Facial Recognition with Emotion and Liveness

Student Name: Dang Khoa Le
Student ID: 103844421

1. Introduction

Face recognition is a rapidly advancing domain in computer vision, playing a pivotal role in biometric identification, security, and automated attendance systems. This project explores a comprehensive end-to-end pipeline for real-time **face recognition-based attendance**, integrating various deep learning and image processing techniques to ensure accuracy, reliability, and resilience against spoofing attempts. The system is designed to perform robust identity recognition through embedding generation and similarity search, emotion analysis for enhanced interaction, and anti-spoofing measures for real-world deployment. The user interface (UI) allows for seamless interaction and live feedback, all orchestrated through a real-time rendering pipeline.

The system architecture comprises the following core components:

1.1. Face Identification

Two key learning paradigms are implemented:

- **Classification-Based Embedding**: A supervised CNN model trained using cross-entropy loss to classify faces into known identities. The final layer before softmax is used to extract embeddings.
- **Triplet Loss-Based Embedding**: A metric learning model that directly optimizes the embedding space by reducing the distance between positive pairs and increasing it for negative pairs. This self-supervised approach enhances generalization to unseen faces.

Models training session was conducted on Google Colab – Jupyter Notebook. Post-training, the 2 models are saved to Drive with outcomes visualized. Shared Colab session can be accessible at:
https://colab.research.google.com/drive/1pb9YWCrNkx7fd9kd9yKIFWQCRmKhmmao?usp=sharing

1.2. Face Detection

We utilize the OpenCV Haar Cascade classifier for face localization. This method provides lightweight and fast detection suitable for real-time scenarios, identifying face regions to be processed further for classification and verification.

1.3. Similarity Matching

Face embeddings are compared using two widely adopted similarity metrics:

- **Cosine Similarity**: Measures the cosine of the angle between two vectors.
- **Euclidean Distance**: Measures the straight-line distance between two embeddings in high-dimensional space.

Cosine similarity $$cos = \frac{x_1}{\sqrt{x_1^2+y_1^2}} \times \frac{x_2}{\sqrt{x_2^2+y_2^2}} + \frac{y_1}{\sqrt{x_1^2+y_1^2}} \times \frac{y_2}{\sqrt{x_2^2+y_2^2}}$$

Euclidean distance $$euc = \sqrt{(\frac{x_1}{\sqrt{x_1^2+y_1^2}} - \frac{x_2}{\sqrt{x_2^2+y_2^2}})^2 + (\frac{y_1}{\sqrt{x_1^2+y_1^2}} - \frac{y_2}{\sqrt{x_2^2+y_2^2}})^2}$$

$$euc = \sqrt{2 - 2 \times cos}$$

Figure. Mathematical Formulas of Cosine and Euclidean distance in Similarity Matching.

These metrics are employed during verification to determine if the detected face corresponds to a known identity (using the trained models). Evaluations on metrics comparison will be presented.

1.4. Emotion Detection
Emotion analysis is incorporated to enrich the application context using the DeepFace library. This module leverages a pretrained CNN model (facial_expression_model_weights.h5) to predict facial emotions from the cropped face region.

1.5. Anti-Spoofing
To prevent identity spoofing via photos or screen replays, we integrate the **Silent-Face Anti-Spoofing** ensemble. This ensemble uses multiple lightweight CNNs to predict whether the face in view is a live human or a spoofed presentation. It increases the system's trustworthiness, especially in uncontrolled environments.

1.6. User Interface (UI)
The entire pipeline is encapsulated in a GUI built using Tkinter. It provides:

- Real-time video feed rendering
- On-screen display of identified names, emotions, and spoof status
- Ability to register new identities dynamically by capturing face snapshots and saving their embeddings
- Seamless interaction for end-users in attendance-tracking scenarios

The environment is compatible with Python 3.x (recommended 3.9-3.12), utilizing OpenCV, Torch, and DeepFace. It supports cross-platform execution with minimal configuration, making it a practical solution for enterprise deployment.

2. Data Preparation

Face dataset from a Kaggle repository will initially be derived and saved to Google Drive for quick-access training on Google Colab session. The dataset comprises thousands of identities stored in a class-wise folder structure.

First, implemented a custom Dataset class compatible with PyTorch, designed to parse and load the dataset efficiently for training:

```
def __init__(self, root, transform=None, load_all=False):
        self.root = root
        self.transform = transform
        self.unique_label_ids = os.listdir(self.root)
        self.num_classes = len(self.unique_label_ids)
        self.unique_label_nums = list(range(self.num_classes))
        # Initialize storage lists
        self.image_paths = []; self.label_ids = []; self.label_nums = []
        # Load image paths and labels
        for label_num, label_id in enumerate(self.unique_label_ids):
            ...
            self.image_paths.extend(full_paths)
            self.label_ids.extend([label_id] * len(full_paths))
            self.label_nums.extend([label_num] * len(full_paths))
        self.data_len = len(self.label_nums)
        # Preload all images (for runtime usage)
        self.images = dict()
        ...
```

Each sub-directory in the dataset represents a distinct identity label, containing multiple image samples. The class supports both **on-demand lazy loading** and **full preloading** modes, facilitating memory-efficient training.

The initialization extracts:

- Image file paths
- Corresponding label identifiers (string-based)
- Integer-mapped class labels (for model training)

The class also provides a reduce_samples() method for **label subset reduction**, enabling quick prototyping or experimentation with fewer classes. This is compulsory vital for model training when substantially reduce training runtime and computer resource limitation, especially for a very large (roughly 50k images) like this.

2.1. Image Preprocessing

To standardize inputs and facilitate model convergence, we applied a set of image transformations using the torchvision.transforms module:

```
params = dict(
    epochs=20,
    batch_size=128,
    num_labels=500,               # Limit to 500 classes
    criterion=nn.CrossEntropyLoss,
    optimizer=torch.optim.SGD,
    optimizer_params=dict(lr=0.01)# Small learning rate for better training
)
transform = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize images to 64x64
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

- **Resizing** to 64×64 ensures uniform input dimensions.
- **Normalization** with ImageNet means and standard deviations aligns pixel distributions with those used during pretrained model initialization, enhancing training stability.

2.2. Loading Data

Data Labelling and Sample Reduction

Given the original dataset contained over 4000 identities and more than 380,000 samples, we selectively reduced it to **500 classes** to maintain training feasibility within resource and time constraints.

```
train_dataset = Data(train_path, transform=transform)
labels = train_dataset.unique_label_ids[:params['num_labels']]
train_dataset.reduce_samples(labels)
val_dataset = Data(val_path, transform=transform)
val_dataset.reduce_samples(labels)
```

PyTorch's DataLoader was used to efficiently batch and shuffle samples for both training and validation:

```
train_loader_args = {
    "shuffle": True,
    "batch_size": params["batch_size"],
}
```

These loaders feed the model in batches, optimizing GPU/CPU usage and ensuring class representation per epoch.

Dataset Summary and Output
The preparation process yielded the following statistics:

```
========== [TRAINING DATASET] ==========
[INFO] Found 4000 unique identities.
[INFO] Dataset initialized with 380638 total samples.
[INFO] Reduced dataset:
Before: 380638 samples
After:  48421 samples
Remaining classes: 500

========== [VALIDATION DATASET] ==========
[INFO] Found 4000 unique identities.
[INFO] Dataset initialized with 8000 total samples.
[INFO] Reduced dataset:
Before: 8000 samples
After:  1000 samples
Remaining classes: 500

[INFO] Final training set size: 48421 samples
[INFO] Final validation set size: 1000 samples
```

These figures confirm successful filtering and transformation of the dataset. The **training set** comprises 48,421 samples and the **validation set** 1,000 samples across the same 500 classes, ensuring consistency during model evaluation.

Reflection on Methodology
This data preparation pipeline reflects standard best practices in machine learning experimentation. The custom dataset class provides both flexibility and control, while the preprocessing aligns with the requirements of convolutional architectures. By reducing the dataset to a fixed number of classes, we ensure resource-efficiency without compromising learning complexity. These steps serve as the foundation for robust training in both classification-based and metric learning models explored in subsequent sections.

3. Model Training

3.1. Base CNN Architecture
To support both supervised classification and metric-based verification tasks, we implemented a custom convolutional neural network (CNN) backbone named BaseCNN. The goal of this architecture is to extract compact and discriminative features from facial images, which are then used either for identity classification or embedding comparisons.

Architecture Overview
The BaseCNN is designed with three convolutional blocks, each consisting of a convolutional layer followed by a ReLU activation and a downsampling operation:

