FACE RECOGNITION ATTENDENCE

TRAN DUC ANH DANG

Abstract

This report presents the development and evaluation of a face recognition attendance system with an integrated anti-spoofing module. The system employs Convolutional Neural Networks (CNNs) and the Triplet Loss function to generate and compare face embeddings for verification purposes. Hyperparameter tuning, including increasing the margin to 1.2 and reducing the learning rate to 0.0001, significantly improved the model's stability and performance. The system was trained on a reduced dataset of 1000 samples with less than 50 epochs, achieving a Receiver Operating Characteristic (ROC) Area Under the Curve (AUC) score of 0.92 on the test dataset.

To enhance security, the system incorporates an anti-spoofing module that uses Eye Aspect Ratio (EAR) to detect blinks and advanced facial analysis techniques to identify deepfakes, facial expressions, and valence arousal estimations. The Chebyshev distance metric was found to provide the best results for face verification. The user interface, developed using Streamlit, offers an intuitive and interactive experience for face registration, verification, and liveness detection.

Overall, the system demonstrates high accuracy in distinguishing between similar and dissimilar faces, robust anti-spoofing capabilities, and user-friendly interaction, making it a reliable solution for face recognition-based attendance verification. Future work will focus on improving triplet selection strategies, further hyperparameter optimization, advanced data augmentation techniques, and real-world deployment to ensure robustness and scalability.

Table of Contents

Introduction

Overview of Face Recognition

Face recognition technology has become a critical component of modern security and authentication systems. It can be broadly classified into two main tasks: **face classification** and **face verification**.

Face Classification involves identifying and categorizing a person's face into a predefined set of face IDs. This is typically a closed-set problem where the model is trained to recognize faces from a fixed database. For example, in a corporate environment, the system would classify an employee's face into their corresponding ID based on the training data.

Face Verification, on the other hand, focuses on determining whether two face images belong to the same person. This is an open-set problem, where the model may encounter new face identities not present in the training dataset. The goal here is to produce a similarity score that quantifies the likelihood that the faces in the two images are of the same person. This process is crucial for applications that require confirming identities, such as unlocking devices, accessing secure areas, or verifying attendance.

The distinction between face classification and face verification is essential as it highlights the different approaches and challenges involved in designing effective face recognition systems.

Project Objective

The primary objective of this project is to design and implement an end-to-end Face Recognition Attendance System for an enterprise environment. This system will integrate face verification techniques using convolutional neural networks (CNNs) to ensure accurate and reliable identification of individuals. Additionally, to enhance the security and robustness of the system, an anti-spoofing module will be incorporated to detect and prevent attempts to deceive the system using fake or non-real faces.

Key Objectives

1. **Face Registration**: Enable the system to register new face IDs for employees as they are hired. This ensures that the database of recognized faces is continually updated.
2. **Face Verification**: Implement a CNN-based face verification system that can accurately determine whether two face images belong to the same person, providing a numerical similarity score.
3. **Anti-Spoofing Module**: Develop and integrate an anti-spoofing mechanism to detect and prevent spoofing attempts using images or videos of faces, ensuring that only live, real faces are recognized.
4. **User Interface**: Create a user friendly interface that allows easy interaction with the system for registration, verification, and detection processes.

5. **Evaluation**: Evaluate the system's performance using standard metrics such as the Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) to ensure its accuracy and reliability.

System Components

The system comprises several key components, each designed to fulfill a specific function within the Face Recognition Attendance System.

Face Verification Model (models.py):

- Utilizes the **InceptionV3** architecture, which includes several **convolutional layers**, **pooling layers**, **inception modules (comprising parallel convolutions with different kernel sizes)**, and **fully connected layers**. The architecture is modified to produce face embeddings.
- The final layer of **InceptionV3** is replaced with a fully connected layer followed by batch normalization, which outputs embeddings. These embeddings are normalized vectors representing the unique features of each face.

Dataset Handling (datasets.py):

- Provides custom dataset classes to manage **training**, **validation**, and **testing** data for face verification tasks.
- Preprocesses images by **resizing** them to a consistent size, **normalizing pixel values**, and **applying data augmentations** to enhance model training robustness.
- Implements efficient data loading mechanisms using PyTorch's **DataLoader**, which streamlines the training process by enabling batch processing and shuffling of data.

Training Pipeline (train.py):

- Includes a custom training loop that handles model **training**, **validation**, and **evaluation** across multiple epochs.
- Utilizes **TensorBoard** for logging training metrics, such as **loss** and **accuracy**, and for **visualizing** training progress and model performance.
- Saves the best model based on **validation loss** to ensure optimal performance, ensuring that the model with the lowest validation loss is preserved for deployment.

Application Script (app.py):

- Integrates various system components to create an interactive user interface using Streamlit, enabling users to interact with the face recognition system easily.
- Implements **live face detection** and **verification** using a webcam feed, providing realtime feedback to the user.

- Ensures **liveness detection** through **eye aspect ratio (EAR)** calculations, which helps to detect blinks and prevent **spoofing** attempts using static images or videos.
- Provides functionalities for **face registration**, **verification**, and **displaying results**, allowing users to register new faces, **verify identities**, and **view** the outcomes of the verification process.

Helper Functions (helper.py):

- Contains utility functions for **loading images**, **creating embeddings** from face images, and **identifying** faces by comparing embeddings.
- Provides methods for **drawing face frames** on images, **cropping images** to focus on detected faces, and **performing** model inference to predict face embeddings.
- Implements **anti-spoofing** checks by **analyzing facial expressions**, **deepfake detection**, and **valence-arousal estimation**, enhancing the security and reliability of the face recognition system.

Methodology

System Design

The system design focuses on creating a robust and scalable face recognition attendance system integrated with a liveness detection module. The primary components include a face verification model, a dataset handling mechanism, a training pipeline, an application script for user interaction, and various helper functions for processing and inference.

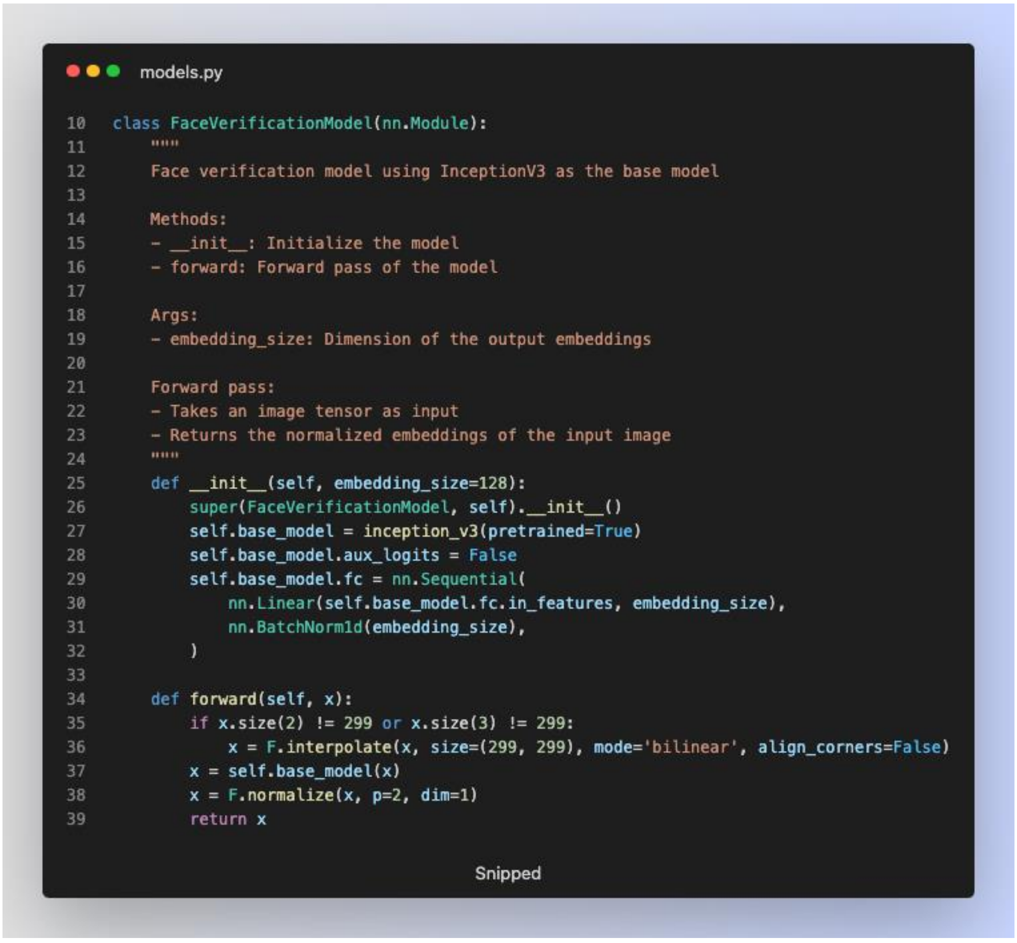
Face Verification

Face Embedding

The face embedding process involves transforming raw face images into compact, low-dimensional vectors that capture the unique features of each face. This is achieved using a **convolutional neural network (CNN)** based on the **InceptionV3** architecture.

InceptionV3 Architecture:

- The **InceptionV3** model includes several **convolutional layers, pooling layers, inception modules (which comprise parallel convolutions with different kernel sizes)**, and **fully connected layers**.
- In our system, the final fully connected layer of **InceptionV3** is replaced with a new fully connected layer followed by **batch normalization**, specifically designed to **output face embeddings**.
- **These embeddings** are **normalized** to unit length, producing vectors that effectively capture the unique features of each face, facilitating comparison and verification.



```python
class FaceVerificationModel(nn.Module):
    """
    Face verification model using InceptionV3 as the base model

    Methods:
    - __init__: Initialize the model
    - forward: Forward pass of the model

    Args:
    - embedding_size: Dimension of the output embeddings

    Forward pass:
    - Takes an image tensor as input
    - Returns the normalized embeddings of the input image
    """
    def __init__(self, embedding_size=128):
        super(FaceVerificationModel, self).__init__()
        self.base_model = inception_v3(pretrained=True)
        self.base_model.aux_logits = False
        self.base_model.fc = nn.Sequential(
            nn.Linear(self.base_model.fc.in_features, embedding_size),
            nn.BatchNorm1d(embedding_size),
        )

    def forward(self, x):
        if x.size(2) != 299 or x.size(3) != 299:
            x = F.interpolate(x, size=(299, 299), mode='bilinear', align_corners=False)
        x = self.base_model(x)
        x = F.normalize(x, p=2, dim=1)
        return x
```

Triplet Loss Function:

- To train the face verification model, we implement the Triplet Loss function. This function aims to minimize the distance between an anchor image (same person) and a positive image while maximizing the distance between the anchor and a negative image (different person).
- The formula for the triplet loss function is:

$$L(a,p,n) = \max\{d(a_i,p_i) - d(a_i,n_i) + \mathrm{margin}, 0\}$$

where

$$d(x_i,y_i) = \left\|\mathbf{x}_i - \mathbf{y}_i\right\|_p$$

The norm is calculated using the specified p p value, and a small constant ϵ ϵ is added for numerical stability.

