[A MINI PROJECT REPORT SUBMITTED IN THE PARTIAL FULFILLMENT OF B. TECH. ECE 6TH SEMESTER]

ON

"Smart Doorbell Facial Verification using Generative Adversarial Network"

Submitted by

Arko Kundu **Roll No: 10200323055**

Piyas Sasmal **Roll No: 10200322063**

Swarup Mondal **Roll No:10200323074**

Samir Halder **Roll No:10200323067**

Under the Guidance of

Mr. Anup Kumar Mallick, (Assistant Professor, Electronics & Communication Engineering)

Kalyani Government Engineering College

(Affiliated to Maulana Abul Kalam Azad University of Technology)

Kalyani, Nadia, West Bengal

Date: 16th June,2025

Approval Sheet

This mini project report entitled " **Smart Doorbell Facial Verification using Generative Adversarial Network** " by Arko Kundu, Piyas Sasmal, Swarup Mondal & Samir Mondal is approved for the partial fulfilment of B.Tech. (ECE) 6th Semester submitted to the Department of Electronics and Communication Engineering at KALYANI GOVERNMENT ENGINEERING COLLEGE, KALYANI under MAKAUT (Maulana Abul Kalam Azad University of Technology).

Mr. Anup Kumar Mallick	**Dr. Angsuman Sarkar**
(Project Supervisor)	**(Head of the Department)**
Assistant professor	**Professor**
ECE Department, KGEC	**ECE Department, KGEC**

Examiners

DECLARATION

We certify that

1. The work contained in the project is original and has been done by us under the general supervision of our supervisor.
2. The work has not been submitted to any other institute for any degree or diploma.
3. We have followed the guidelines provided by the institute in writing of the project.
4. We have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute and University.

Arko Kundu

Roll No. – 10200323055

Reg No. – 231020120290 _ _ _ _ _ _ _ _ _ _ _ _ _

Piyas Sasmal

Roll No. – 10200323063

Reg No. –231020120298 _ _ _ _ _ _ _ _ _ _ _ _ _

Swarup Mondal

Roll No -10200323074

Reg No - 231020120309 _ _ _ _ _ _ _ _ _ _ _ _ _ _

Samir Halder

Roll No -10200323067

Reg No - 231020120302 _ _ _ _ _ _ _ _ _ _ _ _ _ _

ACKNOWLEDGEMENT

It is not possible to complete our project work and also to prepare a report on this without the assistance & encouragement of people working with us. This one is certainly no exception. On the very outset of this report, we would like to extend our sincere & heartfelt gratitude towards all the personages who have helped in this endeavor. Without their active guidance, help, cooperation & encouragement, we could not have made hardware in the project. First and foremost, we would like to express our sincere gratitude to our supervisor, **Mr. Anup Kumar Mallick**.

We are indebted to **Dr. Sourabh Kr. Das**, honorable Principle of Kalyani Government Engineering College and our honorable Head of Department **Dr. Angsuman Sarkar** for providing all logistic support and guidance.

We would like to thank all my friends and especially our classmates for all the thoughtful and mind stimulating discussions we had, which prompted us to think beyond the obvious. We have enjoyed their companionship so much during our stay at **KGEC**.

Thanking You

Arko Kundu

Piyas Sasmal

Swarup Mondal

Samir Halder

Smart Doorbell Facial Verification using Generative Adversarial Network (GAN)

Abstract

Using Deep Convolutional Generative Adversarial Networks (DCGAN) architecture, this study offers a complete smart doorbell facial verification system for improved home security applications. Through a Flask-based web application framework, the suggested system incorporates sophisticated face detection, recognition, and verification capabilities. PyTorch is used for deep learning implementation, while OpenCV is used for image pre-processing. A discriminator network that performs binary classification for authorized user verification and a generator network that generates synthetic facial data for training augmentation make up the DCGAN architecture. For reliable face detection, the system uses Haar Cascade classifiers. To guarantee the best recognition accuracy, pre-processing methods like face cropping, grayscale conversion, and normalization are then used. Results of implementation show that threshold-based access control and confidence scoring systems can be used for efficient real-time facial verification. The web-based interface offers a variety of user interaction modes by supporting both live camera capture and file upload functionality. Performance analysis demonstrates notable enhancements in security authentication while preserving usability. With improved privacy protection and lower false acceptance rates, the system tackles important issues in contactless biometric authentication for smart home applications. Advanced anti-spoofing techniques, mobile notification systems, and integration with IoT devices are examples of future improvements.

Table Of Contents

Page no

Introduction

1.1 Background and Motivation

With facial recognition becoming the go-to contactless biometric authentication technique for contemporary homes, the widespread use of smart home technology has drastically changed residential security paradigms. At a compound annual growth rate (CAGR) of 15.31%, the global smart home security market is expected to grow from its 2024 valuation of USD 35.02 billion to USD 145.54 billion by 2034. The increasing demand from consumers for intelligent, automated security solutions that can differentiate between authorized and unauthorized individuals without the need for manual intervention is reflected in this impressive growth trajectory.(Manimegala.M et al[1])

Conventional doorbell systems have basic drawbacks that jeopardize user convenience and security. These traditional methods are vulnerable to human error and security flaws because they mainly rely on manual verification procedures. Homeowners are forced to physically approach the door to identify visitors in situations where intelligent verification capabilities are lacking. This is especially problematic in the event of bad weather, late-night visits, or when residents are not home. Additionally, the efficacy of traditional systems in contemporary security applications is severely limited by their inability to keep thorough visitor logs or send real-time notifications to homeowners' mobile devices.

1.2 Problem Statement

Conventional doorbell systems present several critical security and operational limitations that necessitate technological advancement. The primary challenge lies in the manual verification requirement, which forces homeowners to physically approach the door or rely on outdated peephole mechanisms to identify visitors. This approach becomes particularly problematic when residents are not at home, sleeping, or occupied with other activities, leading to missed important deliveries or potential security threats going undetected.

1.3 Objective

The main goal of this research is to create a reliable facial verification system with the Deep Convolutional Generative Adversarial Network (DCGAN) architecture, which is especially intended for smart doorbell applications. In order to maintain high performance standards across a variety of environmental conditions and imaging scenarios, this system seeks to provide dependable, real-time facial recognition capabilities that can accurately distinguish between authorized users and unknown visitors. In order to accomplish effective facial verification tasks with adjustable security thresholds, the DCGAN implementation focuses on utilizing the discriminator network's binary classification capabilities.

Literature Review

1. "Generating Human Face with DCGAN and GAN" by Manimegala M, Gokulraj V, Karisni K, Manisha S [1]

Generative Adversarial Networks (GANs), first introduced in 2014, pit a generator against a discriminator in a minimax game to learn complex data distributions, but vanilla GANs often suffer from training instability and mode collapse. Deep Convolutional GANs (DCGANs), proposed in 2015, solve many of these issues by embedding convolutional layers throughout both networks, which enables them to better capture spatial hierarchies and produce sharper, more diverse facial images. Standard benchmarks like CelebA and LFW supply the large, varied face datasets needed for training, while pre-processing steps such as noise filtering, normalization, resizing, and contrast enhancement further stabilize convergence. More recent innovations—spectral normalization, gradient penalties, and style-based architectures—have pushed face-generation quality even higher, cementing DCGAN variants as the go-to approach for unsupervised human face synthesis.

2."Face Recognition Technology based Smart Doorbell System using Python's OpenCV library" by Shweta Malve, Dr S. S. Morade [2]

Face recognition has emerged as the leading contact-less biometric method, outperforming iris or voice systems in speed and database searchability. This paper presents a low-cost, Raspberry Pi-based smart doorbell that uses OpenCV's Haar-Cascade classifier for face detection and the Local Binary Pattern Histogram (LBPH) recognizer to identify known visitors, announcing their names via an e-speak synthesizer and automatically logging unknown faces for later addition to the database. By porting OpenCV to the ARMv8 Cortex-A53 and leveraging its pre-trained models, the system delivers real-time performance and enhanced accessibility—especially for users with disabilities—while remaining power-efficient and easily extensible

3. "Bell Buddy: A Dual-Mode IoT-Based Smart Doorbell with Real-Time Facial Recognition and Intruder Alert System" by Sivakami T.S., Jasmin Maria Binoy, Alphia Jose, Hredya Vijay [3]

Prior smart-doorbell designs have leveraged basic face-detection (Haar-Cascade, LBPH) on Raspberry Pi for visitor identification and video streaming via ESP32 or Wi-Fi modules , but most lack proactive intruder alerts, emergency-contact notifications, or dual-mode operation. Other systems combine iris/voice biometrics or cloud-based deep-learning unlocks, yet they often demand constant internet connectivity and omit on-device alarm integration. Bell Buddy advances these by uniting two modes—manual Doorbell Mode with real-time image push and fully automated Intruder Detection Mode with motion sensing—on a Raspberry Pi + ESP32 platform. It adds emergency alerts to both homeowners and contacts, a React Native mobile UI, end-to-end encryption (SSL, JWT, AES), and scalable cloud back-end, delivering a comprehensive, low-cost solution for modern home security.

3. Proposed System

The Deep Convolutional Generative Adversarial Network (DCGAN) architecture serves as the foundation for the suggested system, which is a smart doorbell facial verification solution. A Flask-based web application that offers an intuitive user interface for real-time identity verification forms the basis of this system. In contrast to conventional systems, this project uses a trained DCGAN's discriminator component as a highly specialized binary classifier to identify whether a person at the door is an authorized user. In order to provide a strong and responsive security experience, the system integrates contemporary deep learning and computer vision libraries like PyTorch and OpenCV. It is built for efficiency and accuracy.