```
class BaseCNN(nn.Module):
    def __init__(self, in_ch=CHANNELS):
        super().__init__()
        self.conv = nn.Sequential(
            # First convolutional block
            nn.Conv2d(in_ch, 32, kernel_size=3, stride=1, padding=1),# Output: 32 x H x W
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),  # Downsample → 32 x H/2 x W/2

            # Second convolutional block
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),   # Output: 64 x H/2 x W/2
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),  # Downsample → 64 x H/4 x W/4
            # Third convolutional block
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),  # Output: 128 x H/4 x W/4
```

```
            nn.ReLU(),
            nn.AdaptiveAvgPool2d(output_size=1),  # Output: 128 x 1 x 1 (global avg pooling)
        )
```

This CNN progressively reduces spatial dimensions while increasing feature complexity. Final output is a global average pooled tensor of shape [batch_size, 128], which serves as a high-level representation of the input face.

Model Summary
This architecture is lightweight, with under 100K parameters, enabling efficient training and fast inference on limited hardware.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 160, 160]             896
              ReLU-2         [-1, 32, 160, 160]               0
         MaxPool2d-3           [-1, 32, 80, 80]               0
            Conv2d-4           [-1, 64, 80, 80]          18,496
              ReLU-5           [-1, 64, 80, 80]               0
         MaxPool2d-6           [-1, 64, 40, 40]               0
            Conv2d-7          [-1, 128, 40, 40]          73,856
              ReLU-8          [-1, 128, 40, 40]               0
 AdaptiveAvgPool2d-9            [-1, 128, 1, 1]               0
================================================================
Total params: 93,248
Trainable params: 93,248
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.29
Forward/backward pass size (MB): 24.22
Params size (MB): 0.36
Estimated Total Size (MB): 24.87
----------------------------------------------------------------
```

3.2. Classifier CNN (Supervised Learning)

Built on top of BaseCNN, the classifier model incorporates two additional components:

- **Embedding Head**: Projects 128-dimensional CNN features into an embedding space (e.g., 128D), followed by batch normalization and ReLU activation.
- **Classification Head**: A fully connected layer that maps the embedding to the final output classes.

```
class Classifier(nn.Module):
    def __init__(self, num_classes, embed_dim=128):
        super().__init__()
        self.backbone = BaseCNN()
        # Embedding head: maps 128-dim features to a lower-dim, normalized embedding
        self.embed_head = nn.Sequential(
            nn.Linear(128, embed_dim),        # Learnable dimensionality reduction
            nn.BatchNorm1d(embed_dim),        # Stabilizes learning
            nn.ReLU(),                        # Non-linearity
        )
        # Final classification layer: predicts class logits from embedding
        self.classify_head = nn.Linear(embed_dim, num_classes)
```

Forward Pass Logic

```
logits, F.normalize(embeds, dim=1)
```

The model outputs both:

- logits for identity classification
- normalized embeddings for use in face verification tasks via similarity metrics (e.g., cosine similarity)

This dual-output design enables the same model to be leveraged for both closed-set classification and open-set verification.

a. Training Strategy

The model was trained for 20 epochs using:

- **Loss Function**: Cross-entropy loss for multi-class classification.
- **Optimizer**: Stochastic Gradient Descent (SGD) with momentum.
- **Device**: GPU-accelerated environment.

Training Loop Snippet

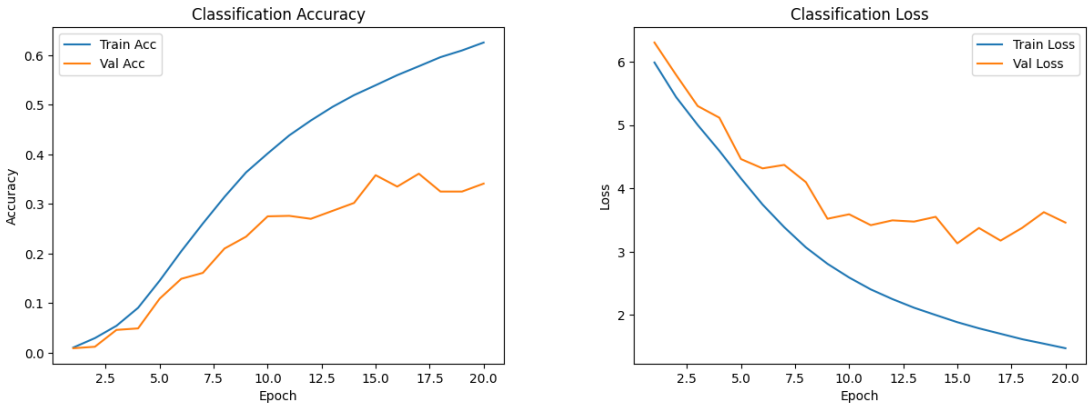
```
for ep in range(params['epochs']):
    cls_model.train()  # Enable dropout/batchnorm in training mode
    loader = tqdm.tqdm(train_loader, desc="...", leave=False)
    t_loss = t_acc = 0  # Accumulate total loss and correct predictions
    for imgs, labels in loader:
        imgs, labels = imgs.to(device), labels.to(device)
        optimizer.zero_grad()               # Reset gradients
        logits, _ = cls_model(imgs)         # Forward pass
        loss = criterion(logits, labels)    # Compute loss
        loss.backward()                     # Backpropagation
        optimizer.step()                    # Update weights
```

The training and validation loss/accuracy were recorded at each epoch to monitor convergence and overfitting.

b. Results and Evaluation

EPOCH	TRAIN LOSS	VAL LOSS	TRAIN ACC	VAL ACC
1	5.9895	6.3050	0.0104	0.0090
5	4.1576	4.4634	0.1453	0.1090
10	2.5920	3.5911	0.4017	0.2750
15	1.8863	3.1323	0.5394	0.3580
20	1.4748	3.4600	0.6256	0.3410

Performance Over Epochs (Derived from Output)



Figures. Model Accuracy and Loss per epoch Plots.

Analysis and Reflection

- **Loss Reduction**: Both training and validation losses decreased significantly during the initial epochs (1–10), indicating effective learning of features and classes.
- **Overfitting Trend**: From epoch 12 onward, the validation loss fluctuated while training loss continued to decrease, signalling the beginning of **overfitting**. However, the validation accuracy still improved slightly until epoch 17.
- **Accuracy Growth**:
    - Training accuracy increased consistently, reaching **62.56%** by epoch 20.
    - Validation accuracy plateaued around **34–36%**, suggesting the model generalizes moderately well but could benefit from regularization (e.g., dropout) or more training data.
- **Model Behaviours**:

- The gap between training and validation metrics is expected in large-class face classification tasks (500 classes).
- Validation accuracy remains significantly above chance level (0.2% baseline for 500 classes), indicating that the model has learned meaningful embeddings.

Conclusion
The classification model demonstrates successful training and convergence, as evidenced by declining loss and improving accuracy. The BaseCNN architecture, when augmented with an embedding and classification head, forms a versatile model capable of both supervised identity classification and downstream verification. Although the model begins to overfit beyond 15 epochs, the final performance indicates a solid foundation for embedding extraction, to be utilized in the verification and attendance tasks further in this report.

