face-detection

April 19, 2025

1 Face Detection using Convolutional Neural Networks

1.1 Project Overview

This project aims to build a binary image classifier that detects whether an image contains a human face or not, using a Convolutional Neural Network (CNN) implemented in PyTorch. The system is trained on a labeled dataset of face and non-face images and includes data preprocessing, model training, validation, and evaluation components. The project serves as a foundational computer vision task applicable to surveillance, authentication systems, and real-time detection tools.

1.1.1 Key Steps

1. **Data Preprocessing:**
   - All input images are resized to **48x48 pixels** and converted to PyTorch tensors.
   - **Data augmentation** techniques such as random horizontal flipping and rotation are applied to improve model robustness.
   - The dataset is divided into **Training**, **Validation**, and **Testing** sets.
2. **Model Architecture:**
   - A custom **CNN** is constructed using stacked convolutional layers, **ReLU activations**, and **max pooling** operations.
   - A **dropout layer** is added to the fully connected portion of the network to mitigate overfitting.
   - The final layer uses a **Sigmoid activation function** for binary classification between face and non-face classes.
3. **Training:**
   - The model is trained using **Binary Cross-Entropy Loss**, optimized via the **Adam optimizer**.
   - Losses are tracked separately for training and validation datasets over 50 epochs.
   - **Checkpointing** is used to save the model with the best validation performance.
4. **Evaluation Metrics:**
   - The final model is evaluated using the **accuracy score** and a **confusion matrix**.
   - A **heatmap** is generated to visualize the confusion matrix and interpret class-wise performance.
5. **Result Visualization:**
   - **Loss curves** are plotted to visualize the learning process over training epochs.
   - Predictions on test data are used to assess classification quality and potential model limitations.

1.1.2 Model Summary

- A lightweight and effective CNN-based classifier was built to differentiate between face and non-face images.
- The model achieved satisfactory performance on unseen data, showcasing its ability to generalize.
- Key features such as data augmentation, dropout regularization, and validation monitoring were incorporated to enhance training stability and prevent overfitting.

1.1.3 Future Work

- **Model Upgrade**: Integrate more advanced architectures like **ResNet** or **EfficientNet** for improved accuracy.
- **Data Scaling**: Train on a larger, more diverse dataset to improve performance in real-world scenarios.
- **Augmentation Enhancements**: Apply more varied augmentation techniques for better generalization.
- **Real-Time Application**: Deploy the model into a **live camera feed** or web application for practical use.
- **Model Explainability**: Implement tools such as **Grad-CAM** to visualize model decision areas on input images.

1.1.4 Step 1: Import Required Libraries

To begin, we import all the necessary libraries. This includes PyTorch for model development and training, torchvision for data transformations, and PIL for image loading. Additionally, we use Seaborn and Matplotlib for data visualization, and Scikit-learn to compute accuracy and the confusion matrix for evaluation.

[1]:
```python
# Importing necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms
from torch.utils.data import DataLoader
from PIL import Image
import os
import seaborn as sns
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
```

1.1.5 Step 2: Define Hyperparameters and Device

Here, we define key hyperparameters for our training process, including the image size, batch size, number of epochs, and learning rate. We also configure the device, using a GPU if available to accelerate training; otherwise, the model will run on the CPU.

[2]:
```python
# Hyperparameters
IMG_SIZE = 48
```

```
BATCH_SIZE = 32
EPOCHS = 50
LR = 0.001
```

[3]:
```
# Select GPU if available
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

1.1.6 Step 3: Define Image Transformations

In this step, we define the transformations to apply to each image in the dataset. Images are resized to 48x48 pixels, and random horizontal flips and rotations are used to augment the dataset, improving model generalization. Finally, the images are converted into tensors, making them compatible with PyTorch models.

[4]:
```
# Transformations for data
transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ToTensor()
])
```

1.1.7 Step 4: Create Custom Dataset Loader

In this step, a custom dataset loader is created to handle face and non-face images. The dataset loader takes the directories of face and non-face images as input, assigns labels (1 for faces, 0 for non-faces), and applies the previously defined transformations. This loader ensures that each image is returned along with its corresponding label during training or evaluation.

[5]:
```
# Dataset loader
class FaceDataset(torch.utils.data.Dataset):
    def __init__(self, face_dir, non_face_dir, transform=None):
        self.images = []
        self.labels = []
        self.transform = transform

        for img in os.listdir(face_dir):
            self.images.append((os.path.join(face_dir, img), 1))  # Label 1 for␣
 ↪faces
        for img in os.listdir(non_face_dir):
            self.images.append((os.path.join(non_face_dir, img), 0))  # Label 0␣
 ↪for non-faces

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

    def __getitem__(self, idx):
```

```
        path, label = self.images[idx]
        image = Image.open(path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, torch.tensor(label, dtype=torch.float32)
```

1.1.8 Step 5: Define CNN Model

This step involves designing the CNN model architecture. The model consists of two convolutional layers followed by max pooling, which helps reduce the image dimensions. The output is then flattened and passed through fully connected layers. A dropout layer is included for regularization, and the final layer uses a Sigmoid activation function to predict probabilities for binary classification (face or non-face).

[6]:
```
# CNN model for face detection
class FaceDetectorCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),
            nn.Conv2d(32, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
            nn.Conv2d(64, 64, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2)
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 12 * 12, 128), nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(128, 1), nn.Sigmoid()
        )


    def forward(self, x):
        x = self.features(x)
        return self.classifier(x)
```

1.1.9 Step 6: Load Training, Validation, and Test Data

The datasets for training, validation, and testing are loaded using the custom dataset loader. This step involves preparing the data for training, validation, and testing, making use of the DataLoader class to batch the data and shuffle it during training. This ensures that the model is trained efficiently and generalizes well to unseen data.

[7]:
```python
# Load datasets
train_faces_dir = 'train/faces'
train_non_faces_dir = 'train/non-faces'
test_faces_dir = 'test/faces'
test_non_faces_dir = 'test/non-faces'
val_faces_dir = 'val/faces'
val_non_faces_dir = 'val/non-faces'

train_dataset = FaceDataset(train_faces_dir, train_non_faces_dir, transform)
test_dataset = FaceDataset(test_faces_dir, test_non_faces_dir, transform)
val_dataset = FaceDataset(val_faces_dir, val_non_faces_dir, transform)

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

1.1.10 Step 7: Define Training Loop

The training loop is defined in this step. During each epoch, the model is trained on the training data and evaluated on the validation data. The model's performance is monitored by tracking the loss for both training and validation. The best-performing model, based on validation loss, is saved during training to prevent overfitting.

