

Swinburne University of Technology



COS30082 - Applied Machine Learning

Face Recognition Attendance System

Group 7

Nguyen Anh Tuan - 103805949

Nghiem Tuan Linh - 10418733

Phan Huy Quang - 104177128

Hanoi, Vietnam

Table of content

1. Instruction.....	3
2. Introduction.....	4
3. Methodology.....	4
3.1. Dataset.....	4
3.2. Metric Learning method.....	4
3.2.1. Data preprocessing.....	4
3.2.2. Model architecture.....	6
3.2.3. Loss function.....	6
3.2.4. Training scheme.....	7
3.3. Classification method.....	7
3.3.1. Data preprocessing.....	7
3.3.2. Model Architecture.....	8
3.3.3. Loss function and Training scheme.....	8
3.4. Anti-spoofing.....	9
3.4.1. System Design.....	9
3.4.2. Model Architecture.....	10
3.4.3. Model Inference.....	10
4. Result and Discussion.....	10
4.1. Comparison of the Three Models and Justification for Metric Learning.....	10
4.1.1. Siamese Network with Euclidean Distance Loss.....	11
4.1.2. Siamese Network with Cosine Distance Loss.....	11
4.1.3. Classification Model.....	12
4.2. Why Metric Learning is Chosen Over Classification.....	12
4.2.1. Theoretical Justification.....	12
4.2.2. Result-Based Justification.....	12
4.2.3. Conclusion.....	13
5. Reference.....	13

1. Instruction

Make sure you download, unzip the code, and run the code on Google Colab.
Get the recognition model and anti-spoofing model path and replace this line:

```
embedding_model = load_model('/content/drive/MyDrive/Submission/embedding_cosine.keras') # Face recognition model  
anti_spoofing_model = torch.load("/content/drive/MyDrive/Submission/anti_spoofing.pth", map_location=device) # Anti-
```

Run this:

```
[16] %%writefile app.py  
## Step 2: Import necessary libraries  
import streamlit as st  
import cv2  
import dlib  
import numpy as np  
from PIL import Image  
from torchvision import transforms  
import torch  
from tensorflow.keras.models import load_model  
import os  
  
# Step 3: Load Models  
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  
embedding_model = load_model('/content/drive/MyDrive/Submission/embedding_cosine.keras') # Face recognition model  
anti_spoofing_model = torch.load("/content/drive/MyDrive/Submission/anti_spoofing.pth", map_location=device) # Anti-  
anti_spoofing_model.eval()  
  
# Step 4: Dlib Setup for Face Detection  
detector = dlib.get_frontal_face_detector()  
landmark_predictor = dlib.shape_predictor("/content/drive/MyDrive/Submission/shape_predictor_68_face_landmarks.dat")  
  
# Constants  
distance_threshold = 0.6  
transform = transforms.Compose([  
    transforms.Resize((224, 224)),  
    transforms.ToTensor(),  
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),  
)
```

Then

```
!streamlit run app.py &>/content/logs.txt & npx localtunnel --port 8501
```

A url will appear, click on it. It will lead you to the localtunnel. Paste your PUBLIC ip address to access the demo.

It is quite slow and clunky so be patient

In the demo you can capture the image of your face with the webcam, then register it or check in if you have already registered.

2. Introduction

This project presents the development of a Face Attendance System that leverages face verification techniques to automate attendance tracking in enterprise settings. The system uses a Convolutional Neural Network (CNN) to generate compact and discriminative face embeddings, accurately comparing face pairs for identity verification.

To ensure reliability, the system incorporates an anti-spoofing module to detect and prevent unauthorized access via counterfeit facial inputs. The project also includes a user-friendly interface for easy face registration and attendance verification. This report outlines the methodologies, results, and comparative evaluation of the system's components, highlighting its effectiveness and potential applications.

3. Methodology

3.1. Dataset

For the dataset in this project, our group used the dataset from Kaggle, which will be referred to in the Reference part of this report.

3.2. Metric Learning method

Our group uses a triplet loss technique to implement metric learning. The goal is to create embeddings that reduce the distance between the anchor and positive face pairs, which are similar, and maximize the distance between the anchor and negative face pairs, which are dissimilar.