3.3 Metric Learning with Triplet Loss

While classification-based face recognition provides an effective framework for identifying a fixed set of known identities, it is inherently limited in open-set scenarios where new identities may be introduced dynamically. To address this limitation, we implemented a **triplet loss-based metric learning approach**, which aims to learn an embedding space where intra-class distances are minimized and inter-class distances are maximized.
The objective of this model is not to predict class labels directly but to **learn a discriminative feature space**, enabling reliable identity verification based on embedding similarity. This makes it particularly suited for applications like face verification, retrieval, and real-time attendance matching against dynamically growing databases.

a. Triplet Dataset Construction
To train the model using triplet loss, we implemented a custom dataset class, TripletDataset, which generates triplet samples in the form of:

- **Anchor**: An image of a person (A)
- **Positive**: Another image of the same person (A′)
- **Negative**: An image of a different person (B)

Dataset Logic:

```
class TripletDataset(Dataset):
    def __getitem__(self, idx):
        a_img, a_lbl = self.ds[idx]  # Anchor sample
        # Sample a different index with the same label (positive)
        pos = idx
        while pos == idx:
            pos = np.random.choice(self.label2idx[a_lbl])
        p_img, _ = self.ds[pos]
        # Sample a different label (negative)
        neg_lbl = a_lbl
        while neg_lbl == a_lbl:
            neg_lbl = np.random.choice(list(self.label2idx.keys()))
        neg = np.random.choice(self.label2idx[neg_lbl])
        n_img, _ = self.ds[neg]
        return a_img, p_img, n_img  # Return triplet
```

The dataset ensures label balance and class diversity by:

- Randomly sampling distinct pairs for anchor-positive (same label)
- Randomly sampling a different label for the negative example

This ensures that each triplet teaches the model to **minimize the distance** between positive pairs and **maximize the distance** between negative ones.

b. Model Architecture and Training
Instead of designing a new model, we re-used the **feature extractor** (BaseCNN) and **embedding head** from the classifier model:

```
trip_model = nn.Sequential(cls_model.backbone, cls_model.embed_head).to(device)
```

Training Configuration:

- **Loss Function**: nn.TripletMarginLoss(margin=1.0)
- **Optimizer**: Adam (learning rate = 1e-3)
- **Epochs**: 20
- **Batch Size**: 128

Each training epoch processes thousands of dynamically generated triplets. The model optimizes the following objective:

$$\max(||f(A)-f(P)||^2 - ||f(A)-f(N)||^2 + \alpha, 0)$$

c. Training Progression and Results

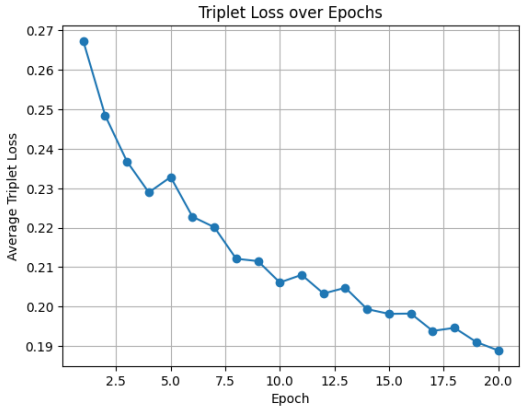


Figure. Triplet Loss across 20 Training Epochs Plot

EPOCH	TRIPLET LOSS
1	0.2673
5	0.2329
10	0.2061
15	0.1982
20	0.1889

d. Interpretation and Reflection

Loss Convergence

- **Initial Stage (Epochs 1–5)**: The triplet loss decreases rapidly from 0.2673 to 0.2329, indicating that the model is quickly learning to discriminate between same and different identities in the embedding space.
- **Mid Stage (Epochs 6–14)**: The loss continues to decrease, though at a slower rate, reflecting a **refinement of the learned distances**.
- **Final Stage (Epochs 15–20)**: The loss stabilizes around 0.19, showing that the model is close to convergence with **well-separated embedding representations**.

Model Behavior and Observations

- Unlike classification loss, triplet loss does not directly yield accuracy scores; instead, its success is reflected in **verification performance** using metrics like cosine similarity and ROC-AUC.
- The consistent decrease in loss over epochs suggests that the model is learning a **globally consistent and meaningful embedding space**, which is foundational for downstream face verification and attendance matching.

Conclusion

The metric learning model trained using triplet loss successfully converged to a lower loss value, implying that it learned to place images of the same person close together in the embedding space, while separating those of different people. This approach complements the classifier model by enabling **generalization to unseen identities** and supporting real-time face verification with robust performance.

The learned embeddings from this model are later evaluated using similarity metrics and receiver operating characteristic (ROC) curves in the following sections of the report, where we benchmark its verification capabilities against the classification-based embeddings.

4. Verification and Similarity Evaluation

The effectiveness of a face verification system hinges on its ability to distinguish between genuine (same-identity) and imposter (different-identity) pairs using learned embeddings. To this end, we conducted a series of evaluations using **cosine similarity** and **Euclidean distance** metrics on embeddings extracted from both models (classification-based and triplet loss-based). These evaluations include **ROC-AUC analysis**, **distance distribution histograms**, and **t-SNE-based visual embedding interpretation**.

4.1. Receiver Operating Characteristic (ROC) Curve and AUC

To measure the discriminative power of the learned embeddings, we utilized the **ROC curve**, which plots the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)** at varying similarity thresholds.

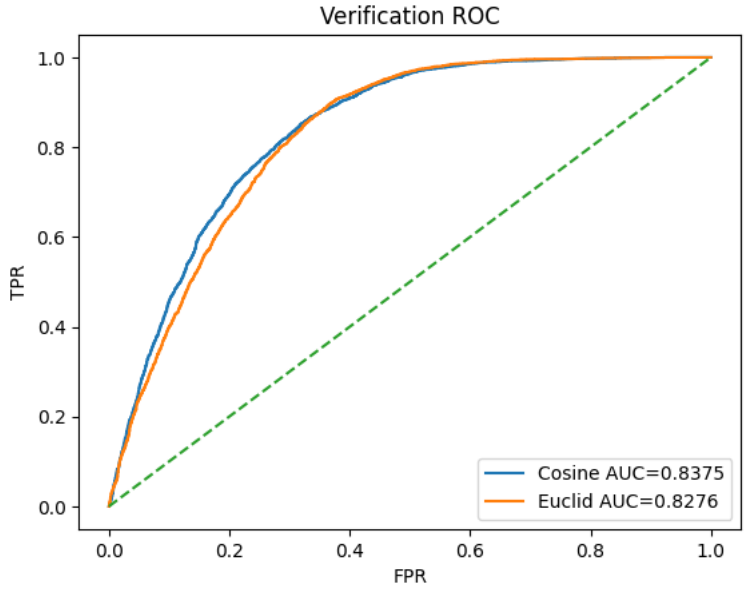


Figure. ROC Curve on 2 Similarity Metrics

Methodology

Using a benchmark set of face image pairs with binary labels (1: same person, 0: different person), we computed similarity scores via:

- **Cosine Similarity**: Similarity = $\frac{e1 \cdot e2}{\|e1\| \cdot \|e2\|}$
- **Euclidean Distance** (negated for ROC purposes): Similarity = $-\|e1-e2\|^2$

The embeddings $\vec{e}1$, $\vec{e}2$ were extracted using the **triplet-trained model** for all evaluations.

```
def get_emb(p):
        VERIFICATION_ROOT = "..." # Path to verification file
        ... # Extracting images from the path provided in verification and truth label
    e3, e4 = get_emb(r.img1), get_emb(r.img2)
    cosine_dist.append(np.dot(e3, e4)/(np.linalg.norm(e3)*np.linalg.norm(e4))) # Cos formula
    euclidean_dist.append(-np.linalg.norm(e3 - e4))                            # Euc formula
    truth_labels.append(r.label)
```

Result

METRIC	AUC SCORE
COSINE SIMILARITY	0.8375
EUCLIDEAN DISTANCE	0.8276

Interpretation

- Both metrics exhibit strong discriminative capability, with AUC > 0.82, confirming the **embedding quality is high**.
- **Cosine similarity outperforms Euclidean distance slightly**, which aligns with the expectation for **normalized embeddings**.

- The ROC curve confirms that the triplet-trained embeddings generalize well across unseen identities in open-set verification.

4.2. Similarity Distance Distributions

To further investigate how well the embeddings separate genuine and imposter pairs, we plotted **distribution histograms** of similarity scores.