[8]:
```python
# Training Loop
def train_model(model, train_loader, val_loader, epochs, lr,␣
↪save_path='best_model.pth'):
    model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.BCELoss()

    train_losses = []
    val_losses = []
    best_val_loss = float('inf')  # Initialize to a high value

    for epoch in range(epochs):
        model.train()
        epoch_train_loss = 0

        for images, labels in train_loader:
            images = images.to(device)
            labels = labels.to(device).unsqueeze(1).float()  # Ensure float␣
↪labels

            outputs = model(images)
            loss = criterion(outputs, labels)

            optimizer.zero_grad()
```

```
            loss.backward()
            optimizer.step()
            epoch_train_loss += loss.item()

        avg_train_loss = epoch_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        # ---- Validation ----
        model.eval()
        epoch_val_loss = 0

        with torch.no_grad():
            for val_images, val_labels in val_loader:
                val_images = val_images.to(device)
                val_labels = val_labels.to(device).unsqueeze(1).float()

                val_outputs = model(val_images)
                val_loss = criterion(val_outputs, val_labels)
                epoch_val_loss += val_loss.item()

        avg_val_loss = epoch_val_loss / len(val_loader)
        val_losses.append(avg_val_loss)

        # ---- Checkpointing ----
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            torch.save(model.state_dict(), save_path)
            print(f"  Saved new best model (val loss: {avg_val_loss:.4f})")

        print(f"Epoch {epoch+1}/{epochs} | Train Loss: {avg_train_loss:.4f} |␣
↪Val Loss: {avg_val_loss:.4f}")

    return train_losses, val_losses
```

1.1.11 Step 8: Define Evaluation Function

The evaluation function is used to assess the model's performance on the test dataset after training. The model's predictions are compared with the true labels, and metrics such as accuracy and confusion matrix are calculated. This function also visualizes the confusion matrix to provide insight into how well the model is distinguishing between faces and non-faces.

[9]:
```
# Evaluation function
def evaluate_model(model, test_loader):
    model.eval()
    all_preds, all_labels = [], []

    with torch.no_grad():
```

```
        for images, labels in test_loader:
            images = images.to(device)
            outputs = model(images).cpu().numpy()
            preds = (outputs > 0.5).astype(int).flatten()
            all_preds.extend(preds)
            all_labels.extend(labels.numpy())

    acc = accuracy_score(all_labels, all_preds)
    cm = confusion_matrix(all_labels, all_preds)

    print(f"\n  Test Accuracy: {acc:.4f}")
    print("Confusion Matrix:")
    print(cm)

    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Non-Face",␣
↪"Face"], yticklabels=["Non-Face", "Face"])
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(f"Confusion Matrix (Accuracy: {acc:.2%})")
    plt.tight_layout()
    plt.show()

    return acc, cm
```

1.1.12 Step 9: Train the Model and Save

In this step, the CNN model is instantiated and trained using the previously defined training loop. The best-performing model, based on validation loss, is saved to a file. The trained model can later be loaded for inference or further fine-tuning.

[10]:
```
# Run training
model = FaceDetectorCNN()
train_losses, val_losses = train_model(
    model,
    train_loader,  # Training data loader
    val_loader,    # Validation data loader
    epochs=EPOCHS,  # Number of epochs
    lr=LR,          # Learning rate
    save_path='best_model.pth'  # Path to save the best model
)
```

```
c:\Users\durus\miniconda3\envs\ee655\lib\site-packages\PIL\Image.py:1045:
UserWarning: Palette images with Transparency expressed in bytes should be
converted to RGBA images
  warnings.warn(
```

```
 Saved new best model (val loss: 0.5146)
```

```
Epoch 1/50 | Train Loss: 0.3707 | Val Loss: 0.5146
  Saved new best model (val loss: 0.4589)
Epoch 2/50 | Train Loss: 0.2122 | Val Loss: 0.4589
Epoch 3/50 | Train Loss: 0.1769 | Val Loss: 0.7556
Epoch 4/50 | Train Loss: 0.1608 | Val Loss: 0.5973
Epoch 5/50 | Train Loss: 0.1410 | Val Loss: 0.5170
Epoch 6/50 | Train Loss: 0.1315 | Val Loss: 0.4999
Epoch 7/50 | Train Loss: 0.1223 | Val Loss: 0.5717
Epoch 8/50 | Train Loss: 0.1021 | Val Loss: 0.6974
  Saved new best model (val loss: 0.4354)
Epoch 9/50 | Train Loss: 0.1067 | Val Loss: 0.4354
Epoch 10/50 | Train Loss: 0.0984 | Val Loss: 0.4620
Epoch 11/50 | Train Loss: 0.0917 | Val Loss: 0.4449
Epoch 12/50 | Train Loss: 0.0931 | Val Loss: 0.4417
Epoch 13/50 | Train Loss: 0.0815 | Val Loss: 0.4946
Epoch 14/50 | Train Loss: 0.0799 | Val Loss: 0.5119
Epoch 15/50 | Train Loss: 0.0749 | Val Loss: 0.5626
Epoch 16/50 | Train Loss: 0.0723 | Val Loss: 0.6020
Epoch 17/50 | Train Loss: 0.0770 | Val Loss: 0.6414
Epoch 18/50 | Train Loss: 0.0677 | Val Loss: 0.4643
  Saved new best model (val loss: 0.4157)
Epoch 19/50 | Train Loss: 0.0658 | Val Loss: 0.4157
Epoch 20/50 | Train Loss: 0.0588 | Val Loss: 0.5449
Epoch 21/50 | Train Loss: 0.0619 | Val Loss: 0.5194
  Saved new best model (val loss: 0.4154)
Epoch 22/50 | Train Loss: 0.0571 | Val Loss: 0.4154
Epoch 23/50 | Train Loss: 0.0567 | Val Loss: 0.5631
Epoch 24/50 | Train Loss: 0.0533 | Val Loss: 0.5669
Epoch 25/50 | Train Loss: 0.0524 | Val Loss: 0.4198
Epoch 26/50 | Train Loss: 0.0475 | Val Loss: 0.5384
Epoch 27/50 | Train Loss: 0.0478 | Val Loss: 0.4765
Epoch 28/50 | Train Loss: 0.0500 | Val Loss: 0.4326
Epoch 29/50 | Train Loss: 0.0453 | Val Loss: 0.7698
Epoch 30/50 | Train Loss: 0.0393 | Val Loss: 0.5437
Epoch 31/50 | Train Loss: 0.0503 | Val Loss: 0.6100
Epoch 32/50 | Train Loss: 0.0404 | Val Loss: 0.5976
Epoch 33/50 | Train Loss: 0.0418 | Val Loss: 0.9329
Epoch 34/50 | Train Loss: 0.0431 | Val Loss: 0.4562
Epoch 35/50 | Train Loss: 0.0444 | Val Loss: 0.9414
Epoch 36/50 | Train Loss: 0.0429 | Val Loss: 0.6282
Epoch 37/50 | Train Loss: 0.0414 | Val Loss: 0.5632
Epoch 38/50 | Train Loss: 0.0457 | Val Loss: 0.8891
Epoch 39/50 | Train Loss: 0.0307 | Val Loss: 1.0157
Epoch 40/50 | Train Loss: 0.0389 | Val Loss: 0.6081
Epoch 41/50 | Train Loss: 0.0328 | Val Loss: 0.5214
Epoch 42/50 | Train Loss: 0.0364 | Val Loss: 0.5656
Epoch 43/50 | Train Loss: 0.0393 | Val Loss: 0.5966
Epoch 44/50 | Train Loss: 0.0357 | Val Loss: 0.7313
```

```
Epoch 45/50 | Train Loss: 0.0368 | Val Loss: 0.7238
Epoch 46/50 | Train Loss: 0.0369 | Val Loss: 0.9565
Epoch 47/50 | Train Loss: 0.0298 | Val Loss: 0.7935
Epoch 48/50 | Train Loss: 0.0279 | Val Loss: 0.8123
Epoch 49/50 | Train Loss: 0.0375 | Val Loss: 0.8397
Epoch 50/50 | Train Loss: 0.0284 | Val Loss: 1.0175
```