3.2.1. Data preprocessing

A Siamese network is trained for facial recognition using a triplet dataset. Three image triplets make up the dataset: an anchor, a positive image featuring the anchor's person, and a negative image with a different person. To create these triplets from the image directory, a unique TripletFace class is created, making sure that for every triplet:

- The positive photos and the anchor are both owned by the same person.
- A different person who was chosen at random provided the negative image. The network can learn to minimize the distance between similar images (anchor and positive) and maximize the distance between different ones (anchor and negative) with the help of this configuration. To facilitate effective training, the triplet dataset is produced in batches.

```

class TripletFace(keras.utils.Sequence):

    def __init__(self, image_dir, batch_size=64, image_size=(64, 64), seed=42):
        self.image_dir = image_dir
        self.batch_size = batch_size
        self.image_size = image_size

        self.imgs_path = {
            person: os.listdir(os.path.join(image_dir, person)) for person in os.listdir(image_dir)
        }
        self.random = random.Random(seed)

        self.__produce_triplet_batches()

    def __produce_triplet_batches(self):
        self.triplets = []
        temp = []

        for person in self.imgs_path:
            anchor_paths = self.random.sample(self.imgs_path[person], len(self.imgs_path[person]) // 2)
            positive_paths = {*self.imgs_path[person]} - {*anchor_paths}
            for anchor, positive in zip(anchor_paths, positive_paths):
                anchor_img = os.path.join(self.image_dir, person, anchor)
                positive_img = os.path.join(self.image_dir, person, positive)
                negative_img = os.path.join(
                    self.image_dir,
                    negative_person := self.random.choice( list( {*self.imgs_path.keys()} - {person} ) ),

```

Figure 1: Data preprocessing for Metric Learning method

```

    def __produce_triplet_batches(self):
        self.triplets = []
        temp = []

        for person in self.imgs_path:
            anchor_paths = self.random.sample(self.imgs_path[person], len(self.imgs_path[person]) // 2)
            positive_paths = {*self.imgs_path[person]} - {*anchor_paths}
            for anchor, positive in zip(anchor_paths, positive_paths):
                anchor_img = os.path.join(self.image_dir, person, anchor)
                positive_img = os.path.join(self.image_dir, person, positive)
                negative_img = os.path.join(
                    self.image_dir,
                    negative_person := self.random.choice( list( {*self.imgs_path.keys()} - {person} ) ),
                    self.random.choice(self.imgs_path[negative_person])
                )
                temp.append((anchor_img, positive_img, negative_img))

        self.random.shuffle(temp)

        for i in range(0, len(temp), self.batch_size):
            self.triplets.append( temp[i:i + self.batch_size] )

    def __preprocess_image(self, image_path):
        image = tf.io.read_file(image_path)
        image = tf.image.decode_jpeg(image, channels=3)
        image = tf.image.resize(image, self.image_size)
        image = keras.applications.resnet.preprocess_input(image)

```

Figure 2: Data preprocessing for Metric Learning method

```

        self.triplets.append( temp[i:i + self.batch_size] )

    def __preprocess_image(self, image_path):
        image = tf.io.read_file(image_path)
        image = tf.image.decode_jpeg(image, channels=3)
        image = tf.image.resize(image, self.image_size)
        image = keras.applications.resnet.preprocess_input(image)
        return image

    def on_epoch_end(self):
        self.__produce_triplet_batches()

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

    def __getitem__(self, idx):
        batch = self.triplets[idx]
        anchor_paths, positive_paths, negative_paths = zip(*batch)
        return (
            tf.stack([self.__preprocess_image(path) for path in anchor_paths]),
            tf.stack([self.__preprocess_image(path) for path in positive_paths]),
            tf.stack([self.__preprocess_image(path) for path in negative_paths]),
        )

```

Figure 3: Data preprocessing for Metric Learning method

3.2.2. Model architecture

- **ResNet 50 model:** Feature extraction is based on the pre-trained ResNet50 model (without top layers). The top layer is excluded (`include_top=False`) and ResNet50 is loaded with `weights='imagenet'`. A 128-dimensional embedding vector is created by passing the ResNet50 output through a **GlobalAveragePooling2D** layer and a Dense layer.

```
[ ] from tensorflow.keras.applications import ResNet50
    from tensorflow.keras.models import Model
    from tensorflow.keras.layers import Dense, GlobalAveragePooling2D

    def resnet50():
        base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(64, 64, 3))
        x = GlobalAveragePooling2D()(base_model.output)
        outputs = Dense(128)(x)
        return Model(inputs=base_model.input, outputs=outputs)
```

