1. Introduction

This project focuses on building an efficient face verification system using a Siamese neural network model and TensorRT for enhanced performance. The goal of this system is to confirm or authenticate a person's identity based on facial images, thus promoting financial inclusion through reliable and scalable technology.

Facial analysis encompasses various tasks, each serving a distinct purpose. These tasks include face verification, face detection, and face recognition. Understanding the differences between these tasks is essential for comprehending the scope and application of face verification systems:

- **Face Verification:**
  Face verification is the process of confirming or authenticating an individual's identity by comparing two facial images. The system answers the question, "Are these two images of the same person?" This is particularly useful in scenarios where security and identity validation are critical, such as in financial services, access control, and authentication systems.
- **Face Detection:**
  Face detection involves identifying and locating all the faces within an image. It does not determine the identity of the faces but simply finds where the faces are. This process is foundational for both face verification and face recognition as it isolates facial regions from the background for further processing.
- **Face Recognition:**
  Face recognition goes a step further than face verification by not only determining whether two images are of the same person but also identifying the individual in the image from a known database of faces. It answers the question, "Who is this person?" This technology is widely used in security systems, surveillance, and user authentication.

This project leverages modern deep learning techniques to optimize the performance of the face verification model, ensuring rapid and accurate identity verification while maintaining low latency, which is crucial for real-world applications.

2. LFW Dataset

The dataset used for this project is the Labeled Faces in the Wild (LFW), a well-known database designed for studying unconstrained face recognition. Created by researchers at the University of Massachusetts, Amherst, LFW contains 13,233 images of 5,749 people, collected from the web and processed using the Viola-Jones face detector to ensure each face is centered and uniformly sized. Among the individuals pictured, 1,680 have two or more distinct photos.

Context

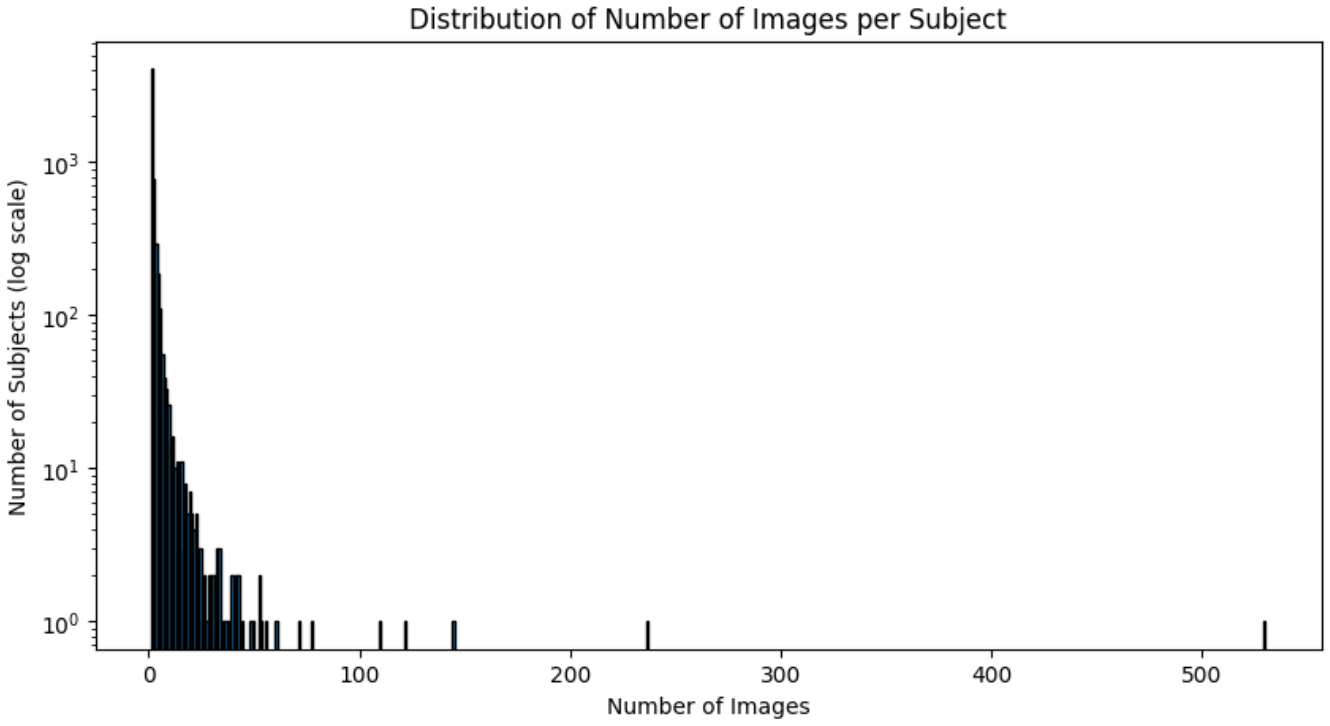
LFW provides a challenging dataset for face recognition and verification due to the variability in lighting, facial expressions, and image quality. The images are categorized into different sets, including the "deep-funneled" version, which has shown superior results for most face verification algorithms. This project utilizes the deep-funneled images to ensure high performance.