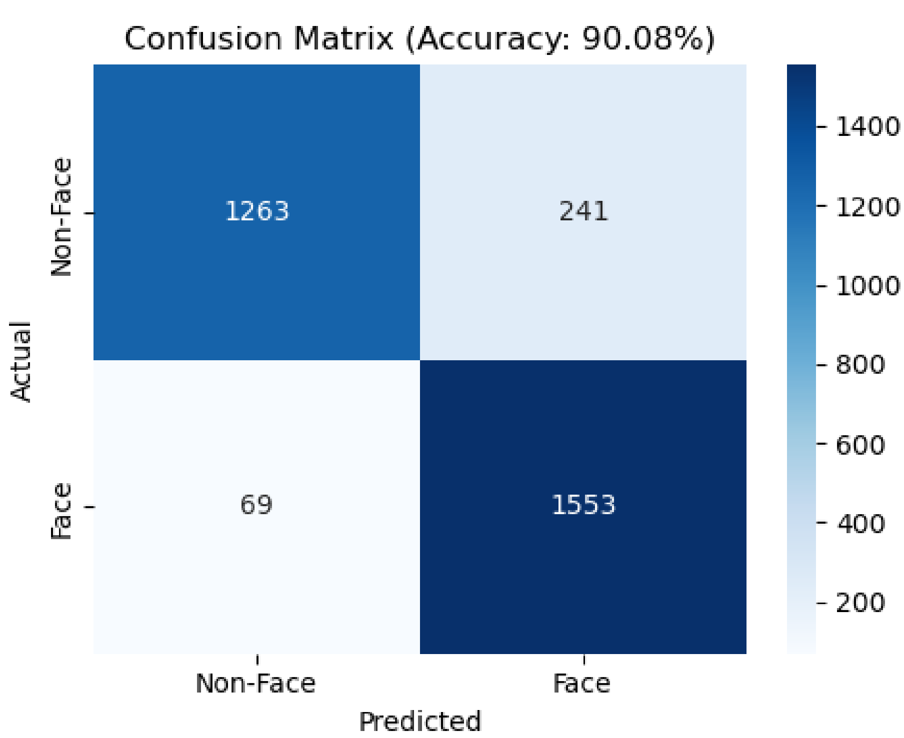
[11]:
```python
# Save model
torch.save(model.state_dict(), 'model.pth')
```

1.1.13 Step 10: Evaluate the Model

After training, the model is evaluated on the test dataset to assess how well it generalizes to unseen data. This is done by computing the accuracy and displaying the confusion matrix, which provides a deeper understanding of the model's performance.

[12]:
```python
# Run evaluation
evaluate_model(model, test_loader)
```

```
 Test Accuracy: 0.9008
Confusion Matrix:
[[1263  241]
 [  69 1553]]
```