Figure 4: ResNet 50 model for the Metric Learning method

- **Siamese Network Architecture:** The structure of the Siamese network is defined by the `triplet_head` function. Three inputs, anchor, positive, and negative, are fed into the same embedding network to produce their corresponding embeddings.

```
def triplet_head(embedding):
    anchor_input = keras.Input(name="anchor", shape=(64, 64, 3))
    positive_input = keras.Input(name="positive", shape=(64, 64, 3))
    negative_input = keras.Input(name="negative", shape=(64, 64, 3))

    anchor_embedding = embedding(anchor_input)
    positive_embedding = embedding(positive_input)
    negative_embedding = embedding(negative_input)

    return keras.Model(inputs=[anchor_input, positive_input, negative_input], outputs=[anchor_embedding, positive_embedding, negative_embedding])
```

Figure 5: Siamese Network Architecture for the Metric Learning method

3.2.3. Loss function

The **SiameseModel** class implements triplet loss as the loss function. The anchor-positive (ap) and anchor-negative (an) pairs' Euclidean distances are calculated, and the triplet loss formula is used to minimize the difference between the anchor-positive and anchor-negative distances and maximize the difference between the two. Using either the Euclidean or cosine distance, the **__distance** function determines the separation between anchor and positive and anchor and negative embeddings. The **__triplet_loss** function applies the margin-based loss to enforce the required separation between the distances.

```
[ ] class SiameseModel(keras.Model):

    def __init__(self, embedding, loss="euclidean", margin=0.5):
        super().__init__()
        assert loss == "euclidean" or loss == "cosine", "loss must be either 'euclidean' or 'cosine'"
        self.siamese = triplet_head(embedding)
        self.loss = loss
        self.margin = margin
        self.loss_tracker = keras.metrics.Mean(name="loss")
        self.auc_metric = keras.metrics.AUC(name="auc")

    def __distance(self, anchor, other):
        if self.loss == "euclidean":
            return keras.ops.sum(keras.ops.square(anchor - other), axis=-1)
        # Normalize embeddings before cosine similarity
        anchor = tf.math.l2_normalize(anchor, axis=-1)
        other = tf.math.l2_normalize(other, axis=-1)
        return 1 + keras.losses.cosine_similarity(anchor, other)

    def __triplet_loss(self, ap_distance, an_distance):
        return keras.ops.maximum(ap_distance - an_distance + self.margin, 0)

    def call(self, inputs):
        return self.siamese(inputs)

    def train_step(self, data):
        with tf.GradientTape() as tape:
```

Figure 6: Loss Function for the Metric Learning method

3.2.4. Training scheme

The **Adam optimizer** is used to construct the model, and the triplet loss is used for training. The optimal weights are saved during training using **ModelCheckpoint**, and the optimizer is applied during the training loop to minimize the loss.

The Siamese model is trained using the fit method, using the **triplets_dataset** as the training dataset and the **val_triplets_dataset** as the validation dataset. The **ModelCheckpoint** guarantees that, depending on validation performance, the optimal model is preserved.

```
[ ] siamese_cosine_history = siamese_cosine.fit(
    triplets_dataset,
    validation_data=val_triplets_dataset,
    epochs=5,
    callbacks=[
        keras.callbacks.ModelCheckpoint("model_siamese_cosine.weights.h5", save_best_only=True, save_weights_only=True)
    ]
)
```

