```
import numpy as np
import pandas as pd
# import file utilities
import os
import glob
import random

# import charting
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, ArtistAnimation
%matplotlib inline

from IPython.display import HTML

# import computer vision
import cv2
```

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

```
Mounted at /content/drive
```

```
realdata = "/content/drive/MyDrive/celebs/real"
fakedata = "/content/drive/MyDrive/celebs/fake"
```

```
def save_frames_from_video(video_path, output_folder):
    # Open the video file
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
        print(f"Error opening video file: {video_path}")
        return

    # Create the output folder if it doesn't exist
    os.makedirs(output_folder, exist_ok=True)

    # Read and save frames from the video
    frame_count = 0
    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Save the frame as an image
        frame_path = os.path.join(output_folder, f"frame_{frame_count}.jpg")
        cv2.imwrite(frame_path, frame)

        frame_count += 1

    # Release the video capture object
    cap.release()

# Output folders for saving frames
real_output_folder = "real_frames"
fake_output_folder = "fake_frames"

# Save frames from one video in realdata folder
save_frames_from_video(os.path.join(realdata, os.listdir(realdata)[0]), real_output_folder)
# Save frames from one video in fakedata folder
save_frames_from_video(os.path.join(fakedata, os.listdir(fakedata)[0]), fake_output_folder)
```

```
import os
import shutil
from sklearn.model_selection import train_test_split

# Define your input folders
real_folder = "real_frames"
fake_folder = "fake_frames"
combined_folder = "combined_data"

# Create the combined folder if it doesn't exist
if not os.path.exists(combined_folder):
    os.makedirs(combined_folder)
```

```
# Move real images to the combined folder
for filename in os.listdir(real_folder):
    src_path = os.path.join(real_folder, filename)
    dst_path = os.path.join(combined_folder, filename)
    shutil.copy(src_path, dst_path)

# Move fake images to the combined folder
for filename in os.listdir(fake_folder):
    src_path = os.path.join(fake_folder, filename)
    dst_path = os.path.join(combined_folder, filename)
    shutil.copy(src_path, dst_path)

# Split the combined data into train and test sets
all_images = os.listdir(combined_folder)
train_images, test_images = train_test_split(all_images, test_size=0.2, random_state=42)

# Create train and test folders
train_folder = "train_data"
test_folder = "test_data"
os.makedirs(train_folder, exist_ok=True)
os.makedirs(test_folder, exist_ok=True)

# Move train images
for filename in train_images:
    src_path = os.path.join(combined_folder, filename)
    dst_path = os.path.join(train_folder, filename)
    shutil.copy(src_path, dst_path)

# Move test images
for filename in test_images:
    src_path = os.path.join(combined_folder, filename)
    dst_path = os.path.join(test_folder, filename)
    shutil.copy(src_path, dst_path)

print("Data split successfully!")
```

```
Data split successfully!
```

```
import os
import shutil
import random

# Define your input folders
real_folder = "real_frames"
fake_folder = "fake_frames"

# Define output folders
train_folder = "train"
test_folder = "test"

# Create output directories
os.makedirs(os.path.join(train_folder, "real"), exist_ok=True)
os.makedirs(os.path.join(train_folder, "fake"), exist_ok=True)
os.makedirs(os.path.join(test_folder, "real"), exist_ok=True)
os.makedirs(os.path.join(test_folder, "fake"), exist_ok=True)

# Function to split files into train and test sets
def split_data(source_folder, train_subfolder, test_subfolder, test_size=0.2):
    files = os.listdir(source_folder)
    random.shuffle(files)  # Shuffle files to ensure randomness
    split_index = int(len(files) * (1 - test_size))  # Calculate index for train-test split

    train_files = files[:split_index]
    test_files = files[split_index:]

    # Move files to the respective folders
    for file in train_files:
        shutil.copy(os.path.join(source_folder, file), os.path.join(train_subfolder, file))

    for file in test_files:
        shutil.copy(os.path.join(source_folder, file), os.path.join(test_subfolder, file))

# Split the real and fake frames
split_data(real_folder, os.path.join(train_folder, "real"), os.path.join(test_folder, "real"))
split_data(fake_folder, os.path.join(train_folder, "fake"), os.path.join(test_folder, "fake"))
```

```
print("Data organization completed!")
```

```
Data organization completed!
```

Start coding or generate with AI.