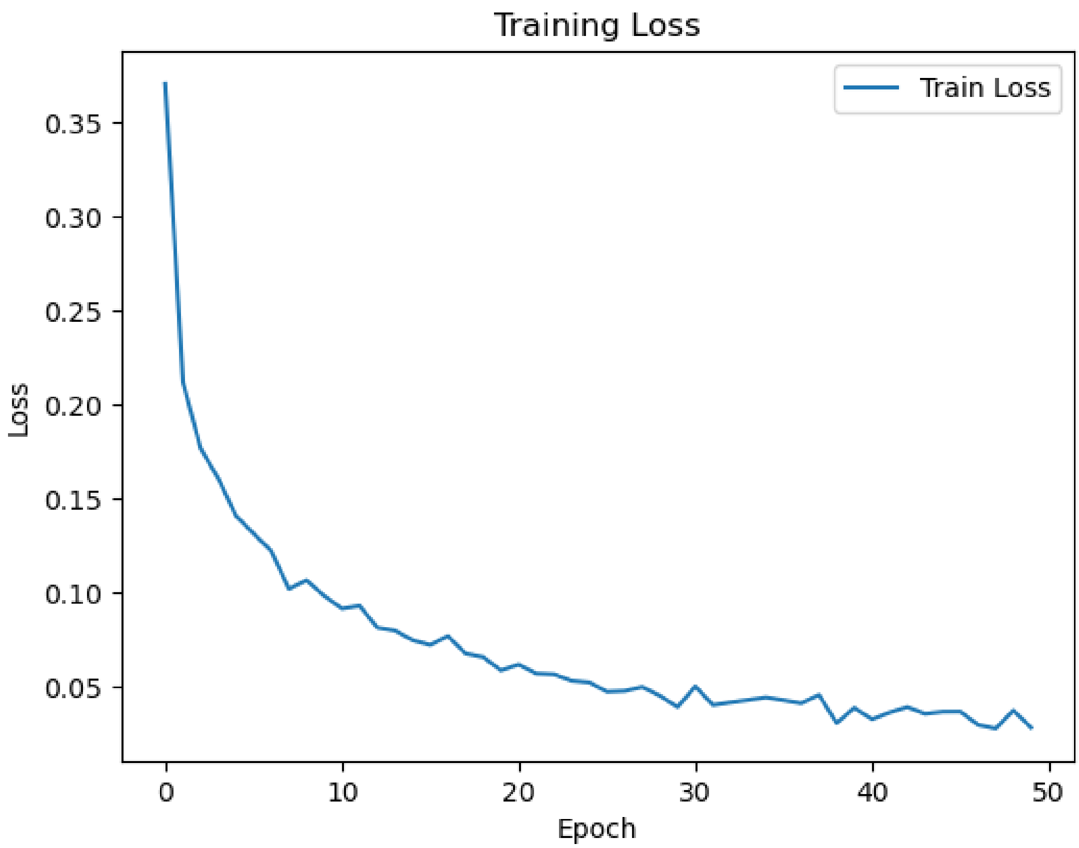


[12]: (0.9008317338451696,
 array([[1263, 241],
        [69, 1553]], dtype=int64))

1.1.14 Step 11: Plot Loss Curves

The loss curves for training and validation are plotted to visualize the model's learning process over the epochs. This helps to identify potential issues such as overfitting or underfitting. By comparing the training and validation loss curves, we can gain insights into the stability of the training process.

[13]:
```
# Plot losses
plt.plot(train_losses, label='Train Loss')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("Training Loss")
plt.show()
```



1.1.15 Conclusion

In this project, we successfully built a CNN model from scratch using PyTorch for face detection. We utilized a custom dataset loader, applied data augmentation to improve generalization, and evaluated the model using metrics such as accuracy and confusion matrix. The results were visualized through loss curves and the confusion matrix, providing a comprehensive view of the model's performance.

deepfake-detection

April 19, 2025

1 DeepFake Detection using CNN + Squeeze-and-Excitation Networks

1.1 Project Overview

This project aims to build a machine learning model capable of detecting deepfake images using Convolutional Neural Networks (CNNs). The core idea behind this approach is to leverage the pre-trained **EfficientNet-B0** model, a state-of-the-art CNN architecture, to extract features from images. To enhance the model's performance, we introduce a **Squeeze-and-Excitation (SE) block**, which dynamically recalibrates the channel-wise feature responses by learning the importance of each feature.

1.1.1 Key Steps:

1. **Data Preprocessing:**
   - The images are resized to 256x256 pixels and converted to tensor format for processing.
   - The dataset is split into three sets: **Training**, **Validation**, and **Testing**.
2. **Model Architecture:**
   - **EfficientNet-B0** is used as the backbone for feature extraction, which is pre-trained on ImageNet and fine-tuned for deepfake detection.
   - The **Squeeze-and-Excitation block** (SE Block) is added to the network to recalibrate the feature maps by learning the importance of each channel.
   - A **Fully Connected (FC) layer** is used at the end of the model to classify images into two categories: **Real** or **Fake**.
3. **Training:**
   - The model is trained using the **CrossEntropy Loss**, and **Adam optimizer** is used for weight updates.
   - During training, **checkpointing** is implemented to save the model at regular intervals and at the point of best validation performance.
4. **Evaluation Metrics:**
   - The model's performance is evaluated on the test set using various metrics:
     - **Classification Report**: Precision, Recall, F1-Score for both classes.
     - **Confusion Matrix**: To visualize the model's predictions.
     - **ROC-AUC Score**: To evaluate the model's ability to distinguish between Real and Fake images.
     - **IoU (Jaccard) Score**: To compute the overlap between the predicted and true labels.

1.1.2 Key Features:

- **EfficientNet**: An advanced architecture providing efficient performance with fewer parameters.
- **Squeeze-and-Excitation (SE) Block**: A lightweight attention mechanism that recalibrates the feature maps for better model focus.
- **GPU Support**: The code automatically selects **CUDA-enabled GPUs** if available, ensuring faster training times.
- **Checkpointing**: The model saves checkpoints during training and stores the best performing model, preventing data loss during long training runs.

1.1.3 Step 1: Import Required Libraries

Libraries like `torch`, `torchvision`, and `sklearn` are essential for creating the deep learning model, performing transformations on images, and computing evaluation metrics like classification reports, confusion matrices, and ROC AUC scores. We also import `matplotlib` and `seaborn` for visualizing results such as loss curves and confusion matrices. The `warnings` library is used to suppress unnecessary warnings during training.

[1]:
```
# Importing necessary libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models
import torchvision.transforms as transforms
from torchvision import datasets
from torch.utils.data import DataLoader, ConcatDataset
import cv2
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix,␣
↪roc_auc_score, jaccard_score
import matplotlib.pyplot as plt
import seaborn as sns
import os
import warnings
warnings.filterwarnings('ignore')
```

1.1.4 Step 2: Select Device (GPU or CPU)

To ensure efficient model training, this step checks whether a GPU is available for computation. If a GPU is available, it is used to speed up the training process. Otherwise, the model will fall back on the CPU. The device selection is printed to the console, so the user can confirm which device is being used for the training.

[2]:
```
# Select GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cuda
```

1.1.5 Step 3: Define the Squeeze-and-Excitation (SE) Block

The Squeeze-and-Excitation block (SEBlock) is a lightweight attention mechanism designed to improve the performance of the model. It works by recalibrating feature maps using global information from the feature maps themselves. The `SEBlock` takes the output of convolution layers and applies a global average pooling operation to summarize the feature map's global context. The output is passed through two fully connected (FC) layers to enhance the channel-wise feature representations, followed by a sigmoid activation function to scale the input feature map accordingly.