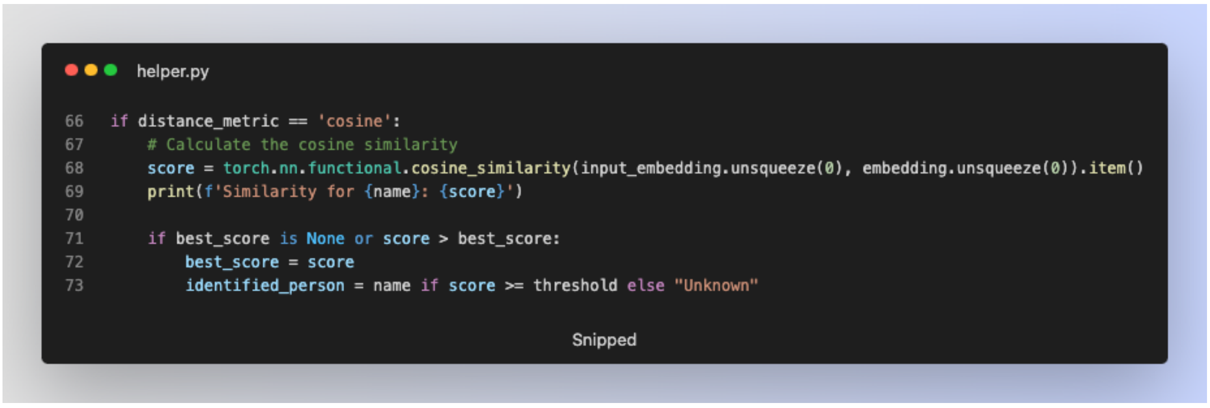
- This loss function helps the model learn to effectively differentiate between similar and dissimilar faces.
- Training with triplet loss is challenging because it requires careful selection of triplets (anchor, positive, and negative samples) to ensure effective learning. If the triplets are not well-selected, the model may not learn meaningful differences between similar and dissimilar faces. This selection process is computationally intensive and often involves strategies like hard negative mining to identify the most challenging negative examples for effective training.

Similarity Metrics

To verify whether two face images belong to the same person, we compute the similarity between their embeddings using distance metrics.

Cosine Similarity:

- Measures the cosine of the angle between two non-zero vectors, producing a similarity score. A score of 1 indicates identical embeddings, while a score of -1 indicates completely dissimilar embeddings.
- Cosine similarity is particularly useful when the magnitude of the vectors is not important, and the focus is on the orientation of the vectors in the embedding space.

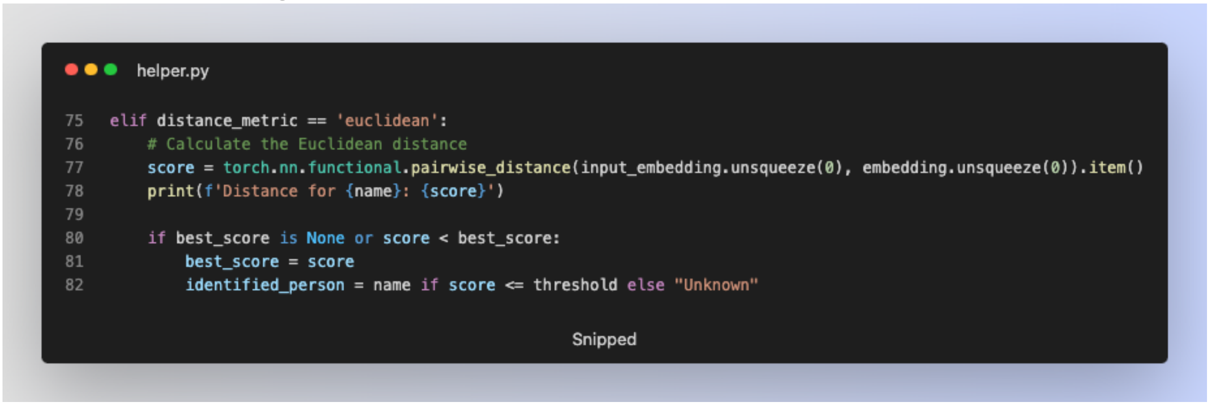

helper.py

```python
if distance_metric == 'cosine':
    # Calculate the cosine similarity
    score = torch.nn.functional.cosine_similarity(input_embedding.unsqueeze(0), embedding.unsqueeze(0)).item()
    print(f'Similarity for {name}: {score}')

    if best_score is None or score > best_score:
        best_score = score
        identified_person = name if score >= threshold else "Unknown"
```

Snipped

Euclidean Distance:

- Computes the straight-line distance between two points in the embedding space. Smaller distances indicate higher similarity, and larger distances indicate lower similarity.
- Euclidean distance is intuitive and straightforward, making it a common choice for measuring similarity.

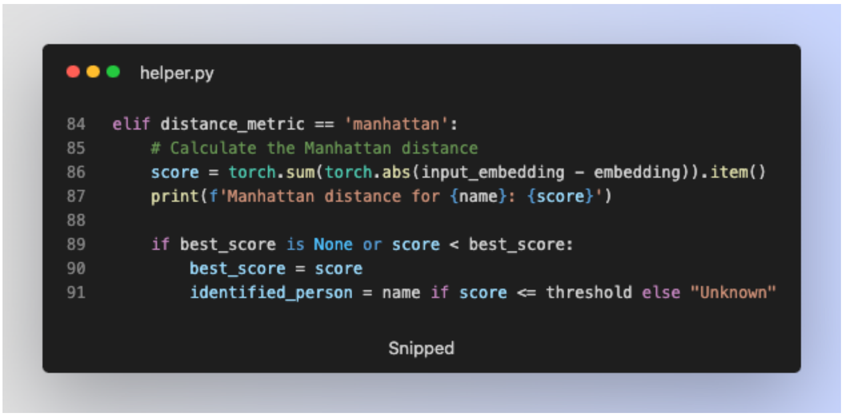

helper.py

```python
elif distance_metric == 'euclidean':
    # Calculate the Euclidean distance
    score = torch.nn.functional.pairwise_distance(input_embedding.unsqueeze(0), embedding.unsqueeze(0)).item()
    print(f'Distance for {name}: {score}')

    if best_score is None or score < best_score:
        best_score = score
        identified_person = name if score <= threshold else "Unknown"
```

Snipped

Manhattan Distance:

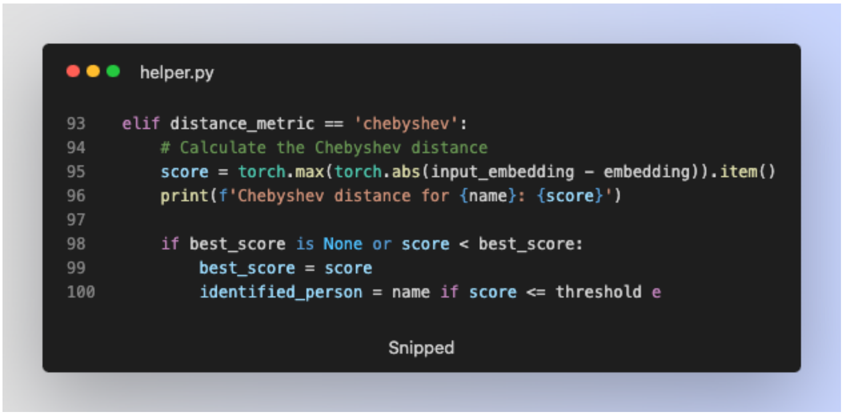
- Also known as L1 distance, it calculates the sum of the absolute differences between the coordinates of the points.
- Manhattan distance is useful in high-dimensional spaces where the differences in each dimension are additive.


helper.py

```python
elif distance_metric == 'manhattan':
    # Calculate the Manhattan distance
    score = torch.sum(torch.abs(input_embedding - embedding)).item()
    print(f'Manhattan distance for {name}: {score}')

    if best_score is None or score < best_score:
        best_score = score
        identified_person = name if score <= threshold else "Unknown"
```

Chebyshev Distance:

- Measures the maximum absolute difference between the coordinates of the points.
- Chebyshev distance is effective when the focus is on the most significant difference between any dimensions.


helper.py

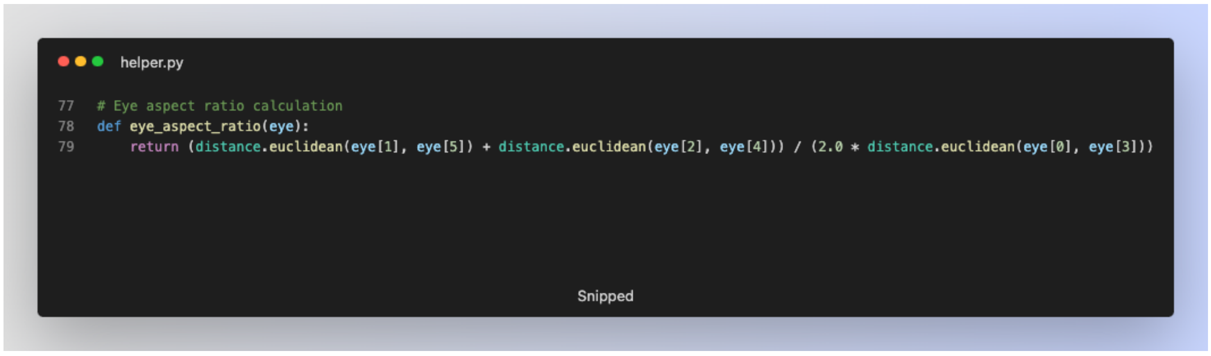
```python
elif distance_metric == 'chebyshev':
    # Calculate the Chebyshev distance
    score = torch.max(torch.abs(input_embedding - embedding)).item()
    print(f'Chebyshev distance for {name}: {score}')

    if best_score is None or score < best_score:
        best_score = score
        identified_person = name if score <= threshold e
```

Anti-Spoofing Module

The anti-spoofing module is crucial for ensuring the security and reliability of the face recognition system. It prevents fraudulent attempts to deceive the system using static images or videos of faces.

