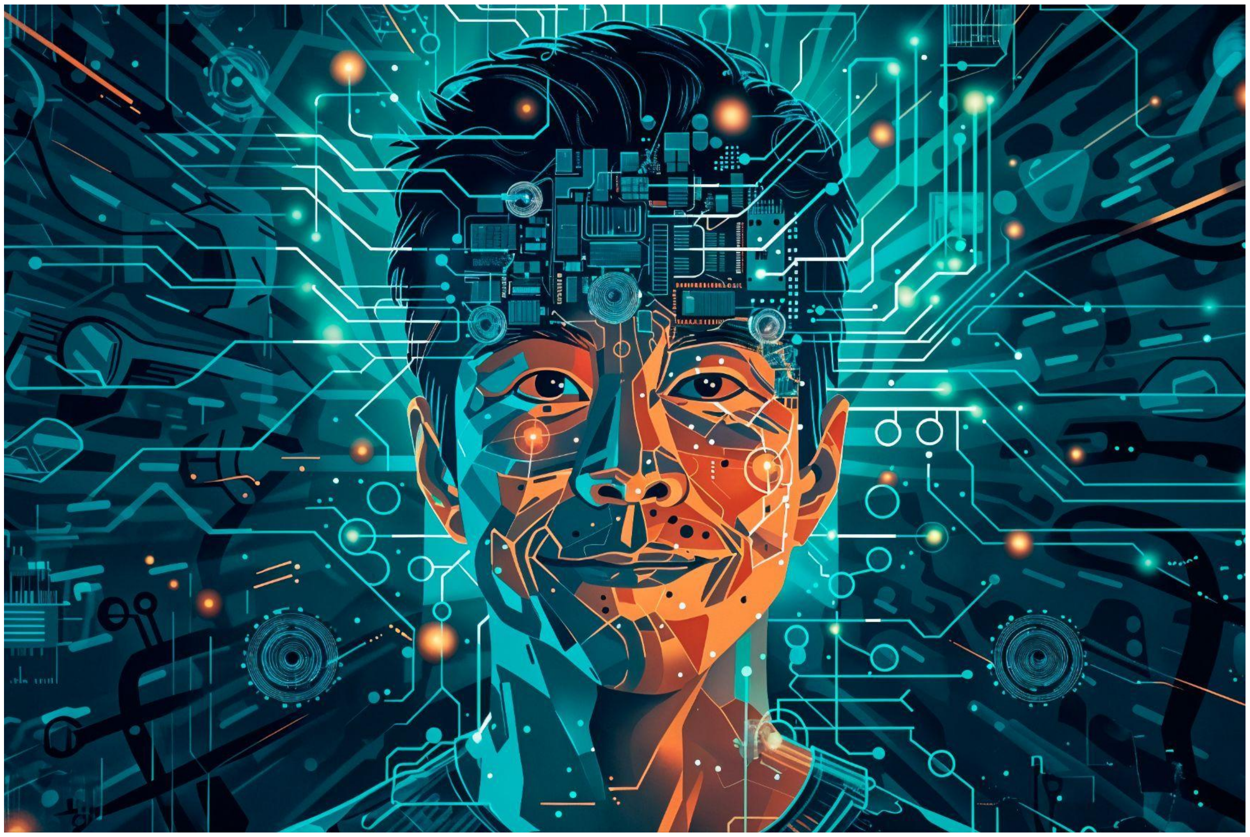


FaceFusion: Blending Features from Different Ethnic Groups

—

Madhurachanna Halayya Bellad

Sai Shailesh Gadde

Index

Introduction

This report outlines the development and implementation of a face generation system using a diffusion-based model. The project aims to create new faces by blending features from three ethnic groups: Orientals, Indians, and White Europeans. Leveraging deep learning techniques, specifically conditional diffusion models, the system will generate realistic faces by learning and recombining features from an image dataset. The model will produce 10 new faces for each ethnic combination and allow for adjustable class proportions, enabling the creation of faces that represent multiple ethnic groups.

Project Description

The project consists of three main steps:

1. **Data Preparation:** Collecting 500 to 1000 images from the three ethnic groups: Orientals, Indians, and white Europeans.
2. **Model Development:** Training a conditional diffusion model to generate new faces by combining features from these groups.
3. **Image Generation:** Create 10 images for each combination—Orientals/Indians, Orientals/Europeans, Indians/Europeans, and all three groups combined—to showcase diversity.

The project will be evaluated on its ability to generate realistic faces and effectively combine features across the groups.

Summary of Completed Tasks

The tasks completed in this project include:

1. **Building the Architecture**: Designed a custom **UNet-based diffusion model** for generating images. This model takes class information as an additional input.
2. **Class Conditioning**: Converted class labels (Orientals, Indians, Europeans) into numerical embeddings using `nn.Embedding`. These embeddings were added as extra channels to the image input. Class conditioning is needed to guide the model in generating images specific to a desired class or combination of classes, enabling controlled and meaningful outputs.

3. **Training the Model**: The model learned to generate realistic faces by combining features from different classes through a step-by-step process of refining noisy images into clear ones.
4. **Output Channels**: The model output was a 64×64 image with three color channels (RGB).
5. **Evaluation and Results:** Generated images were evaluated for their realism and diversity. The results were visualized and analyzed to determine the model's performance.

