the loss function at the time of training.

Flows and Procedures: Implementation and Challenges

Diffusion based generative model was also successfully applied and was implemented in several steps with unique issues arising with each of them.

Data Collection and Preprocessing: In the first step, facial images were searched from Google Image Search and these images were then grouped into three racial groups by the author (White European, Oriental, and Indian). Labeling was a particularly important issue in this phase as the consistency of the different dataset was critical. It was periodically found that some images were improperly sorted or contained wrong tags. The images were preprocessed and resized in order to have a more uniform size that can be feed into the model.

Model Architecture Development: Since the task involved generating images, we opted for a UNet-based architecture due to its effectiveness in image generation tasks. During training, Gaussian noise was added to the images to simulate the diffusion process. The noise level was randomly selected within a specified range (between 0.05 and 0.2) to provide the model with varying degrees of difficulty in denoising. In initial runs aimed at training the model for facial image generation, the loss values were high, and the generated images lacked clarity. Adjustments were made to the noise level range and other training parameters, such as increasing the number of epochs and fine-tuning the learning rate, to improve the model's performance.

Training and Evaluation: Training loop identified involved making changes to the images by adding noise to it and minimizing the loss with actual noise. One of the main difficulties related to the model was the possibility of its capacity to produce good quality images, at the same time avoiding overfitting, as the training process was time consuming. It was considered that observing the trends of losses will facilitate understanding of whether the model is converging or not.

Generation of Synthetic Images: After training the model, I proceeded to generate synthetic images by blending images from different demographic groups. By averaging images from the Orientals, Indians, and Europeans datasets, I created mixed inputs that were then processed by the trained UNet model. The generated images exhibited a combination of features from the various groups. Over the course of developing this technique, the two main areas requiring attention were the model architecture and the adjustment of noise levels during training. Fine-tuning the UNet architecture and modifying the range of noise added to the inputs were crucial steps. By refining these aspects, I aimed to enhance the model's ability to generate a diverse set of synthetic faces that effectively blend characteristics from multiple demographics.