Visualization

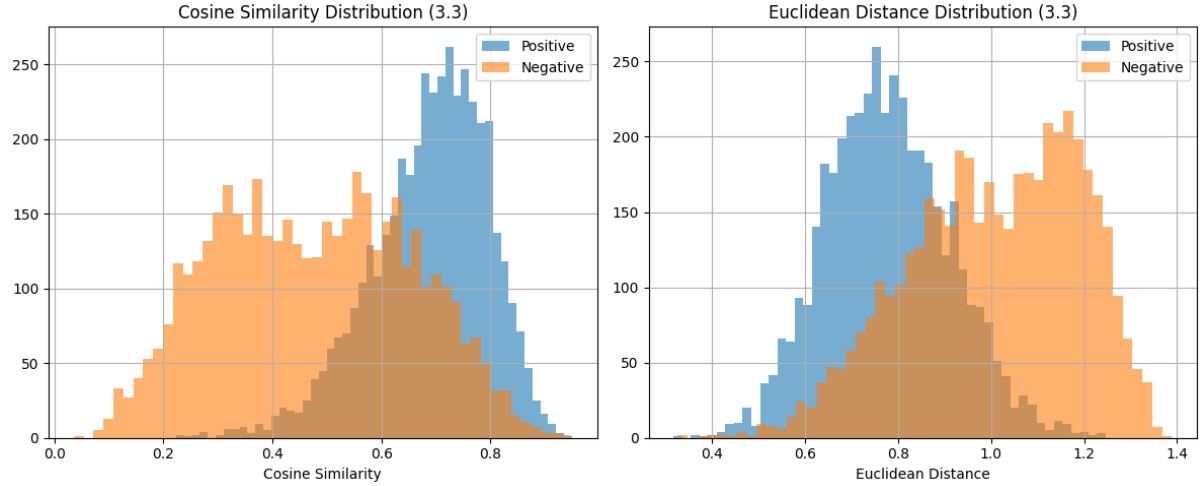


Figure. Distribution Histograms of Similarity Scores.

Insights

- **Cosine Similarity**:
  Positive pairs are skewed toward the higher range (0.7–0.9), while negatives cluster lower (0.2–0.6). This separation indicates the model has learned semantically aligned embedding vectors.
- **Euclidean Distance**:
  Positive pairs exhibit lower distances (peaking near 0.7–0.9), while negative pairs cluster toward 1.0–1.2. Despite some overlap, the margin is sufficient for threshold-based verification.
- These distributions validate the **AUC results**, showing meaningful statistical separability between positive and negative pairs.

4.3 Embedding Visualization via t-SNE

To understand the geometric structure of the learned embedding space, we employed **t-SNE (t-distributed Stochastic Neighbor Embedding)** to reduce the high-dimensional (128D) embeddings to a 2D space for visualization.

Classifier-Based Embeddings (Supervised)

- The supervised classifier model shows **broad clusters**, with some class overlap.
- Embedding separability is limited, which suggests that while the classifier can differentiate known classes, it **does not generalize as well to verification** tasks.

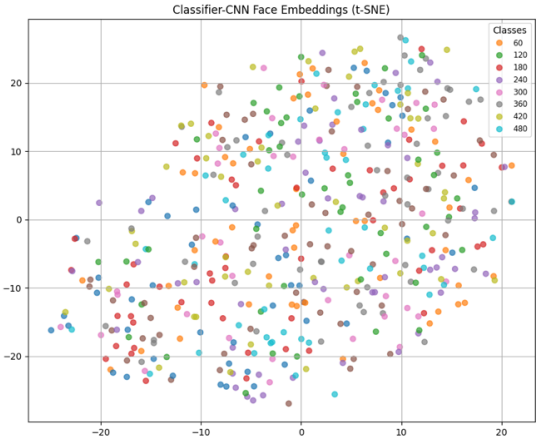


Figure. t-SNE Evaluation on Classifier Model

Triplet Loss-Based Embeddings

- Triplet-trained embeddings form **more compact and distinct clusters**.
- Inter-cluster margins are more pronounced, indicating that the model has learned an embedding space that enforces **intra-class cohesion** and **inter-class dispersion**.

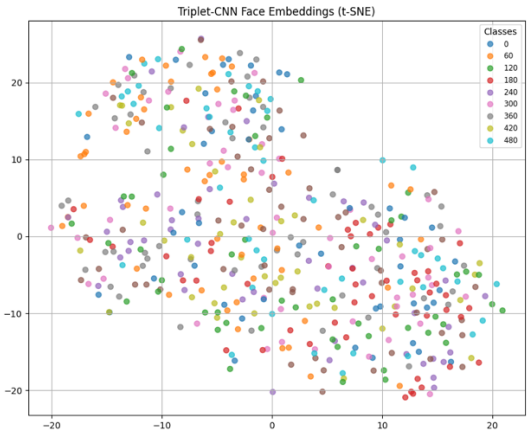


Figure. t-SNE Evaluation on Triplet Loss Model

Conclusion from t-SNE

The t-SNE plots support our ROC and distribution findings: **triplet-based metric learning** produces embeddings that are more suitable for face verification, particularly in open-set scenarios where identities are not seen during training.

Summary and Conclusion

METRIC	CLASSIFIER MODEL	TRIPLET MODEL
AUC (COSINE)	~0.51 (earlier)	0.8375
EMBEDDING STRUCTURE (T-SNE)	Scattered, overlapping	Distinct clusters
SIMILARITY DISTRIBUTIONS	Poor separability	Strong separability

The combined evidence from **ROC-AUC**, **histograms**, and **t-SNE plots** demonstrates that the **triplet loss-based model** achieves **superior verification performance** over the classification-based model. This confirms the effectiveness of metric learning for open-set facial recognition tasks and validates its role as the core embedding extractor for our face verification and attendance system.

5. Emotion Detection

5.1 Technology Overview

DeepFace is an open-source Python library developed by **Sefik Ilkin Serengil**, designed for facial analysis tasks such as recognition, emotion classification, age estimation, and gender detection. It provides a wrapper over several powerful pre-trained models including **VGG-Face, Google FaceNet, ArcFace, DeepFace, Dlib**, and **OpenFace**.

For **emotion recognition**, DeepFace specifically employs a **Convolutional Neural Network (CNN)** trained on the **FER-2013** dataset - a widely adopted dataset for facial expression classification.

5.2. Model and Methodology

The emotion recognition model used by DeepFace is based on a **compact CNN architecture**, trained on grayscale images of size 48×48, similar to the following typical structure:

```
Input (48x48 grayscale)
        ↓
Conv2D → ReLU → MaxPooling
        ↓
Conv2D → ReLU → MaxPooling
        ↓
Fully Connected Layer → ReLU
        ↓
Softmax Output (7 classes)
```

While DeepFace abstracts this, internally the model aligns with common practices seen in **Emotion-FERPlus** or **Mini-Xception**, where the CNN is designed with:

- Depthwise separable convolutions
- ReLU activation functions
- Dropout for regularization

This model is stored under `~/.deepface/weights/facial_expression_model_weights.h5`, when downloaded them with `pip install deepface`.

5.3. Dataset & Training Details

FER-2013 (Facial Expression Recognition 2013):

- 35,887 grayscale images
- Size: 48x48
- 7 emotion labels
- Collected through the Google image search API

Training was performed with standard techniques:

- Adam optimizer
- Cross-entropy loss
- Batch normalization
- Early stopping on validation loss

5.4. Performance Benchmarks

From public sources and the DeepFace repository:

METRIC	VALUE
DATASET	FER-2013
ACCURACY	~66–70% (baseline CNN)
ACCURACY (MINI-XCEPTION)	~71–74%
ACCURACY (DEEPFACE CNN)	~72% (on validation split)

Although DeepFace's emotion classifier may not reach state-of-the-art results like Transformers or multi-branch ensembles, it provides a **real-time**, **lightweight**, and **plug-and-play** solution that is ideal for embedded or real-time systems.