[3]:
```
# Squeeze-and-Excitation block
class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction=16):
        super(SEBlock, self).__init__()
        # Fully connected layers for squeeze and excitation
        self.fc1 = nn.Linear(in_channels, in_channels // reduction)  # Squeeze phase
        self.fc2 = nn.Linear(in_channels // reduction, in_channels)  # Excitation phase

    def forward(self, x):
        se = F.adaptive_avg_pool2d(x, 1).view(x.size(0), -1)  # Global average pooling
        se = F.relu(self.fc1(se))  # Squeeze phase
        se = torch.sigmoid(self.fc2(se)).view(x.size(0), x.size(1), 1, 1)  # Excitation phase
        return x * se  # Return recalibrated features
```

1.1.6 Step 4: Define the DeepFake Detection Model

In this step, we define the `DeepFakeModel` class, which is the main deep learning model for detecting fake images. We use EfficientNet as the backbone for feature extraction due to its efficiency and accuracy. The last few layers of EfficientNet are fine-tuned, allowing the model to learn from the specific dataset. We then pass the features through the `SEBlock` to enhance important features. The model also includes global average pooling and a final classifier that outputs the probability of an image being "Real" or "Fake."

[4]:
```
# DeepFake Model
class DeepFakeModel(nn.Module):
    def __init__(self):
        super(DeepFakeModel, self).__init__()
        # EfficientNet-B0 backbone pre-trained on ImageNet for feature extraction
        backbone = models.efficientnet_b0(pretrained=True)

        # Fine-tuning the last few layers
        for param in backbone.features[-5:].parameters():
            param.requires_grad = True  # Fine-tune the last few layers
```

```
        self.feature_extractor = backbone.features  # Extract features from␣
↪EfficientNet
        self.se_block = SEBlock(1280)  # Apply SE block to recalibrate features
        self.pool = nn.AdaptiveAvgPool2d(1)  # Global average pooling layer
        self.classifier = nn.Linear(1280, 2)  # Classifier to distinguish Real␣
↪vs Fake

    def forward(self, x):
        x = self.feature_extractor(x)  # Extract features from the input image
        x = self.se_block(x)  # Apply SE block to recalibrate the features
        x = self.pool(x).view(x.size(0), -1)  # Flatten the pooled features
        x = self.classifier(x)  # Pass through classifier to get final␣
↪prediction
        return x
```

1.1.7 Step 5: Define Training Function with Checkpointing

The `train` function is responsible for training the model. It iterates through the dataset for a specified number of epochs and computes the loss at each step. To prevent losing progress if the training is interrupted, checkpointing is implemented. This function saves the model's weights and optimizer states periodically. If a checkpoint exists, the training resumes from the last saved state. The function also tracks the training and validation losses and saves the best model based on validation performance.

[5]:
```
# Training loop
def train(model, train_loader, val_loader, criterion, optimizer, epochs,␣
↪device, checkpoint_path):
    model.train()  # Set model to training mode
    train_losses, val_losses = [], []  # Track training and validation losses
    best_val_loss = float('inf')  # Keep track of best validation loss for␣
↪model saving

    # Load checkpoint if exists
    if os.path.exists(checkpoint_path + "/checkpoint.pth"):
        checkpoint = torch.load(os.path.join(checkpoint_path, "checkpoint.pth"))
        model.load_state_dict(checkpoint['model_state_dict'])  # Load model␣
↪state
        optimizer.load_state_dict(checkpoint['optimizer_state_dict'])  # Load␣
↪optimizer state
        start_epoch = checkpoint['epoch'] + 1  # Start from the next epoch
        print(f"Resuming from epoch {start_epoch}")
    else:
        start_epoch = 0  # Start fresh if no checkpoint is found
        print("Starting training...")

    # Main training loop
    for epoch in range(start_epoch, epochs):
```

```
        model.train()  # Set model to training mode
        running_loss = 0.0  # Track loss during training
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)  # Move data␣
↪to the device
            optimizer.zero_grad()  # Zero the gradients before backward pass
            outputs = model(images)  # Perform forward pass
            loss = criterion(outputs, labels)  # Calculate loss
            loss.backward()  # Backpropagate gradients
            optimizer.step()  # Update weights
            running_loss += loss.item()  # Accumulate loss for the epoch
        avg_train_loss = running_loss / len(train_loader)  # Average training␣
↪loss
        avg_val_loss = validate(model, val_loader, criterion)  # Calculate␣
↪validation loss
        train_losses.append(avg_train_loss)  # Store training loss
        val_losses.append(avg_val_loss)  # Store validation loss
        print(f"Epoch {epoch+1}/{epochs}, Train Loss: {avg_train_loss:.4f}, Val␣
↪Loss: {avg_val_loss:.4f}")

        # Save best model if validation loss improves
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            torch.save(model.state_dict(), os.path.join(checkpoint_path,␣
↪"best_model.pth"))
            print("Saved best model!")

        # Save checkpoint at every epoch
        torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': avg_train_loss,
        }, os.path.join(checkpoint_path, "checkpoint.pth"))

    print("Training completed!")
    # Plot and save loss curves
    plt.figure(figsize=(10, 6))
    plt.plot(train_losses, label='Training Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training vs Validation Loss")
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(checkpoint_path, "loss_plot.png"))
    plt.show()
```

1.1.8 Step 6: Define Validation Function

The validation function is used to evaluate the model on the validation set after each epoch. It runs the model in evaluation mode (i.e., no weight updates) and calculates the loss using the same loss function as in training. This helps track how well the model is generalizing to unseen data during the training process.

[6]:
```
# Validation loop
def validate(model, val_loader, criterion):
    model.eval()  # Set model to evaluation mode
    val_loss = 0.0  # Initialize validation loss
    with torch.no_grad():  # Disable gradient computation during validation
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)  # Move data␣
↪to device
            outputs = model(images)  # Forward pass
            loss = criterion(outputs, labels)  # Calculate loss
            val_loss += loss.item()  # Accumulate validation loss
    return val_loss / len(val_loader)  # Return average validation loss
```

1.1.9 Step 7: Define Test and Evaluation Function

The `test` function evaluates the final model on the test set. It computes the predictions for the test images and compares them to the true labels. A classification report is generated to show the precision, recall, and F1-score for both classes (Real and Fake). Additionally, a confusion matrix is plotted to visualize how well the model distinguishes between the two classes.