Image Collection Method

The images were collected from **four free and open platforms**, including Kaggle and Roboflow, to ensure diversity, quality, and representation across ethnic groups.
The dataset from Roboflow was downloaded in the **YOLO format**. **OpenCV** was used to detect and crop the facial region from the images. This ensured that only the face was retained, removing any unnecessary background.
For Indian:

https://www.kaggle.com/datasets/iamsouravbanerjee/indian-actor-images-dataset?resource=download

For Orientals:

https://www.kaggle.com/datasets/lukexng/aisanfaces

https://app.roboflow.com/indianfaccedataset/behavioral-analysis-cnxiz/1

For European:

https://universe.roboflow.com/ai-hackathon-face-recognition/caucasian-mid-light

https://www.kaggle.com/datasets/axondata/nist-dataset-3m-unique-biometric-faces

Proposed Method

The proposed method will generate new faces using a custom-built conditional diffusion model instead of a pre-trained model like Stable Diffusion. The model will combine features from different ethnic groups, such as Orientals, Indians, and Europeans.

Model Design:

The model developed in this work is a Class-Conditional UNet-based diffusion model, with class conditioning that enables controlled face generation. The class labels, such as Oriental, Indian, and European, are embedded using learned embeddings and then concatenated with image features to control the generation process. This model thus can generate faces by mixing and matching facial features across ethnicities, which are dictated by the input class conditioning.

Data:

The model is trained on a diverse dataset of open-source data from Kaggle and Roboflow. *OpenCV* was used to detect and crop the facial region from the images. This ensured that only the face was retained, removing any unnecessary background.

Face Generation:

The model, after being trained, synthesizes new faces by incorporating features from different ethnicities based on textual prompts (e.g., "Oriental and Indian face features" or "European and Indian combination"). Thus, the model produces results with a face having a proper proportion of ethnic features and allows controlled, realistic, and diverse face generation. It ensures that the generated faces maintain high quality while reflecting a realistic mixture of features from the selected ethnic groups.

Implementation Details

1. Image Resizing and Normalization

Procedure:

All the collected images were resized to a standard resolution (64 × 64 pixels) due to computational constraints. Although higher resolutions would provide better details, the adjustment was necessary due to hardware limitations.

Challenges:

Dataset Imbalance: The Orientals dataset had relatively clean images, while there were significant issues in the European and Indian datasets.

Background Noise: In the European dataset, the background details were mostly long when compared to the required face region, which caused the model to focus not only on facial features but also on the irrelevant elements. A similar problem was there in the Indian dataset.

Solution:

Using the annotations provided by Roboflow, OpenCV was used to crop out the face region and discard the background. This preprocessing improved the model's ability to focus more on the face features themselves, thus giving good results.

2. Model Training

Model Choice:

A Class-Conditional UNet was chosen to be trained. Given the task of label mapping, the architecture was ideal for the generation of images conditioned on a particular class label. Diffusion models are considered to be among the powerful tools for generating images; thus, UNet formed their core.

Training:

The model was trained on the preprocessed dataset based on the following considerations:

Learning rate, batch size, and number of epochs are the tuned hyperparameters, in view of balancing the training time and accuracy.

Noise was reduced as much as possible in the dataset so that the model learns only from facial areas.

Challenges:

Results were highly noisy; this required several reworks.

A good model quality improvement in performance and stability needed fine-tuning on the learning rate, batch sizes, and reduction of noise.

3. Image Generation

Process:

The model is then able to create images after training by specifying class percentages. Example:

You can create an image that has 50% of Class 1 features (Orientals) and 50% Class 2 features (Indian). You can also combine all three classes, such as 40% Orientals, 30% Indian, and 30% European.

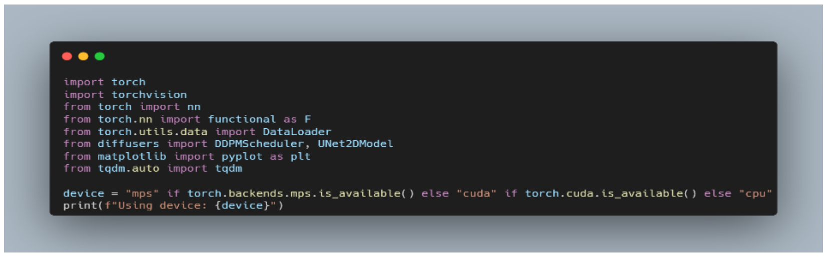
Challenges:

Initially, the encoding of prompts and how they would be integrated into the model were problematic.