Figure 7: Training Scheme for the Metric Learning method

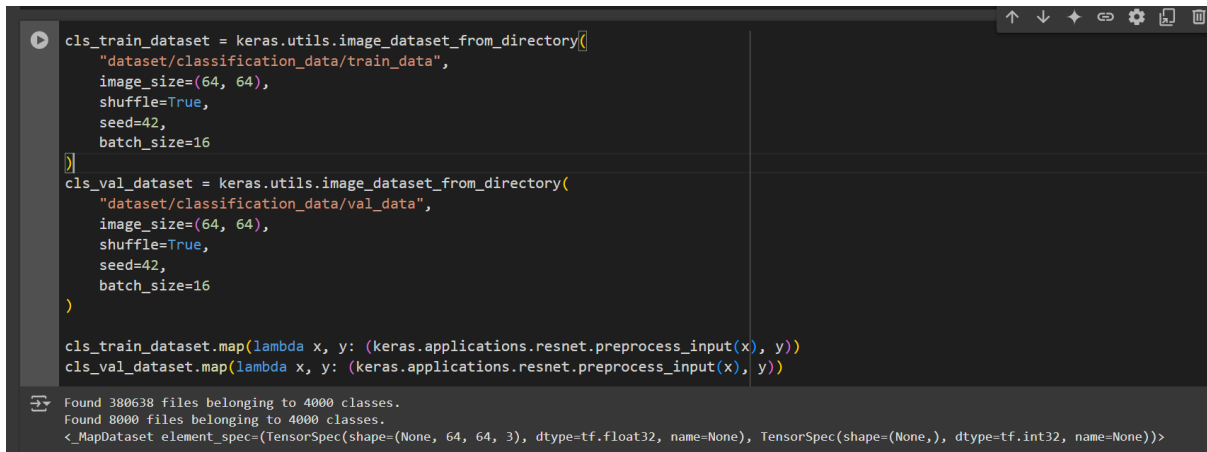
3.3. Classification method

Additionally, a supervised learning technique was used to categorize face photos into pre-established identity classes. This method works better for closed-set problems.

3.3.1. Data preprocessing

For the data preprocessing, first, the code loads the training and validation datasets from directories, where each subdirectory corresponds to a class. The images are resized to 64x64 pixels for consistency, and random shuffling is applied to the data to prevent any order-based patterns from affecting model training. The datasets are then divided into batches of 16 images for efficient training.

Key preprocessing steps are performed on both the training and validation datasets using the `map()` function. This applies the **keras.applications.resnet.preprocess_input** method to each image, which normalizes the pixel values to ensure compatibility with the ResNet model. The main goal of this preprocessing is to adjust the image data into the format expected by the pre-trained model.

A screenshot of a Jupyter Notebook interface. The code cell contains two lines of code to load training and validation datasets using Keras' `image_dataset_from_directory` function. Both datasets are configured with `image_size=(64, 64)`, `shuffle=True`, `seed=42`, and `batch_size=16`. The training dataset is loaded from `dataset/classification_data/train_data` and the validation dataset from `dataset/classification_data/val_data`. Below the code, the output shows that 388638 files were found for 4000 classes in the training set and 8000 files for 4000 classes in the validation set. The output also displays the `MapDataset` element specification, indicating that the training data is a `TensorSpec` with shape `(None, 64, 64, 3)` and dtype `tf.float32`, and the validation data is a `TensorSpec` with shape `(None,)` and dtype `tf.int32`.

```
cls_train_dataset = keras.utils.image_dataset_from_directory(  
    "dataset/classification_data/train_data",  
    image_size=(64, 64),  
    shuffle=True,  
    seed=42,  
    batch_size=16  
)  
  
cls_val_dataset = keras.utils.image_dataset_from_directory(  
    "dataset/classification_data/val_data",  
    image_size=(64, 64),  
    shuffle=True,  
    seed=42,  
    batch_size=16  
)  
  
cls_train_dataset.map(lambda x, y: (keras.applications.resnet.preprocess_input(x), y))  
cls_val_dataset.map(lambda x, y: (keras.applications.resnet.preprocess_input(x), y))
```

Found 388638 files belonging to 4000 classes.
Found 8000 files belonging to 4000 classes.
<MapDataset element_spec=(TensorSpec(shape=(None, 64, 64, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None,), dtype=tf.int32, name=None))>

Figure 8: Data preprocessing for the Classification method.