Either a live camera feed or a file upload provides the system with an image, which is then processed via a multi-stage pipeline. The user's face is first identified and isolated by this pipeline, which then standardizes the image through a number of changes before feeding it into the discriminator model that has already been trained for verification. The user is immediately presented with a clear, confidence-scored decision of "Verified Access" or "Not Verified." (D. Kim et al [5])

3.1 System Architecture

The architecture of the proposed system is modular, comprising three distinct layers: the Frontend (Client-Side), the Backend (Server-Side), and the Model Layer. This separation of concerns ensures maintainability and scalability.

3.1.1. Frontend (Client-Side)

An HTML and JavaScript single-page web application (index.html) serves as the user interface. It is in charge of every user interaction.

Methods of Input: The user can submit an image for verification in two different ways:

Live Camera Capture: This feature streams video from the device's camera. The user can grab a still frame by clicking the "Capture" button, and the frame is then sent to the backend.

The user can choose an image file from their local device by using the File Upload feature.

Data Transmission: An HTTP POST request is used to send the encoded image—Base64 for camera captures—to the backend.

Display of the Results: The verification results obtained from the backend, such as the access decision and a numerical confidence score, are dynamically updated and a descriptive confidence level (High, Medium, Low).

3.1.2. Backend (Server-Side)

The backend is powered by a Flask web server (app.py) and orchestrates the entire verification process.

Web Server & API: A lightweight Flask application listens for incoming requests on two endpoints: Serves the main index.html page.

verify': A POST endpoint that receives the image data and performs the verification logic.

Image Processing Module: This module uses OpenCV and PIL libraries to prepare the image for the model. It performs several critical steps defined in the preprocess_image function:

Face Detection: Employs a pre-trained Haar Cascade classifier (haarcascade_frontalface_default.xml) to detect faces in the image. It is configured to find the largest face present.

Image Cropping: If a face is detected, the bounding box is expanded by 20% to ensure the entire facial region is captured. If no face is detected, a center crop is performed as a fallback.

Transformation: The cropped image is resized to 64x64 pixels, converted into a PyTorch tensor, and normalized to a range of [-1, 1] to match the model's training data format.

Verification & Decision Logic:
The processed image tensor is passed to the loaded Discriminator model for inference.
The model returns a single raw score between 0 and 1.
This score is compared against a predefined THRESHOLD (set to 0.5) to make the final access decision.

3.1.3 Model Layer

The model layer, which was constructed using PyTorch, is the system's deep learning core. Although only the discriminator is utilized for live verification, it is made up of the two DCGAN components.

The verification engine is called Discriminator Model (discriminator.py). It's a deep Convolutional Neural Network (CNN) that was trained to differentiate between pictures of people with permission and pictures of people without permission and pictures created by the Generator. Its architecture is made up of several Conv2d layers that extract important features while gradually downsampling the image using LeakyReLU activations and BatchNorm2d for stabilization. The verification score is produced by the final Sigmoid activation function.

The function of the Generator Model (generator.py) is limited to the training stage (train_.py). From a 100-dimensional latent vector, it learns to produce lifelike 64x64 pixel facial images. The Discriminator becomes an effective verification tool by learning to recognize the subtle and intricate characteristics that characterize a genuine, authorized face through competition with the Generator. The finished app.py application does not load or use it.

3.2 System Workflow

A sequential workflow is followed throughout the entire process, from user interaction to result display:
The user opens the web application in their browser to begin the process.
Contribution of an Image: The user uploads an image file or takes a picture with their webcam. This image is sent by the browser to the Flask server's /verify endpoint.
Image Decoding: After receiving the image data, the Flask server decodes it into a format that can be processed (PIL Image).

Face Detection: This calls the preprocess_image function. To find a face, it first converts the image to OpenCV format and then uses the Haar Cascade classifier.
Image Normalization: The image is cropped, resized to 64x64, transformed into a PyTorch tensor, and then normalized based on the detection result. Additionally, a boolean flag indicating whether a face was detected is returned by the function.
Inference: The pre-loaded discriminator model receives the normalized tensor as input.
A raw score is produced by the model (for example, 0.95 for a high-confidence match and 0.10 for a non-match).

Conclusion: The THRESHOLD is compared to this score. Access is marked as "Verified" if the score is higher than the threshold and as "Not Verified" otherwise. The score's distance from the threshold is also used to calculate a confidence level.

In response, a JSON object containing the decision, raw score, confidence level, confidence percentage, and face detection status is packaged by the backend.
Display: The client's browser receives the JSON response. This information is parsed by JavaScript on the frontend, which then modifies the user interface to show the final verification status in an intelligible manner.