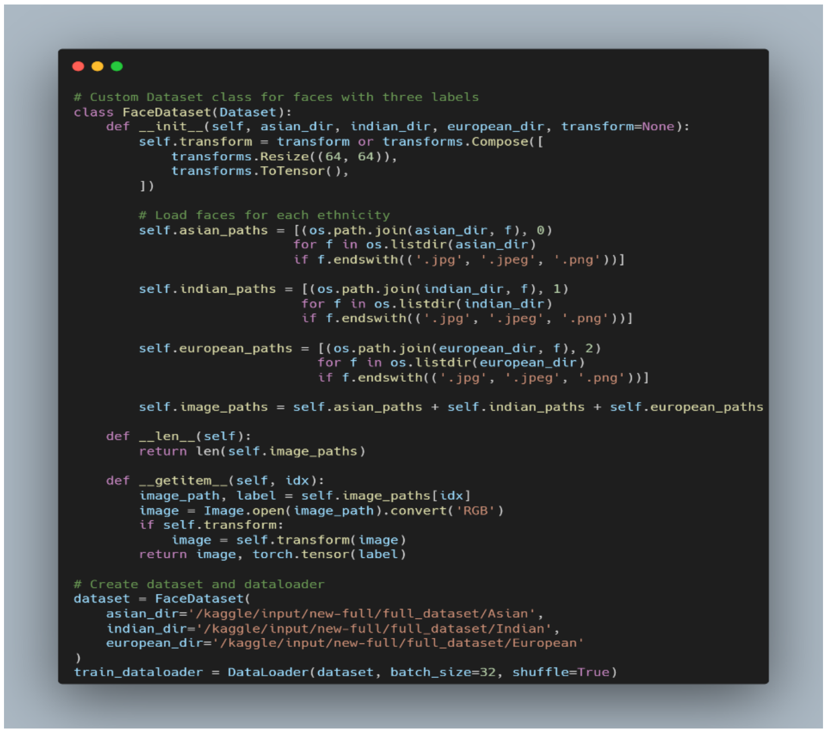
Solution: Class labels were then encoded and input to the model through the Class-Conditional UNet, ensuring that the model leverages label information effectively during generation.

Program Code

Setup and Loading the dataset :

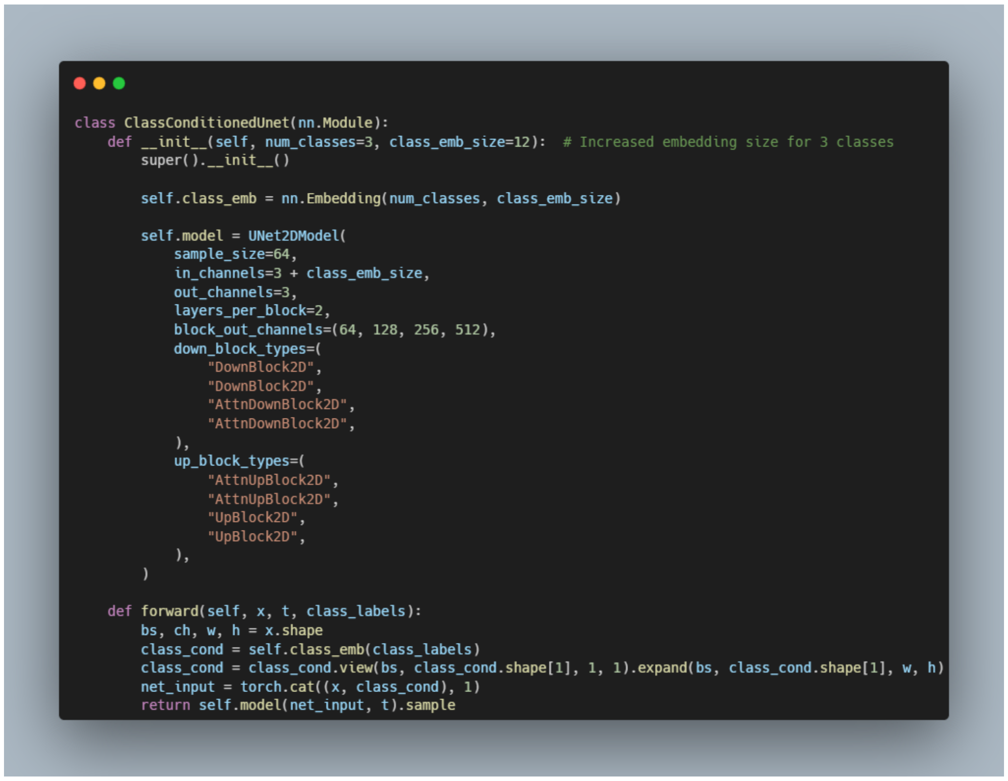


```python
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from torch.utils.data import DataLoader
from diffusers import DDPMScheduler, UNet2DModel
from matplotlib import pyplot as plt
from tqdm.auto import tqdm

device = "mps" if torch.backends.mps.is_available() else "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")
```