Program with proper Documentation

```python
# Import necessary libraries
import os
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms, utils
from torchvision.utils import make_grid
from PIL import Image
import matplotlib.pyplot as plt
from google.colab import drive

# Mounting Google Drive to save generated images directly
#drive.mount('/content/drive')

# Define dataset and output directories
dataset_directory = '/content/drive/MyDrive/Images-Dataset'
output_directory = '/content/drive/MyDrive/Generated_Mixed_Images'
os.makedirs(output_directory, exist_ok=True)

# Ensuring if the dataset directory exists; raise an error if not found
if not os.path.exists(dataset_directory):
    raise FileNotFoundError(f"The dataset path {dataset_directory} does not exist.")

# Custom Dataset Class
class ImageDataset(Dataset):
    def __init__(self, folder_path, transform=None):
        self.folder_path = folder_path
        self.image_list = [
            os.path.join(folder_path, img_file)
```

```
        self.image_list = [
            os.path.join(folder_path, img_file)
            for img_file in os.listdir(folder_path)
            if img_file.endswith(('png', 'jpg', 'jpeg'))
        ]
        self.transform = transform

    def __len__(self):
        return len(self.image_list)

    def __getitem__(self, index):
        image_path = self.image_list[index]
        image = Image.open(image_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image

# Define image transformations:
# Resize all images to 64x64 pixels and normalize pixel values to the range [-1, 1]
data_transforms = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load datasets for each demographic group (Orientals, Indians, Europeans).
# Each dataset contains images preprocessed using the defined transformations.
orientals_dataset_full = ImageDataset(os.path.join(dataset_directory, 'Orientals'), transform=data_transforms)
indians_dataset = ImageDataset(os.path.join(dataset_directory, 'Indians'), transform=data_transforms)
europeans_dataset = ImageDataset(os.path.join(dataset_directory, 'Europeans'), transform=data_transforms)
```

```
# Split the Orientals dataset into training (80%) and testing (20%) subsets
train_size = int(0.8 * len(orientals_dataset_full))
test_size = len(orientals_dataset_full) - train_size
orientals_train_dataset, orientals_test_dataset = random_split(orientals_dataset_full, [train_size, test_size])

# Create DataLoaders for training and testing
train_loader = DataLoader(orientals_train_dataset, batch_size=16, shuffle=True)
test_loader = DataLoader(orientals_test_dataset, batch_size=16, shuffle=False)

# Verify dataset sizes
print(f"Orientals dataset size: {len(orientals_dataset_full)}")
print(f"Training set size: {len(orientals_train_dataset)}")
print(f"Testing set size: {len(orientals_test_dataset)}")
print(f"Indians dataset size: {len(indians_dataset)}")
print(f"Europeans dataset size: {len(europeans_dataset)}")

# Define the UNet-based Diffusion Model:
# The model includes:
# - Encoder: Extracts high-level features from input images
# - Bottleneck: Processes these features to add/remove noise
# - Decoder: Reconstructs images from denoised features
class SimpleUNet(nn.Module):
    def __init__(self):
        super(SimpleUNet, self).__init__()

        # Encoder layers
        self.encoder_layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
```

```
            nn.ReLU(inplace=True)
        )
        self.pool1 = nn.MaxPool2d(2, 2)

        self.encoder_layer2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
        self.pool2 = nn.MaxPool2d(2, 2)

        # Bottleneck layer
        self.bottleneck_layer = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        # Decoder layers
        self.upconv2 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.decoder_layer2 = nn.Sequential(
            nn.Conv2d(256, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        self.upconv1 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.decoder_layer1 = nn.Sequential(
```

```
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )

        self.output_layer = nn.Conv2d(64, 3, kernel_size=1)

    def forward(self, x):
        # Encoding path
        enc1 = self.encoder_layer1(x)
        enc2 = self.encoder_layer2(self.pool1(enc1))

        # Bottleneck
        bottleneck = self.bottleneck_layer(self.pool2(enc2))

        # Decoding path
        dec2 = self.upconv2(bottleneck)
        dec2 = torch.cat((dec2, enc2), dim=1)
        dec2 = self.decoder_layer2(dec2)

        dec1 = self.upconv1(dec2)
        dec1 = torch.cat((dec1, enc1), dim=1)
        dec1 = self.decoder_layer1(dec1)

        # Output layer:
        # Reduces the final feature map to a 3-channel RGB image
        output = self.output_layer(dec1)
        return output
```

```
# Set device (CPU or GPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Initialize the model, loss function, and optimizer
model = SimpleUNet().to(device)
loss_function = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)  # Smaller learning rate

# Training Loop:
# The model is trained for `num_epochs` iterations to learn the denoising process.
# Loss values are computed for both training and testing data.
# Loss curves are logged to visualize model performance over time.
num_epochs = 100  # Number of epochs as needed
train_loss_history = []
test_loss_history = []

for epoch in range(num_epochs):
    # Training phase
    model.train()
    total_train_loss = 0
    for batch_images in train_loader:
        batch_images = batch_images.to(device)
        # Add Gaussian noise to the input images:
        # The noise is randomly scaled between 0.05 and 0.2 to simulate the diffusion process.
        noise = torch.randn_like(batch_images) * np.random.uniform(0.05, 0.2)
        noisy_inputs = batch_images + noise

        # Forward pass
        denoised_outputs = model(noisy_inputs)
```