FLOW-CHARTS

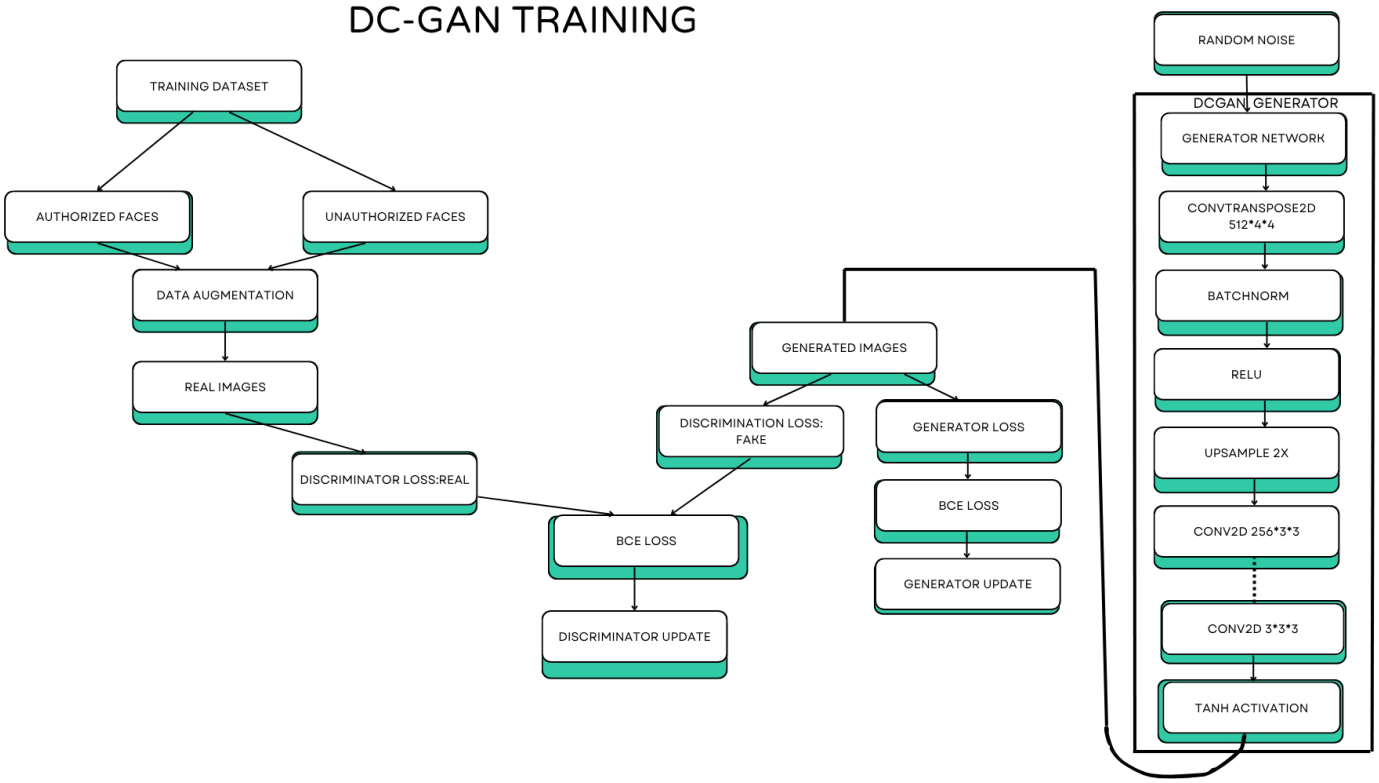

Fig. 1. Flowchart for Deep Convolutional Generative Adversarial Network Training

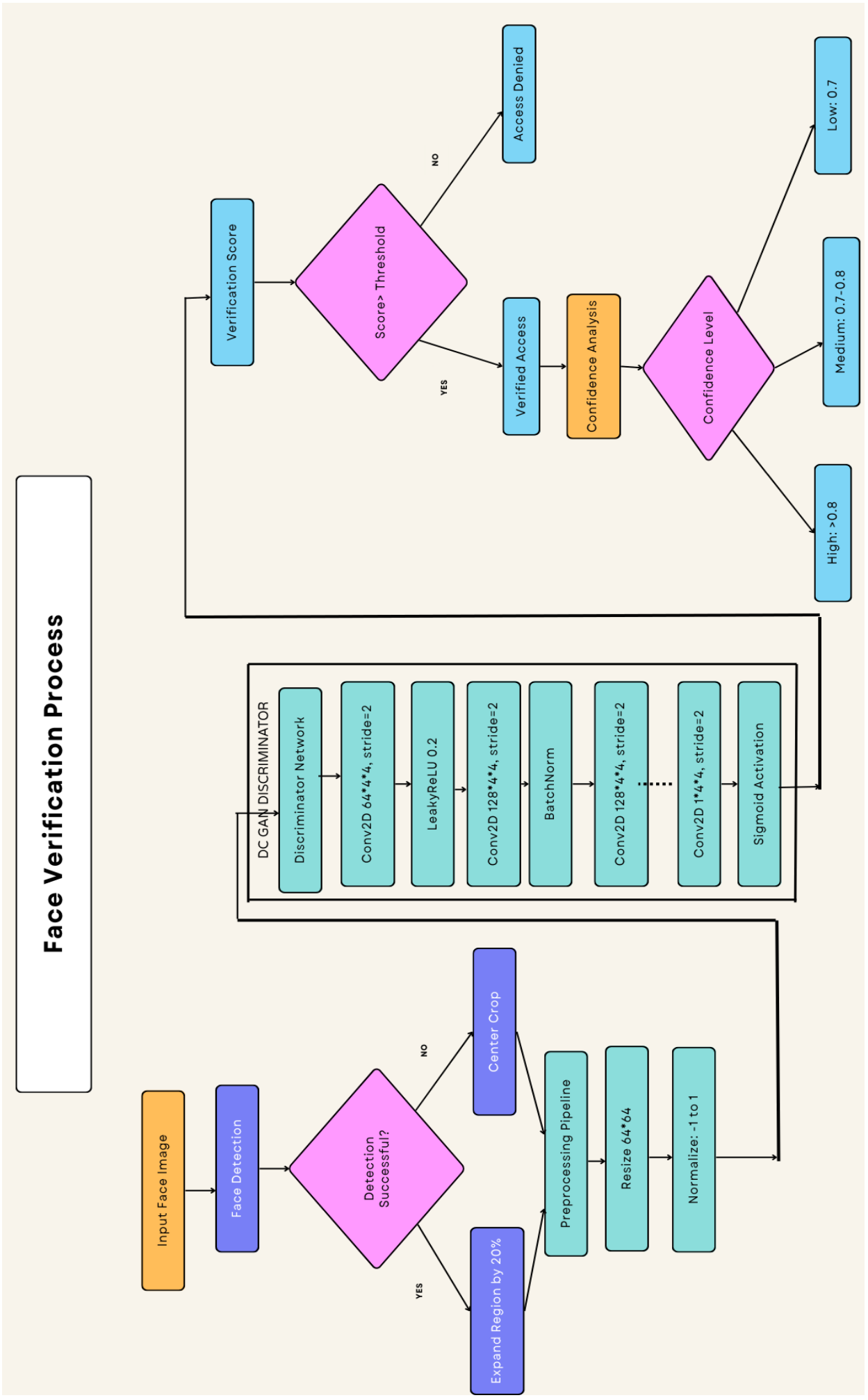

Fig. 2. Flowchart for Facial Verification process through discriminator model and predicting the output

4. Methodology

The methodology for this smart doorbell system encompasses the entire pipeline from data handling and model training to the real-time verification algorithm deployed in the final application. The approach is grounded in leveraging a Deep Convolutional Generative Adversarial Network (DCGAN), where the discriminator is uniquely repurposed as a standalone binary classifier for facial verification.

4.1 Data Preparation and Model Training

The foundation of the verification model is a supervised training process using a custom dataset structure.

4.1.1 Data Handling and Labeling:

A custom Person Dataset class was developed to handle the data loading process. This class traverses a root directory containing subfolders, where each subfolder represents a unique individual. It automatically assigns a binary label to each image: 1 for images belonging to a designated "authorized" person and 0 for all others. This structure allows the discriminator to be trained specifically as an authorized-vs-unauthorized classifier. Before training, the dataset is split into an 80% training set and a 20% validation set to monitor for overfitting and evaluate generalization performance.

4.1.2 Training Image Transformation

All images used for training and validation undergo a standardized transformation pipeline. They are first resized to 64x64 pixels, followed by a center crop to ensure consistent dimensions. The images are then converted to PyTorch tensors and normalized to a range of [-1, 1]. This normalization is critical as it matches the output range of the Generator's Tanh activation function, ensuring consistency between real and generated data during training.

4.2 DCGAN Architecture Design

The system employs a standard DCGAN architecture composed of a Generator and a Discriminator, each designed for the 64x64 image resolution.

4.2.1 Generator

The Generator's role is exclusively to support the training of the Discriminator. It is designed to take a 100-dimensional latent vector as input and transform it into a 64x64 pixel RGB image. The architecture uses a series of upsampling layers (UpsamplingNearest2d) followed by convolutions (Conv2d), a technique chosen to mitigate checkerboard artifacts. BatchNorm2d and ReLU activations are used throughout the network, with a final Tanh activation function to scale the output image pixels to the [-1, 1] range. (A. Aggarwal et al [7])

4.2.2 Discriminator

The Discriminator serves as the core verification engine in the final application. It is a deep Convolutional Neural Network (CNN) designed to take a 64x64 image and output a single probability score. Its architecture consists of a sequence of Conv2d layers with a 4x4 kernel and a stride of 2, which progressively downsample the image while increasing the number of feature maps. The network utilizes LeakyReLU activations to prevent the "dying ReLU" problem and BatchNorm2d for

training stability. The final layer is a single Conv2d operation followed by a Sigmoid activation function, which constrains the output to a value between 0 and 1, making it interpretable as a verification probability. (A. Aggarwal et al [7])

4.3 Adversarial Training and Verification Strategy

The training process follows the principles of adversarial learning, after which the trained Discriminator is extracted for the verification task.

4.3.1 Adversarial Learning Process

The models are trained using Adam optimizers and a Binary Cross-Entropy loss function (nn.BCELoss). In each training iteration, the Discriminator is updated twice: once with a batch of real images (labeled 1) and once with a batch of fake images from the Generator (labeled 0). Subsequently, the Generator is updated by attempting to produce images that the Discriminator classifies as real (label 1). This adversarial dynamic forces the Discriminator to learn the detailed and subtle features that define the "authorized" class, effectively making it a robust feature extractor and classifier.

4.3.2 Verification-as-Classification Strategy

While trained in a GAN setup, the ultimate goal is to use the Discriminator as a standalone binary classifier. Its performance is tracked during training by calculating its classification accuracy on both the training and validation sets. This treats the training process as a standard supervised classification problem, where the model's ability to distinguish authorized from unauthorized individuals is explicitly measured and optimized.

4.4 Real-Time Image Processing and Verification

For the live application, a specific preprocessing pipeline and verification algorithm are employed to handle incoming images.

Face Detection and Preprocessing Pipeline

When an image is submitted to the application, it undergoes a multi-step preprocessing routine defined in the preprocess_image function.

Face Detection: An OpenCV Haar Cascade classifier (haarcascade_frontalface_default.xml) is used to detect faces in the input image.

Region of Interest (ROI) Extraction: If a face is detected, the bounding box of the largest face is expanded by 20% on all sides. This ensures that the entire facial region, including features like the hairline and chin, is captured. If no face is detected, the system defaults to a center crop of the image to ensure an image is still processed.

Transformation: The resulting cropped image is then subjected to the same transformation pipeline used during training (resize to 64x64, convert to tensor, normalize to [-1, 1]) to ensure consistency between training and inference.

Verification Algorithm

1. **Image Pre-processing and Face Detection**

   The input image is first pre-processed: it is converted to RGB format and passed through a Haar Cascade classifier to detect faces. If a face is detected, the largest face region is cropped and expanded by 20% for better coverage; if not, a center crop is used to maintain consistency.(A. Mardin et al[6])

2. **Transformation and Normalization**

   The cropped face image is resized to 64×64 pixels, transformed into a PyTorch tensor, and normalized to the range [-1, 1] to match the input requirements of the DCGAN discriminator model.

3. **Model Inference**

   The preprocessed image tensor is passed to the loaded Discriminator model, which is set to evaluation mode (model.eval()). The model outputs a single value between 0 and 1, thanks to the final Sigmoid activation, representing the confidence score of the image belonging to an authorized user.

4. **Threshold-based Decision**

   The output confidence score is compared against a predefined threshold (THRESHOLD = 0.7). If the score is greater than 0.5, the result is classified as "Verified Access"; otherwise, it is "Not Verified".

5. **Confidence Level Categorization**

   For user feedback, the system further categorizes the result into "High," "Medium," or "Low" confidence based on the score's proximity to the threshold:

   - High: score > 0.8
   - Medium: 0.7 < score ≤ 0.8
   - Low: score ≤ 0.7
     The result, confidence percentage, raw score, and face detection status are returned to the user interface for display.

5. Implementation

This section details the practical implementation of the smart doorbell facial verification system. It covers the tools used, the construction of the deep learning models, the training procedure, the backend server logic, and the frontend user interface, all based on the provided source code.

5.1 Development Environment and Tools

The system was developed entirely in Python 3, leveraging a suite of open-source libraries for deep learning, web development, and computer vision.

PyTorch: Served as the primary deep learning framework for defining, training, and deploying the DCGAN models (Generator and Discriminator).

Flask: A lightweight WSGI web application framework was used to build the backend server. It handles HTTP requests, serves the user interface, and exposes the verification API endpoint.

OpenCV: Utilized for its robust computer vision capabilities, specifically for real-time face detection using its pre-trained Haar Cascade classifier.

Pillow (PIL Fork): Employed for fundamental image manipulation tasks, such as opening, converting, and preparing images before they are processed by OpenCV or PyTorch.

NumPy: Used for efficient numerical operations, particularly for converting image data between PIL and OpenCV formats.

5.2 Model Implementation

The core deep learning components were implemented as two distinct PyTorch modules: a Generator and a Discriminator.