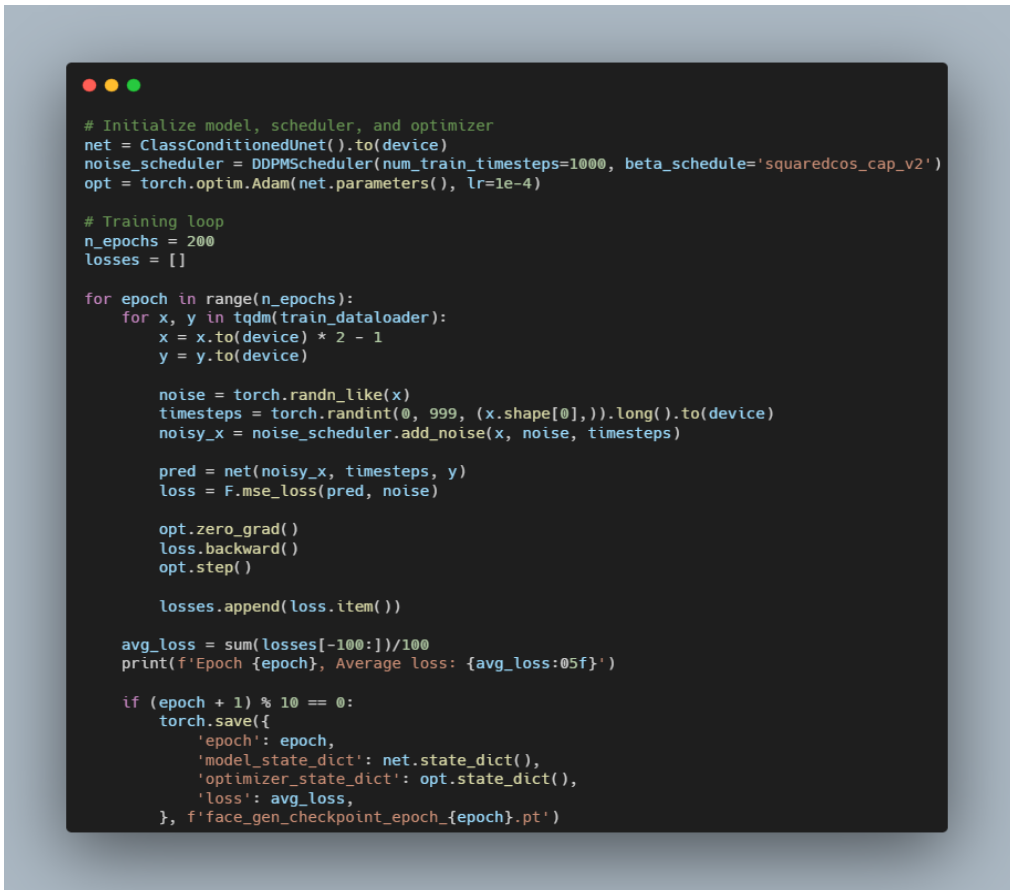


```python
# Custom Dataset class for faces with three labels
class FaceDataset(Dataset):
    def __init__(self, asian_dir, indian_dir, european_dir, transform=None):
        self.transform = transform or transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
        ])

        # Load faces for each ethnicity
        self.asian_paths = [(os.path.join(asian_dir, f), 0)
                           for f in os.listdir(asian_dir)
                           if f.endswith(('.jpg', '.jpeg', '.png'))]

        self.indian_paths = [(os.path.join(indian_dir, f), 1)
                            for f in os.listdir(indian_dir)
                            if f.endswith(('.jpg', '.jpeg', '.png'))]

        self.european_paths = [(os.path.join(european_dir, f), 2)
                              for f in os.listdir(european_dir)
                              if f.endswith(('.jpg', '.jpeg', '.png'))]

        self.image_paths = self.asian_paths + self.indian_paths + self.european_paths

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

    def __getitem__(self, idx):
        image_path, label = self.image_paths[idx]
        image = Image.open(image_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, torch.tensor(label)

# Create dataset and dataloader
dataset = FaceDataset(
    asian_dir='/kaggle/input/new-full/full_dataset/Asian',
    indian_dir='/kaggle/input/new-full/full_dataset/Indian',
    european_dir='/kaggle/input/new-full/full_dataset/European'
)
train_dataloader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Create a Class Conditioned Unet with Training & Sampling :

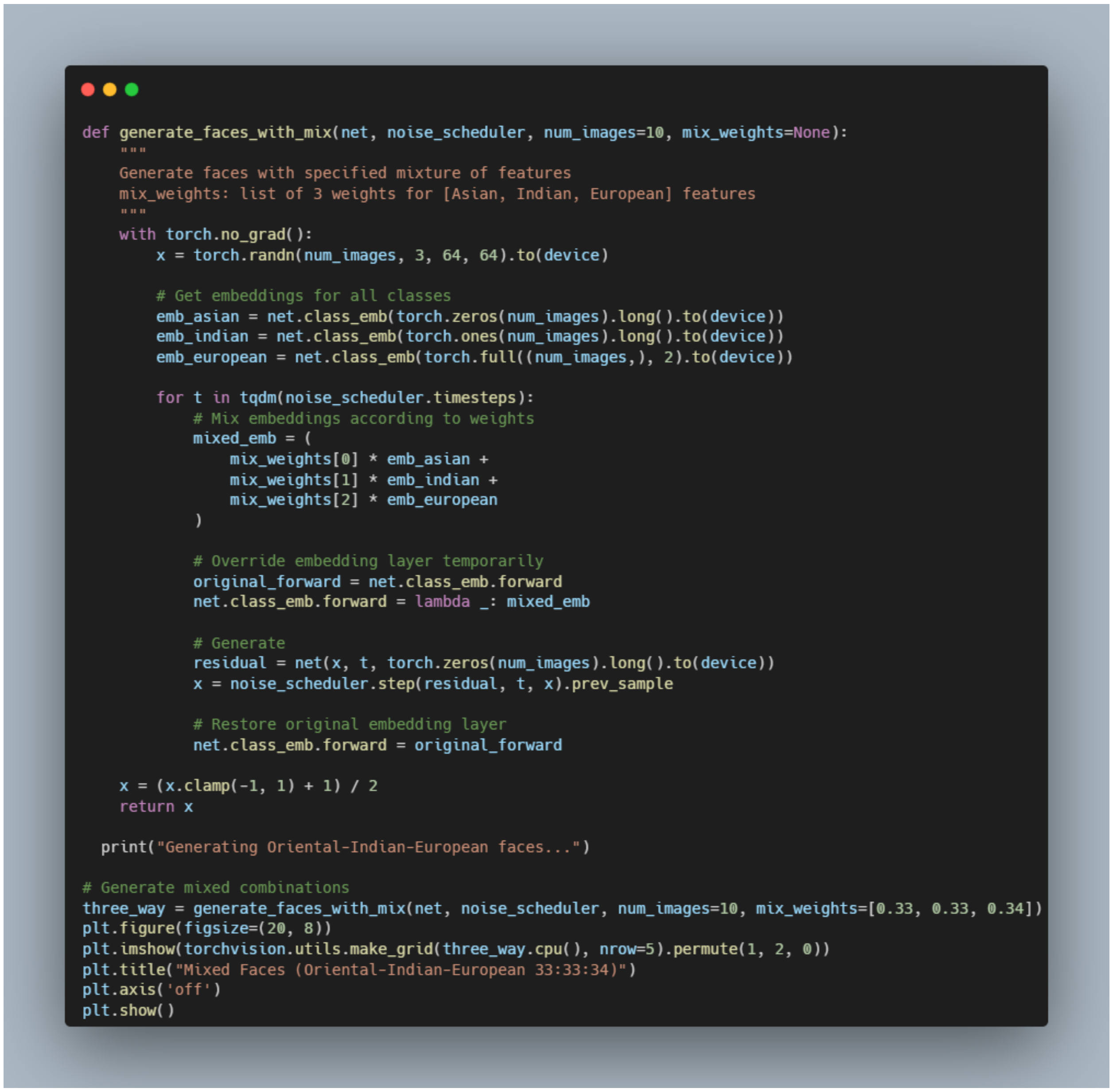