Eye Aspect Ratio (EAR) Calculation:

- The EAR is used to detect blinks, ensuring that the face being verified is live. This is calculated by analyzing the eye landmarks detected using dlib's shape predictor.
- The EAR threshold is set, and if a certain number of blinks are detected within a time frame, the face is considered live.



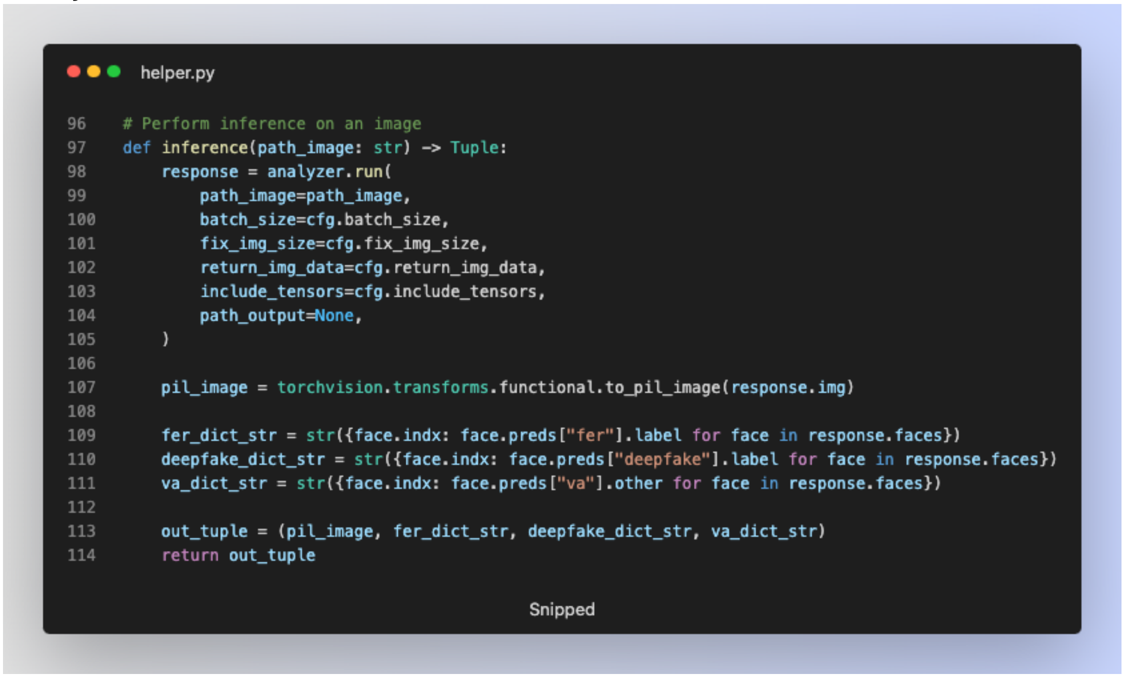
helper.py

```python
# Eye aspect ratio calculation
def eye_aspect_ratio(eye):
    return (distance.euclidean(eye[1], eye[5]) + distance.euclidean(eye[2], eye[4])) / (2.0 * distance.euclidean(eye[0], eye[3]))
```

Snipped

Facial Expression and DeepFake Detection:

- The system analyzes **facial expressions** to detect unusual patterns that may indicate spoofing attempts.
- **DeepFake** detection techniques are employed to identify manipulated or synthetic faces that are not real.



helper.py

```python
# Perform inference on an image
def inference(path_image: str) -> Tuple:
    response = analyzer.run(
        path_image=path_image,
        batch_size=cfg.batch_size,
        fix_img_size=cfg.fix_img_size,
        return_img_data=cfg.return_img_data,
        include_tensors=cfg.include_tensors,
        path_output=None,
    )

    pil_image = torchvision.transforms.functional.to_pil_image(response.img)

    fer_dict_str = str({face.indx: face.preds["fer"].label for face in response.faces})
    deepfake_dict_str = str({face.indx: face.preds["deepfake"].label for face in response.faces})
    va_dict_str = str({face.indx: face.preds["va"].other for face in response.faces})

    out_tuple = (pil_image, fer_dict_str, deepfake_dict_str, va_dict_str)
    return out_tuple
```

Snipped

User Interface

The user interface is designed to be intuitive and user-friendly, allowing easy interaction with the system for face registration, verification, and liveness detection.

Streamlit:

- **Streamlit** is used to create the interactive user interface, providing realtime feedback and control to the user.
- Users can initiate face **verification**, **reset** the system, and **register** new faces through simple buttons and input fields.

Live Webcam Feed:

- The system captures **live video frames** through **webcam**, processes them in realtime, and displays the results back to the user.
- **Face detection**, **frame drawing**, and **liveness verification** are performed continuously to provide an interactive experience.

Face Registration:

- **New faces** can be registered by uploading an image through the interface. The system detects the face in the **image**, **processes** it, and **saves** for future verification.

Implementation

Code Structure

```
.
├── app.py
├── datasets.py
├── facetorch
│   └── config.merged.yml
├── helper.py
├── known_face
│   ├── biden.jpg
│   ├── ducanh.jpg
│   ├── obama.jpg
│   └── trump.jpg
├── models.py
├── predictor
│   └── shape_predictor_68_face_landmarks.dat
├── saved
│   └── captured_face.jpg
├── test.py
└── train.py
```

Directories

Main Files:

- **app.py:** The main application script that integrates the various components and provides the user interface using Streamlit.
- **datasets.py:** Contains custom dataset classes and data loading mechanisms for training, validation, and testing.
- **helper.py:** Includes utility functions for image processing, embedding creation, face identification, and anti-spoofing checks.
- **models.py:** Defines the face verification model used for training.

- **test.py:** Script for testing the trained model on the test dataset.
- **train.py:** Script for training the face verification model.

Subdirectories:

facetorch:

- **config.merged.yml**: Configuration file for the FaceTorch analyzer used in inference.

known_face:

- **Database** folder contain images of known faces for embedding creation and verification.

predictor:

- **shape_predictor_68_face_landmarks.dat**: Pretrained model for facial landmarks detection used in liveness detection.

saved:

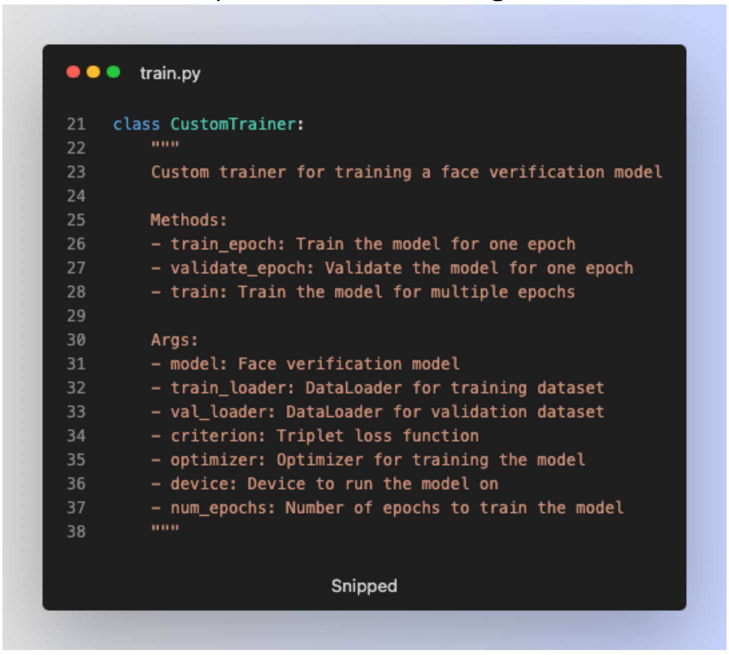
- **captured_face.jpg:** Stores the captured face images during verification.

Model Training

The model training process involves several key steps, encapsulated in the **train.py** script.

Custom Trainer Class:

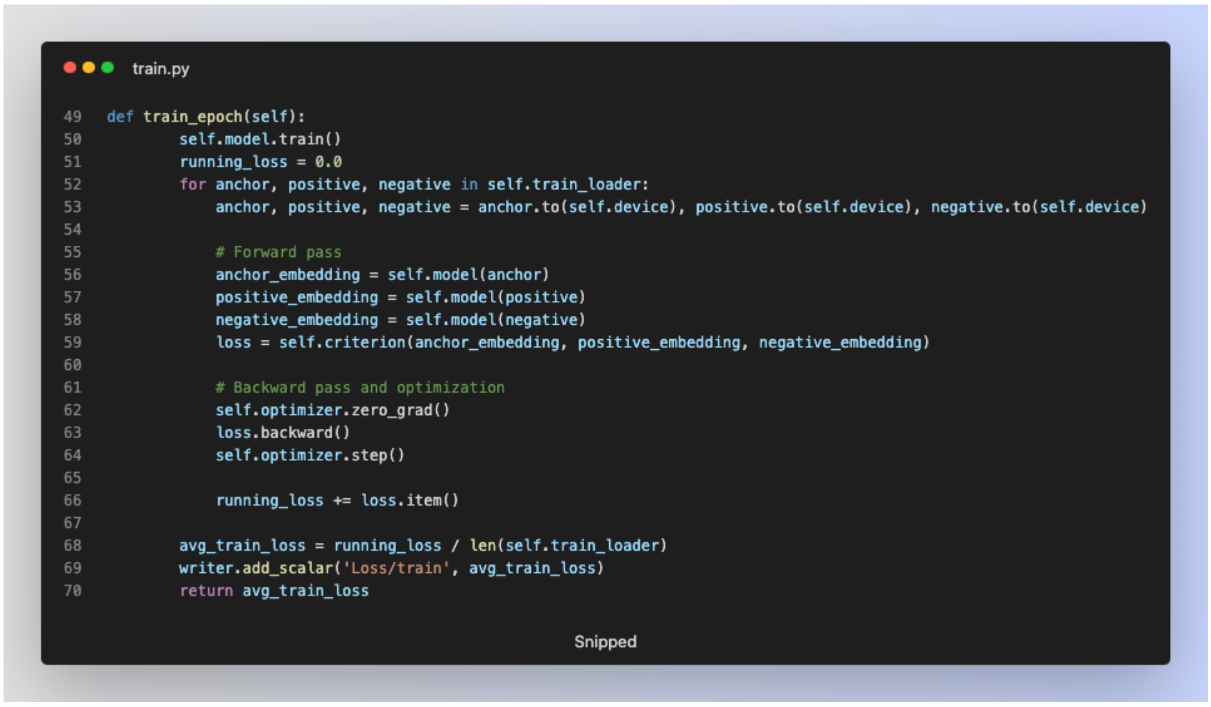
- A CustomTrainer class is defined to handle the training process, including training and validation loops, and model saving.



```python
class CustomTrainer:
    """
    Custom trainer for training a face verification model

    Methods:
    - train_epoch: Train the model for one epoch
    - validate_epoch: Validate the model for one epoch
    - train: Train the model for multiple epochs

    Args:
    - model: Face verification model
    - train_loader: DataLoader for training dataset
    - val_loader: DataLoader for validation dataset
    - criterion: Triplet loss function
    - optimizer: Optimizer for training the model
    - device: Device to run the model on
    - num_epochs: Number of epochs to train the model
    """
```