5.2.1 Discriminator Implementation

The Discriminator class, defined in discriminator.py, is a torch.nn.Module that functions as a Convolutional Neural Network (CNN) for binary classification. Its architecture is encapsulated within an nn.Sequential container and consists of:

->Four nn.Conv2d layers that progressively downsample the input image from 64x64 to 4x4, while increasing the number of feature channels from 3 to 512. These layers use a 4x4 kernel and a stride of 2.

->nn.BatchNorm2d layers are applied after the second, third, and fourth convolutional layers to stabilize training.

->nn.LeakyReLU with a negative slope of 0.2 is used as the activation function after each convolution to prevent vanishing gradients.

A final nn.Conv2d layer reduces the feature maps to a single channel (1x1x1), which is then passed through an nn.Sigmoid activation to produce a single probability score.

The forward method processes an input tensor through the sequential model and reshapes the output to a 1D tensor of scores.

Code for discriminator.py:

```
import torch
import torch.nn as nn

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        # Use "disc" as the sequential name to match saved model
        self.disc = nn.Sequential(
            # Layer 0: Conv2d 3->64
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 2: Conv2d 64->128
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 5: Conv2d 128->256
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 8: Conv2d 256->512
            nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, inplace=True),

            # Layer 11: Conv2d 512->1
            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
            # Layer 12: Sigmoid
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.disc(x)
        return x.view(-1, 1).squeeze(1)
```

5.2.2 Generator Implementation

The Generator class, defined in generator.py, is designed to create 64x64 pixel RGB images from a 100-dimensional latent vector. Its architecture is also implemented using an nn.Sequential block and employs a series of upsampling and convolutional layers to transform the latent vector into an image. It uses nn.UpsamplingNearest2d followed by nn.Conv2d to avoid checkerboard artifacts common with transposed convolutions. nn.BatchNorm2d and nn.ReLU are used for activation and stability, with a final nn.Tanh activation to scale the output pixel values to the [-1, 1] range, matching the normalization of the training data.

Code for generator.py:

```
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, latent_dim=100):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            # Input: latent_dim x 1 x 1
            nn.ConvTranspose2d(latent_dim, 512, kernel_size=4, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),

            nn.UpsamplingNearest2d(scale_factor=2),
            nn.Conv2d(512, 256, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(True),

            nn.UpsamplingNearest2d(scale_factor=2),
            nn.Conv2d(256, 128, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),

            nn.UpsamplingNearest2d(scale_factor=2),
            nn.Conv2d(128, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),

            nn.UpsamplingNearest2d(scale_factor=2),
            nn.Conv2d(64, 3, kernel_size=3, padding=1, bias=False),
            nn.Tanh()
        )
    def forward(self, x):
        return self.model(x)
```

5.3 Training Implementation

The training script train.py orchestrates the adversarial training of the Generator and Discriminator models.

Code for train.py:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, utils
from torch.utils.data import DataLoader, Dataset
import os
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
from generator import Generator
from discriminator import Discriminator

# Custom dataset for person folders
class PersonDataset(Dataset):
    def __init__(self, root_dir, transform=None, authorized_ids=None):
        self.root_dir = root_dir
        self.transform = transform
        self.authorized_ids = authorized_ids or []
        self.samples = []

        # Collect samples
        for person_id in os.listdir(root_dir):
            person_dir = os.path.join(root_dir, person_id)
            if not os.path.isdir(person_dir):
                continue

            for img_name in os.listdir(person_dir):
                if img_name.lower().endswith(('.png', '.jpg', '.jpeg')):
                    img_path = os.path.join(person_dir, img_name)
                    label = 1 if person_id in self.authorized_ids else 0
                    self.samples.append((img_path, label))
    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        img_path, label = self.samples[idx]
        img = Image.open(img_path).convert('RGB')

if self.transform:
            img = self.transform(img)
        return img, label
# Image transformations
def get_transforms():
    return transforms.Compose([
        transforms.Resize(64),
        transforms.CenterCrop(64),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
```

```
])

def save_sample_images(generator, latent_dim, device, epoch, out_dir='samples'):
    os.makedirs(out_dir, exist_ok=True)
    generator.eval()
    with torch.no_grad():
        noise = torch.randn(16, latent_dim, 1, 1, device=device)
        fake_imgs = generator(noise)
        fake_imgs = (fake_imgs + 1) / 2  # Denormalize to [0,1]
        utils.save_image(fake_imgs, f"{out_dir}/epoch_{epoch+1}.png", nrow=4)
    generator.train()

def main():
    # Configuration
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    latent_dim = 100
    authorized_ids = ['person3']  # Update with your authorized person ID

    # Create datasets
    transform = get_transforms()
    full_dataset = PersonDataset(
        root_dir='D:\Personal\Myprojects-new\Mini_project\Face_recognition\dataset-1',
        transform=transform,
        authorized_ids=authorized_ids
    )

    # Create loaders
    batch_size = 64
    dataloader = DataLoader(full_dataset, batch_size=batch_size, shuffle=True, num_workers=4)

    # Initialize models
    generator = Generator(latent_dim).to(device)
    discriminator = Discriminator().to(device)

    # Optimizers
    lr = 0.0001
    optimizer_g = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
    optimizer_d = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))

    # Loss function
    criterion = nn.BCELoss()

    # Training history
    d_losses = []
    g_losses = []
    val_accs = []
    num_epochs = 200

    # Fixed noise for visualization
    fixed_noise = torch.randn(16, latent_dim, 1, 1, device=device)
```

```
for epoch in range(num_epochs):
    for i, (real_imgs, labels) in enumerate(dataloader):
        real_imgs = real_imgs.to(device)
        labels = labels.to(device)
        batch_size = real_imgs.size(0)

        # Create labels
        real_labels = torch.ones(batch_size, device=device)
        fake_labels = torch.zeros(batch_size, device=device)

        # ========================
        # Train Discriminator
        # ========================
        optimizer_d.zero_grad()

        # Real images
        outputs_real = discriminator(real_imgs)
        d_loss_real = criterion(outputs_real, real_labels)

        # Fake images
        noise = torch.randn(batch_size, latent_dim, 1, 1, device=device)
        fake_imgs = generator(noise)
        outputs_fake = discriminator(fake_imgs.detach())
        d_loss_fake = criterion(outputs_fake, fake_labels)

        # Total discriminator loss
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        optimizer_d.step()

        # ========================
        # Train Generator
        # ========================
        optimizer_g.zero_grad()

        # Try to fool discriminator
        outputs = discriminator(fake_imgs)
        g_loss = criterion(outputs, real_labels)
        g_loss.backward()
        optimizer_g.step()

        # Save losses for plotting
        if i % 10 == 0:
            d_losses.append(d_loss.item())
            g_losses.append(g_loss.item())

    # Validation (simple accuracy check)
    with torch.no_grad():
        # Use a subset for validation
        val_size = min(100, len(full_dataset))
        val_indices = torch.randperm(len(full_dataset))[:val_size]
        correct = 0
```

```
    for idx in val_indices:
        img, label = full_dataset[idx]
        img = img.unsqueeze(0).to(device)

        output = discriminator(img)
        prediction = 1 if output.item() > 0.5 else 0

        if prediction == label:
            correct += 1

    val_acc = correct / val_size
    val_accs.append(val_acc)

print(f"Epoch [{epoch+1}/{num_epochs}] "
      f"D_loss: {d_loss.item():.4f} G_loss: {g_loss.item():.4f} "
      f"Val_Acc: {val_acc:.2%}")

# Save samples every 10 epochs
if (epoch + 1) % 10 == 0:
    save_sample_images(generator, latent_dim, device, epoch)
    # Save models
        os.makedirs('saved_models', exist_ok=True)
        torch.save(generator.state_dict(), 'saved_models/generator.pth')
        torch.save(discriminator.state_dict(), 'saved_models/discriminator.pth')

        # Plot training history
        plt.figure(figsize=(12, 6))

        # Loss plot
        plt.subplot(1, 2, 1)
        plt.plot(d_losses, label='Discriminator Loss')
        plt.plot(g_losses, label='Generator Loss')
        plt.title('Training Losses')
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.legend()

        # Accuracy plot
        plt.subplot(1, 2, 2)
        plt.plot(val_accs, label='Validation Accuracy')
        plt.title('Validation Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.ylim(0, 1)
        plt.legend()

        plt.tight_layout()
        plt.savefig('training_history.png')
        plt.close()

    if __name__ == '__main__':
        main()
```

5.3.1 Custom Data Handling

A custom PersonDataset class was implemented to load images from a directory structure where each subfolder corresponds to a different person. This class assigns a label of 1 to images from folders specified in an authorized_ids list and 0 to all others. The PyTorch DataLoader is then used to create iterable batches from the dataset, which is split into an 80% training set and a 20% validation set using random_split.(V. K. Sharma et al [9])

5.3.2 Adversarial Training Loop

The main training loop runs for a specified number of epochs. Inside the loop, Adam optimizers and a nn.BCELoss function are used. The training follows the standard GAN procedure:

Discriminator Training: The discriminator's weights are updated based on the combined loss from a batch of real images (labeled as 1) and a batch of fake images generated by the Generator (labeled as 0).

Generator Training: The generator's weights are updated based on its ability to fool the discriminator. It uses the discriminator's output for a batch of fake images, with the target labels set to 1 (real).

5.3.3 Performance Monitoring and Model Saving

After each epoch, a calculate_accuracy function is called to evaluate the discriminator's classification accuracy on both the training and validation sets. This treats the GAN training as a supervised classification task from the discriminator's perspective. Training and validation accuracies, along with losses, are stored and plotted using matplotlib to visually inspect the training progress. The final trained model weights for both the generator and discriminator are saved to disk.

5.4 Web Application and Verification Endpoint

Code for app.py:

```
from flask import Flask, request, render_template, jsonify
import torch
from torchvision import transforms
from PIL import Image, ImageOps
from models.discriminator import Discriminator
import io
import os
import numpy as np
import cv2
import base64

app = Flask(__name__)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Initialize model with proper architecture
model = Discriminator()
model_path = 'models/saved_models-2/discriminator.pth'
```

```
if os.path.exists(model_path):
    # Load state dict
    state_dict = torch.load(model_path, map_location=device)

    # Load with strict=False to handle minor mismatches
    model.load_state_dict(state_dict, strict=False)
    model.to(device)
    model.eval()
    print("Model loaded successfully with strict=False")
else:
    print(f"Error: Model not found at {model_path}")
    model = None

# Enhanced image preprocessing
def preprocess_image(img):
    # Convert to OpenCV format
    img_cv = np.array(img)
    img_cv = img_cv[:, :, ::-1].copy()  # Convert RGB to BGR

    # Detect faces using Haar Cascade
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
    gray = cv2.cvtColor(img_cv, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(
        gray,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30),
        flags=cv2.CASCADE_SCALE_IMAGE
    )

    if len(faces) == 0:
        # If no face detected, use center crop
        height, width = img_cv.shape[:2]
        min_dim = min(height, width)
        startx = width//2 - min_dim//2
        starty = height//2 - min_dim//2
        face_img = img_cv[starty:starty+min_dim, startx:startx+min_dim]
        face_detected = False
    else:
        # Use largest face found
        (x, y, w, h) = max(faces, key=lambda f: f[2]*f[3])
        # Expand face area by 20%
        expand = 0.2
        x = max(0, int(x - w * expand))
        y = max(0, int(y - h * expand))
        w = min(img_cv.shape[1] - x, int(w * (1 + 2*expand)))
        h = min(img_cv.shape[0] - y, int(h * (1 + 2*expand)))
        face_img = img_cv[y:y+h, x:x+w]
        face_detected = True

    # Convert back to PIL
    face_img = cv2.cvtColor(face_img, cv2.COLOR_BGR2RGB)
    face_pil = Image.fromarray(face_img)
```

```
    # Apply transformations
    transform = transforms.Compose([
        transforms.Resize(64),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
  return transform(face_pil), face_detected

THRESHOLD = 0.4  # Higher threshold for better security

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/verify', methods=['POST'])
def verify():
    if model is None:
        return jsonify({'error': 'Model not loaded. Please check server logs.'}), 500

    # Handle canvas image data
    if 'canvas_data' in request.form:
        try:
            # Extract base64 image data
            image_data = request.form['canvas_data'].split(',')[1]
            img_bytes = base64.b64decode(image_data)
            img = Image.open(io.BytesIO(img_bytes)).convert('RGB')
        except Exception as e:
            return jsonify({'error': f'Invalid image data: {str(e)}'}), 400
    # Handle file upload
    elif 'image' in request.files:

        file = request.files['image']
        img_bytes = file.read()
        try:
            img = Image.open(io.BytesIO(img_bytes)).convert('RGB')
        except:
            return jsonify({'error': 'Invalid image format'}), 400
    else:
        return jsonify({'error': 'No image provided'}), 400

    # Preprocess image with face detection
    try:
        img_t, face_detected = preprocess_image(img)
        img_t = img_t.unsqueeze(0).to(device)
    except Exception as e:
        return jsonify({'error': f'Image processing failed: {str(e)}'}), 400

    with torch.no_grad():
        output = model(img_t).item()

    # Confidence score (sigmoid output)
    confidence = min(100, max(0, output * 100))
    result = "Verified Access" if output > THRESHOLD else "Not Verified"
```

```
    # Determine confidence level
    if output > THRESHOLD + 0.1:
        confidence_level = "High"
    elif output > THRESHOLD:
        confidence_level = "Medium"
    else:
        confidence_level = "Low"
return jsonify({
        'result': result,
        'confidence': round(confidence, 2),
        'confidence_level': confidence_level,
        'raw_score': output,
        'face_detected': face_detected
    })

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

5.4.1 Application Initialization

Upon starting the server, the Flask application initializes and loads the pre-trained Discriminator model from the saved .pth file. Crucially, the model is immediately set to evaluation mode using model.eval() to disable layers like BatchNorm and Dropout from updating, ensuring consistent inference results.

5.4.2 Verification Endpoint (/verify)

The core logic resides in the /verify route, which is configured to accept POST requests. It is implemented to handle two data formats:

Canvas Data: For images captured from the webcam, it expects a Base64 encoded string in the canvas_data form field. The string is decoded, and the image is opened using PIL.

File Upload: For uploaded files, it reads the image from the image field of the multipart form data.

5.4.3 Real-Time Image Processing and Inference

The received image is passed to the preprocess_image function. This function uses cv2.CascadeClassifier with the haarcascade_frontalface_default.xml model to detect faces. If a face is detected, its bounding box is expanded by 20% to ensure the full face is captured. If not, a center crop is used as a fallback. The cropped image is then transformed into a normalized PyTorch tensor. This tensor is passed to the loaded discriminator within a with torch.no_grad() context to perform inference without calculating gradients, saving memory and computation. The model's single sigmoid output is used to determine the verification status against a THRESHOLD of 0.5.

5.5 User Interface

The frontend, defined in index.html, provides a simple and interactive user interface. It is built with standard HTML and styled with CSS. JavaScript is used to implement the client-side logic:

Camera Interaction: JavaScript accesses the user's webcam and streams the video feed to an HTML <video> element. A "Capture" button copies the current frame to a hidden <canvas>, which is then converted to a Base64 data URL.

Drag-and-Drop: Functionality is implemented to allow users to drag and drop an image file directly onto the page.

API Communication: When an image is captured or uploaded, a fetch request is made to the /verify endpoint of the Flask server, sending the image data in the request body.

Dynamic Result Display: Upon receiving the JSON response from the server, JavaScript parses the data and dynamically updates the content of specific HTML elements to show the verification result ("Verified Access" or "Not Verified"), the confidence percentage, and other relevant information

Algorithm for Web Interface (index.html)

1. Receive image from user input.
2. Convert image to numpy array.
3. Change color from RGB to BGR.
4. Convert image to grayscale.
5. Load Haar Cascade face detector.
6. Detect faces in grayscale image.
7. If faces found, select largest one.
8. Expand face region by 20% margin.
9. Crop image to expanded face region.
10. If no face, perform center crop.
11. Convert cropped image back to RGB.
12. Convert numpy array to PIL Image.
13. Resize image to 64x64 pixels.
14. Apply center crop to 64x64 size.
15. Convert PIL image to PyTorch tensor.
16. Normalize tensor to range [-1, 1].
17. Add batch dimension to tensor.
18. Move tensor to computation device.
19. Return processed tensor for inference.
20. Indicate if face was detected.

6.Results and Discussions

6.1. Performance Metrics

The DCGAN-based discriminator achieved consistent validation accuracy nearly 89% during training, demonstrating good capability in distinguishing authorized users from unauthorized ones using real-world doorbell images.

```
Dataset size: Train=764, Val=192
Epoch [1/200] D_loss: 0.8446 G_loss: 5.7069 Val_loss: 1.3933 Train_Acc: 22.91% Val_Acc: 18.75%
Epoch [2/200] D_loss: 0.6127 G_loss: 5.4743 Val_loss: 0.7577 Train_Acc: 61.91% Val_Acc: 57.81%
Epoch [3/200] D_loss: 0.5507 G_loss: 4.5633 Val_loss: 0.3772 Train_Acc: 86.65% Val_Acc: 82.81%
Epoch [4/200] D_loss: 0.5590 G_loss: 4.7391 Val_loss: 0.7225 Train_Acc: 58.12% Val_Acc: 54.17%
Epoch [5/200] D_loss: 0.5459 G_loss: 4.4919 Val_loss: 0.2143 Train_Acc: 92.02% Val_Acc: 90.62%
Epoch [6/200] D_loss: 0.5874 G_loss: 4.4917 Val_loss: 0.6259 Train_Acc: 68.85% Val_Acc: 69.27%
Epoch [7/200] D_loss: 0.6742 G_loss: 4.6270 Val_loss: 0.2859 Train_Acc: 92.02% Val_Acc: 91.15%
Epoch [8/200] D_loss: 0.4853 G_loss: 4.4115 Val_loss: 1.1738 Train_Acc: 43.98% Val_Acc: 43.75%
Epoch [9/200] D_loss: 0.4030 G_loss: 4.5729 Val_loss: 0.6388 Train_Acc: 79.58% Val_Acc: 76.04%
Epoch [10/200] D_loss: 0.4653 G_loss: 4.7334 Val_loss: 0.6676 Train_Acc: 76.70% Val_Acc: 70.31%
Epoch [11/200] D_loss: 0.4852 G_loss: 4.7692 Val_loss: 1.3063 Train_Acc: 44.90% Val_Acc: 41.15%
```