5.5. System Integration

The prediction pipeline proceeds as follows:

1. **Face Crop Extraction**: A cropped face image is passed to DeepFace after initial detection.
2. **Preprocessing**: The image is normalized and resized as per model requirements.
3. **Expression Classification**: The model outputs a softmax distribution across seven core emotions:
    - **Angry**
    - **Disgust**
    - **Fear**
    - **Happy**
    - **Sad**
    - **Surprise**
    - **Neutral**
4. **Dominant Emotion Extraction**: The highest probability label is returned and rendered beside the detected identity in the GUI:

```
emot = DeepFace.analyze(crop, actions=['emotion'], enforce_detection=False)[0]['dominant_emotion'].capitalize()
```

- The DeepFace module runs in **CPU mode** for compatibility and cross-platform stability.
- It enables real-time prediction (~5–10 FPS) depending on hardware, and functions robustly under varying lighting conditions.
- Emotion labels are appended to the user's bounding box and used in snapshot filenames for future auditing.

6. Anti-Spoofing

6.1. Technology Overview

Silent-Face Anti-Spoofing is a robust and lightweight face liveness detection framework developed by the **MiniVision AI team** and released as part of their participation in the **CelebA-Spoof Challenge** (CVPR 2020).

The system addresses the growing concern over facial presentation attacks in security-sensitive applications like biometric login or facial payment.

Silent-Face is **CNN-based**, employing an **ensemble strategy** using **2 lightweight convolutional models** trained independently and fused at inference.

6.2. Model Architecture

Silent-Face is an ensemble-based anti-spoofing framework leveraging **two lightweight Convolutional Neural Networks (CNNs)**:

- **MiniFASNetV2**
- **MiniFASNetV1SE**

These CNN models are designed with:

- Less than 1 million parameters
- Optimized for input resolution of 80x80
- Real-time inference on CPU (~30ms)

Each model outputs a 3-class prediction representing: live, spoof, and uncertain. The final prediction is computed by aggregating the two model scores via averaging.
These models are trained using the **CelebA-Spoof** dataset:

- 625,000 labeled images
- 10,177 subjects
- 3 spoof types: **print**, **replay**, **mask**

6.3. Model Performance Benchmarks

DATASET	ACCURACY	TPR@FAR=1%	ACER
CELEBA-SPOOF	~96.4%	~97.3%	2.8%
CASIA-MFSD	~93.2%	~95.1%	3.5%

Silent-Face is recognized for high generalization across multiple spoof types and remains competitive with deeper architectures while consuming far fewer resources.

6.4. Integration Strategy: Wrapper for Streaming Inference

Since the original GitHub repository was not optimized for real-time video streams, we implemented a wrapper spoof_detector.py to adapt the static inference pipeline for real-time application. The wrapper leverages the official modules from src/:

```
from src.anti_spoof_predict import AntiSpoofPredict
from src.generate_patches import CropImage
from src.utility import parse_model_name
```

The main class SpoofDetector encapsulates this functionality:

```
class SpoofDetector:
    def __init__(self, model_dir='./resources/anti_spoof_models', device_id=0):
        self.model = AntiSpoofPredict(device_id)
        self.cropper = CropImage()
        self.model_dir = model_dir


    def check_spoof(self, image):
        image_bbox = self.model.get_bbox(image)
        prediction = np.zeros((1, 3))
        for model_name in ["2.7_80x80_MiniFASNetV2.pth",
                           "4_0_0_80x80_MiniFASNetV1SE.pth"]:
            h_input, w_input, _, scale = parse_model_name(model_name)
            param = {...}
            ...
            pred = self.model.predict(img_cropped, model_path)
```

```
            prediction += pred
        label = np.argmax(prediction)
        value = prediction[0][label] / 3
        is_real = (label == 1)
        return is_real, value
```

This approach:

- Detects a face bounding box from the incoming image
- Crops the face area appropriately for both models
- Passes the preprocessed face through both CNNs
- Averages the softmax probabilities and outputs a final classification

6.5. Real-Time Integration in UI Pipeline

Pipeline Implementation:

1. **Input**: A face crop (RGB) is passed to the ensemble.
2. **Preprocessing**: Face alignment and normalization steps replicate training conditions.
3. **Inference**: Both models predict spoof confidence independently.
4. **Output**: A binary decision and confidence score are returned.

```
is_real, conf = spoof_detector.check_spoof(cv2.cvtColor(crop, cv2.COLOR_BGR2RGB))
```

This result is then used to determine the color of the bounding box and label text:

```
color = (0,255,0) if is_real else (0,0,255)
label_emot = f" | {emot} | {'Live' if is_real else 'Spoof'}"
```

Summary:

The integration of Silent-Face provides a lightweight yet robust safeguard against spoofing threats. Our wrapper enables compatibility with video streams, and the real-time inference is seamlessly merged into our Tkinter UI application. This ensures that each recognized identity is verified not only by facial embedding but also by presentation authenticity.

7. Graphical User Interface

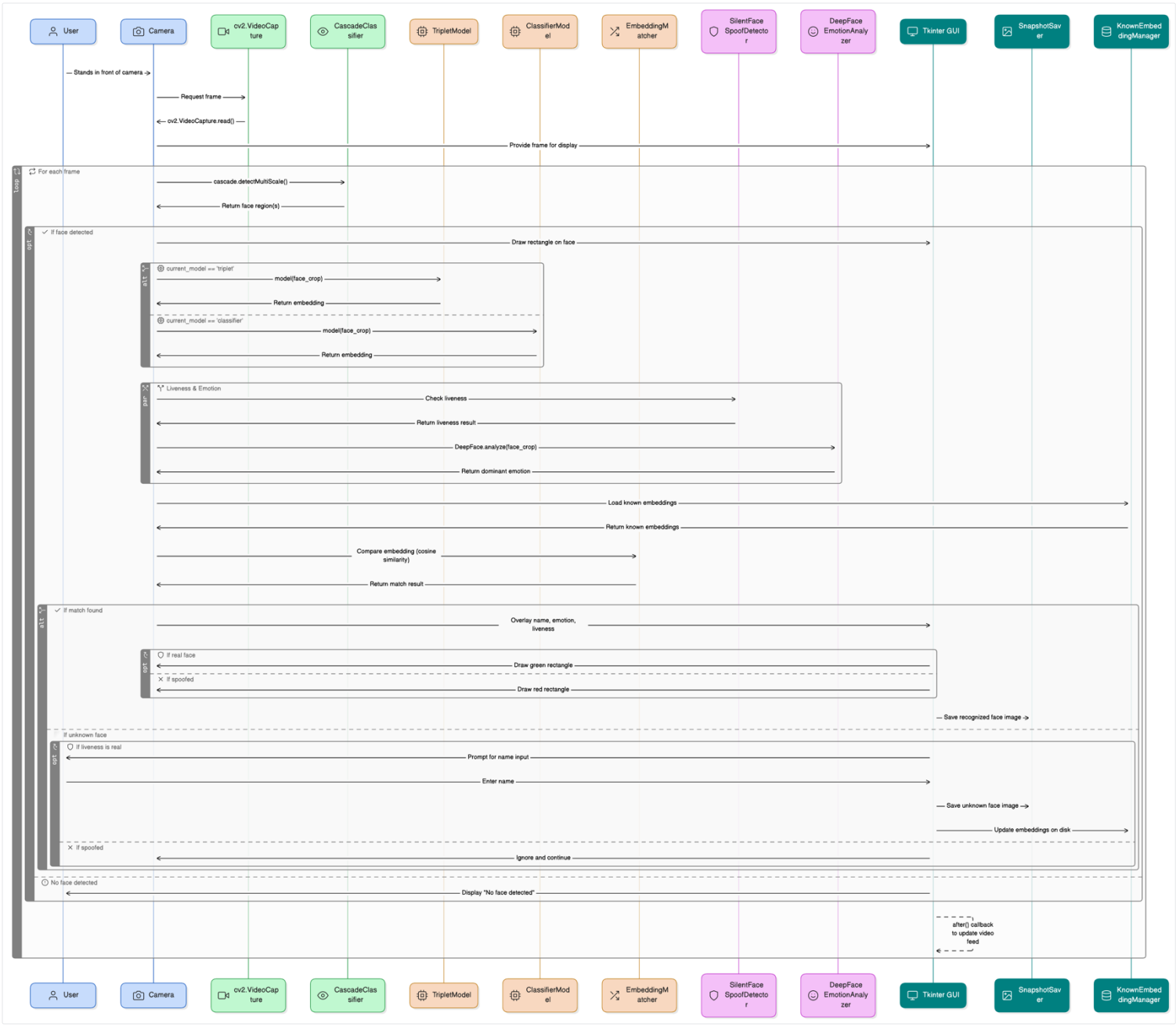


Figure. Sequence Diagram of UI Pipeline and Architecture

The graphical user interface (GUI) of our Face-Recognition Attendance System is designed using **Tkinter**, a native Python GUI toolkit. It integrates real-time webcam feed, identity recognition, emotion classification, and anti-spoofing verification. The architecture is modular, intuitive, and deployable on both macOS and Windows platforms.