Training Loop:

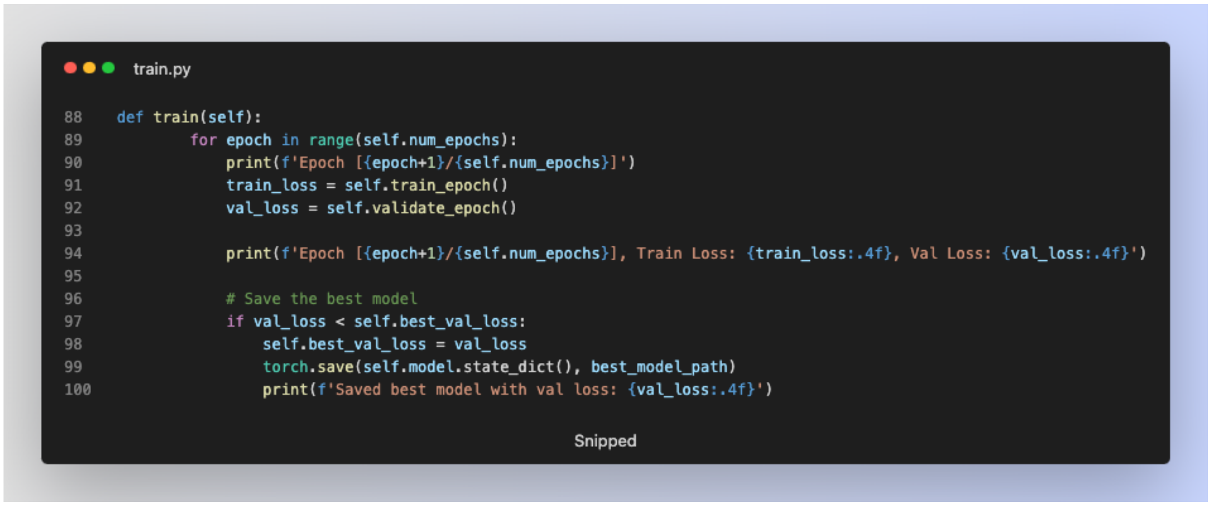
- The training loop iterates over the dataset for a specified number of epochs.
- During each epoch, the model processes batches of data, computes embeddings, and calculates the triplet loss.
- The optimizer updates the model weights to minimize the loss.



train.py

```python
def train_epoch(self):
        self.model.train()
        running_loss = 0.0
        for anchor, positive, negative in self.train_loader:
            anchor, positive, negative = anchor.to(self.device), positive.to(self.device), negative.to(self.device)

            # Forward pass
            anchor_embedding = self.model(anchor)
            positive_embedding = self.model(positive)
            negative_embedding = self.model(negative)
            loss = self.criterion(anchor_embedding, positive_embedding, negative_embedding)

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

            running_loss += loss.item()

        avg_train_loss = running_loss / len(self.train_loader)
        writer.add_scalar('Loss/train', avg_train_loss)
        return avg_train_loss
```

Snipped



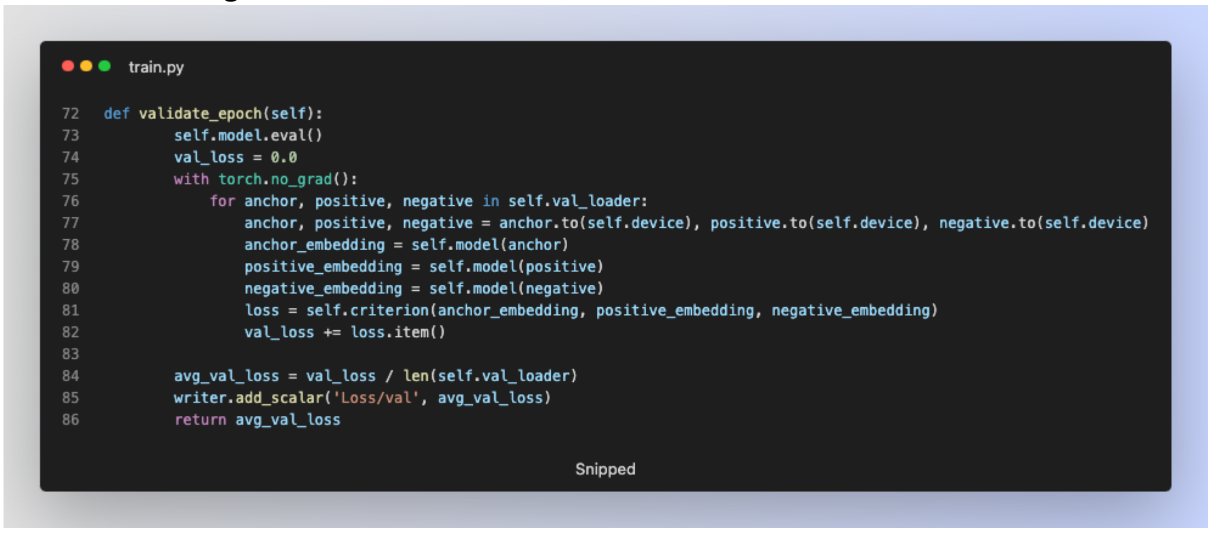
train.py

```python
def train(self):
        for epoch in range(self.num_epochs):
            print(f'Epoch [{epoch+1}/{self.num_epochs}]')
            train_loss = self.train_epoch()
            val_loss = self.validate_epoch()

            print(f'Epoch [{epoch+1}/{self.num_epochs}], Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}')

            # Save the best model
            if val_loss < self.best_val_loss:
                self.best_val_loss = val_loss
                torch.save(self.model.state_dict(), best_model_path)
                print(f'Saved best model with val loss: {val_loss:.4f}')
```

Snipped

Validation Loop:

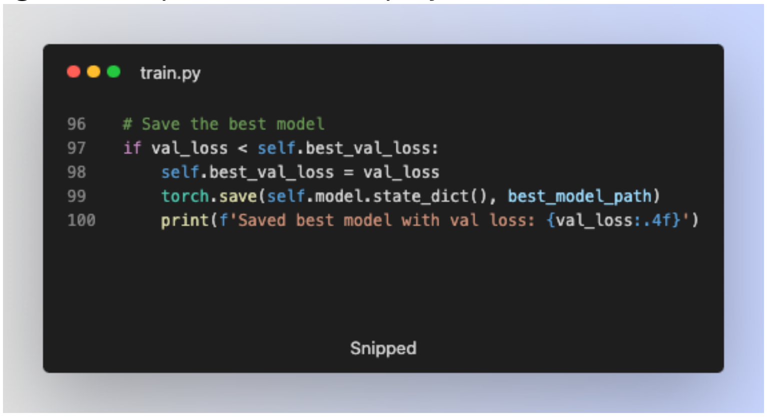
- After each training epoch, the model is evaluated on the validation dataset.
- Validation loss is computed to monitor the model's performance and prevent overfitting.



```python
def validate_epoch(self):
    self.model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for anchor, positive, negative in self.val_loader:
            anchor, positive, negative = anchor.to(self.device), positive.to(self.device), negative.to(self.device)
            anchor_embedding = self.model(anchor)
            positive_embedding = self.model(positive)
            negative_embedding = self.model(negative)
            loss = self.criterion(anchor_embedding, positive_embedding, negative_embedding)
            val_loss += loss.item()

    avg_val_loss = val_loss / len(self.val_loader)
    writer.add_scalar('Loss/val', avg_val_loss)
    return avg_val_loss
```

Saving the Best Model:

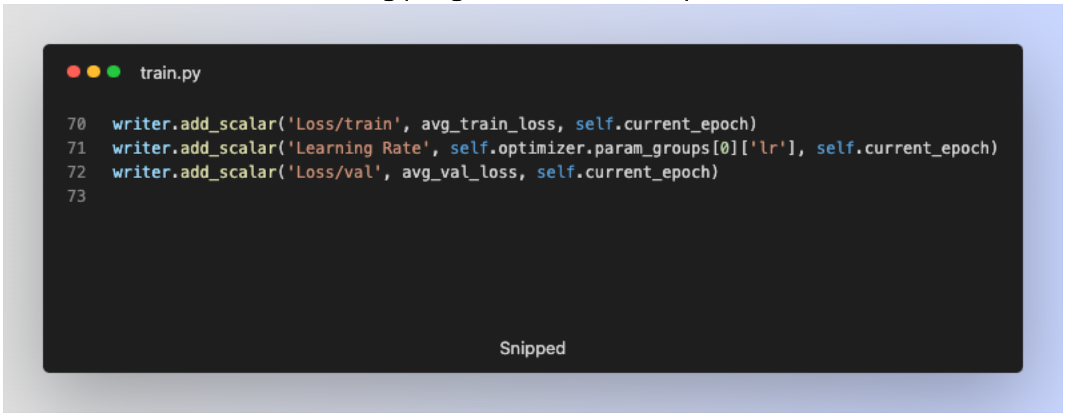
- The model with the lowest validation loss is saved to ensure that the best performing model is preserved for deployment.



```python
# Save the best model
if val_loss < self.best_val_loss:
    self.best_val_loss = val_loss
    torch.save(self.model.state_dict(), best_model_path)
    print(f'Saved best model with val loss: {val_loss:.4f}')
```

TensorBoard Logging:

- TensorBoard is used for logging training and validation metrics, enabling visualization of the training progress and model performance.