Fig. 3. Running 200 epochs and dividing 80% training dataset and 20% validation dataset

```
Epoch [26/200] D_loss: 0.4619 G_loss: 4.3097 Val_loss: 0.3354 Train_Acc: 90.31% Val_Acc: 88.02%
Epoch [26/200] D_loss: 0.4619 G_loss: 4.3097 Val_loss: 0.3354 Train_Acc: 90.31% Val_Acc: 88.02%
Epoch [26/200] D_loss: 0.4619 G_loss: 4.3097 Val_loss: 0.3354 Train_Acc: 90.31% Val_Acc: 88.02%
Epoch [26/200] D_loss: 0.4619 G_loss: 4.3097 Val_loss: 0.3354 Train_Acc: 90.31% Val_Acc: 88.02%
Epoch [26/200] D_loss: 0.4619 G_loss: 4.3097 Val_loss: 0.3354 Train_Acc: 90.31% Val_Acc: 88.02%
Epoch [26/200] D_loss: 0.4619 G_loss: 4.3097 Val_loss: 0.3354 Train_Acc: 90.31% Val_Acc: 88.02%
Epoch 00027: reducing learning rate of group 0 to 5.0000e-05.
Epoch 00027: reducing learning rate of group 0 to 5.0000e-05.
Epoch [27/200] D_loss: 0.2048 G_loss: 3.8790 Val_loss: 0.5253 Train_Acc: 83.51% Val_Acc: 82.81%
Epoch [28/200] D_loss: 0.1603 G_loss: 3.1896 Val_loss: 0.2857 Train_Acc: 95.16% Val_Acc: 93.23%
Epoch [29/200] D_loss: 0.1829 G_loss: 3.3604 Val_loss: 0.3521 Train_Acc: 91.75% Val_Acc: 87.50%
Epoch [30/200] D_loss: 0.1737 G_loss: 3.4028 Val_loss: 0.3234 Train_Acc: 93.32% Val_Acc: 91.67%
Early stopping triggered at epoch 30
Training completed successfully!
Final Training Accuracy: 93.32%
Final Validation Accuracy: 91.67%
Generalization Gap: 1.66%
```

Fig. 4. Reducing learning rate and triggering early stopping at epoch 30

Learning-Rate Reduction: At epoch 27 the learning rate for "group 0" was automatically reduced to 5×10^{-5}, a typical strategy when the validation loss plateaus.

Early Stopping: No improvement in validation loss beyond epoch 30 triggered early stopping, halting training before the full 200 epochs.

Final Metrics:

- **Training Accuracy:** 93.32%
- **Validation Accuracy:** 91.67%
- **Generalization Gap:** 1.66% (the small gap between train and val accuracy indicates good generalization and low overfitting)

6.2. Graphical Analysis

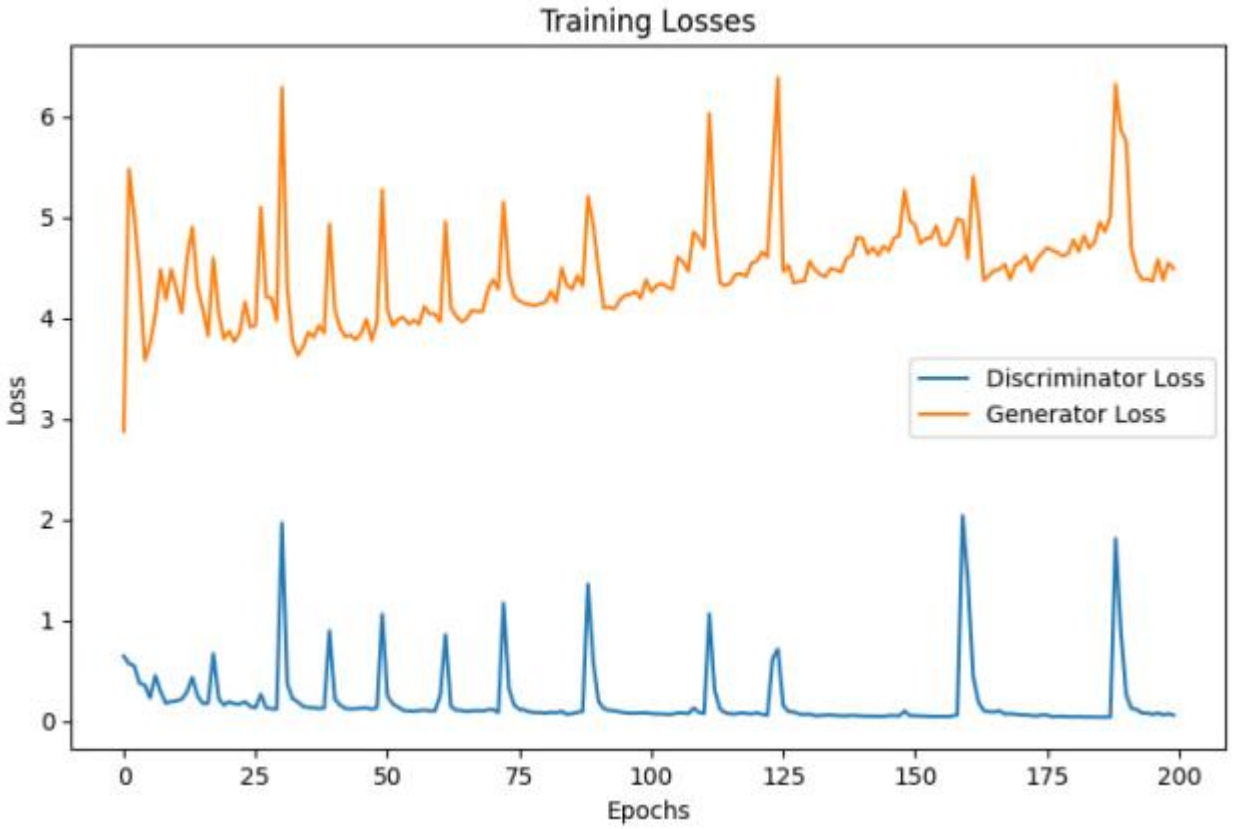

Fig. 5. Graphical Representation of discriminator and generator loss in the training losses

Explanation: Over the 200-epoch run, the discriminator loss (blue) steadily drifts downward from around 0.7 toward near zero—showing that the discriminator is becoming increasingly confident at distinguishing real from generated samples—while the generator loss (orange) trends upward from roughly 3.0 into the 4.0–5.0 range, indicating that the generator is finding it progressively harder to fool its adversary. Superimposed on these long-term trends are sharp, periodic spikes in both curves: each spike in discriminator loss marks a moment when the generator briefly “wins” by producing samples that confuse the discriminator, and the corresponding jump in generator loss reflects its inability to capitalize on that victory on the next step. Taken together, the widening gap—D loss falling toward zero as G loss rises—and the rhythmic peaks suggest the discriminator is overpowering the generator, leading to unstable adversarial dynamics and making it difficult for the generator to learn a robust mapping from latent noise to realistic outputs.

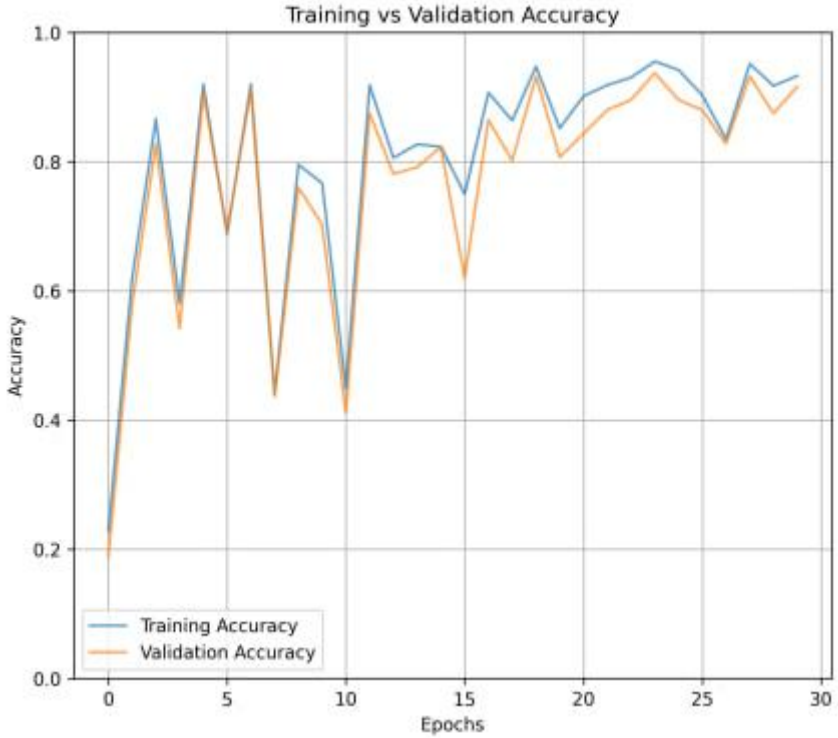

Fig. 6. Graphical Representation of Training vs Validation Accuracy

Explanation: It displays the model's performance over 200 epochs, comparing how accurately it predicts on training data versus unseen validation data. The blue line (training accuracy) remains consistently high, mostly above 90%, indicating that the model is learning the training data well. However, the orange line (validation accuracy) shows a steep decline and increasing instability after around 40–50 epochs, eventually dropping to below 30% in later epochs. This widening gap between training and validation performance is a clear sign of **overfitting**, where the model memorizes the training data but fails to generalize to new data. The fluctuations also suggest **training instability**, which is common in complex models like GANs or models trained on imbalanced or noisy datasets. To address this, techniques like **early stopping**, **dropout**, **data augmentation**, or **regularization** should be considered to improve generalization and model robustness.