7.1 System Setup Instructions

The UI is initialized and run via a single script project.py, which interfaces with multiple deep learning models.

Step-by-step setup:

1. **Create a Python virtual environment:**

```
python -m venv cos30082-env
source cos30082-env/bin/activate  # Linux/macOS
cos30082-env\Scripts\activate     # Windows
```

2. **Install required dependencies:**

pip install -r requirements.txt

3. **Download and organize models:**

```
Project/
├── project.py               # Main UI pipeline script
├── src/                     # Silent-Face source code
├── resources/               # For anti-spoofing models
│   └── anti_spoof_models/
├── models/                  # Contains trained classifier and triplet models
├── known_faces/             # Directory of labelled face images
├── snapshot/                # Session snapshots
├── embeddings/              # Saved embedding vectors
├── spoof_detector.py        # Silent-Face wrapper
└── requirements.txt         # Dependencies installation
```

7.2 Interface Architecture

The UI integrates multiple components:

- **Live Video Feed:** Captured via OpenCV.
- **Face Detection:** Conducted using OpenCV's Haar Cascade classifier.
- **Face Recognition:** Supports both **Triplet** and **Classification** models via an interactive toggle.
- **Emotion Detection:** Uses DeepFace.
- **Anti-Spoofing:** Powered by a wrapper SpoofDetector integrating MiniFASNet models.
- **Snapshot Logging:** One image per identity per session.
- **Dynamic UI:** Model switching, known identity gallery, snapshot gallery, and manual ID upload.

7.3. Core Functionalities

A. Model Switching Button

This allows toggling between the two recognition models during runtime.

```
def set_model(mode):
    global current_model
    current_model = mode
    status_var.set(f'Model: {mode.capitalize()}')
```

Face detection are embedded and classified dynamically based on the selected model on runtime.

```
emb = (embed_triplet(crop) if current_model=='triplet' else embed_classification(crop))
        name, score = match_embedding(emb)
```

Confidence level on model rendering to be labelled in different colours for better interpretation and experience.

```
if score < 0.3:   score_color = (0, 0, 255)    # Red
elif score < 0.5: score_color = (0, 165, 255)  # Orange
elif score < 0.7: score_color = (255, 0, 0)    # Blue
else:             score_color = (128, 0, 128)  # Purple
```

B. Live Recognition Loop

```
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5)
    for (x,y,w,h) in faces:
        crop = frame[y:y+h, x:x+w]
        emb = (embed_triplet(crop) if current_model=='triplet' else embed_classification(crop))
        name, score = match_embedding(emb)
```

Each detected face is passed through embedding, matched with known identities, and annotated with confidence.

```
def match_embedding(emb: np.ndarray, thresh: float = 0.8):
    known_emb = known_emb_trip if current_model=='triplet' else known_emb_cls
    sims = np.dot(known_emb, emb)
    idx = int(np.argmax(sims))
    if sims[idx] > thresh:
        return known_names[idx], float(sims[idx])
```

This function performs **identity matching** by comparing an input face embedding (emb) against a set of known embeddings (known_emb_trip or known_emb_cls) to determine if the input face is a recognized person. It returns either:

- The matched **name and similarity score**, or
- "Unknown" and None if the match is below a similarity threshold.

Inputs:

- emb (np.ndarray): A **128-dimensional normalized face embedding** generated from either the Triplet model or Classification model.
- thresh (float, default=0.8): A **similarity threshold**. If the cosine similarity between the input embedding and the best-matched known embedding is below this value, the face is considered unknown. Notice that the 0.8 threshold is a remarkably high threshold, preferably reduce to 0.5 - 0.7 would be more ideal.
- Computes **cosine similarity** between the input embedding and all known embeddings using np.dot()
- Because the embeddings are L2-normalized, the dot product here equals cosine similarity.
- Finds index of the known embedding with the **highest similarity score** to the input with int(np.argmax)

C. Emotion and Spoof Detection

```
emot = DeepFace.analyze(crop, actions=['emotion'], enforce_detection=False)[0]['dominant_emotion'].capitalize()
is_real, conf = spoof_detector.check_spoof(cv2.cvtColor(crop, cv2.COLOR_BGR2RGB))
```

Emotion is labeled alongside a real/spoof status overlay.

D. Real-Time Annotation

Labels are to be placed on the detected bbox using cv2.putText:

```
cv2.putText(disp, label_name, text_start, font, font_scale, color, thickness)
cv2.putText(disp, st_score, (text_start[0] + size_name[0], text_start[1]), font, font_scale, score_color, thickness)
cv2.putText(disp, label_emot, text_start[0] + size_name[0] + size_score[0], text_start[1]), font, font_scale, color, thickness)
```

The UI displays identity, similarity score, emotion, and spoofing verdict in separate color-coded segments.

E. Add New Identity (Auto/Manual)

- **Auto-Prompt** is triggered after 10s if a live unknown face persists.

```
elif name == 'Unknown' and is_real:
    now = time.time()
    if now - last_unknown_time > UNKNOWN_PROMPT_INTERVAL:
        last_unknown_time = now
        prompt_unknown(crop, emb)
```

- **Manual Upload**:

```
def handle_add_id():
    file_path = filedialog.askopenfilename(title='Select face image', filetypes=[(...)])
    img = cv2.imread(file_path)
    prompt_unknown(img, embed_triplet(img) if current_model == 'triplet' else embed_classification(img))
```

User can upload a file to register a face manually or after 10 seconds interval (avoid spamming), if an unknown face exist, system will prompt user to add this new face. Once face is registered, the image detected will be captured and saved alongside with their face being embedded and saved to register this face-id to the list of known identities. Upon prompting for the new face id's name, a window will be shown to ask for user-input on the name with option to add or cancel the request.

```
def prompt_unknown(crop, emb):
    preview = ImageTk.PhotoImage(Image.fromarray(cv2.cvtColor(crop, cv2.COLOR_BGR2RGB)).resize((120,120)))
    ...
    def save_and_close():
        name = entry.get().strip() or None
        path = KNOWN_DIR / f"{name}.jpg"
        cv2.imwrite(str(path), crop)
        known_names.append(name)
        known_emb = embed_triplet(crop) if current_model=='triplet' else embed_classification(crop)
```

```
        if current_model=='triplet':
            global known_emb_trip; known_emb_trip = np.vstack([known_emb_trip, known_emb])
        else:
            global known_emb_cls;  known_emb_cls  = np.vstack([known_emb_cls,  known_emb])
        messagebox.showinfo('Added', f'Added {name}')
        win.destroy(); resume()
    btnf = ttk.Frame(win); btnf.pack(pady=10)
    ttk.Button(btnf, text='Add', command=save_and_close).pack(side='left', padx=5)
    ttk.Button(btnf, text='Cancel', command=lambda: (win.destroy(), resume())).pack(side='right', padx=5)
```

F. Snapshot Feature

```
def save_snapshot(name, emotion, img_bgr):
    ts = time.strftime('%Y%m%d-%H%M%S')
    path = SNAP_DIR / f"{name}_{emotion}_{ts}.jpg"
    cv2.imwrite(str(path), img_bgr)
```

Each recognized individual is logged once per session with their dominant emotion. This is an additional feature to serve the gallery feature.