[7]:
```
# Test and evaluation
def test(model, dataloader):
    model.eval()  # Set model to evaluation mode
    all_preds = []  # List to store predictions
    all_labels = []  # List to store true labels
    with torch.no_grad():  # No gradient calculation during testing
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)  # Move data␣
↪to device
            outputs = model(images)  # Get predictions
            _, predicted = torch.max(outputs.data, 1)  # Get predicted class
            all_preds.extend(predicted.cpu().numpy())  # Store predictions
            all_labels.extend(labels.cpu().numpy())  # Store true labels

    # Print classification report
    print("\nClassification Report:")
    print(classification_report(all_labels, all_preds, target_names=["Real",␣
↪"Fake"]))

    # Plot confusion matrix
    cm = confusion_matrix(all_labels, all_preds)
```

```
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Real",␣
↪"Fake"], yticklabels=["Real", "Fake"])
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title("Confusion Matrix")
    plt.savefig(os.path.join(checkpoint_path, "confusion_matrix.png"))
    plt.show()
```

1.1.10 Step 8: Define a Function to Compute the ROC AUC Score

The Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) score are commonly used metrics for evaluating the performance of binary classification models. In this step, we compute the AUC score, which measures the model's ability to distinguish between the "Real" and "Fake" classes. A higher AUC indicates better performance, where a score of 1.0 represents perfect classification.

[8]:
```
# Compute ROC AUC Score
def compute_roc_auc(model, dataloader):
    model.eval()  # Set model to evaluation mode
    all_labels = []  # True labels
    all_probs = []  # Predicted probabilities
    with torch.no_grad():  # No gradient calculation during evaluation
        for images, labels in dataloader:
            images = images.to(device)  # Move data to device
            outputs = model(images)  # Get model outputs
            probs = F.softmax(outputs, dim=1)[:, 1]  # Get probability for the␣
↪"Fake" class
            all_probs.extend(probs.cpu().numpy())  # Store predicted␣
↪probabilities
            all_labels.extend(labels.numpy())  # Store true labels
    auc = roc_auc_score(all_labels, all_probs)  # Compute ROC AUC score
    print(f"ROC AUC Score: {auc:.4f}")
```

1.1.11 Step 9: Define a Function to Compute Intersection over Union (IoU) Score

Intersection over Union (IoU), also known as the Jaccard Index, is another evaluation metric for binary classification problems. This metric measures the overlap between the predicted and actual classes. A higher IoU indicates that the model's predictions align more closely with the true labels.

[9]:
```
# Compute IoU (Jaccard) Score
def compute_iou(model, dataloader):
    model.eval()  # Set model to evaluation mode
    all_preds = []  # Store predictions
    all_labels = []  # Store true labels
    with torch.no_grad():  # Disable gradient computation during evaluation
        for images, labels in dataloader:
```

```
            images = images.to(device)  # Move data to device
            outputs = model(images)  # Get predictions
            _, preds = torch.max(outputs, 1)  # Get predicted classes
            all_preds.extend(preds.cpu().numpy())  # Store predictions
            all_labels.extend(labels.numpy())  # Store true labels
    iou = jaccard_score(all_labels, all_preds, average='binary')  # Compute IoU␣
 ↪score
    print(f"IoU (Jaccard) Score: {iou:.4f}")
```

1.1.12 Step 10: Define Image Transformations

In this step, we define the image transformations that will be applied to the dataset. The images are resized to a fixed size of 256x256 pixels and then converted to tensor format. This transformation ensures that the input images are in a format that the model can process efficiently.

[10]:
```
# Image transformation
transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor()
])
```

1.1.13 Step 11: Load Dataset and Create DataLoaders

The dataset for training, validation, and testing is loaded from directories containing images. The `ImageFolder` class from `torchvision.datasets` is used to automatically assign labels based on the folder names. After the datasets are loaded, DataLoader objects are created to handle batching, shuffling, and parallel data loading during training and evaluation.

[11]:
```
# Checkpoint path
checkpoint_path = "/kaggle/working/"
```

[12]:
```
# Dataset
train_dataset = datasets.ImageFolder("/kaggle/input/deepfake-and-real-images/
 ↪Dataset/Train", transform=transform)
val_dataset = datasets.ImageFolder("/kaggle/input/deepfake-and-real-images/
 ↪Dataset/Test", transform=transform)
test_dataset = datasets.ImageFolder("/kaggle/input/deepfake-and-real-images/
 ↪Dataset/Validation", transform=transform)
```

[13]:
```
# DataLoader
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

1.1.14 Step 12: Initialize Model, Loss Function, Optimizer

Here, we initialize the deep learning model (`DeepFakeModel`), the loss function (`CrossEntropyLoss`), and the optimizer (`Adam`). The model is transferred to the selected device (GPU or CPU). The loss function computes the error between the predicted and actual labels, while the optimizer updates the model weights based on the computed gradients.

[14]:
```
# Model, Loss, Optimizer
model = DeepFakeModel().to(device)
criterion = nn.CrossEntropyLoss()  # Cross-entropy loss for classification
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  # Adam optimizer␣
↪with learning rate 0.001
```

```
Downloading:
"https://download.pytorch.org/models/efficientnet_b0_rwightman-7f5810bc.pth" to
/root/.cache/torch/hub/checkpoints/efficientnet_b0_rwightman-7f5810bc.pth
100%|      | 20.5M/20.5M [00:00<00:00, 86.8MB/s]
```

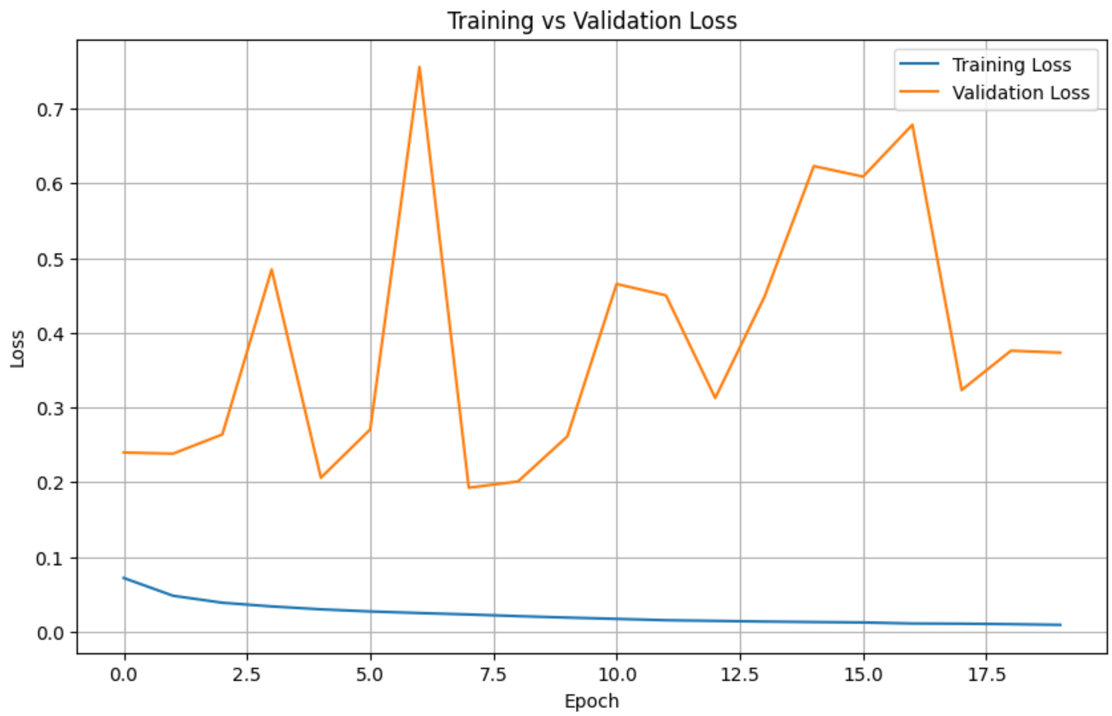
1.1.15 Step 13: Train the Model

The training process begins with the `train` function. This function will train the model for a specified number of epochs (20 in this case). During each epoch, the model will learn from the training data and adjust its weights accordingly. The training loss and validation loss are printed after every epoch to monitor the model's progress.

[15]:
```
# Start the training process
train(model, train_loader, val_loader, criterion, optimizer, epochs=20,␣
↪device=device, checkpoint_path=checkpoint_path)
```