Image Information

- **Format:** Each image is available as "lfw-dataset/lfw-deepfunneled/name/name_xxxx.jpg", where "xxxx" is the image number padded to four characters. For example, the 10th image of George W. Bush can be found at "lfw/George_W_Bush/George_W_Bush_0010.jpg".
- **Dimensions:** Each image is a 250x250 pixel JPG, centered and resized using the Viola-Jones face detector with an enlargement factor of 2.2 applied to capture more of the head.

Here are some sample images from the dataset:

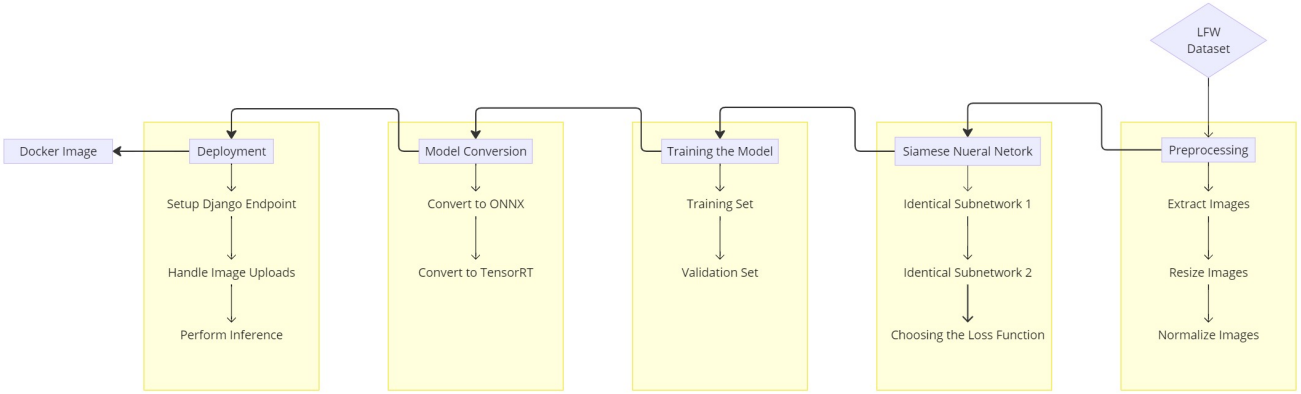
And below is a visualization of the distribution of image counts per subject:



It can clearly be seen that very few subjects have more than 100 images, while the majority have a single image.

3. System Architecture

The system architecture for this face verification project is designed to ensure high accuracy and efficiency, involving several key steps. Below is an overview of each step, from data loading to deploying the model using a Django endpoint.



The system architecture for the face verification task using the LFW dataset is illustrated in the diagram. The process begins with the preprocessing of the LFW dataset. This involves extracting images, resizing them, and normalizing them to ensure uniformity and optimal input for the model.

Following preprocessing, the images are fed into a Siamese Neural Network, which consists of two identical subnetworks. These subnetworks are designed to learn useful representations for the face verification task. The output of the subnetworks is used to compute a similarity score, with the choice of a suitable loss function being crucial for effective learning.

After training the Siamese Neural Network, the model undergoes a conversion process. Initially, it is converted to the ONNX format, enabling compatibility across different platforms. Further, it is converted to TensorRT for optimization and faster inference on compatible hardware.

The final step is deployment. A Django endpoint is set up to handle the image uploads and perform inference. This allows for a seamless interface for users to verify faces by uploading images and receiving verification results.

This architecture ensures a systematic approach, from data preparation to model training, conversion, and deployment, ultimately facilitating efficient face verification.

4. Methodology

In this section, we will delve into the methodology employed to build and optimize the face verification system using the restricted configuration of the LFW dataset and a Siamese neural network model. This includes detailed steps on data preprocessing, model architecture, training process, model optimization, and deployment strategy.