Start coding or generate with AI.

```
!git clone https://github.com/polimi-ispl/icpr2020dfdc
!pip install efficientnet-pytorch
!pip install -U git+https://github.com/albu/albumentations > /dev/null
%cd icpr2020dfdc/notebook
```

```
Cloning into 'icpr2020dfdc'...
remote: Enumerating objects: 656, done.
remote: Counting objects: 100% (119/119), done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 656 (delta 101), reused 87 (delta 83), pack-reused 537 (from 1)
Receiving objects: 100% (656/656), 99.64 MiB | 32.47 MiB/s, done.
Resolving deltas: 100% (341/341), done.
Collecting efficientnet-pytorch
  Downloading efficientnet_pytorch-0.7.1.tar.gz (21 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from efficientnet-pytorch) (2.5.0+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch->efficientnet-pytorch) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch->efficientnet-pytorch) (4
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->efficientnet-pytorch) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch->efficientnet-pytorch) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch->efficientnet-pytorch) (2024.6.1)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch->efficientnet-pytorch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch->efficientnet-py
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch->efficientnet-pytorch) (3.
Building wheels for collected packages: efficientnet-pytorch
  Building wheel for efficientnet-pytorch (setup.py) ... done
  Created wheel for efficientnet-pytorch: filename=efficientnet_pytorch-0.7.1-py3-none-any.whl size=16424 sha256=19ea038767b02cf4901883
  Stored in directory: /root/.cache/pip/wheels/03/3f/e9/911b1bc46869644912bda90a56bcf7b960f20b5187feea3baf
Successfully built efficientnet-pytorch
Installing collected packages: efficientnet-pytorch
Successfully installed efficientnet-pytorch-0.7.1
  Running command git clone --filter=blob:none --quiet https://github.com/albu/albumentations /tmp/pip-req-build-y2_47szv
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the source
albumentations 1.4.20 requires albucore==0.0.19, but you have albucore 0.0.20 which is incompatible.
/content/icpr2020dfdc/notebook
```

```
import torch
from torch.utils.model_zoo import load_url
import matplotlib.pyplot as plt
from scipy.special import expit

import sys
sys.path.append('..')

from blazeface import FaceExtractor, BlazeFace, VideoReader
from architectures import fornet,weights
from isplutils import utils
```

Parameters

```
"""
Choose an architecture between
- EfficientNetB4
- EfficientNetB4ST
- EfficientNetAutoAttB4
- EfficientNetAutoAttB4ST
- Xception
"""
net_model = 'EfficientNetAutoAttB4ST'

"""
Choose a training dataset between
- DFDC
- FFPP
"""
```

```
train_db = 'DFDC'
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'



device = torch.device('cpu')
face_policy = 'scale'
face_size = 224
frames_per_video = 32
```