3.3.2. Model Architecture

ResNet50 is used as the backbone, with the following layers and components for the classification method:

- **Input Layer:** A Keras input layer with the shape (64, 64, 3) is made to take 64x64 pixel pictures with three RGB color channels.
- **Embedding Layer:** The input undergoes the application of the embedding function, which in this case is a pre-trained ResNet50 model. This functions as an extractor of features.
- **Dense Layer (1000 units):** A 1000-unit fully linked layer is added, acting as the initial layer following the ResNet model. It converts the features that have been retrieved into a higher-dimensional space.
- **Batch Normalization:** To increase the speed and stability of training, this layer normalizes the activations.
- **Activation Layer:** A ReLU activation function is applied.
- **Output Layer:** With a softmax activation function and 4000 units, the final layer is used for multi-class classification tasks. For each of the 4000 classes, probabilities are generated by this layer.

3.3.3. Loss function and Training scheme

The model is compiled with:

- **Optimizer:** The **Adam optimizer** is used with a very small learning rate (0.00005) to ensure gradual updates and avoid large oscillations during training.
- **Loss Function:** **SparseCategoricalCrossentropy** is chosen as the loss function, which is appropriate for multi-class classification where the labels are integers.
- **Metrics:** The model will track accuracy during training to measure performance.

The model is also trained using the `fit()` function and the following components:

- **Training Data:** Training is conducted using the preprocessed `cls_train_dataset`.
- **Validation Data:** Performance is tracked throughout training using the `cls_val_dataset`.
- **Epochs:** The model is trained for 10 epochs

- **Callbacks:** Every time the validation performance improves, the model's weights are saved using a **ModelCheckpoint** callback; the optimal model is saved in a file called **model_classification.weights.h5**.

```
[ ] def classification_head(embedding):
    inputs = keras.Input(shape=(64, 64, 3))
    x = embedding(inputs)
    x = keras.layers.Dense(1000)(x)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Activation("relu")(x)
    outputs = keras.layers.Dense(4000, activation="softmax")(x)

    return keras.Model(inputs=inputs, outputs=outputs)

[ ] embedding_classification = resnet50()

[ ] classification = classification_head(embedding_classification)

[ ] classification.compile(
    optimizer=keras.optimizers.Adam(0.00005),
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)
```

Figure 9: Resnet 50 model Architecture for the Classification method.

```
[ ] classification.fit(
    cls_train_dataset,
    validation_data=cls_val_dataset,
    epochs=10,
    callbacks=[
        keras.callbacks.ModelCheckpoint("model_classification.weights.h5", save_best_only=True, save_weights_only=True)
    ]
)
```

Figure 10: Resnet 50 model Architecture for the Classification method.

3.4. Anti-spoofing

Designed to detect efforts at spoofing in biometric authentication systems, the anti-spoofing module The module analyzes video or picture data using a deep learning model educated on "real" and "fake" labels to classify whether the input is authentic or a faked effort. The implementation specifics, procedures, and choices taken during the evolution of the anti-spoofing model we created are described in this paper.

3.4.1. System Design

- **Input Types**
 - **Images:** Static images provided for verification.
 - **Videos:** Videos submitted for anti-spoofing detection.
- **Processing Workflow**
 - Extract frames from videos (if applicable).
 - Preprocess frames to match the model's input requirements.
 - Run frames through the trained model for classification.
 - Aggregate frame-level predictions to determine the overall label for a video.
- **Output**
 - Binary classification:

- **Class 0:** "Fake" (spoof attempt).
- **Class 1:** "Real" (genuine biometric input).

3.4.2. Model Architecture

- **Base Model**
 - **ResNet-18:** A lightweight and efficient convolutional neural network pre-trained on ImageNet.
 - Fine-tuned for binary classification on the provided dataset with "real" and "fake" labels.
- **Training Data:**
 - The dataset was structured into two classes:
 - **Class 0:** Spoofed images and videos, including printed photos and replay attacks.
 - **Class 1:** Genuine images and videos.
- **Training Configuration:**
 - **Input Size:** Images resized to 224x224.
 - **Loss Function:** CrossEntropyLoss.
 - **Optimizer:** Adam with a learning rate of 0.001.
 - **Batch Size:** 16.
 - **Epochs:** Trained for 50 epochs until convergence.

3.4.3. Model Inference

- Each frame is passed through the model for classification:
 - **Output:** Logits for "real" and "fake."
 - **Prediction:** The class with the highest probability.