4.1 Data Loading and Preprocessing

Data Loading:

The process begins with defining the root directory for the LFW dataset, containing all face images. These images are retrieved, shuffled to ensure randomness, and then split into two sets: training and testing. The split ratio is **80%** for training and **20%** for testing, providing a robust dataset for both model training and evaluation.

Creating Image Pairs:

For the Siamese Neural Network, pairs of images are created for both training and testing. For the training set, **5000 pairs** are generated, equally divided between positive pairs (same person) and negative pairs (different people). Similarly, **1000 pairs** are created for the testing set. **Positive pairs** are formed by selecting an image and another of the same person, while **negative pairs** are created by selecting an image and another of a different person. These pairs are then saved to text files for easy loading during the training process.

Preprocessing:

Face alignment is performed using MTCNN (Multi-task Cascaded Convolutional Networks) to detect and align faces properly. The images are then transformed into tensors using a set of predefined transformations. For the training set, these transformations include resizing, random affine transformations, vertical flipping, and color jittering to introduce variability and enhance robustness. For the testing set, only resizing and conversion to tensors are performed to maintain consistency during evaluation.

Custom Dataset Class:

A custom dataset class is implemented to handle the loading and transformation of the image pairs. This class reads the pairs from the text files, applies the necessary transformations, and returns the processed images along with their labels.

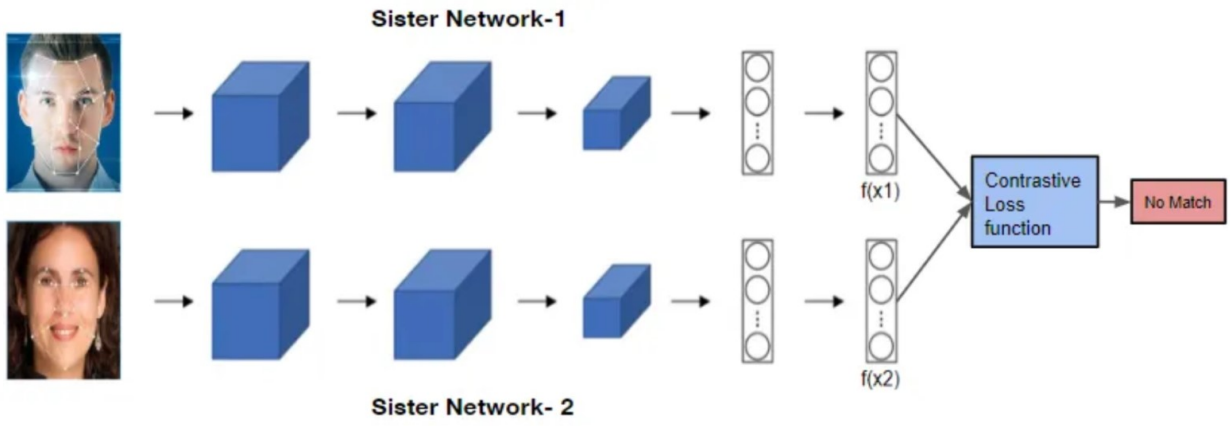
4.2 Model Architecture

The core of the face verification system is a Siamese neural network, which is particularly effective for tasks that involve comparing two inputs. The architecture of the model consists of the following components:

Base Network:

The **Base Network** serves as the foundation for the Siamese Network. It is composed of several layers designed to extract meaningful features from the input images.

- Convolutional Layers:
  - **Conv1:** 3 channels to 64 filters, 3x3 kernel.
  - **Conv2:** 64 filters to 128 filters, 3x3 kernel.
  - **Conv3:** 128 filters to 256 filters, 3x3 kernel.
- Pooling Layer:
  - Max pooling with a 2x2 kernel and a stride of 2, applied after each convolutional layer.
- Fully Connected Layers



Siamese Network:

The **Siamese Network** leverages the Base Network to create a system capable of comparing two input images.

- Structure:
  - Composed of two identical subnetworks (Base Networks) that share the same weights.
  - Each subnetwork processes one of the input images.
- Function:
  - Extracts feature vectors (embeddings) for each image.
  - The feature vectors are then compared to determine the similarity between the images.

4.3 Choosing the Loss Function

Triplet Loss:

Triplet loss is commonly used in face verification tasks due to its effectiveness in learning feature embeddings that minimize the distance between positive pairs (same person) and maximize the distance between negative pairs (different people). It achieves this by considering three images at a time: an anchor, a positive image (same person as the anchor), and a negative image (different person).