```python
class ClassConditionedUnet(nn.Module):
    def __init__(self, num_classes=3, class_emb_size=12):  # Increased embedding size for 3 classes
        super().__init__()

        self.class_emb = nn.Embedding(num_classes, class_emb_size)

        self.model = UNet2DModel(
            sample_size=64,
            in_channels=3 + class_emb_size,
            out_channels=3,
            layers_per_block=2,
            block_out_channels=(64, 128, 256, 512),
            down_block_types=(
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",
                "AttnDownBlock2D",
            ),
            up_block_types=(
                "AttnUpBlock2D",
                "AttnUpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ),
        )

    def forward(self, x, t, class_labels):
        bs, ch, w, h = x.shape
        class_cond = self.class_emb(class_labels)
        class_cond = class_cond.view(bs, class_cond.shape[1], 1, 1).expand(bs, class_cond.shape[1], w, h)
        net_input = torch.cat((x, class_cond), 1)
        return self.model(net_input, t).sample
```



```python
# Initialize model, scheduler, and optimizer
net = ClassConditionedUnet().to(device)
noise_scheduler = DDPMScheduler(num_train_timesteps=1000, beta_schedule='squaredcos_cap_v2')
opt = torch.optim.Adam(net.parameters(), lr=1e-4)

# Training loop
n_epochs = 200
losses = []

for epoch in range(n_epochs):
    for x, y in tqdm(train_dataloader):
        x = x.to(device) * 2 - 1
        y = y.to(device)

        noise = torch.randn_like(x)
        timesteps = torch.randint(0, 999, (x.shape[0],)).long().to(device)
        noisy_x = noise_scheduler.add_noise(x, noise, timesteps)

        pred = net(noisy_x, timesteps, y)
        loss = F.mse_loss(pred, noise)

        opt.zero_grad()
        loss.backward()
        opt.step()

        losses.append(loss.item())

    avg_loss = sum(losses[-100:])/100
    print(f'Epoch {epoch}, Average loss: {avg_loss:05f}')