Initialization

```
model_url = weights.weight_url['{:s}_{:s}'.format(net_model,train_db)]
net = getattr(fornet, net_model)()
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'

# Try moving a single parameter to the GPU for debugging
try:
    # Get a single parameter from the model
    param = next(net.parameters())
    # Move the parameter to the GPU
    param.data = param.data.to(device)
except RuntimeError as e:
    print(f"Error moving parameter to GPU: {e}")
    # If this fails, it might indicate a problem with the model architecture or data type.

# Move the entire model to the GPU after the single parameter test
net = net.eval().to(device)

net.load_state_dict(load_url(model_url, map_location=device, check_hash=True))
```

```
Downloading: "https://github.com/lukemelas/EfficientNet-PyTorch/releases/download/1.0/efficientnet-b4-6ed6700e.pth" to /root/.cache/torc
100%|██████████| 74.4M/74.4M [00:00<00:00, 134MB/s]
Downloading: "https://f002.backblazeb2.com/file/icpr2020/EfficientNetAutoAttB4ST_DFDC_bestval-4df0ef7d2f380a5955affa78c35d0942ac1cd65229
Loaded pretrained weights for efficientnet-b4
100%|██████████| 33.9M/33.9M [00:01<00:00, 24.8MB/s]
<All keys matched successfully>
```

```
transf = utils.get_transformer(face_policy, face_size, net.get_normalizer(), train=False)


facedet = BlazeFace().to(device)
facedet.load_weights("../blazeface/blazeface.pth")
facedet.load_anchors("../blazeface/anchors.npy")
videoreader = VideoReader(verbose=False)
video_read_fn = lambda x: videoreader.read_frames(x, num_frames=frames_per_video)
face_extractor = FaceExtractor(video_read_fn=video_read_fn,facedet=facedet)
```

```
/content/icpr2020dfdc/notebook/../blazeface/blazeface.py:164: FutureWarning: You are using `torch.load` with `weights_only=False` (the c
  self.load_state_dict(torch.load(path))
```

```
import os
import cv2
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

class DeepFakeDataset(Dataset):
    def __init__(self, directory, transform=None):
        self.directory = directory
        self.transform = transform
        self.images = []  # Initialize an empty list to store image paths

        # Recursively search for images in subdirectories
        for root, _, files in os.walk(directory):
            for file in files:
                if file.endswith('.jpg'):
                    self.images.append(os.path.join(root, file))

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

    def __getitem__(self, idx):
        img_path = self.images[idx]
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB

        if self.transform:
            image = self.transform(image)

        label = 1 if "fake" in img_path else 0  # Assuming fake images have "fake" in the filename
        return image, label


transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.RandomAffine(0, shear=10, scale=(0.8, 1.2)),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),
    transforms.ToTensor(),
])


train_dataset = DeepFakeDataset('/content/train', transform=transform) # Using DeepFakeDataset class to create the dataset
test_dataset = DeepFakeDataset('/content/test', transform=transform) # Using DeepFakeDataset class to create the dataset
print(f"Train dataset size: {len(train_dataset)}")
print(f"Test dataset size: {len(test_dataset)}")
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

```
Train dataset size: 617
Test dataset size: 155
```

```
import os
import cv2
import torch
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, models
import torch.nn as nn
import torch.optim as optim
import time

# Custom Dataset Definition
class DeepFakeDataset(Dataset):
    def __init__(self, directory, transform=None):
        self.directory = directory
        self.transform = transform
        self.images = [os.path.join(root, file)
                       for root, _, files in os.walk(directory)
                       for file in files if file.endswith('.jpg')]

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

    def __getitem__(self, idx):
        img_path = self.images[idx]
        image = cv2.imread(img_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        if self.transform:
            image = self.transform(image)

        label = 1 if "fake" in img_path else 0
        return image, label

# Data Augmentation Transform
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.2),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.RandomGrayscale(p=0.2),
    transforms.ToTensor(),
])
```

```
# Initialize Dataset and DataLoader
train_dataset = DeepFakeDataset('/content/train', transform=transform)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

# Set up model, loss, and optimizer
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = models.efficientnet_b4(weights=models.EfficientNet_B4_Weights.IMAGENET1K_V1)

# Freeze early layers
for param in net.features.parameters():
    param.requires_grad = False

# Update the classifier for binary classification
num_ftrs = net.classifier[1].in_features
net.classifier = nn.Sequential(
    nn.Dropout(0.5),
    nn.Linear(num_ftrs, 1)
)
net.to(device)

# Define loss function and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(net.parameters(), lr=1e-5)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=3, verbose=True)

# Training loop
train_accuracies = []  # List to store training accuracies for each epoch
num_epochs = 15
for epoch in range(num_epochs):
    net.train()
    running_loss, correct, total = 0.0, 0, 0
    start_time = time.time()

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = net(images)
        loss = criterion(outputs.squeeze(1), labels.float())
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        preds = torch.round(torch.sigmoid(outputs.squeeze(1)))
        correct += (preds == labels).sum().item()
        total += labels.size(0)

    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct / total
    train_accuracies.append(train_accuracy)  # Append training accuracy for this epoch

    # Learning Rate Adjustment
    scheduler.step(train_loss)

    # Print training results
    print(f"Epoch [{epoch+1}/{num_epochs}], "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_accuracy:.2f}%, "
          f"Time: {time.time() - start_time:.2f}s")
```