However, triplet loss requires multiple images of the same subject to form the triplets. Since the LFW dataset has many subjects with only a single image, it is infeasible to compute triplet loss on this dataset.

Contrastive Loss:

Given the constraints of the LFW dataset, contrastive loss is selected as a more suitable alternative. Contrastive loss works with pairs of images rather than triplets, making it applicable even when there are only single images per subject. The contrastive loss function is defined as:

$$L(Y, D) = (1 - Y)\frac{1}{2}D^2 + Y\frac{1}{2}\max(0, m - D)^2$$

where:

- L is the label indicating whether the pair of images are of the same person (0) or different people (1).
- D is the Euclidean distance between the feature vectors of the two images.
- m is the margin that defines the distance threshold for dissimilar pairs.

This loss function helps in pulling the feature vectors of similar images closer and pushing the feature vectors of dissimilar images apart, thus improving the model's ability to verify faces.

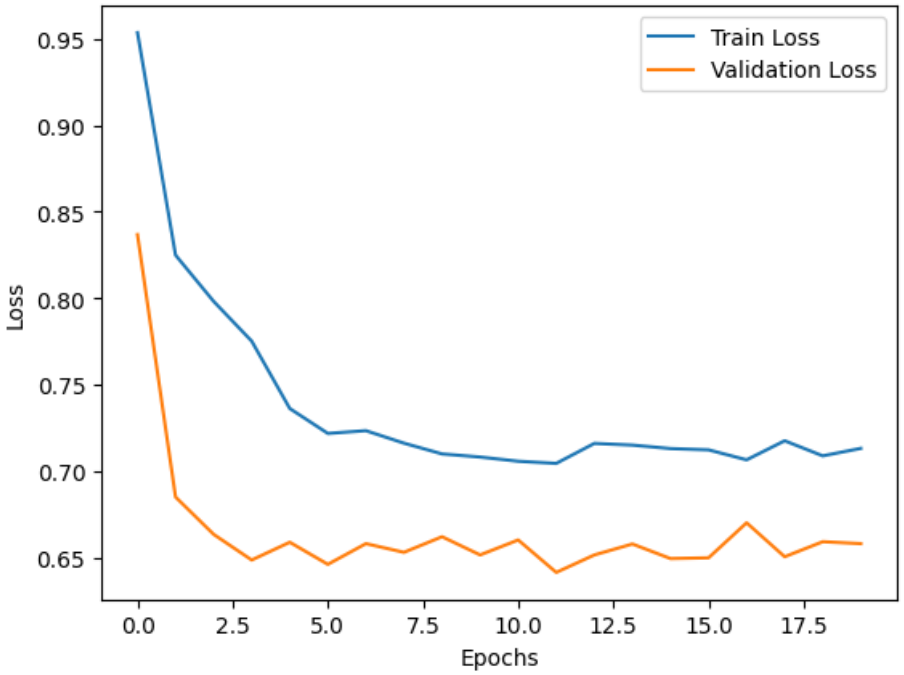
4.4 Training the Model

The training process involves the following steps:

1. **Batch Preparation:** Training is conducted in batches, with each batch containing a mix of matched and mismatched pairs. The batch size is set to 32.
2. **Hyperparameters:**
    - **Learning Rate:** 0.01
    - **Batch Size:** 32
    - **Epochs:** 20
    - **Logging Interval:** Every 50 batches
3. **Optimization:**
    - The model is optimized using the SGD (Stochastic Gradient Descent) optimizer.
    - A learning rate scheduler is used to adjust the learning rate every 4 epochs with a decay factor of 0.1.
4. **Environment:** The model training was conducted in a Google Colab environment with the following specifications: T4 GPU and 52 GB RAM.

4.5 Model Evaluation and Results

The best model, saved during the training process, is loaded for evaluation. The following graph shows how loss decreased over epochs.