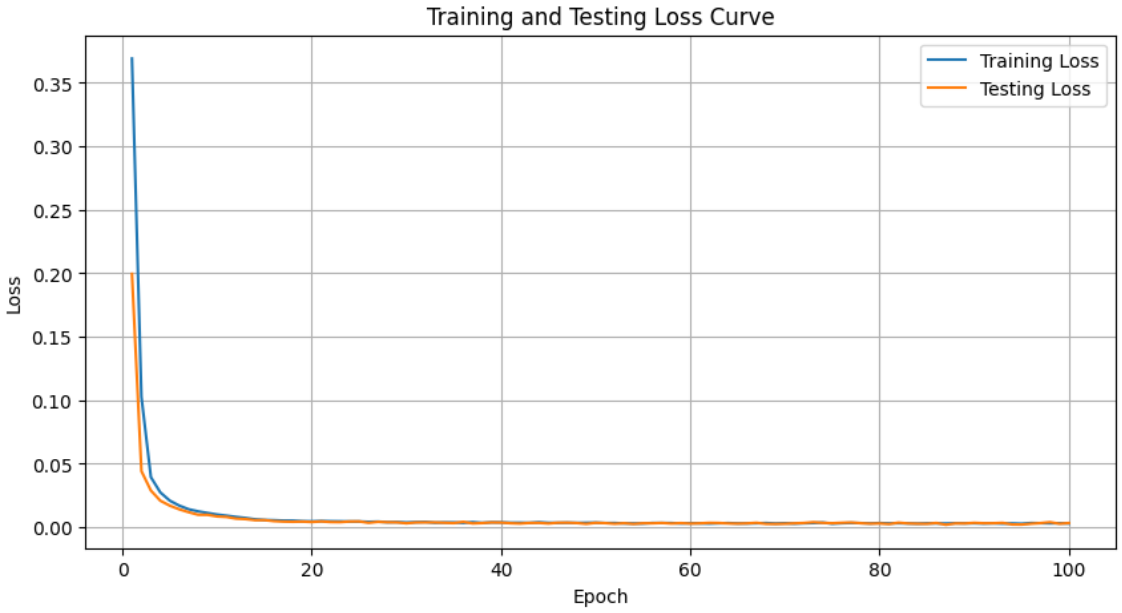
```
        # Compute Mean Squared Error (MSE) loss:
        # It measures the difference between the model's denoised output and the original clean image.
        loss = loss_function(denoised_outputs, batch_images)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_train_loss += loss.item()

    average_train_loss = total_train_loss / len(train_loader)
    train_loss_history.append(average_train_loss)

    # Testing phase
    model.eval()
    total_test_loss = 0
    with torch.no_grad():
        for batch_images in test_loader:
            batch_images = batch_images.to(device)
            # Adding random noise to the images
            noise = torch.randn_like(batch_images) * np.random.uniform(0.05, 0.2)
            noisy_inputs = batch_images + noise

            # Forward pass
            denoised_outputs = model(noisy_inputs)

            # Compute the loss
            loss = loss_function(denoised_outputs, batch_images)
            total_test_loss += loss.item()
```

```
        average_test_loss = total_test_loss / len(test_loader)
        test_loss_history.append(average_test_loss)

        print(f"Epoch [{epoch + 1}/{num_epochs}], "
              f"Training Loss: {average_train_loss:.4f}, "
              f"Testing Loss: {average_test_loss:.4f}")

    # Plotting of training and testing loss curves
    plt.figure(figsize=(10, 5))
    plt.plot(range(1, num_epochs + 1), train_loss_history, label='Training Loss')
    plt.plot(range(1, num_epochs + 1), test_loss_history, label='Testing Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Testing Loss Curve')
    plt.grid(True)
    plt.legend()
    plt.show()

    # Function: `generate_and_save_images`
    # This function takes datasets of images and generates blended outputs by:
    # 1. Randomly selecting images from each dataset
    # 2. Mixing features of the selected images (averaging pixel values)
    # 3. Passing the mixed input through the trained model to generate denoised images
    # The function also:
    # - Saves generated images in Google Drive for future use
    # - Displays them as a grid in the notebook
    def generate_and_save_images(model, dataset_a, dataset_b=None, dataset_c=None, title=""):
        model.eval()
        generated_images = []
```

```
        # Create a subfolder in the output directory to store generated images for each combination.
        # The folder name is based on the title of the combination (e.g., Orientals/Indians).
        combination_folder = os.path.join(output_directory, title.replace('/', '_'))
        os.makedirs(combination_folder, exist_ok=True)

        for idx in range(10):
            img_a = dataset_a[np.random.randint(0, len(dataset_a))].unsqueeze(0).to(device)
            if dataset_b:
                img_b = dataset_b[np.random.randint(0, len(dataset_b))].unsqueeze(0).to(device)
            else:
                img_b = 0

            if dataset_c:
                img_c = dataset_c[np.random.randint(0, len(dataset_c))].unsqueeze(0).to(device)
            else:
                img_c = 0

            # Mixing of the images
            num_images = 1 + (dataset_b is not None) + (dataset_c is not None)
            mixed_input = (img_a + img_b + img_c) / num_images

            # Generation of the output image
            with torch.no_grad():
                generated_output = model(mixed_input).squeeze(0).cpu()
                generated_images.append(generated_output)

                # Unnormalize and save the image
                image = generated_output.clone()
                image = (image * 0.5) + 0.5  # Unnormalize from [-1, 1] to [0, 1]
                image = image.clamp(0, 1)
```

```
                image = transforms.ToPILImage()(image)
                image.save(os.path.join(combination_folder, f"generated_image_{idx + 1}.png"))

        # Create a grid of images
        image_grid = make_grid(generated_images, nrow=5, normalize=True, scale_each=True)
        plt.figure(figsize=(15, 15))
        plt.imshow(np.transpose(image_grid, (1, 2, 0)))
        plt.title(title)
        plt.axis('off')
        plt.show()

    # Generate and Save Synthetic Images:
    # For each combination of demographic groups, the function:
    # - Randomly selects images from the input datasets
    # - Generates mixed images using the trained diffusion model
    # - Saves the generated images to Google Drive
    # - Displays the generated images in a grid format
    # Orientals + Indians
    generate_and_save_images(model, orientals_dataset_full, indians_dataset, title="Orientals/Indians")

    # Orientals + Europeans
    generate_and_save_images(model, orientals_dataset_full, europeans_dataset, title="Orientals/Europeans")

    # Indians + Europeans
    generate_and_save_images(model, indians_dataset, europeans_dataset, title="Indians/Europeans")

    # Orientals + Indians + Europeans
    generate_and_save_images(model, orientals_dataset_full, indians_dataset, europeans_dataset, title="Orientals/Indians/Europeans")

    print("Image generation and saving completed.")
```