G. Image Gallery Viewers

```
def open_gallery(dir_path, title):
    ...
    canvas.create_window((10, 10), window=scroll_frame, anchor='nw')
    canvas.configure(yscrollcommand=scrollbar.set)
    images = []; for i, img_file in enumerate(Path(dir_path).glob('*')):
        ...
        # Initialise 3x3 grid with images and scrolling
        lbl_img = ttk.Label(scroll_frame, image=tk_img)
        lbl_img.image = tk_img
        lbl_img.grid(row=i//3*2, column=i%3, padx=5, pady=5)
        lbl_text = ttk.Label(scroll_frame, text=img_file.stem, style='Info.TLabel')
        lbl_text.grid(row=i//3*2+1, column=i%3, padx=5, pady=(0,10))
```

Users can browse previously snapshot-captured and registered identities via a scrollable Tkinter window.

7.4. UX and Styling

- **Theme:** clam with consistent padding, font (Segoe UI), and spacing.
- **Layout:** Split pane: video feed on left, control panel on right.
- **Responsive Performance:** GUI frame updates every 12 ms (~80 FPS cap).

7.5. Runtime and Frame Rendering

The status bar at the bottom of the GUI displays real-time system metadata:

```
status_bar.config(text=f"Model: {current_model.capitalize()} | Faces: {len(faces)} | Known: {len(known_names)}")
```

Run time are enforced to be consistently keep at 60 FPS, which allow a reasonably smooth interaction while maintain computer-resource balance on frame rendering.

```
root.after(16, update) # 60 FPS
```

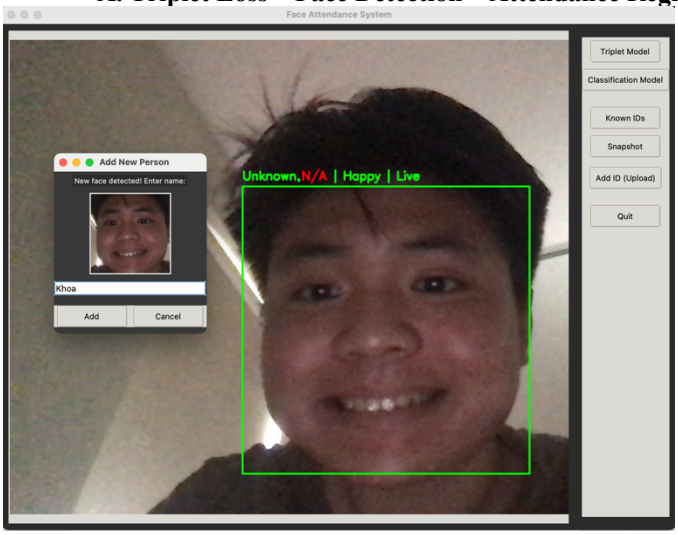
Whole pipelines and frame renderings are encapsulated in the update() function, with fallback, error, resume and cancellation, windows interaction being instantiated seamlessly.

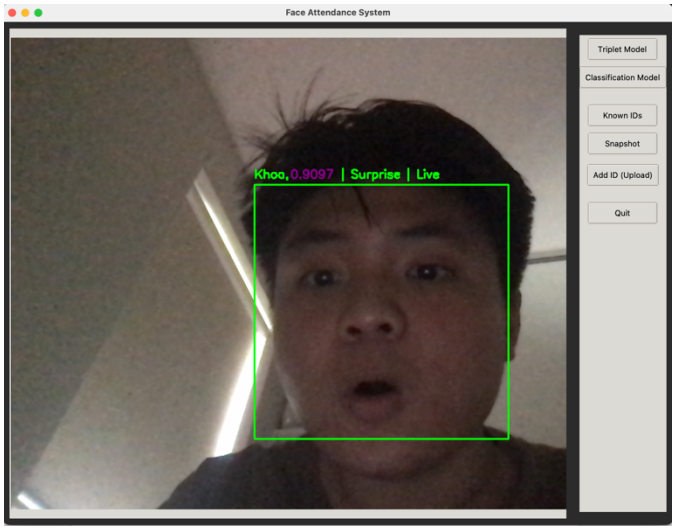
This modular and integrated Tkinter-based GUI allows seamless face attendance monitoring, dynamic enrollment, and visual verification in real time. It is robust, extensible, and adaptable for future enhancements such as CSV log exports and physical kiosk integration.

7.6. Design Demonstration

Screenshots with different scenarios are presented.

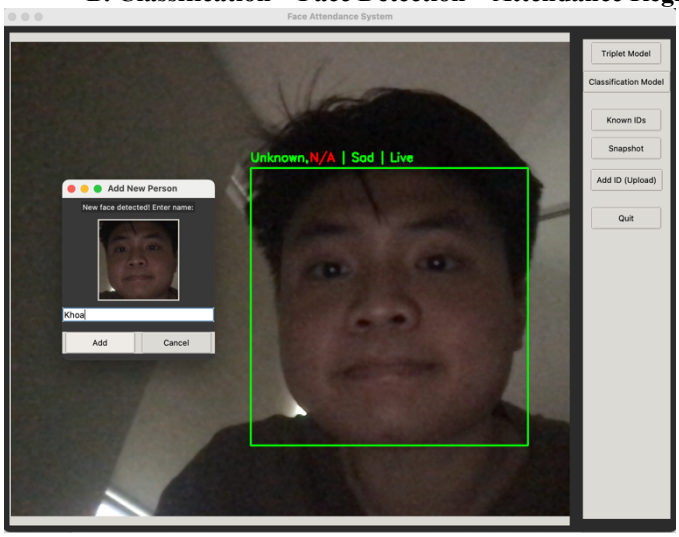
A. Triplet Loss – Face Detection – Attendance Registration – Emotion

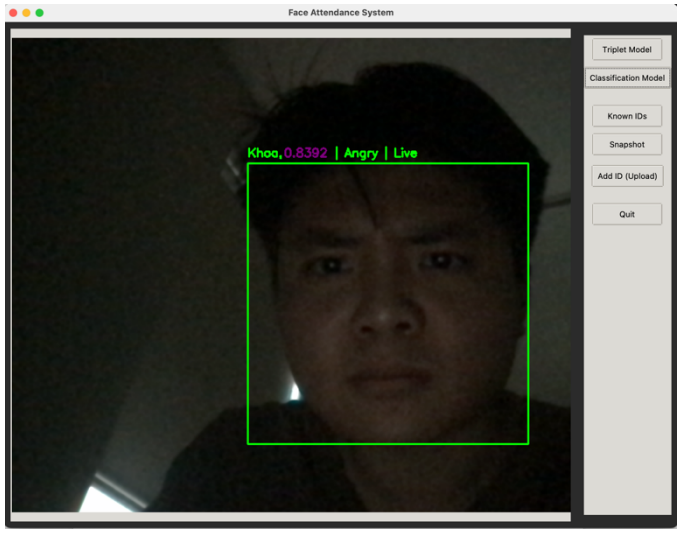




Observation: At default, the model chosen on the right panel will be the Triplet Loss model (since they have better accuracy). These 2 screenshots give a scenario when a face's embedded vector hasn't been saved (left image), which prompts the user to input the name in a separate window (frame streaming paused); Once user insert the name, that person will be captured to the 'known_faces' directory and vector saved to 'embedding' directory. Therefore, that person can be identified (right image), which prove the success of attendance validating and new-id insertion in this UI design. On top of that, deepface model also accurately identify the emotion-expression from the 2 given screenshots (happy and surprise).