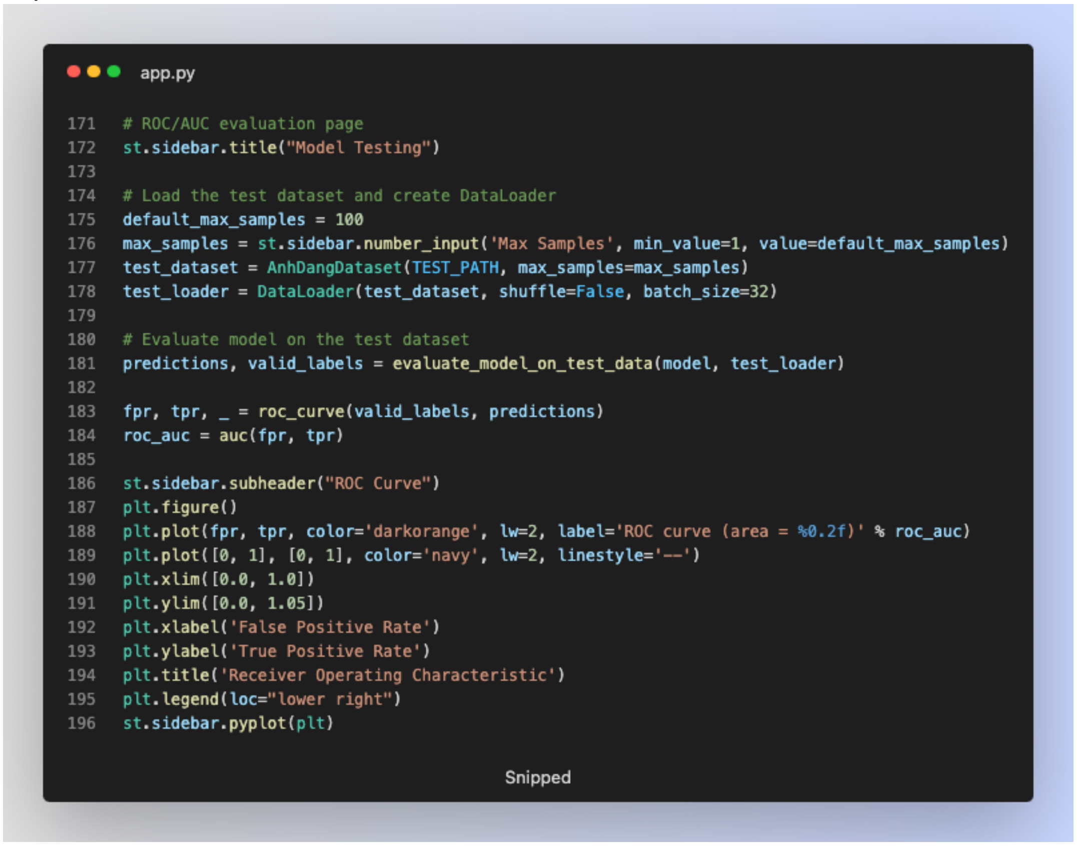
```python
writer.add_scalar('Loss/train', avg_train_loss, self.current_epoch)
writer.add_scalar('Learning Rate', self.optimizer.param_groups[0]['lr'], self.current_epoch)
writer.add_scalar('Loss/val', avg_val_loss, self.current_epoch)

```

Evaluation

The evaluation of the face verification model is performed using the test dataset and various performance metrics.

ROC Curve and AUC:

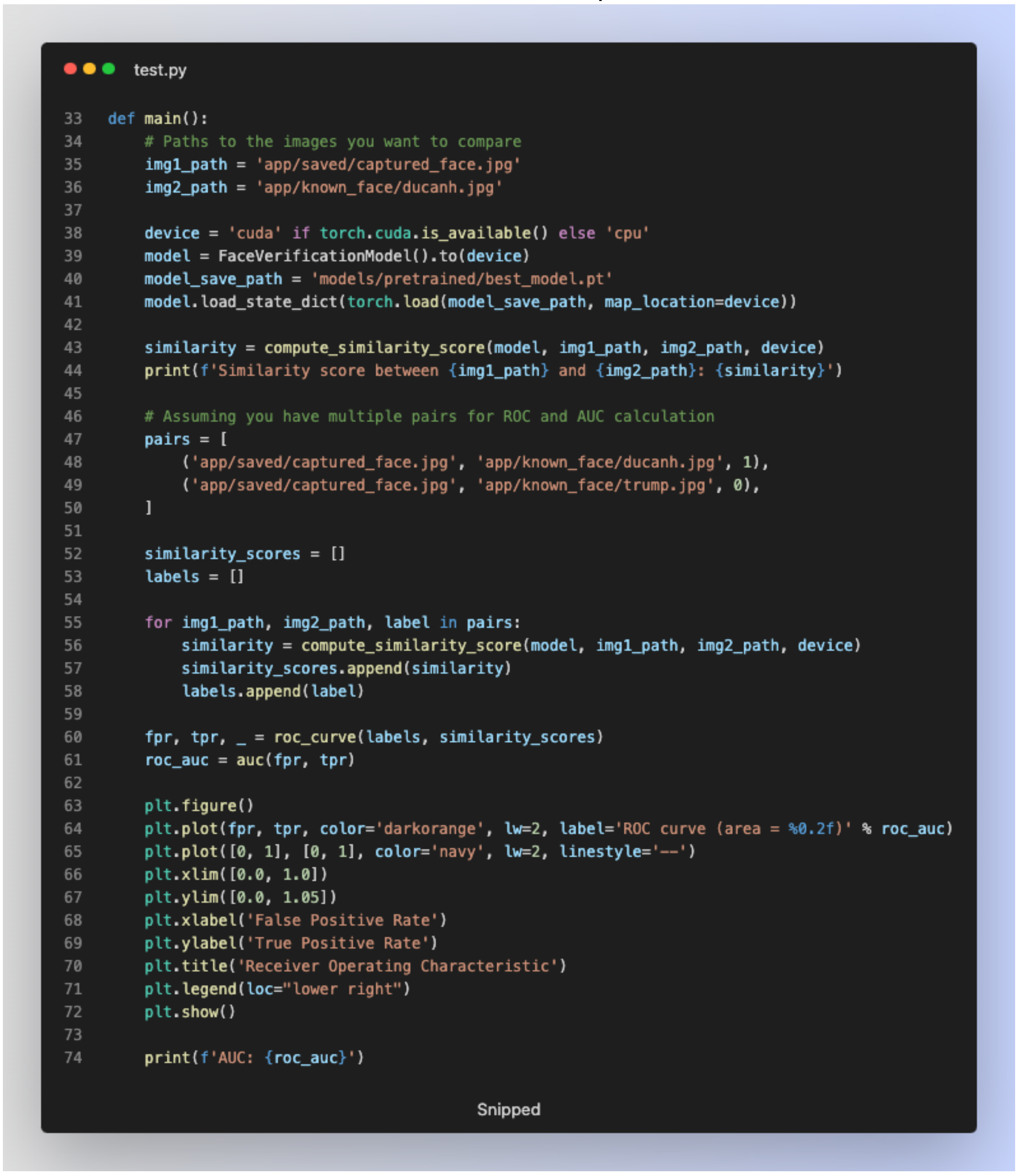
- **Receiver Operating Characteristic (ROC) curve** is plotted to visualize the model's performance across different threshold values.
- **Area Under the Curve (AUC)** is calculated to provide a single metric that summarizes the model's ability to differentiate between positive and negative pairs.



```python
# ROC/AUC evaluation page
st.sidebar.title("Model Testing")

# Load the test dataset and create DataLoader
default_max_samples = 100
max_samples = st.sidebar.number_input('Max Samples', min_value=1, value=default_max_samples)
test_dataset = AnhDangDataset(TEST_PATH, max_samples=max_samples)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=32)

# Evaluate model on the test dataset
predictions, valid_labels = evaluate_model_on_test_data(model, test_loader)

fpr, tpr, _ = roc_curve(valid_labels, predictions)
roc_auc = auc(fpr, tpr)

st.sidebar.subheader("ROC Curve")
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
st.sidebar.pyplot(plt)
```

Testing Script (test.py):

- **test.py** script loads the trained model and evaluates it on the **test dataset**.
- Predictions and true labels are used to compute **the ROC curve and AUC**.