```
Epoch [1/15], Train Loss: 0.6959, Train Acc: 48.95%, Time: 233.25s
Epoch [2/15], Train Loss: 0.6905, Train Acc: 52.35%, Time: 239.52s
Epoch [3/15], Train Loss: 0.6875, Train Acc: 55.11%, Time: 246.15s
Epoch [4/15], Train Loss: 0.6852, Train Acc: 57.86%, Time: 243.40s
Epoch [5/15], Train Loss: 0.6817, Train Acc: 57.54%, Time: 254.06s
Epoch [6/15], Train Loss: 0.6770, Train Acc: 63.53%, Time: 241.47s
Epoch [7/15], Train Loss: 0.6741, Train Acc: 65.64%, Time: 241.70s
Epoch [8/15], Train Loss: 0.6704, Train Acc: 66.94%, Time: 249.59s
Epoch [9/15], Train Loss: 0.6708, Train Acc: 67.75%, Time: 242.72s
Epoch [10/15], Train Loss: 0.6689, Train Acc: 69.69%, Time: 243.55s
Epoch [11/15], Train Loss: 0.6696, Train Acc: 69.21%, Time: 242.44s
Epoch [12/15], Train Loss: 0.6650, Train Acc: 69.85%, Time: 242.44s
Epoch [13/15], Train Loss: 0.6541, Train Acc: 76.50%, Time: 244.35s
Epoch [14/15], Train Loss: 0.6566, Train Acc: 74.72%, Time: 236.08s
Epoch [15/15], Train Loss: 0.6518, Train Acc: 76.82%, Time: 239.51s
```

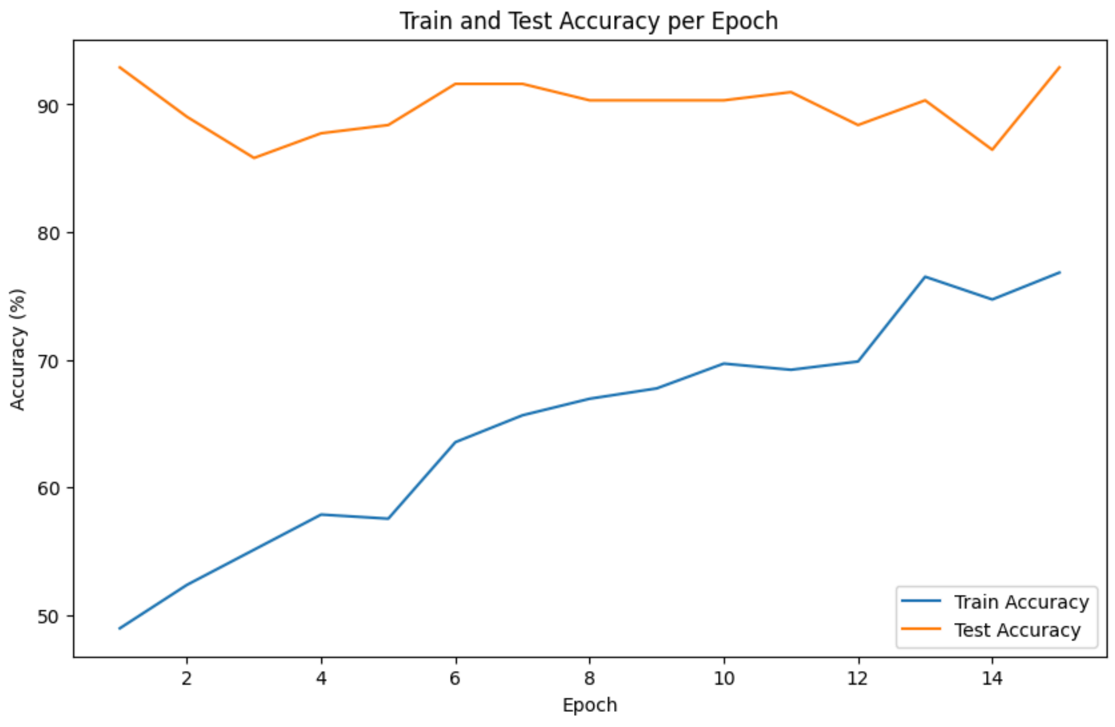
```
num_epochs = 15
test_accuracies=[]
for epoch in range(num_epochs):
    net.eval()  # Set the model to evaluation mode
    test_correct, test_total = 0, 0  # Initialize counters for correct predictions and total samples
    with torch.no_grad():  # Disable gradient calculation during evaluation
        for images, labels in test_loader:  # Iterate over the test data loader
            images, labels = images.to(device), labels.to(device)  # Move data to the device (GPU if available)
            outputs = net(images)  # Get model predictions
            preds = torch.round(torch.sigmoid(outputs.squeeze(1)))  # Apply sigmoid and round to get binary predictions
            test_total += labels.size(0)  # Update total samples count
            test_correct += (preds == labels).sum().item()  # Update correct predictions count

    test_accuracy = 100 * test_correct / test_total  # Calculate test accuracy
    test_accuracies.append(test_accuracy)  # Append test accuracy for this epoch