4. Result and Discussion

4.1. Comparison of the Three Models and Justification for Metric Learning

The provided ROC curves compare the performance of three models: a traditional classification model and two Siamese networks employing different distance metrics for loss computation (Euclidean Distance and Cosine Distance). For easier demonstration, our group applied Matplotlib to show the rate for each model.

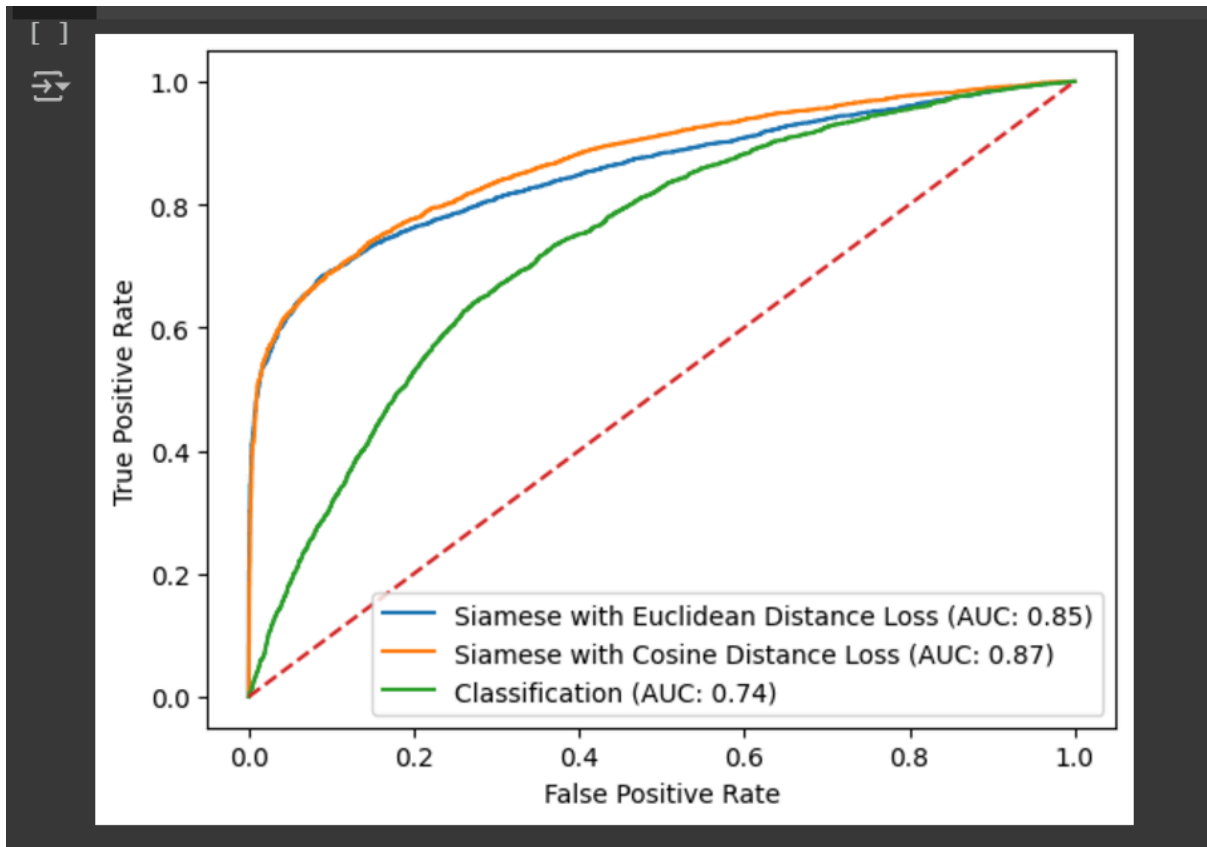


Figure 12: ROC curve and AUC score plot

Here's a detailed comparison and justification for the use of metric learning over classification:

4.1.1. Siamese Network with Euclidean Distance Loss

- **AUC: 0.85**
- **Strengths:**
 - Leverages a pairwise learning framework, focusing on the similarity between data points rather than direct classification.
 - Euclidean distance effectively captures feature-space relationships by considering absolute differences between embeddings.
- **Weaknesses:**
 - While Euclidean distance captures magnitude differences well, it might be less effective in high-dimensional spaces where angles between embeddings matter more (as addressed by Cosine Distance).
- **Performance:** Consistently outperforms the classification model and shows a strong ability to separate classes.