Evaluation Metrics:

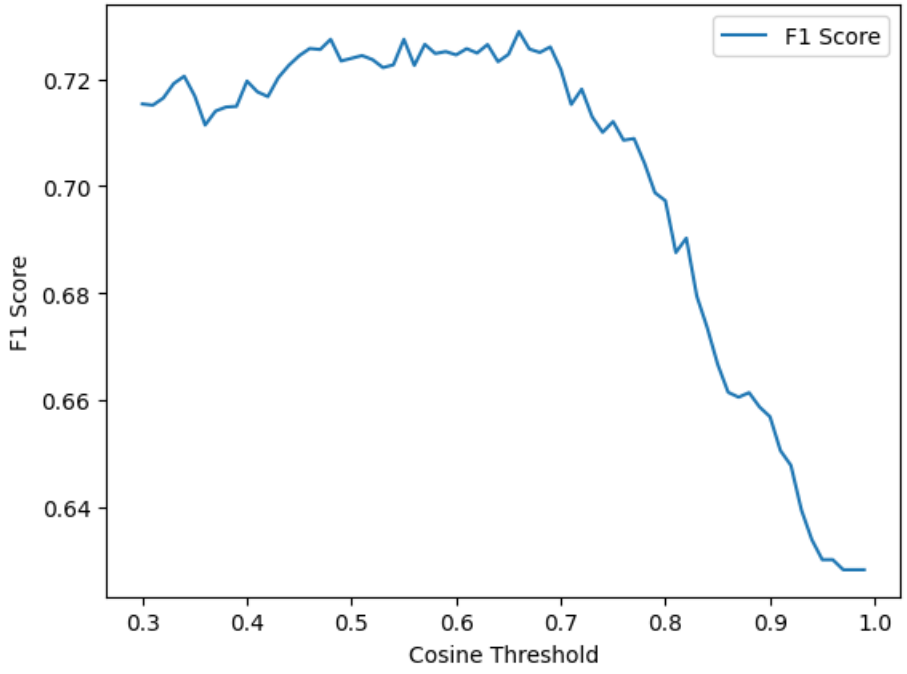
The model is evaluated on the test set using cosine similarity and Euclidean distances between image pairs. The evaluation metrics include:

- **Cosine Similarity:** Measures the cosine of the angle between two vectors, indicating similarity.
- **Euclidean Distance:** Calculates the straight-line distance between two vectors, indicating dissimilarity.

Threshold Analysis:

Different thresholds are tested to determine the best threshold for cosine similarity that maximizes the F1 score.

- **Thresholds:** Range from 0.3 to 1.0, with increments of 0.01.
- **F1 Score:** Calculated at each threshold to evaluate the balance between precision and recall.



At threshold 0.64, we get a maximum F1 score of **0.7289**

4.5 Model Conversion and Optimization

To achieve real-time performance, the trained model is converted to the ONNX format. Although the goal was to also optimize the model using TensorRT, this step was unsuccessful due to the lack of GPU support on the local machine and compatibility issues in the Colab environment.

Conversion to ONNX:

1. **Model Export:** The trained PyTorch model is exported to the ONNX format. This intermediate representation is essential for facilitating further optimization and deployment across different platforms.
2. **Attempted Optimization:** The ONNX model was intended to be converted to a TensorRT engine to leverage GPU acceleration. TensorRT performs various optimizations such as layer fusion, precision calibration, and kernel auto-tuning to achieve low-latency inference.

4.6 Deployment

This section provides an overview of deploying an ONNX model using Django, focusing on setting up the Django project, creating an endpoint for model inference, and containerizing the application with Docker. Users can interact with the deployed model using tools like `curl` or Postman.

Set Up the Django Project

- **Install Dependencies:** Install necessary libraries including Django, ONNX Runtime, Pillow, Torch, and Torchvision.
- **Create Project and App:** Initialize a new Django project and create a new app within it.
- **Configure Settings:** Update the Django settings to include the newly created app.

Create Model Inference Logic

- **Load ONNX Model:** Use ONNX Runtime to load the pre-trained model.
- **Preprocess Input:** Implement preprocessing steps to ensure input images match the model's expected format.
- **Define Inference Function:** Create a function to run the model inference and compute similarity scores between images.

Develop the API Endpoint

- **Define View:** Create a Django view to handle HTTP POST requests, process uploaded images, run model inference, and return similarity scores.
- **Configure URLs:** Map the view to a specific URL in the project's URL configuration.

4.7 Creating a Docker Image

To containerize the Django project and ensure a consistent runtime environment, Docker was used to create an image of the project. Here are the steps that were followed:

1. **Dockerfile Creation**: A `Dockerfile` was created in the root directory of the project. This file contains instructions on how to build the Docker image.
2. **Building the Docker Image**: The Docker image was built using the `docker build` command.
3. **Running the Docker Container**: Once the image was built, a container was started using the `docker run` command.