    # Print test accuracy for each epoch
    print(f"Epoch [{epoch+1}/{num_epochs}], Test Accuracy: {test_accuracy:.2f}%")
```

```
Epoch [1/15], Test Accuracy: 92.90%
Epoch [2/15], Test Accuracy: 89.03%
Epoch [3/15], Test Accuracy: 85.81%
Epoch [4/15], Test Accuracy: 87.74%
Epoch [5/15], Test Accuracy: 88.39%
Epoch [6/15], Test Accuracy: 91.61%
Epoch [7/15], Test Accuracy: 91.61%
Epoch [8/15], Test Accuracy: 90.32%
Epoch [9/15], Test Accuracy: 90.32%
Epoch [10/15], Test Accuracy: 90.32%
Epoch [11/15], Test Accuracy: 90.97%
Epoch [12/15], Test Accuracy: 88.39%
Epoch [13/15], Test Accuracy: 90.32%
Epoch [14/15], Test Accuracy: 86.45%
Epoch [15/15], Test Accuracy: 92.90%
```

```
# Plot the train and test accuracy curves
plt.figure(figsize=(10, 6))
plt.plot(range(1, num_epochs + 1), train_accuracies, label="Train Accuracy")
plt.plot(range(1, num_epochs + 1), test_accuracies, label="Test Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Train and Test Accuracy per Epoch")
plt.legend()
plt.show()
```



```
import cv2
import torch
import numpy as np
import numpy as np
from torchvision import transforms
from scipy.special import expit  # For sigmoid on logits
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
import os

# Frame extraction function
def extract_frames(video_path, num_frames=32):
    cap = cv2.VideoCapture(video_path)
    frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_indices = np.linspace(0, frame_count - 1, num_frames, dtype=int)

    frames = []
    for idx in frame_indices:
        cap.set(cv2.CAP_PROP_POS_FRAMES, idx)
        ret, frame = cap.read()
        if ret:
            frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            frames.append(frame)
    cap.release()
    return frames

# Prediction function for a single video
def predict_video(video_path, model, transform, device, num_frames=32):
    frames = extract_frames(video_path, num_frames=num_frames)
    frame_preds = []

    model.eval()  # Ensure the model is in evaluation mode
    with torch.no_grad():
        for frame in frames:
            frame = transform(frame)
            frame = frame.unsqueeze(0).to(device)  # Add batch dimension
            output = model(frame)
            prob = expit(output.item())  # Convert logits to probability
            frame_preds.append(prob)

    # Average prediction probabilities over all frames
    avg_pred = np.mean(frame_preds)
    label = "Fake" if avg_pred >= 0.5 else "Real"
    print(f"Prediction: {label} (Confidence: {avg_pred:.2f})")
    return label, avg_pred