4.1.2. Siamese Network with Cosine Distance Loss

- **AUC: 0.87**
- **Strengths:**
 - Excels in capturing angular relationships between embeddings, which is particularly useful in high-dimensional or sparse feature spaces.

- Focuses on the direction of the embeddings rather than their magnitude, making it robust to scale variations.
- **Weaknesses:**
 - Might require additional normalization to ensure embeddings are unit vectors for accurate Cosine Distance calculations.
- **Performance:** Best-performing model with the highest AUC score, indicating that this loss function aligns well with the task at hand.

4.1.3. Classification Model

- **AUC:** 0.74
- **Strengths:**
 - Directly predicts the class labels, making it intuitive and straightforward to implement.
 - Suitable for problems where the dataset has clearly defined class boundaries.
- **Weaknesses:**
 - Struggles with data where feature similarities or relationships are more critical than absolute labels.
 - Unable to generalize well for unseen classes, as it does not explicitly learn relationships between data points.
- **Performance:** Performs the worst among the three, suggesting that it fails to capture the nuanced relationships between data points effectively.

4.2. Why Metric Learning is Chosen Over Classification

Based on the models comparison, our group believe that the metric learning method is more suitable than the classification method for several reasons

4.2.1. Theoretical Justification

Metric learning, as implemented in the Siamese networks, focuses on learning relationships (similarities or dissimilarities) between data points rather than just assigning labels. This approach is particularly powerful in tasks involving comparison, such as face verification, signature matching, and recommendation systems.

The loss function in Siamese networks explicitly optimizes for embeddings that are closer for similar pairs and farther for dissimilar pairs, resulting in a robust feature space.

Classification models map data directly to class labels, limiting their ability to generalize to unseen classes. In contrast, metric learning creates a transferable embedding space, allowing the model to effectively handle unseen data by comparing feature similarities.

Metric learning is inherently better at handling class imbalances because it evaluates pairwise similarities, reducing the bias towards dominant classes.

4.2.2. Result-Based Justification

The Siamese networks outperform the classification model significantly in terms of AUC (0.85 and 0.87 vs. 0.74). This indicates that the ability to learn pairwise relationships leads to better performance in distinguishing between classes.

The higher AUC of the Cosine Distance Loss Siamese model suggests that capturing angular relationships in feature space provides an advantage over the traditional Euclidean metric.

The classification model's lower AUC suggests it struggles with the inherent complexity or noise in the dataset. Metric learning, by focusing on relative comparisons, is more robust to such challenges.

4.2.3. Conclusion

Overall, metric learning, through Siamese networks, is chosen over classification for both theoretical and practical reasons. Theoretically, metric learning excels at building a generalizable embedding space that captures nuanced relationships between data points. Practically, the results demonstrate that Siamese networks, especially with Cosine Distance Loss, outperform the classification model, making them a superior choice for tasks where relational understanding between samples is crucial.

5. Reference

- Link to the source code:
<https://drive.google.com/drive/folders/1KBk0lxDh4Hc7VA-VQtX0yYmxtJ7cH6re?usp=sharing>
- Link to the demo video:
<https://drive.google.com/file/d/1OY0-NRCUwW9YEaAvIEvSMb0dQI0g6O7S/view?usp=sharing>
- Link to Kaggle dataset:
<https://www.kaggle.com/c/11-785-fall-20-homework-2-part-2/overview/evaluation>
- Soukupová and Čech, 2016, "Real-Time Eye Blink Detection Using Facial Landmarks",
<https://vision.fe.uni-lj.si/cvww2016/proceedings/papers/05.pdf>
- Hazem Essam and Santiago L. Valdarrama, 2015, "Image similarity estimation using a Siamese Network with a triplet loss": https://keras.io/examples/vision/siamese_network/
- Shivam Chandhok, 2023, "Triplet Loss with Keras and TensorFlow":
<https://pyimagesearch.com/2023/03/06/triplet-loss-with-keras-and-tensorflow/>
- Geeksforgeeks, 2024, "AUC ROC Curve in Machine Learning":
<https://www.geeksforgeeks.org/auc-roc-curve/>