Code

Annotate_MTCNN.ipynb

```
!pip install mtcnn
!pip install pillow

from mtcnn import MTCNN
from PIL import Image, ImageDraw
import os
import numpy as np

def draw_bounding_boxes(image_path, result_list):
    data = Image.open(image_path)
    draw = ImageDraw.Draw(data)
    for result in result_list:
        x, y, width, height = result['box']
        rect = [(x, y), (x + width, y + height)]
        draw.rectangle(rect, outline="red", width=3)

    return data

detector = MTCNN()

def process_directory(input_dir, output_dir):
    for root, dirs, files in os.walk(input_dir):
        for file in files:
            if file.lower().endswith(('.png', '.jpg', '.jpeg')):
                path = os.path.join(root, file)
                try:
                    with Image.open(path) as img:
                        pixels = np.array(img)
                        faces = detector.detect_faces(pixels)
                        if faces:
                            image_with_boxes = draw_bounding_boxes(path, faces)
                            relative_path = os.path.relpath(root, input_dir)
                            save_dir = os.path.join(output_dir, relative_path)
                            if not os.path.exists(save_dir):
                                os.makedirs(save_dir)
                            output_path = os.path.join(save_dir, file)
                            image_with_boxes.save(output_path)
```

```
            except Exception as e:
                print(f"An error occurred with image {path}: {e}")

input_directory = "/content/drive/MyDrive/ArcFace/data/raw"
output_directory = "/content/drive/MyDrive/ArcFace/data/processed"

process_directory(input_directory, output_directory)

print("All images have been processed.")
```

Create_Labels.ipynb

```
import os
import csv

def get_images_and_labels(root_dir):
    images = []
    labels = []
    label_dict = {}
    current_label = 0
    for person_name in sorted(os.listdir(root_dir)):
        class_folder = os.path.join(root_dir, person_name)
        if os.path.isdir(class_folder):
            for img_file in sorted(os.listdir(class_folder)):
                img_path = os.path.join(class_folder, img_file)
                images.append(img_path)
                if person_name not in label_dict:
                    label_dict[person_name] = current_label
                    current_label += 1
                labels.append(label_dict[person_name])
    return images, labels

#root_dir = '/content/drive/MyDrive/Shruti Project/data/processed'
#images, labels = get_images_and_labels(root_dir)

def save_images_and_labels_to_csv(images, labels, csv_filename):
    with open(csv_filename, 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['image_path', 'label'])
```

```
        for img_path, label in zip(images, labels):
            writer.writerow([img_path, label])

root_dir = '/content/drive/MyDrive/ArcFace/data/processed'
images, labels = get_images_and_labels(root_dir)

csv_filename = '/content/drive/MyDrive/ArcFace/data/dataset_labels.csv'
save_images_and_labels_to_csv(images, labels, csv_filename)
print(f"Saved image paths and labels to {csv_filename}")
```