```
Starting training…
Epoch 1/20, Train Loss: 0.0718, Val Loss: 0.2397
Saved best model!
Epoch 2/20, Train Loss: 0.0481, Val Loss: 0.2383
Saved best model!
Epoch 3/20, Train Loss: 0.0387, Val Loss: 0.2639
Epoch 4/20, Train Loss: 0.0338, Val Loss: 0.4848
Epoch 5/20, Train Loss: 0.0299, Val Loss: 0.2060
Saved best model!
Epoch 6/20, Train Loss: 0.0270, Val Loss: 0.2707
Epoch 7/20, Train Loss: 0.0249, Val Loss: 0.7558
Epoch 8/20, Train Loss: 0.0230, Val Loss: 0.1926
Saved best model!
Epoch 9/20, Train Loss: 0.0207, Val Loss: 0.2011
Epoch 10/20, Train Loss: 0.0188, Val Loss: 0.2614
Epoch 11/20, Train Loss: 0.0171, Val Loss: 0.4654
Epoch 12/20, Train Loss: 0.0153, Val Loss: 0.4501
Epoch 13/20, Train Loss: 0.0144, Val Loss: 0.3128
Epoch 14/20, Train Loss: 0.0135, Val Loss: 0.4484
Epoch 15/20, Train Loss: 0.0129, Val Loss: 0.6232
```

```
Epoch 16/20, Train Loss: 0.0123, Val Loss: 0.6091
Epoch 17/20, Train Loss: 0.0109, Val Loss: 0.6785
Epoch 18/20, Train Loss: 0.0106, Val Loss: 0.3235
Epoch 19/20, Train Loss: 0.0100, Val Loss: 0.3760
Epoch 20/20, Train Loss: 0.0091, Val Loss: 0.3735
Training completed!
```



1.1.16 Step 14: Save Final Model

After training is complete, the model's weights are saved to a file (`model.pth`). This allows us to load the model later for inference or further fine-tuning.

[16]:
```python
# Save the trained model
torch.save(model.state_dict(), os.path.join(checkpoint_path, "model.pth"))
```

1.1.17 Step 15: Run Final Evaluation

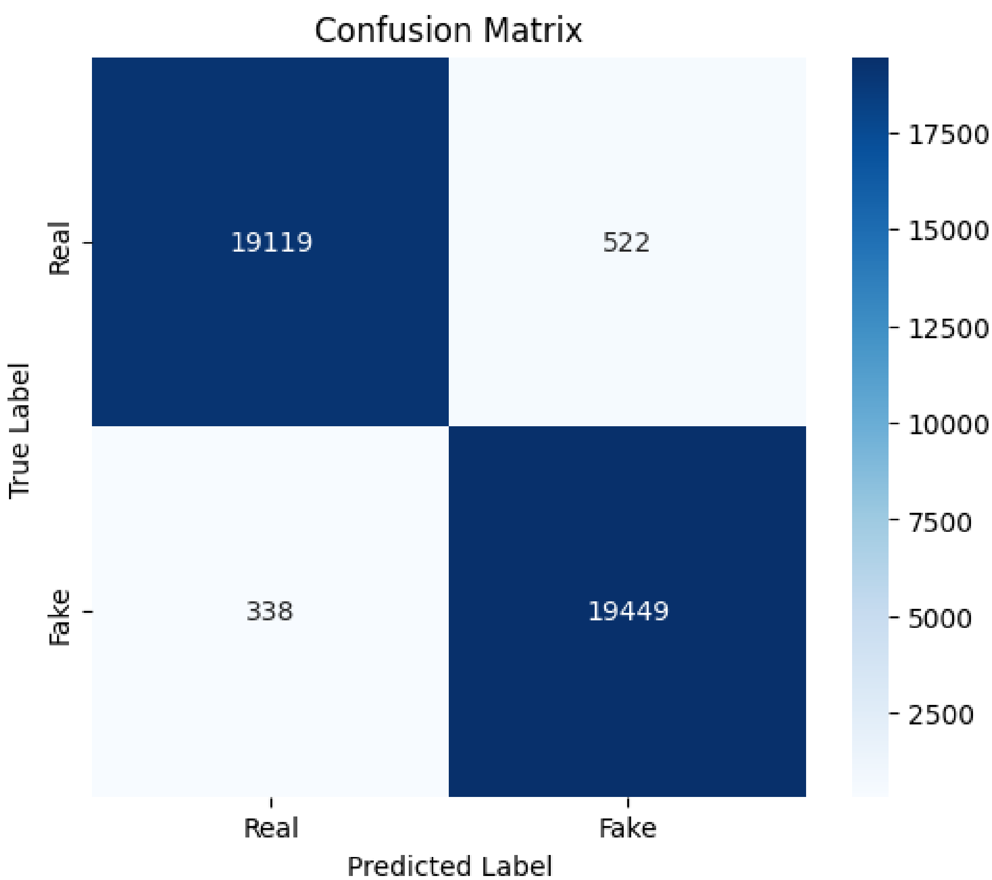
After training, we evaluate the model on the test dataset. This includes running the `test`, `compute_roc_auc`, and `compute_iou` functions to assess the model's classification performance and generalization ability. These evaluations provide insight into how well the model can detect fake images in real-world scenarios.

[17]:
```python
# Evaluate the model on test data
test(model, test_loader)
```