    if (epoch + 1) % 10 == 0:
        torch.save({
            'epoch': epoch,
            'model_state_dict': net.state_dict(),
            'optimizer_state_dict': opt.state_dict(),
            'loss': avg_loss,
        }, f'face_gen_checkpoint_epoch_{epoch}.pt')
```

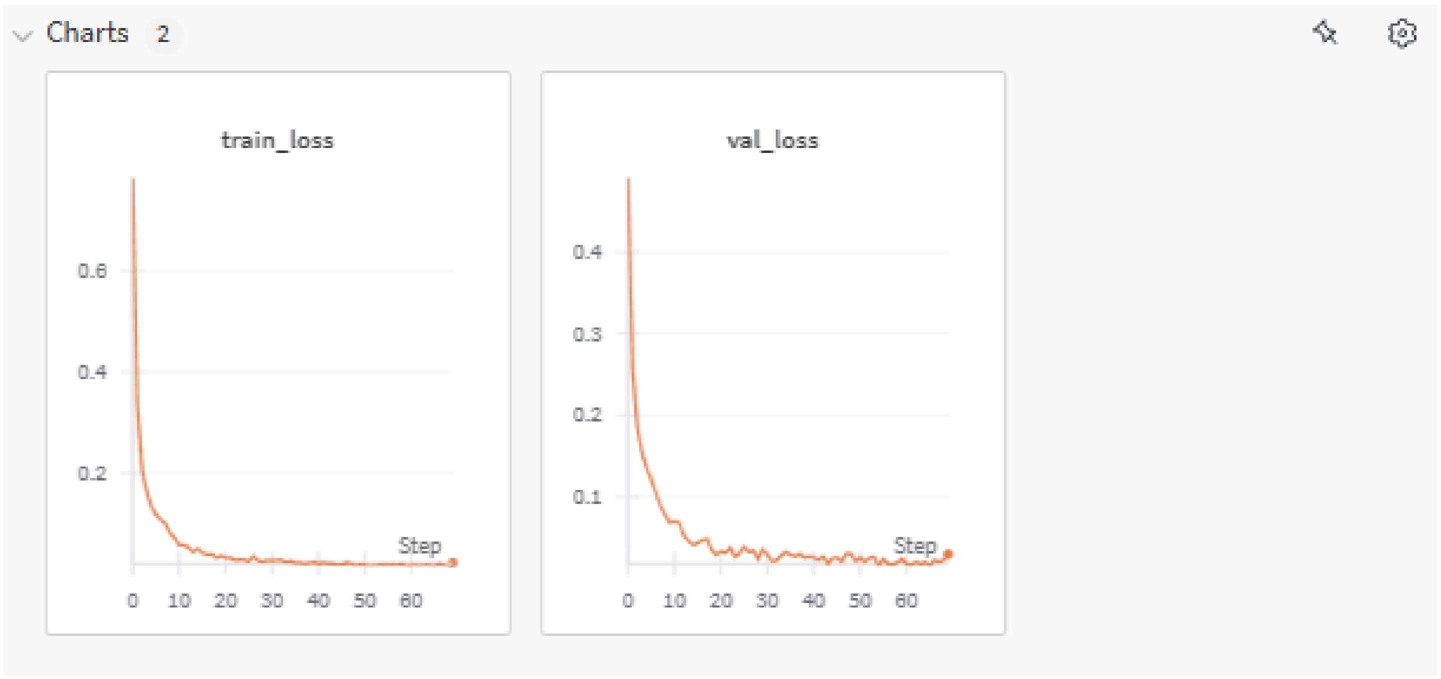
Generate Faces with different Classes / Ethnicities :



```python
def generate_faces_with_mix(net, noise_scheduler, num_images=10, mix_weights=None):
    """
    Generate faces with specified mixture of features
    mix_weights: list of 3 weights for [Asian, Indian, European] features
    """
    with torch.no_grad():
        x = torch.randn(num_images, 3, 64, 64).to(device)

        # Get embeddings for all classes
        emb_asian = net.class_emb(torch.zeros(num_images).long().to(device))
        emb_indian = net.class_emb(torch.ones(num_images).long().to(device))
        emb_european = net.class_emb(torch.full((num_images,), 2).to(device))

        for t in tqdm(noise_scheduler.timesteps):
            # Mix embeddings according to weights
            mixed_emb = (
                mix_weights[0] * emb_asian +
                mix_weights[1] * emb_indian +
                mix_weights[2] * emb_european
            )

            # Override embedding layer temporarily
            original_forward = net.class_emb.forward
            net.class_emb.forward = lambda _: mixed_emb

            # Generate
            residual = net(x, t, torch.zeros(num_images).long().to(device))
            x = noise_scheduler.step(residual, t, x).prev_sample

            # Restore original embedding layer
            net.class_emb.forward = original_forward

    x = (x.clamp(-1, 1) + 1) / 2
    return x

  print("Generating Oriental-Indian-European faces...")