Update1.ipynb

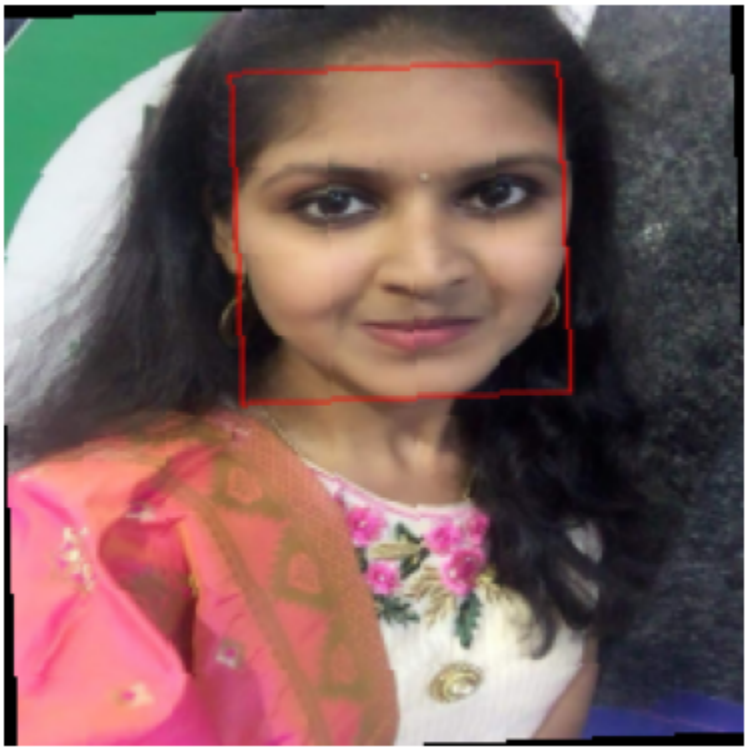
```
import os
import pandas as pd
import matplotlib.pyplot as plt
from torchvision.io import read_image
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader, random_split

class CustomDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform

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

    def __getitem__(self, idx):
      img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
      image = read_image(img_path).float() / 255.
      label = self.img_labels.iloc[idx, 1]
      if self.transform:
        image = self.transform(image)
      return image, label

transform = transforms.Compose([
    transforms.Resize((224, 224), antialias=True),
    transforms.RandomHorizontalFlip(p=0.5),
```

```
    transforms.RandomRotation(10),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

dataset =
CustomDataset(annotations_file='/content/drive/MyDrive/ArcFace/data/dataset_labels.c
sv', img_dir='/content/drive/MyDrive/ArcFace/data/processed', transform=transform)

train_val_split = int(len(dataset) * 0.9)
test_split = len(dataset) - train_val_split
train_val_dataset, test_dataset = random_split(dataset, [train_val_split, test_split])

train_split = int(train_val_split * 0.88)
val_split = train_val_split - train_split
train_dataset, val_dataset = random_split(train_val_dataset, [train_split, val_split])

batch_size = 8
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)

import matplotlib.pyplot as plt
import torch

def show_image(img, mean, std):
    img = img.permute(1, 2, 0)
    img = img * std + mean
    plt.imshow(img)
    plt.axis('off')
    plt.show()

for images, _ in train_loader:
    first_image = images[0]
    show_image(first_image, mean=torch.tensor([0.485, 0.456, 0.406]),
std=torch.tensor([0.229, 0.224, 0.225]))

#Optimization

import torch
import torch.nn as nn
```

```
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import math
import numpy as np

class Arcface(nn.Module):
    def __init__(self, embedding_size=512, classnum=10, s=64., m=0.5):
        super(Arcface, self).__init__()
        self.classnum = classnum
        self.kernel = nn.Parameter(torch.Tensor(embedding_size, classnum))
        nn.init.xavier_uniform_(self.kernel)
        self.m = m
        self.s = s
        self.cos_m = math.cos(m)
        self.sin_m = math.sin(m)
        self.th = math.cos(math.pi - m)
        self.mm = math.sin(math.pi - m) * m

    def forward(self, embeddings, labels):
        embeddings = nn.functional.normalize(embeddings, dim=1)
        kernel_norm = nn.functional.normalize(self.kernel, dim=0)
        cos_theta = torch.mm(embeddings, kernel_norm).clamp(-1, 1)
        sin_theta = torch.sqrt(1.0 - torch.pow(cos_theta, 2))
        cos_theta_m = cos_theta * self.cos_m - sin_theta * self.sin_m
        cos_theta_m = torch.where(cos_theta > self.th, cos_theta_m, cos_theta - self.mm)
        one_hot = torch.zeros(cos_theta.size(), device=embeddings.device)
        one_hot.scatter_(1, labels.view(-1, 1).long(), 1)
        output = (one_hot * cos_theta_m) + ((1.0 - one_hot) * cos_theta)
        output *= self.s
        return output

class EmbeddingModel(nn.Module):
    def __init__(self, embedding_size=512):
        super(EmbeddingModel, self).__init__()
        self.fc = nn.Linear(3 * 28 * 28, embedding_size)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

```
embedding_model = EmbeddingModel()
arcface = Arcface(embedding_size=512, classnum=10, s=64.0, m=0.5)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(list(embedding_model.parameters()) +
list(arcface.parameters()), lr=0.01)

dummy_data = torch.randn(64, 3, 28, 28)  # Mini-batch of 64 images, 3 channels, 28x28
dummy_labels = torch.randint(0, 10, (64,))  # Mini-batch of 64 labels

train_loader = DataLoader(TensorDataset(dummy_data, dummy_labels), batch_size=8)

# Training loop for the first mini-batch
for i, (inputs, labels) in enumerate(train_loader):
    optimizer.zero_grad()
    embeddings = embedding_model(inputs)
    outputs = arcface(embeddings, labels)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # Print loss
    print(f'Batch {i+1}, Loss: {loss.item()}')

    # Break after the first mini-batch
    break
```

Visualization of 1st sample of each minibatch of size 8




SPEED