Results - Plot of Training/Testing Accuracy and the Generated Images



The results of this project focus on two main aspects: the evolution of training and testing loss and the quality of the generated images.

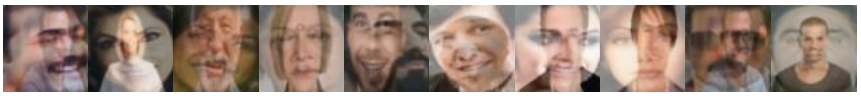
Figure 1 illustrates the training and testing loss curves over 100 epochs, as measured by the Mean Squared Error (MSE) loss. At the beginning of training, the loss started at a high value (~0.36), with a rapid decrease in both training and testing loss within the first 20 epochs. By the 20th epoch, both losses had significantly stabilized, with testing loss closely following the training loss throughout the process. This indicates that the model learned to predict and remove noise effectively during training without signs of overfitting.

By the 100th epoch, the training and testing losses converged to near-zero values (~0.005), indicating that the model successfully learned the diffusion process for generating synthetic images. The close alignment of the training and testing loss curves suggests that the model generalized well to unseen data during the testing phase.

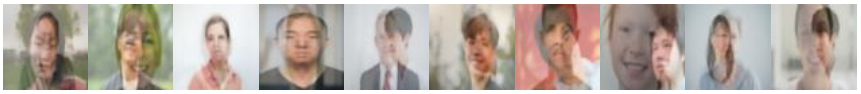
The generated images represent combinations of demographics such as Orientals/Indians, Orientals/Europeans, Indians/Europeans, and Orientals/Indians/Europeans. The images display realistic blending of features across demographic groups, with distinct visual characteristics from each group preserved in the outputs. The training loss curve also demonstrates the model's stability during training. While the generated images are visually coherent and show improved quality, minor artifacts and less-defined facial details in some cases highlight the potential need for further architectural or parameter tuning.

This outcome confirms the model's capability to generate synthetic images effectively while maintaining demographic diversity. Future work may involve additional fine-tuning of the noise addition process and further dataset augmentation to enhance image sharpness and diversity.

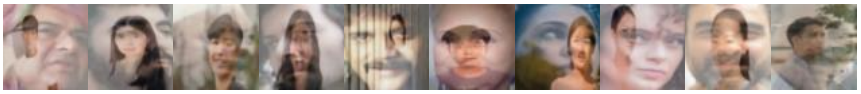
Indians Europeans –



Orientals Europeans –



Orientals Indians –



Orientals Indians Europeans –

Conclusion

The goal of this project was to create a diffusion-based generative model capable of generating synthetic face images by blending features from different demographic groups, such as Orientals, Indians, and Europeans. The training process successfully demonstrated a gradual reduction in loss values, as reflected in both the training and testing loss curves, confirming that the model effectively learned the process of denoising. However, while the generated images showed a distinct mix of demographic features, some outputs lacked sharpness and realistic facial details, indicating the need for further optimization of the generative process.

Several improvements could enhance the current model's performance. Fine-tuning the UNet architecture, such as experimenting with different layer configurations or filters, could yield better image clarity. Additionally, increasing the dataset size and diversity would enable the model to capture finer details of facial structures across demographics. Adjustments to the noise addition process, training duration (e.g., increasing the number of epochs), and hyperparameters such as learning rate may also improve image quality.

Despite these challenges, the results confirm that diffusion-based generative models are a promising approach for synthetic image generation. The training process demonstrated stability, and the generated outputs provided a basis for future improvements. This project highlights the potential of diffusion models in applications such as media content generation and ethical uses like facial recognition. With additional modifications and optimizations, the model has the potential to produce sharper, more realistic, and diverse synthetic face images, advancing the possibilities in generative modeling.