```
Classification Report:
              precision    recall  f1-score   support

        Real       0.98      0.97      0.98     19641
        Fake       0.97      0.98      0.98     19787

    accuracy                           0.98     39428
   macro avg       0.98      0.98      0.98     39428
weighted avg       0.98      0.98      0.98     39428
```



[18]:
```
# Compute and print ROC AUC score
compute_roc_auc(model, test_loader)
```

```
ROC AUC Score: 0.9977
```

[19]:
```
# Compute and print IoU score
compute_iou(model, test_loader)
```

```
IoU (Jaccard) Score: 0.9577
```

[20]:

```python
# Print class-to-index mapping
print(train_dataset.class_to_idx)
```

```
{'Fake': 0, 'Real': 1}
```

Unifies face detection and deepfake detection detection models.

April 19, 2025

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models, transforms
from PIL import Image

# --- Face Detector ---
class FaceDetectorCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # Define the feature extraction layers
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),
            nn.Conv2d(32, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
            nn.Conv2d(64, 64, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2)
        )
        # Define the classifier layers
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 12 * 12, 128), nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(128, 1), nn.Sigmoid()
        )

    def forward(self, x):
        # Pass the input through feature extractor and
           classifier
        x = self.features(x)
        return self.classifier(x)

# --- SE Block ---
class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction=16):
```

```python
        super().__init__()
        # Define the fully connected layers for SE block
        self.fc1 = nn.Linear(in_channels, in_channels // reduction)
        self.fc2 = nn.Linear(in_channels // reduction, in_channels)

    def forward(self, x):
        # Apply SE block: global average pooling, fully connected layers, and scaling
        se = F.adaptive_avg_pool2d(x, 1).view(x.size(0), -1)
        se = F.relu(self.fc1(se))
        se = torch.sigmoid(self.fc2(se)).view(x.size(0), x.size(1), 1, 1)
        return x * se

# --- DeepFake Model using EfficientNet ---
class DeepFakeModel(nn.Module):
    def __init__(self):
        super(DeepFakeModel, self).__init__()
        # Load EfficientNet backbone
        backbone = models.efficientnet_b0(pretrained=True)
        # Freeze all layers except the last 5
        for param in backbone.features[-5:].parameters():
            param.requires_grad = True

        self.feature_extractor = backbone.features
        self.se_block = SEBlock(1280)  # SE block
        self.pool = nn.AdaptiveAvgPool2d(1)  # Adaptive average pooling
        self.classifier = nn.Linear(1280, 2)  # Classifier for real vs deepfake

    def forward(self, x):
        # Pass input through feature extractor, SE block, and classifier
        x = self.feature_extractor(x)
        x = self.se_block(x)
        x = self.pool(x).view(x.size(0), -1)
        x = self.classifier(x)
        return x

# --- Unified Model ---
class UnifiedFaceDeepFakeModel(nn.Module):
    def __init__(self):
        super().__init__()
        # Initialize face detector model
        self.face_detector = FaceDetectorCNN()
        # Initialize EfficientNet backbone for deepfake classification
```

```
        backbone = models.efficientnet_b0(pretrained=True)
        self.df_feature_extractor = backbone.features
        self.se_block = SEBlock(1280)
        self.pool = nn.AdaptiveAvgPool2d(1)
        self.df_classifier = nn.Linear(1280, 2)  # 2 classes : real (1) or deepfake (0)

    def forward(self, face_tensor, df_tensor):
        # Run face detection first
        face_score = self.face_detector(face_tensor)
        if face_score.item() <= 0.5:
            # No face detected, return -1 to indicate no face
            return torch.tensor([-1])

        # If face detected, proceed with deepfake model
        features = self.df_feature_extractor(df_tensor)
        features = self.se_block(features)
        pooled = self.pool(features).view(features.size(0), -1)
        logits = self.df_classifier(pooled)
        probs = F.softmax(logits, dim=1)
        prediction = torch.argmax(probs, dim=1)  # 0 = DeepFake, 1 = Real
        return prediction

# --- Transforms ---
# Transform for the face detection model (resize and normalize)
face_transform = transforms.Compose([
    transforms.Resize((48, 48)),
    transforms.ToTensor()
])

# Transform for the deepfake detection model (resize and normalize)
df_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor()
])

# --- Load Model ---
# Set device for model (GPU or CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = UnifiedFaceDeepFakeModel().to(device)

# Load the pre-trained model weights
# Make sure to provide paths to the saved model weights
model.face_detector.load_state_dict(torch.load("
```

```python
faceclassifier.pth"))
model.df_feature_extractor.load_state_dict(torch.load("deepfake.pth"), strict=False)

# Set the model to evaluation mode (disables dropout layers)
model.eval()

# --- Inference Function ---
def run_inference(image_path):
    """
    Run inference on the given image to detect whether it contains a real face or a deepfake.
    :param image_path: Path to the image file.
    """
    # Open the image and convert it to RGB
    image = Image.open(image_path).convert("RGB")

    # Transform the image for the face detection model and deepfake model
    face_tensor = face_transform(image).unsqueeze(0).to(device)
    df_tensor = df_transform(image).unsqueeze(0).to(device)

    # Disable gradient computation for inference
    with torch.no_grad():
        result = model(face_tensor, df_tensor).item()

    # Print the result of the inference
    if result == -1:
        print("No face detected.")
    elif result == 1:
        print("Real face detected.")
    else:
        print("DeepFake detected.")

# --- Save Unified Model ---
def save_model(model, path="unified_model_final.pth"):
    """
    Save the model state dictionary to a file.
    :param model: The model to be saved.
    :param path: The file path where the model will be saved.
    """
    torch.save(model.state_dict(), path)
    print(f"Model saved at: {path}")

# Save the model
save_model(model)

# --- Test ---
```

```
# Test the model with an image
run_inference("test.jpg")
```