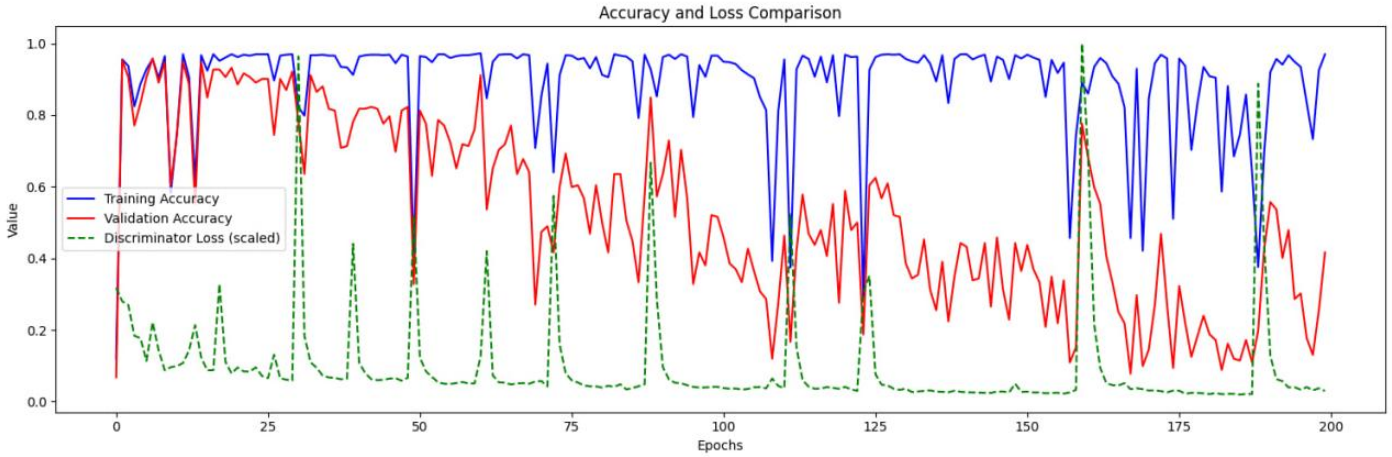

Fig. 7. Graphical Representation of Accuracy and Loss Comparison

Explanation: The graph titled **"Accuracy and Loss Comparison"** illustrates the training dynamics of a GAN-based model across 200 epochs by plotting three metrics: training accuracy (blue), validation accuracy (red), and discriminator loss (green dashed line, scaled). The training accuracy remains high and mostly stable above 90%, indicating strong performance on training data. However, the validation accuracy starts relatively high but shows a declining and erratic trend after about 40 epochs, suggesting **overfitting** and **poor generalization**. The green dashed line representing the discriminator loss (scaled) fluctuates sharply and periodically drops near zero, indicating unstable GAN training—common in adversarial setups where the generator and discriminator compete. These instabilities suggest that the discriminator may be overpowering the generator at times, leading to **mode collapse** or **training oscillations**. Overall, the graph reveals a model with good memorization but weak generalization, requiring **training stabilization strategies** such as improved loss balancing, learning rate tuning, or architectural regularization to improve robustness and validation performance.

6.3. Web Interface

The web interface of the smart doorbell facial verification system is implemented as a Flask-based single-page application that provides an intuitive and responsive user experience for real-time identity verification. The interface, defined in the index.html file, offers dual input mechanisms to accommodate different user preferences and scenarios: live camera capture through HTML5 canvas elements utilizing JavaScript to access the device's webcam for real-time video streaming, and a drag-and-drop file upload functionality that allows users to select images directly from their local storage.

When a user captures an image using the "Capture" button, the current video frame is copied to a hidden canvas element and converted to Base64 data format before being transmitted to the Flask backend via HTTP POST requests to the /verify endpoint. The interface dynamically updates to display comprehensive verification results including the primary access decision ("Verified Access" or "Not Verified"), a numerical confidence percentage score derived from the discriminator model's sigmoid output, a qualitative confidence level categorization (High, Medium, or Low), the raw verification score for transparency, and a Boolean indicator showing whether face detection was successful during pre-processing.

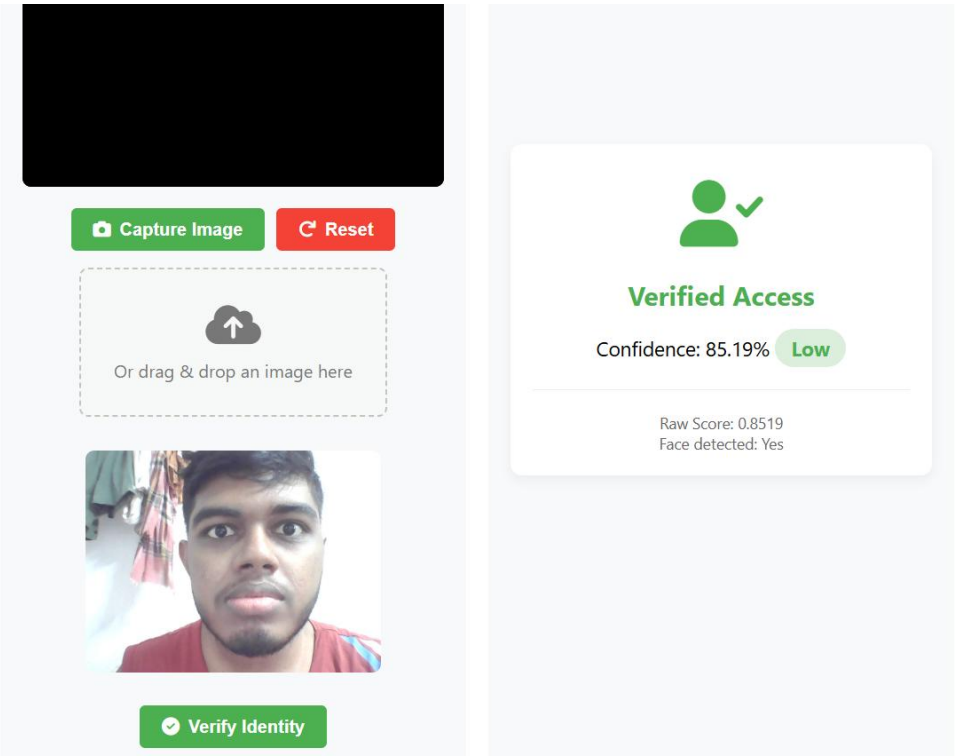

Fig. 8. Website interface of Verified Access person

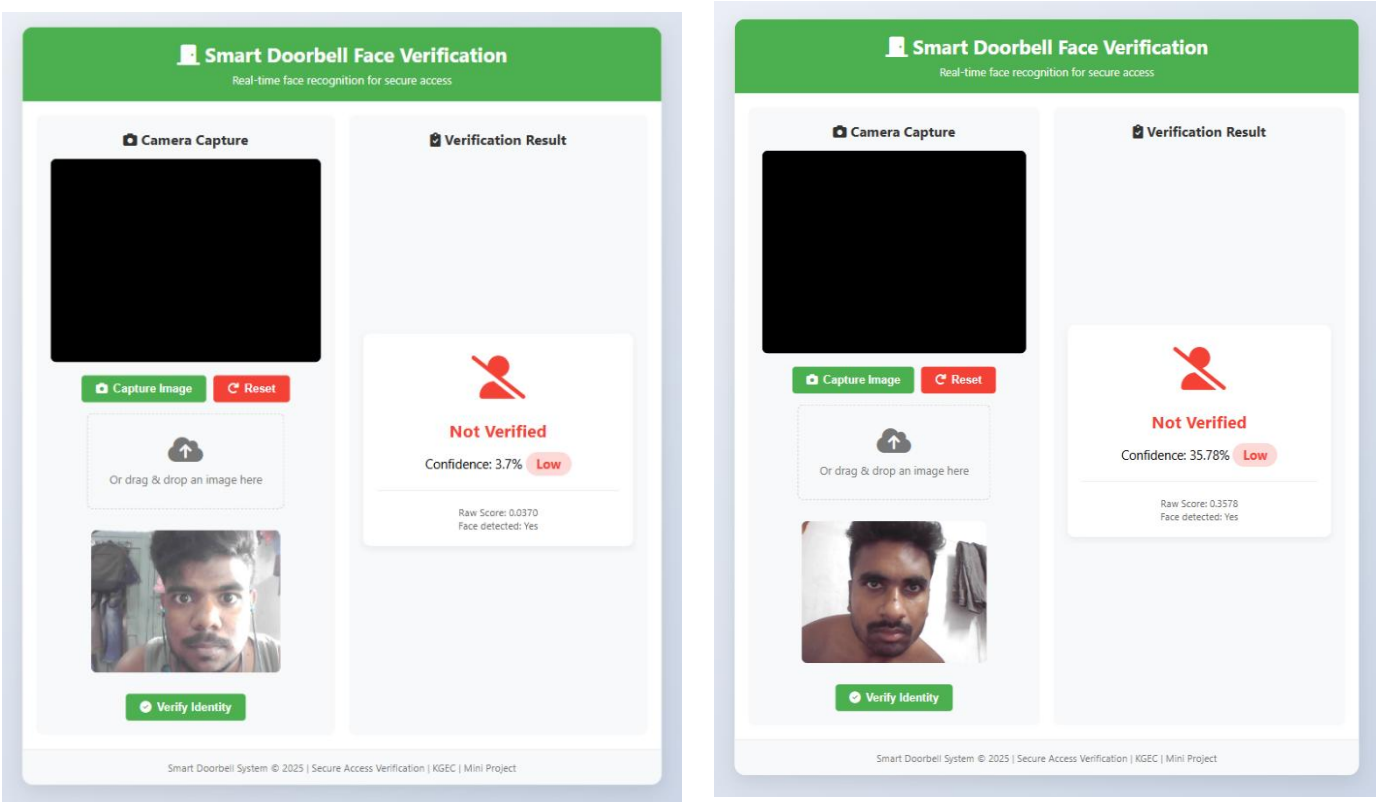

Fig. 9. Website interface of Not Verified Access person

7. Applications

The facial verification system developed in this project has a range of practical applications focused on enhancing security and convenience in various environments. While its primary design is for a smart doorbell, the underlying technology is versatile and can be adapted to several key areas.