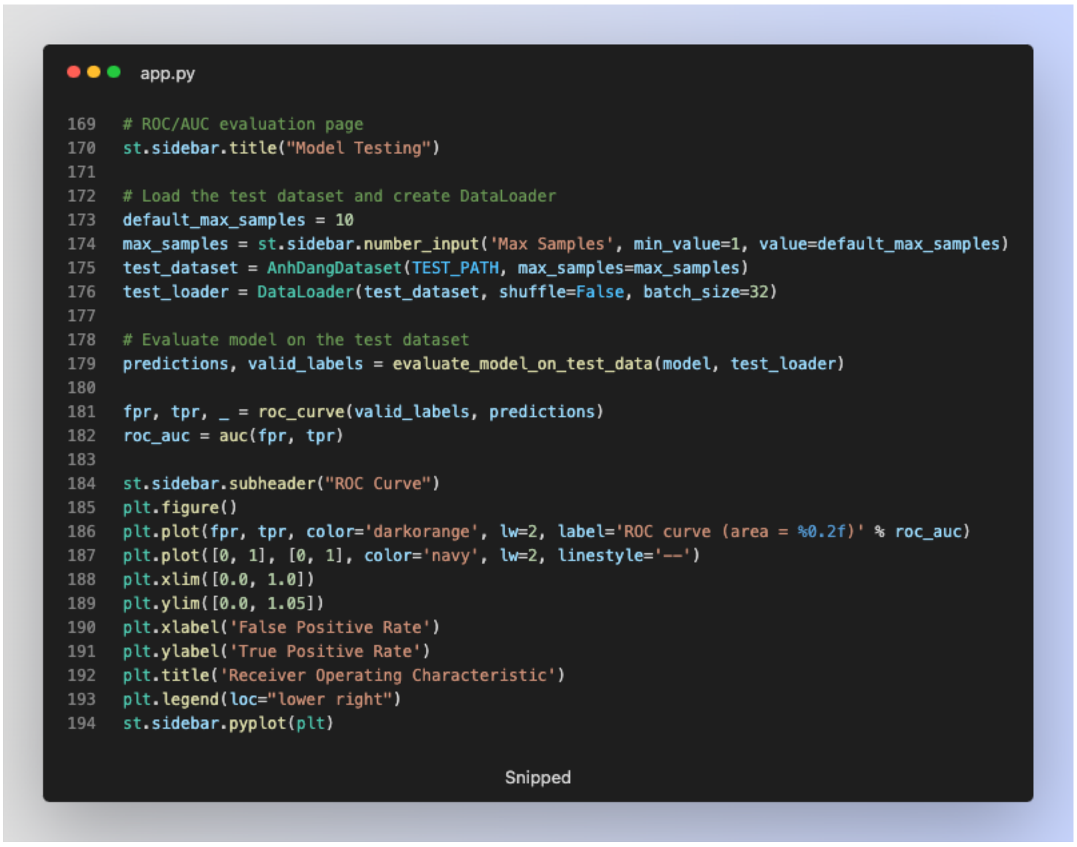
test.py

```python
def main():
    # Paths to the images you want to compare
    img1_path = 'app/saved/captured_face.jpg'
    img2_path = 'app/known_face/ducanh.jpg'

    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model = FaceVerificationModel().to(device)
    model_save_path = 'models/pretrained/best_model.pt'
    model.load_state_dict(torch.load(model_save_path, map_location=device))

    similarity = compute_similarity_score(model, img1_path, img2_path, device)
    print(f'Similarity score between {img1_path} and {img2_path}: {similarity}')

    # Assuming you have multiple pairs for ROC and AUC calculation
    pairs = [
        ('app/saved/captured_face.jpg', 'app/known_face/ducanh.jpg', 1),
        ('app/saved/captured_face.jpg', 'app/known_face/trump.jpg', 0),
    ]

    similarity_scores = []
    labels = []

    for img1_path, img2_path, label in pairs:
        similarity = compute_similarity_score(model, img1_path, img2_path, device)
        similarity_scores.append(similarity)
        labels.append(label)

    fpr, tpr, _ = roc_curve(labels, similarity_scores)
    roc_auc = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic')
    plt.legend(loc="lower right")
    plt.show()

    print(f'AUC: {roc_auc}')
```

Snipped

Integration in app.py:

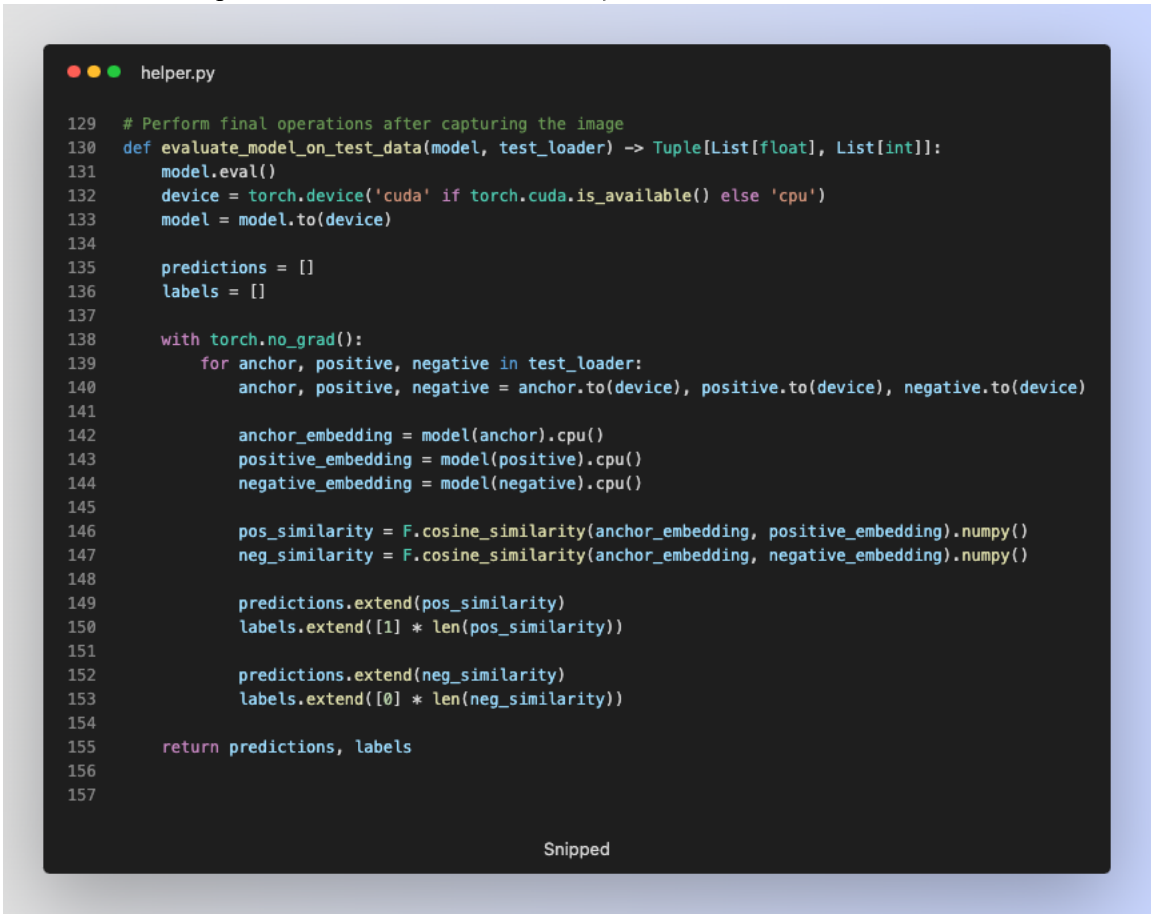
- **app.py** script includes a section for evaluating the model directly within the application interface using Streamlit. This allows users to interactively test the model and visualize the ROC curve.



```python
# ROC/AUC evaluation page
st.sidebar.title("Model Testing")

# Load the test dataset and create DataLoader
default_max_samples = 10
max_samples = st.sidebar.number_input('Max Samples', min_value=1, value=default_max_samples)
test_dataset = AnhDangDataset(TEST_PATH, max_samples=max_samples)
test_loader = DataLoader(test_dataset, shuffle=False, batch_size=32)

# Evaluate model on the test dataset
predictions, valid_labels = evaluate_model_on_test_data(model, test_loader)

fpr, tpr, _ = roc_curve(valid_labels, predictions)
roc_auc = auc(fpr, tpr)

st.sidebar.subheader("ROC Curve")
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
st.sidebar.pyplot(plt)
```

Helper Function (helper.py):

- **evaluate_model_on_test_data** function **in helper.py** is used to evaluate the model on the test dataset. This function computes the cosine similarity between embeddings to determine the model's predictions.



helper.py

```python
# Perform final operations after capturing the image
def evaluate_model_on_test_data(model, test_loader) -> Tuple[List[float], List[int]]:
    model.eval()
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = model.to(device)

    predictions = []
    labels = []

    with torch.no_grad():
        for anchor, positive, negative in test_loader:
            anchor, positive, negative = anchor.to(device), positive.to(device), negative.to(device)

            anchor_embedding = model(anchor).cpu()
            positive_embedding = model(positive).cpu()
            negative_embedding = model(negative).cpu()

            pos_similarity = F.cosine_similarity(anchor_embedding, positive_embedding).numpy()
            neg_similarity = F.cosine_similarity(anchor_embedding, negative_embedding).numpy()

            predictions.extend(pos_similarity)
            labels.extend([1] * len(pos_similarity))

            predictions.extend(neg_similarity)
            labels.extend([0] * len(neg_similarity))

    return predictions, labels
```

Snipped

Result and Discussion

Performance Analysis

Training Challenges with Triplet Loss:

- **Triplet Selection**: One of the main challenges in training with Triplet Loss is the selection of effective triplets. If the triplets are too easy, they do not provide a useful learning signal. Conversely, hard triplets can cause instability in the training process.
- **Convergence Issues**: Training with Triplet Loss can result in high variance in the loss function, leading to instability and difficulty in convergence. Appropriate selection of learning rates and margin values is crucial.
- **Data Imbalance**: Class imbalance in the dataset can affect triplet selection and the overall learning process, leading to poor generalization.

Hyperparameter Tuning:

To address some of the training challenges, hyperparameters were adjusted:

- The margin for the Triplet Loss function was increased from **1.0 to 1.2** to enforce a larger separation between positive and negative pairs.
- The learning rate for the optimizer was reduced from **0.001 to 0.0001** to ensure more stable and gradual updates to the model parameters.

Dataset Preprocessing Information:

```
----- Original Dataset Information -----
Train dataset size: 380638
Validation dataset size: 8000
Test dataset size: 8000
----- Dataset Information After Samples -----
Train dataset size: 1000
Validation dataset size: 1000
Test dataset size: 1000
----- DataLoader Information -----
Train loader size: 32
Validation loader size: 32
Test loader size: 32
```

Original Dataset Information:

- Train dataset size: 380638
- Validation dataset size: 8000
- Test dataset size: 8000

Dataset Information After Sampling:

- Train dataset size: 1000
- Validation dataset size: 1000
- Test dataset size: 1000

DataLoader Information:

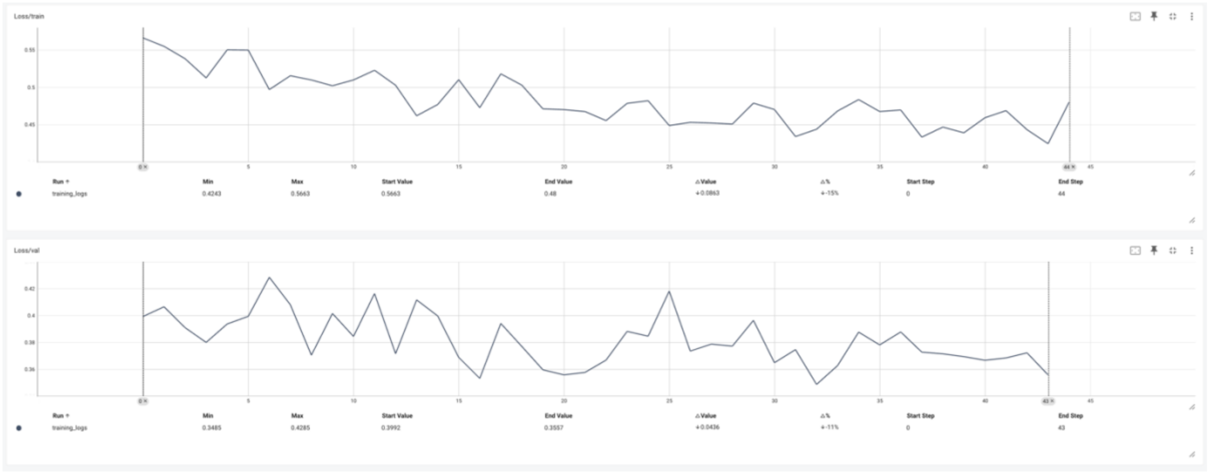
- Train loader size: 32
- Validation loader size: 32
- Test loader size: 32
- Batch Size: 32

Training on Reduced Dataset:

The model was trained on a reduced dataset of 1000 samples with less than 50 epochs. This reduced training dataset size was chosen to expedite the training process and evaluate the model's performance under limited data conditions.