# Generate mixed combinations
three_way = generate_faces_with_mix(net, noise_scheduler, num_images=10, mix_weights=[0.33, 0.33, 0.34])
plt.figure(figsize=(20, 8))
plt.imshow(torchvision.utils.make_grid(three_way.cpu(), nrow=5).permute(1, 2, 0))
plt.title("Mixed Faces (Oriental-Indian-European 33:33:34)")
plt.axis('off')
plt.show()
```

Results

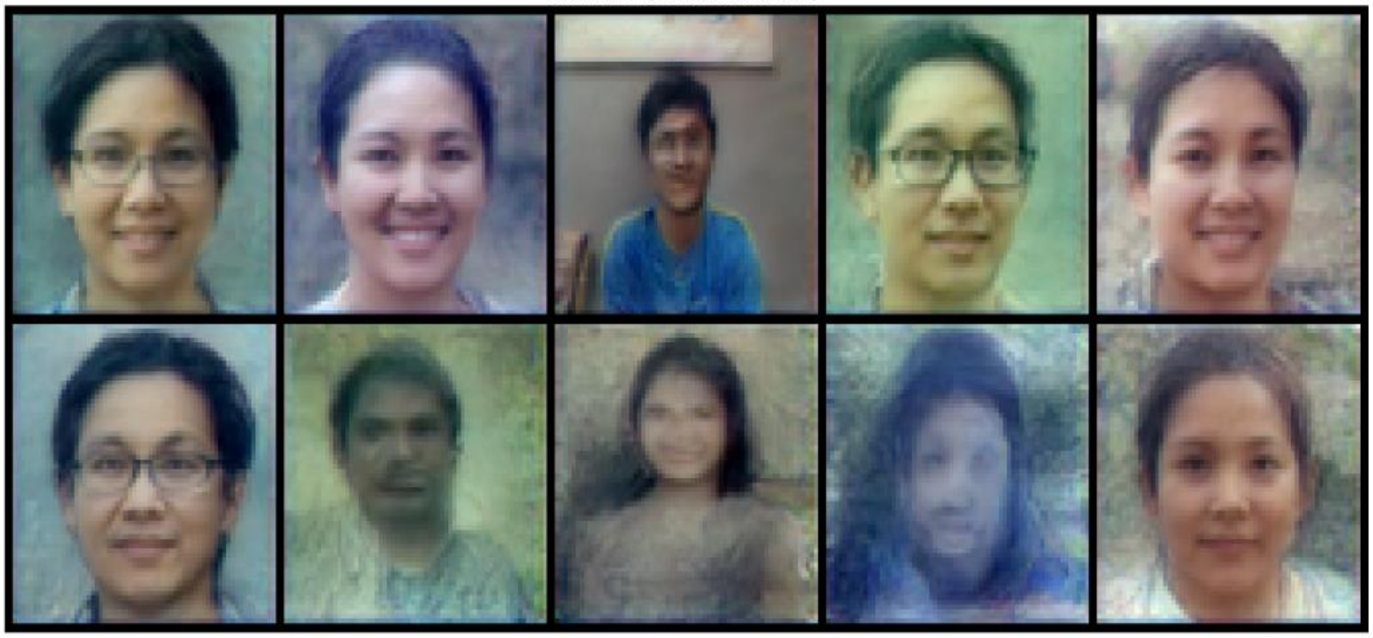
Training and Validation Loss



Orientals/Indians

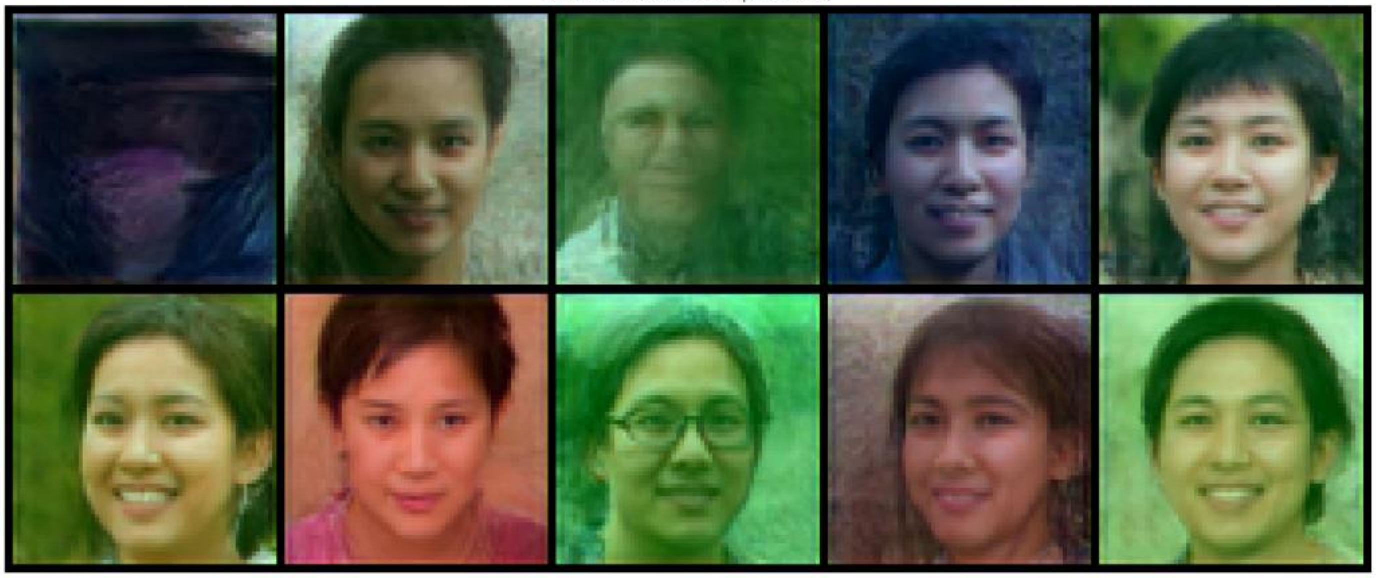
```
Generating Oriental-Indian faces...
Loading widget...
```

Mixed Faces (Oriental-Indian 50:50)



Orientals/Europeans

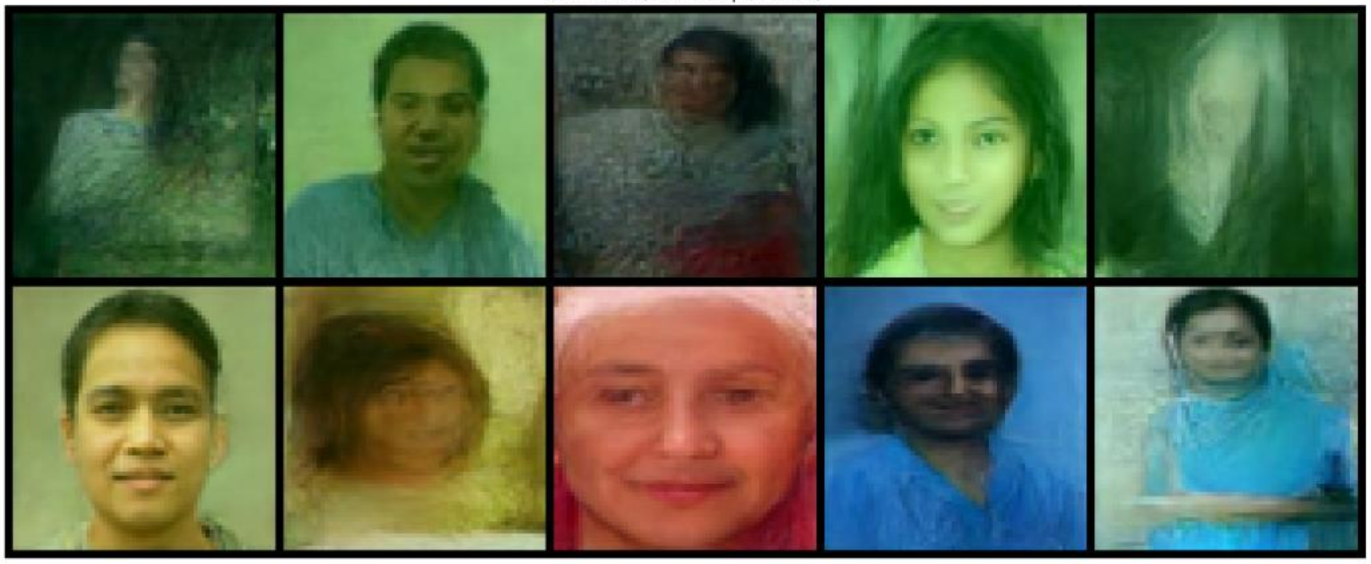
Mixed Faces (Oriental-European 50:50)



Indian/Europeans

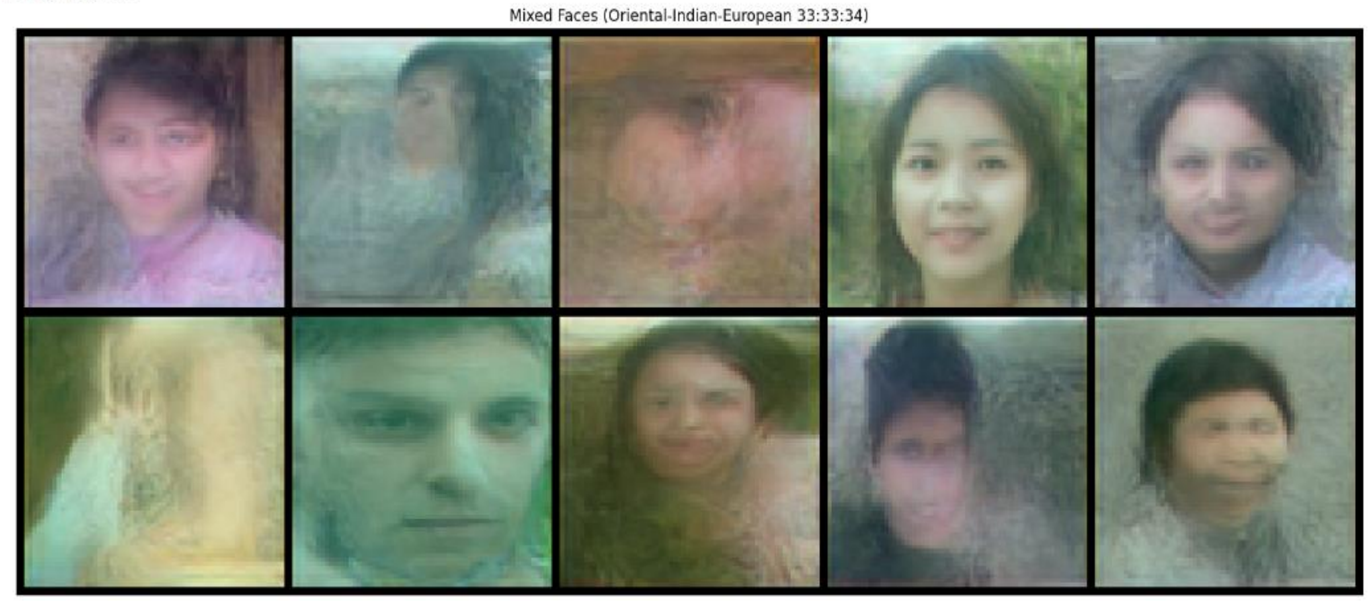
```
 Generating Indian-European faces...
Loading widget...
```

Mixed Faces (Indian-European 80:20)



Orientals/Indians/Europeans

```
Generating Oriental-Indian-European faces...
Loading widget...
```



Conclusion

This project demonstrates the integration of conditional generative models and diffusion processes to synthesize high-quality, class-specific facial images. By leveraging UNet-based architectures and class embeddings, we generated diverse and realistic images, including mixed-ethnicity combinations. The model's capability to incorporate nuanced class information showcases its potential in creative content generation and synthetic data augmentation. Work in the future may investigate increased image resolution, richer class conditioning, and real-world applications in fields such as entertainment, health, and reduction of bias in AI systems.