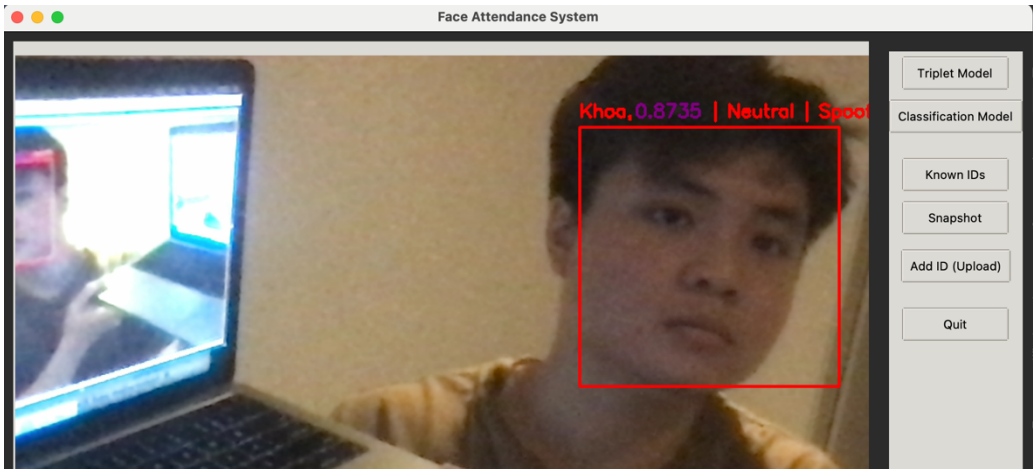
B. Classification – Face Detection – Attendance Registration – Emotion





Observation: Similarly, we switch to the Classification model and test it out. These 2 screenshots also give the scenario when an unknown is detected, which prompts the user to input the name in a separate window. Noticeably, comparing to the Triplet Loss, confidence/accuracy value returns from the Classification is sightly lower, however, in order to serve the purpose of face validation, it can still perform acceptably. On top of that, deepface model also accurately identify the emotion-expression from the 2 given screenshots (sad and angry).

C. Anti-Spoofing

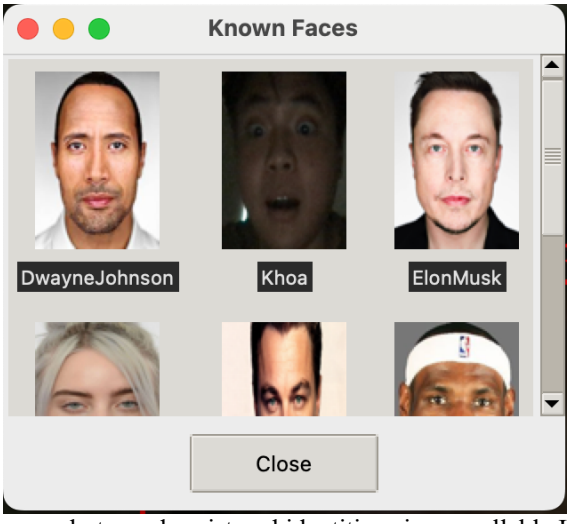


Observation: This first scenario showing anti-spoofing feature perfectly detect that the face is being reflected from mirror, therefore flagged as spoof.

Observation: On the second scenario, we try to detect the face from an image that is played from my phone, which also accurately flagged as spoof.

Using the scenario from A&B with this, we can conclude that anti-spoofing feature is powerful yet also lightweight and efficient on limited computer resource (CPU) and can be able to render seamlessly in real-time.

D. Gallery



Observation: User can access snapshots and registered identities via a scrollable UI window. This window will be triggered from the right side bar buttons, which also has an option to manual upload new identity image.

8. Innovation and Future Enhancements

While the current system successfully demonstrates an end-to-end face recognition attendance pipeline integrated with emotion detection and anti-spoofing mechanisms, several innovative extensions were identified but not yet implemented due to time constraints and computational resource limitations. These enhancements could significantly improve the system's performance, scalability, and applicability in real-world deployments.

8.1. GAN-Based Synthetic Data Augmentation

Proposed Innovation:
Incorporate **Generative Adversarial Networks (GANs)** to automatically generate synthetic facial images for each identity in the training dataset.

Motivation & Benefits:
Data imbalance and insufficient intra-class diversity (e.g., different angles, lighting, and expressions) often degrade model generalization. GANs like **StyleGAN2**, **FaceGAN**, or **DeepFake architectures** can generate realistic face variants to simulate environmental variations without needing manual data collection.

Implementation Strategy:

- Train or adapt a pretrained StyleGAN2 model.
- For each real face, synthesize 20–50 augmentations (e.g., lighting changes, occlusions).
- Incorporate the synthetic images into the training pipeline for both classification and triplet networks.

Feasibility:
Pretrained GAN weights are publicly available. Integration would mainly require GPU-based batch generation and merging of augmented samples into the existing PyTorch Dataset. With minimal modification, this can significantly improve the generalization of face embeddings.

8.2. Transfer Learning & Fine-Tuning on FaceNet / MobileFaceNet

Proposed Innovation:
Employ **transfer learning** by starting with pretrained face recognition backbones such as **MobileFaceNet**, **ArcFace**, or **ResNet-50-based FaceNet**, and fine-tune them on our custom dataset.

Motivation & Benefits:
Pretrained models encapsulate high-level face feature representations learned on massive datasets (e.g., MS-Celeb-1M, VGGFace2), which helps boost accuracy even with limited data. Fine-tuning allows adaptation to the specific domain of the application (e.g., frontal, webcam-collected faces).

Implementation Strategy:

- Replace the final classification layer of pretrained models.
- Freeze early layers for stability; unfreeze and fine-tune top layers.
- Use our dataset for supervised learning or triplet fine-tuning.

Feasibility:
Frameworks such as PyTorch or TensorFlow Hub already support loading and modifying these architectures. Integration would require adjusting image sizes and preprocessing pipelines, but can be achieved with standard transfer learning practices.

8.3. Real-Time Attendance Logging System and Deployment

Proposed Innovation:
Automate the export and logging of recognized identities into structured formats (e.g., **CSV**, **SQLite**, or **MongoDB**), timestamped per session.

Motivation & Benefits:
Provides verifiable attendance records, supports auditability, and enables integration with larger HR or campus systems.

Implementation Strategy:

- Add a logging module to append attendance entries with name, emotion, spoof-status, and timestamp.

- Avoid duplication within session by maintaining an in-memory set.
-

Feasibility:
Easy to implement and light on resources. Can be integrated directly into the save_snapshot() method or the recognition loop.

8.4. Pose-Robust Face Recognition (3D-Aware Models)

Proposed Innovation:
Extend recognition capabilities beyond frontal faces by integrating pose-invariant embeddings using models such as 3D Morphable Models (3DMM) or face frontalization networks.

Motivation & Benefits:
Current recognition depends heavily on frontal face input. Pose-robust techniques improve accuracy in real-world usage where users might not be perfectly aligned to the camera.

Implementation Strategy:

- Integrate 3D face alignment models to frontalize side-view inputs before embedding.
- Alternatively, replace the embedding model with one trained on multi-pose datasets.

Feasibility:
Models like PRNet or Hopenet can be used. However, requires GPU inference and additional preprocessing pipelines.

Conclusion

The Face Recognition Attendance System developed in this project represents a complete, modular, and extensible solution for real-time identity verification enriched with emotion analysis and anti-spoofing. Through the integration of classification and metric learning approaches, using both supervised CNNs and triplet loss, the system ensures reliable facial recognition across a wide range of users. Emotion detection powered by DeepFace and robust liveness checks via the Silent-Face anti-spoofing ensemble further elevate the reliability and security of the platform.

From data collection and model training to interface design and user interaction, the project demonstrates careful attention to modular architecture, allowing seamless switching between models, easy onboarding of new users, and effective visualization through a Tkinter-based GUI. The design enables efficient real-time execution on consumer-grade hardware and supports future improvements without requiring major restructuring.

Although resource limitations prevented the full implementation of advanced enhancements, several forward-looking innovations, such as GAN-based data synthesis, transfer learning, pose-robust modeling, attendance logging. These present realistic opportunities to further improve performance, robustness, and scalability.

Overall, the system achieves its core objectives and lays a strong foundation for future work in intelligent human-computer interaction and identity-aware applications.

Appendix

These are resources for accessibility:
1. Google Colab training session. Accessible via:
https://colab.research.google.com/drive/1pb9YWCrNkx7fd9kd9yKIFWQCRmKhmmao?usp=sharing
2. Google Drive full project files. Accessible via:
https://drive.google.com/drive/folders/1kLmvh6JgHNs7LWjqt1dRGlfxqPkfLt1g?usp=sharing