Improved on Training Performance:

After adjusting the hyperparameters, the training performance improved significantly. The increased margin in the Triplet Loss function helped the model to enforce a larger separation between positive and negative pairs, resulting in better learning. The reduced learning rate allowed for more stable and gradual updates to the model parameters, reducing the chances of overshooting the optimal solution and improving convergence. The validation loss stabilized and decreased, indicating better generalization and reduced overfitting.



	Run ↑	Min	Max	Start Value	End Value	△Value	△%	Start Step	End Step
●	training_logs	0.4243	0.5663	0.5663	0.48	↓0.0863	↓-15%	0	44

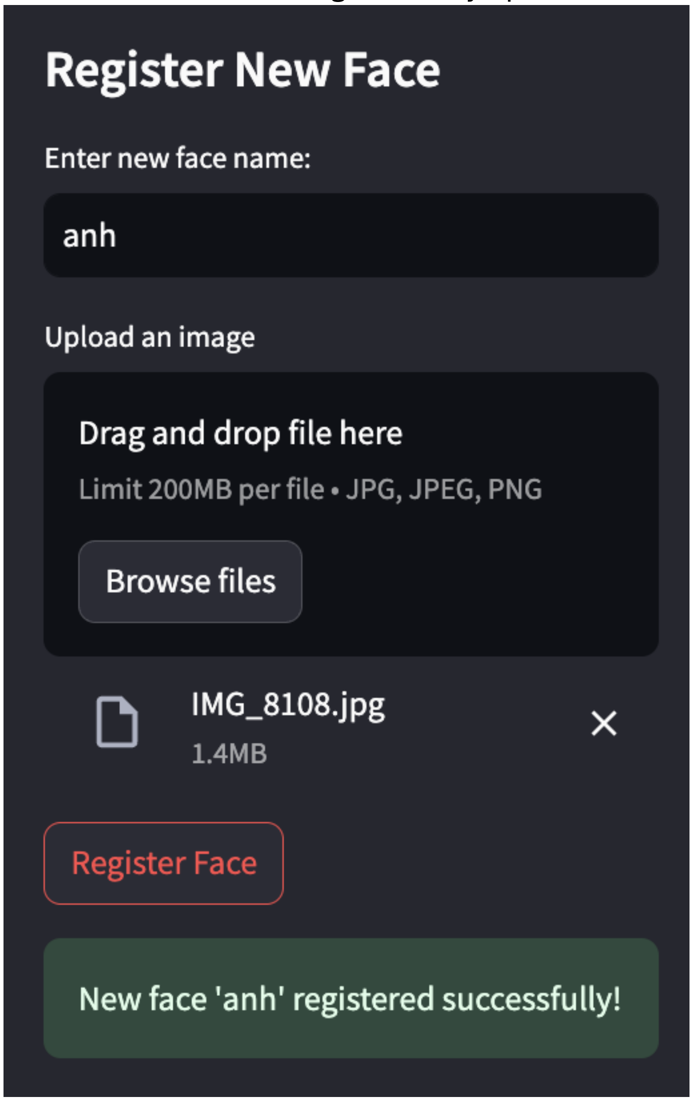
	Run ↑	Min	Max	Start Value	End Value	△Value	△%	Start Step	End Step
●	training_logs	0.3485	0.4285	0.3992	0.3557	↓0.0436	↓-11%	0	43

Generalization to Unseen Faces:

- When new faces are registered that were not included in the training dataset, the model's ability to generalize to these unseen faces is critical.
- The embedding space learned by the model during training allows it to generate embeddings for new faces that can be effectively compared with the embeddings of known faces.
- The model can identify new faces by generating embeddings for the new images and comparing them to the embeddings of registered faces using distance metrics.
- The effectiveness of this generalization can be tested by evaluating the model on a separate validation set containing faces not seen during training.

Example Scenario:

- A new student face is registered by upload their image.



Register New Face

Enter new face name:

anh

Upload an image

Drag and drop file here

Limit 200MB per file • JPG, JPEG, PNG

Browse files

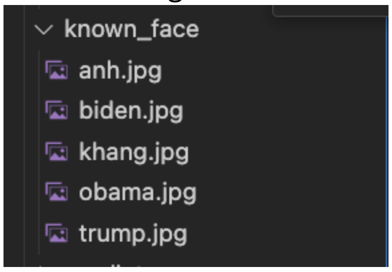
IMG_8108.jpg

1.4MB

Register Face

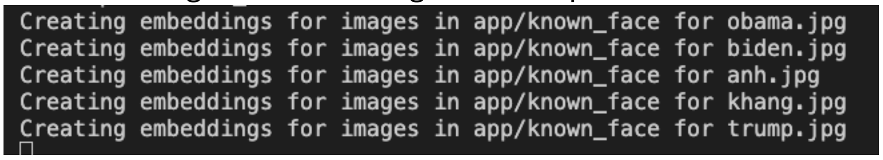
New face 'anh' registered successfully!

- This image is added to the known faces database.



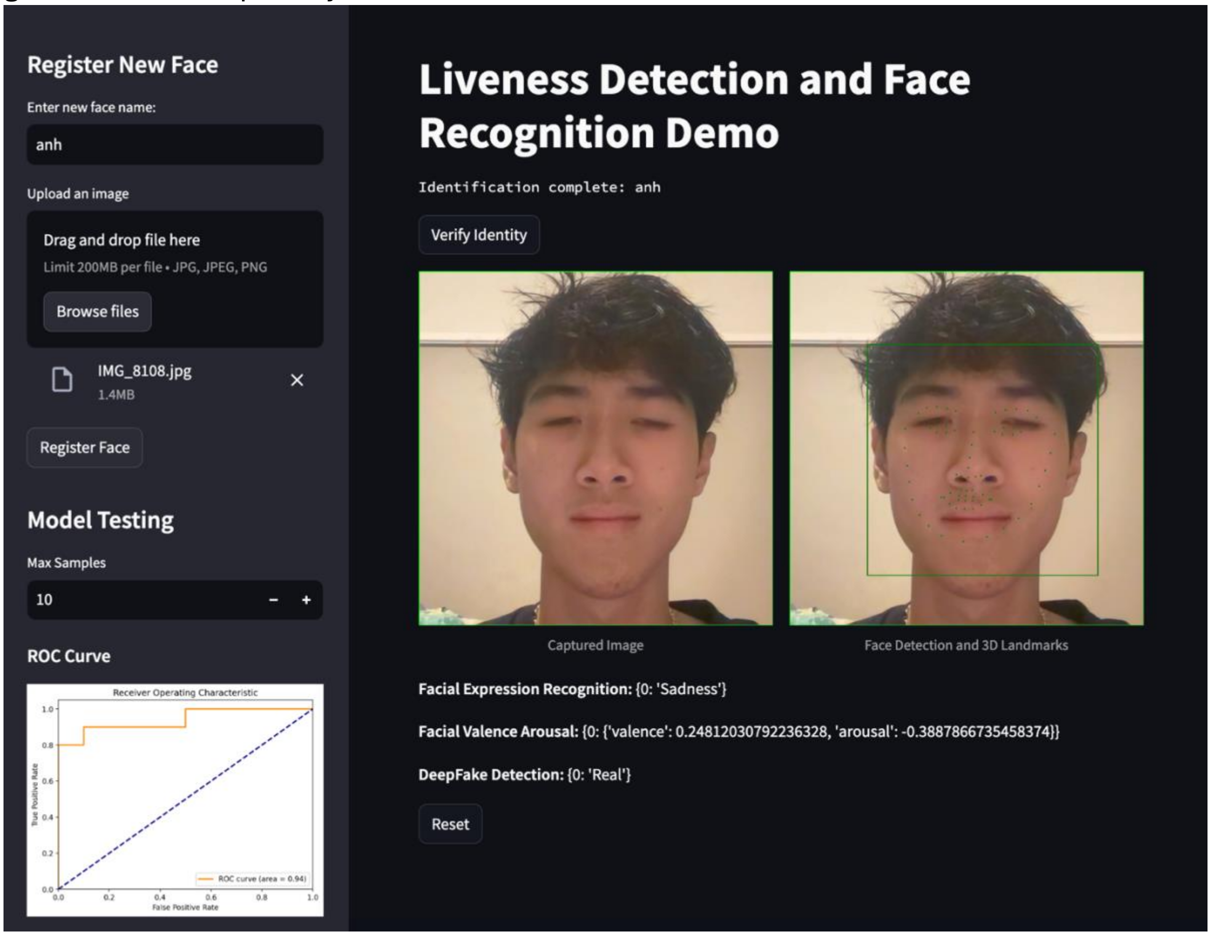
```
known_face
  anh.jpg
  biden.jpg
  khang.jpg
  obama.jpg
  trump.jpg
```

- When this new student's face is presented for verification, the model generates an embedding for the live image and compares it to the stored embedding.



```
Creating embeddings for images in app/known_face for obama.jpg
Creating embeddings for images in app/known_face for biden.jpg
Creating embeddings for images in app/known_face for anh.jpg
Creating embeddings for images in app/known_face for khang.jpg
Creating embeddings for images in app/known_face for trump.jpg
```