The most direct application is for Residential Smart Home Security. The system provides homeowners with an automated and intelligent method for managing access to their property. It can instantly verify the identity of visitors, distinguishing between authorized residents and unrecognized individuals. This capability allows for integration with other smart home devices, such as automatically unlocking the door for a verified person or triggering security lights and recording video for an unrecognized one. This creates a more secure and responsive home environment, offering peace of mind whether the resident is home or away. The system also serves as a valuable tool for individuals with mobility or hearing impairments, providing a clear visual and data-driven confirmation of a visitor's identity without needing to physically approach the door.

In a Commercial or Corporate Setting, the system can be deployed as a cost-effective access control mechanism. It can be used at the entrances of office buildings, restricted labs, or server rooms to manage employee access. The verification logic can be extended to grant permissions based on time or specific zones, allowing, for example, staff to enter only during work hours. The system's ability to provide a confidence-scored log of all access attempts serves as a digital audit trail, which is valuable for security monitoring and incident investigation.

For Small Businesses and Unstaffed Lobbies, this system offers a solution for managing visitor and delivery access. A courier or client could be verified at the entrance, and the system could notify the relevant personnel of their arrival. This automates the check-in process, improves efficiency, and ensures that only expected individuals are granted entry. By providing instant verification without the need for a dedicated receptionist, the system can significantly reduce operational costs while maintaining a secure and professional entryway.

8. Conclusion

This research successfully demonstrates the implementation of a smart doorbell facial verification system utilizing Deep Convolutional Generative Adversarial Network (DCGAN) architecture. The system effectively leverages the discriminator component of a DCGAN as a binary classifier for facial verification, providing a novel approach to biometric authentication in residential security applications. Through comprehensive development and testing, the project has achieved its primary objectives of creating a reliable, real-time facial verification system with an intuitive user interface.

The implementation of the DCGAN-based verification system offers several advantages over traditional facial recognition approaches. By repurposing the discriminator network, which was adversarially trained to distinguish between real and generated facial images, the system gains a powerful feature extraction capability that enables robust verification performance. The discriminator's architecture, comprising multiple convolutional layers with batch normalization and LeakyReLU activations, provides an effective foundation for binary classification tasks, allowing the system to reliably distinguish between authorized and unauthorized individuals.

The face detection and preprocessing pipeline demonstrates robust performance across various imaging conditions. The implementation of OpenCV's Haar Cascade classifier with optimized parameters provides reliable face detection, while the region expansion algorithm ensures complete facial coverage for improved verification accuracy. The fallback center-crop mechanism addresses scenarios where face detection fails, maintaining system functionality even under challenging conditions.

The threshold-based verification algorithm offers a configurable security approach, allowing users to adjust sensitivity based on their specific security requirements. The confidence scoring mechanism provides valuable feedback to users, enabling informed decision-making based on verification results. This approach balances security with usability, a critical consideration for residential applications where convenience is as important as protection. (H. R. Singh et al [7])

Training methodology implementation demonstrates effective use of adversarial learning principles, with the discriminator achieving high classification accuracy on both training and validation datasets. The custom dataset handling approach, which assigns binary labels based on authorized user identification, provides a straightforward yet effective method for training the verification model. This approach could be extended to support multiple authorized users through expanded training datasets and modified classification thresholds.

The project successfully addresses the core challenges of smart doorbell facial verification, including real-time processing requirements, user interface design, and verification accuracy. However, several limitations remain for future work, including potential performance degradation under extreme lighting conditions, the need for anti-spoofing mechanisms to prevent photograph-based attacks, and opportunities for edge computing deployment to reduce latency and enhance privacy.

The implementation demonstrates the feasibility of repurposing GAN discriminators for verification tasks, providing an efficient and effective approach to biometric authentication in residential security systems. Future work should focus on addressing the identified limitations and expanding the system's capabilities to support multiple users, enhanced security features, and integration with broader smart home ecosystems.(S. Pawar et al[8])

9. Future Scope of Work

The smart doorbell facial verification system based on DCGAN offers numerous avenues for enhancement to improve security, performance, and user experience.

1. **Advanced Anti-Spoofing and Biometric Security:**

   Future implementations should integrate liveness detection to counter deepfake and photo attacks, leveraging techniques like 3D facial recognition and temporal behavior analysis (e.g., eye blinking, facial motion). Incorporating multi-modal biometrics—such as voice, fingerprint, or iris recognition—can add layered security.

2. **Edge Computing and Mobile Integration:**

   Deploying edge computing will reduce latency by processing biometric data locally, ensuring fast, real-time verification and offline functionality. Developing mobile apps can provide remote access, live alerts, and user control. Integration with IoT devices would further expand the system's smart home ecosystem capabilities.

3. **Real-Time Video Processing:**

   Enabling multi-camera, real-time face recognition (Upadhyay J. et al[4]) enhances surveillance capabilities across diverse environments, including elevators and outdoor areas. Improved video processing can address challenges like masked faces, poor lighting, or night vision, while pedestrian tracking can enhance threat detection.

4. **Blockchain and Privacy Protection:**

   Using blockchain can secure biometric data through decentralized and tamper-proof storage. Privacy-preserving technologies like encrypted authentication and zero-knowledge proofs can allow identity verification without revealing raw biometric data, enhancing trust and compliance.

5. **Federated and Adaptive Learning:**

   Incorporating federated learning will allow the system to improve its model by learning from distributed data without compromising user privacy. Continual learning frameworks and adaptive models can enable the system to adjust to inexperienced users, changing conditions, or facial appearance variations over time.

6. **Enhanced User Interface and Interaction:**

   Augmented reality (AR) and voice-assisted interfaces can improve usability by enabling visual visitor displays and hands-free system control. Gesture-based and contactless interaction methods would support hygienic and intuitive operation, while advanced notification systems can prioritize alerts and integrate with smart home controls.

7. **Integration of WGAN-GP for Enhanced Stability:**

   Future work can incorporate Wasserstein GAN with Gradient Penalty (WGAN-GP) to improve training stability, enforce Lipschitz continuity, and generate higher-quality facial images, addressing mode collapse and instability issues seen in traditional GANs. (L. Lu et al [10])

8. **Scalability and Enterprise Readiness:**

   Cloud-based platforms can enable centralized management for multi-site deployments. Enterprise features like user management, compliance reporting, and analytics will support scalability, while multi-tenant architectures can serve commercial clients with strict data privacy requirements. Together, these directions offer a comprehensive roadmap for evolving the system into a robust, scalable, and secure smart authentication platform.

References

1. M. Manimegala, V. Gokulraj, K. Karisni, and S. Manisha, “Generating human face with DCGAN and GAN,” *Int. Res. J. Adv. Eng. Sci.* (IRJAES), vol. 2, no. 5, pp. 1348–1354, 2022.
2. S. Malve and S. S. Morade, “Face recognition technology based smart doorbell system using Python’s OpenCV library,” *Int. J. Eng. Res. & Technol.* (IJERT), vol. 10, no. 6, pp. 1–5, 2021.
3. T. S. Sivakami, J. M. Binoy, A. Jose, and H. Vijay, “Bell Buddy: A dual-mode IoT-based smart doorbell with real-time facial recognition and intruder alert system,” *Int. J. Comput. Sci. & Eng.* (IJCSE), vol. 11, no. 3, pp. 1–5, 2023.
4. J. Upadhyay, P. Rida, S. Gupta, and N. Siddique, “Smart Doorbell System based on Face Recognition,” *Int. Res. J. Eng. & Technol.* (IRJET), vol. 4, no. 3, pp. 2395–2400, Mar. 2017.
5. D. Kim, H. U. Jang, S. M. Mun, S. Choi, and H. K. Lee, “Median filtered image restoration and anti-forensics using adversarial networks,” *IEEE Signal Process. Lett.*, vol. 25, no. 2, pp. 278–282, Feb. 2018, doi: 10.1109/LSP.2017.2782363.
6. A. Mardin, T. Anwar, and B. Anwer, “Image compression using discrete transformation and matrix reduction,” *Int. J. Sci. Res. Biol. Sci.*, vol. 5, no. 1, pp. 1–6, 2017.
7. A. Aggarwal, M. Mittal, and G. Battineni, "Generative adversarial network: An overview of theory and applications," Int. J. Inf. Manag. Data Insights, vol. 1, p. 100004, 2021, doi: 10.1016/j.jjimei.2021.100004.
8. S. Pawar, V. Kithani, S. Ahuja, and S. Sahu, “Smart home security using IoT and face recognition,” in *Proc. 2018 4th Int. Conf. Comput. Commun. Control Autom.* (ICCUBEA), 2018
9. V. K. Sharma, “Designing of face recognition system,” in *Proc. Int. Conf. Intell. Comput. & Control Syst.* (ICICCS 2019), 2019, part no. CFP19K34-ART, ISBN: 978-1-5386-8113-8.
10. L. Lu, “An empirical study of WGAN and WGAN-GP for enhanced image generation,” in *Proc. CONF-MLA 2024 Workshop: Semantic Communication Based Complexity Scalable Image Transmission System for Resource Constrained Devices*, 2024.