# Define frame transform (should match training transforms)
frame_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

# Path to the video for prediction
video_path = '/content/id61_id60_0009.mp4'

label, confidence = predict_video(video_path, net, frame_transform, device)
```

```
Prediction: Real (Confidence: nan)
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:3504: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py:129: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

```
import os
import cv2
import torch
import numpy as np
from torchvision import transforms
from scipy.special import expit  # For sigmoid on logits
from blazeface import BlazeFace  # Assuming BlazeFace is available from previous setup

# Initialize the face detector
facedet = BlazeFace().to(device)
facedet.load_weights("../blazeface/blazeface.pth")
facedet.load_anchors("../blazeface/anchors.npy")

# Function to extract frames containing clear faces from a single video
def extract_frames_with_faces(video_path, facedet, num_frames=64):
    cap = cv2.VideoCapture(video_path)
    frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    selected_frames = []
    count, idx = 0, 0

    while count < num_frames and idx < frame_count:
        cap.set(cv2.CAP_PROP_POS_FRAMES, idx)
        ret, frame = cap.read()
        if not ret:
            break

        # Detect faces and select frames with clear faces
        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        # Resize the frame to 128x128 before passing it to BlazeFace
        frame_rgb_resized = cv2.resize(frame_rgb, (128, 128))

        detections = facedet.predict_on_image(frame_rgb_resized) # Pass the resized frame
        if len(detections) > 0:  # If a face is detected
            selected_frames.append(frame_rgb) # Append the original frame
            count += 1

        idx += frame_count // num_frames  # Spread selection evenly

    cap.release()
    return selected_frames

# Prediction function for a single video using ensemble averaging
def predict_video(video_path, model, transform, device, facedet, num_frames=64, num_ensemble=3):
    # Run multiple predictions and average
    ensemble_preds = []
    for _ in range(num_ensemble):
        frames = extract_frames_with_faces(video_path, facedet, num_frames=num_frames)
        frame_preds = []

        model.eval()  # Ensure the model is in evaluation mode
        with torch.no_grad():
            for frame in frames:
                frame = transform(frame)
                frame = frame.unsqueeze(0).to(device)  # Add batch dimension
                output = model(frame)
                prob = expit(output.item())  # Convert logits to probability
                frame_preds.append(prob)

        # Average prediction for this ensemble run
        ensemble_preds.append(np.mean(frame_preds))

    # Final prediction: Average of ensemble predictions
    final_avg_pred = np.mean(ensemble_preds)
    label = "Fake" if final_avg_pred >= 0.6 else "Real"  # Adjust threshold as needed
    return label, final_avg_pred

#for predicting a single video
video_path = '/content/fvideo_1.mp4'

# Define frame transform (should match training transforms)
frame_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

# Perform prediction
label, confidence = predict_video(video_path, net, frame_transform, device, facedet)
print(f"Video: {os.path.basename(video_path)} - Prediction: {label}, Confidence: {confidence:.2f}")
```

```
Video: fvideo_1.mp4 - Prediction: Real, Confidence: 0.47
```

for a set of videos

```
import os
import cv2
import torch
import numpy as np
from torchvision import transforms
from scipy.special import expit  # For sigmoid on logits

# Frame extraction function
def extract_frames(video_path, num_frames=32):
    cap = cv2.VideoCapture(video_path)
    frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_indices = np.linspace(0, frame_count - 1, num_frames, dtype=int)

    frames = []
    for idx in frame_indices:
        cap.set(cv2.CAP_PROP_POS_FRAMES, idx)
        ret, frame = cap.read()
        if ret:
            frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            frames.append(frame)
    cap.release()
    return frames

# Prediction function for a single video
def predict_video(video_path, model, transform, device, num_frames=32):
    frames = extract_frames(video_path, num_frames=num_frames)
    frame_preds = []

    model.eval()  # Ensure the model is in evaluation mode
    with torch.no_grad():
        for frame in frames:
            frame = transform(frame)
            frame = frame.unsqueeze(0).to(device)  # Add batch dimension
            output = model(frame)
            prob = expit(output.item())  # Convert logits to probability
            frame_preds.append(prob)

    # Average prediction probabilities over all frames
    avg_pred = np.mean(frame_preds)
    label = "Fake" if avg_pred >= 0.5 else "Real"
    return label, avg_pred

# Batch prediction function for multiple videos in a folder
def predict_videos_in_folder(folder_path, model, transform, device, num_frames=32):
    results = {}
    video_files = [f for f in os.listdir(folder_path) if f.endswith(('.mp4', '.avi', '.mov'))]

    for video_file in video_files:
        video_path = os.path.join(folder_path, video_file)
        label, confidence = predict_video(video_path, model, transform, device, num_frames)
        results[video_file] = {"Label": label, "Confidence": confidence}
        print(f"Video: {video_file} - Prediction: {label}, Confidence: {confidence:.2f}")
    return results

# Define frame transform (should match training transforms)
frame_transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

# Path to the folder containing multiple videos
folder_path = '/content/drive/MyDrive/new_videos'

results = predict_videos_in_folder(folder_path, net, frame_transform, device)
```

```
Video: video_3.mp4 - Prediction: Real, Confidence: 0.46
Video: video_2.mp4 - Prediction: Real, Confidence: 0.48
Video: video_1.mp4 - Prediction: Real, Confidence: 0.49
```

Start coding or generate with AI.

Start coding or generate with AI.