- The model's ability to correctly identify the new student demonstrates its generalization capability.



```
Identification complete: anh
```

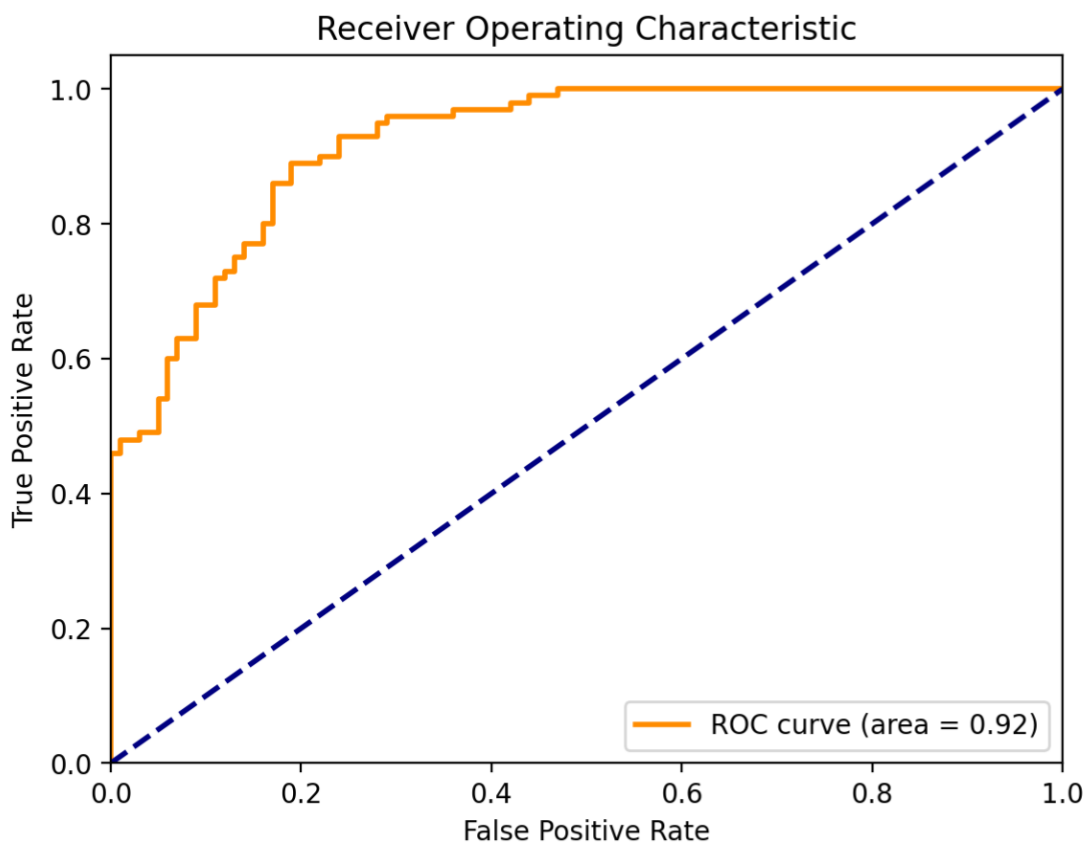
Facial Expression Recognition: {0: 'Sadness'}

Facial Valence Arousal: {0: {'valence': 0.24812030792236328, 'arousal': -0.3887866735458374}}

DeepFake Detection: {0: 'Real'}

Performance Metrics:

- The model's performance is evaluated using the Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC).
- The ROC curve provides a visual representation of the model's performance across different threshold values, while the AUC summarizes the model's ability to distinguish between positive and negative pairs.



The results show the model's ability to differentiate between similar and dissimilar faces using the ROC curve and AUC as performance metrics. For the test dataset with 100 samples, the ROC AUC score was found to be 0.92, indicating a high level of accuracy in distinguishing between positive and negative pairs.

Anti-Spoofing Evaluation

The anti-spoofing module evaluates the system's ability to detect and prevent spoofing attempts using static images or videos. The module uses the Eye Aspect Ratio (EAR) to detect blinks and ensures that only live faces are recognized.

Additional Anti-Spoofing Techniques:

- The system uses a face analyzer configuration loaded from app/facetorch/config.merged.yml.
- The FaceAnalyzer is utilized to perform inference on images to detect facial expressions, deepfakes, and valence-arousal estimations.

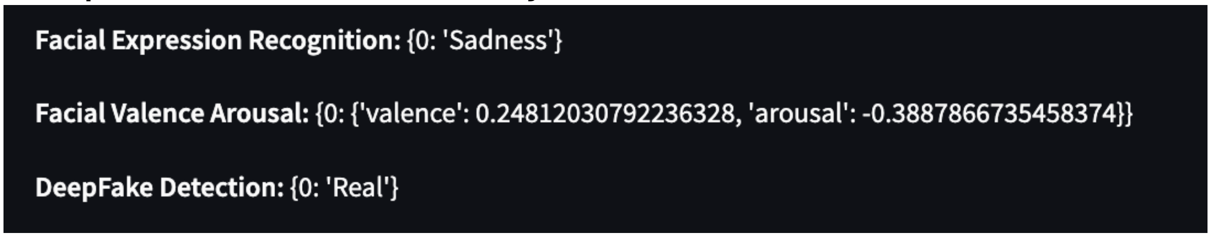
Evaluation Metrics:

- The effectiveness of the anti-spoofing module is tested by presenting various spoofing attempts, including printed images and videos.
- The system's ability to detect these attempts and maintain accurate face verification is evaluated.

Results:

The anti-spoofing module, combined with the additional techniques, effectively detected spoofing attempts, demonstrating the robustness of the system in preventing fraudulent access.

Example results from the facial analysis module:



```
Facial Expression Recognition: {0: 'Sadness'}

Facial Valence Arousal: {0: {'valence': 0.24812030792236328, 'arousal': -0.3887866735458374}}

DeepFake Detection: {0: 'Real'}
```

Innovation Discussions

The face recognition attendance system developed in this project introduces several innovative elements that extend beyond the basic framework, enhancing the system's accuracy, security, and user experience. These innovations include:

1. **Hyperparameter Tuning for Enhanced Performance:**
    - The system benefits from carefully tuned hyperparameters, specifically the margin and learning rate for the Triplet Loss function. By increasing the margin to 1.2 and reducing the learning rate to 0.0001, the model achieved improved stability and performance, as evidenced by a Receiver Operating Characteristic (ROC) Area Under the Curve (AUC) score of 0.92 on the test dataset. This tuning process is crucial for fine-tuning the model's ability to differentiate between similar and dissimilar faces effectively.
2. **Advanced Anti-Spoofing Techniques:**
    - The anti-spoofing module employs a combination of Eye Aspect Ratio (EAR) for blink detection and advanced facial analysis techniques using a face analyzer configuration. This multi-faceted approach enhances the system's ability to detect and prevent spoofing attempts with high accuracy. The integration of deepfake detection, facial expression recognition, and valence-arousal estimation further strengthens the anti-spoofing capabilities.
3. **Chebyshev Distance Metric for Improved Verification:**
    - A novel aspect of this project is the implementation of the Chebyshev distance metric for face verification. This metric measures the maximum absolute differences between the embeddings, providing superior results compared to other distance metrics. The use of Chebyshev distance helped achieve better accuracy in identifying individuals, making it a valuable addition to the system.
4. **User-Friendly Interface with Real-Time Feedback:**
    - The system's interface, developed using Streamlit, offers an intuitive and interactive user experience. It facilitates easy face registration, verification, and liveness detection, providing real-time feedback to users. This real-time

interaction enhances the usability and efficiency of the system, making it accessible and straightforward for users to operate.

5. **Scalability and Robustness**:
    - The system is designed with scalability in mind, ensuring that it can handle large datasets and be deployed in real-world environments. The robust framework supports future enhancements and the integration of additional features, such as more sophisticated triplet mining strategies and advanced data augmentation techniques.

Conclusion

Summary of Findings

- The face recognition attendance system effectively differentiates between similar and dissimilar faces using the **Triplet Loss** function and **CNN** based face embeddings.
- Hyperparameter tuning, including increasing the **margin to 1.2** and **reducing the learning rate to 0.0001**, significantly **improved training stability** and **performance**.
- The **Chebyshev** distance metric provided the best results for face verification, demonstrating its effectiveness in this application.
- The anti-spoofing module enhances system security by detecting and preventing spoofing attempts.
- The user interface provides a seamless and interactive experience, facilitating face registration and verification.
- The model demonstrated a decent generalization to unseen faces, effectively identifying new faces registered after the initial training.
- The ROC AUC score of 0.92 for the test dataset with 100 samples indicates a high level of accuracy in distinguishing between positive and negative pairs.

Future Work

- **Improving Triplet Selection**: Implement more advanced triplet mining strategies to improve the effectiveness of triplet selection and enhance model training.
- **Further Hyperparameter Optimization**: Continue experimenting with different learning rates, margins, and batch sizes to find the optimal combination for training stability and performance.
- **Data Augmentation**: Utilize more advanced data augmentation techniques to increase the variability in the training data and prevent overfitting.
- **Improving Anti-Spoofing Techniques**: Enhance the anti-spoofing module by incorporating more sophisticated methods, such as machine learning based texture analysis and motion detection algorithms, to detect and counter more advanced spoofing attempts.
- **Deployment**: Optimize the system for real-world deployment, ensuring robustness and scalability in various environments.

References

1. Schroff, F., Kalenichenko, D. and Philbin, J. (2015) *FaceNet: A unified embedding for face recognition and clustering*, *arXiv.org*. Available at: https://arxiv.org/abs/1503.03832 (Accessed: 26 May 2024).
2. *11-785-fall-20-homework-2: Part 2* (no date) *Kaggle*. Available at: https://www.kaggle.com/competitions/11-785-fall-20-homework-2-part-2/data (Accessed: 26 May 2024).
3. *Tripletmarginloss¶* (no date) *TripletMarginLoss - PyTorch 2.3 documentation*. Available at: https://pytorch.org/docs/stable/generated/torch.nn.TripletMarginLoss.html (Accessed: 26 May 2024).
4. *Effect of distance measures on the performance of face recognition ...* Available at: https://www.researchgate.net/publication/281684384_Effect_of_Distance_Measures_on_the_Performance_of_Face_Recognition_Using_Principal_Component_Analysis (Accessed: 26 May 2024).