INTRODUCTION TO PYTHON-22AIE205

A Thesis

Submitted by

Group – 05

ESHWANTH KARTHI T R-(CB.EN.U4AIE22118)

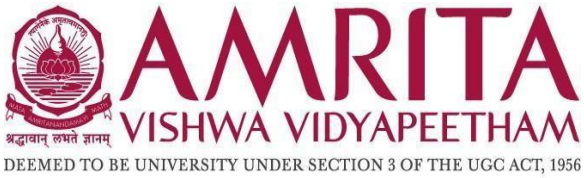
NANDANA GIREESH -(CB.EN.U4AIE22138)

S ANANTHASIVAN -(CB.EN.U4AIE22148)

SNEGA SRI A -(CB.EN.U4AIE22163)

In partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
CSE (AIE)



Centre for Computational Engineering and Networking

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

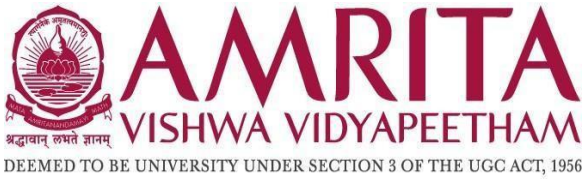
COIMBATORE – 641 112 (INDIA)

DECEMBER– 2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDHYAPEETHAM

COIMBATORE – 641 112



BONAFIDE CERTIFICATE

This is to certify that the report entitled "**Comparative Analysis of Deepfake Generation and Detection Models: A Web-Based Framework**" submitted by ESHWANTH KARTHI (CB.EN.U4AIE22118), SNEGA SRI A (CB.EN.U4AIE22163), NANDANA GIREESH (CB.EN.U4AIE22138), S ANANTHASIVAN (CB.EN.U4AIE22148) for the award of the Degree of Bachelor of Technology in the "CSE (AI)" is a bonafide record of the work carried out by her under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Ms. Sreelakshmi K

Project Guide

Dr.K.P.Soman

Professor and Head CEN

Submitted for the university examination held on 22-12-2023

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDHYAPEETHAM

COIMBATORE – 641 112

DECLARATION

We, Eshwanth Karti T R(CB.EN.U4AIE22118), Snega Sri A (CB.EN.U4AIE22163), Nandana Gireesh(CB.EN.U4AIE22138), S ANANTHASIVAN (CB.EN.U4AIE22148) hereby declare that this thesis entitled "**Comparative Analysis of Deepfake Generation and Detection Models: A Web-Based Framework**", is the record of the original work done by us under the guidance of Ms. Sreelakshmi K, Assistant Professor, Centre for Computational Engineering and Networking (CEN), Amrita School of Artificial Intelligence, Coimbatore. To the best of my knowledge, this work has not formed the basis for the award of any degree/diploma/ associate ship/fellowship/or a similar award to any candidate in any University.

Place: Ettimadai

Date: 22-12-2023 **Signature of the Students**

ACKNOWLEDGEMENT

We would like to express our special thanks of gratitude to our teacher (MS. SREELAKSHMI K ma'am), who gave us the golden opportunity to do this wonderful project on the topic "**Comparative Analysis of Deepfake Generation and Detection Models: A Web-Based Framework**", which also helped us in doing a lot of Research and we came to know about so many new things. We are thankful for the opportunity given. We would also like to thank our group members, as without their cooperation, we would not have been able to complete the project within the prescribed time.

Contents

List of figures

ABSTRACT

With the proliferation of deepfake technology, there is an increasing need for robust and reliable methods for both generation and detection. This project presents a comprehensive analysis of a web-based deepfake generation and detection framework. The generation component leverages the GFPGANv1.4.pth model for realistic face swapping, while the detection module utilizes the inswapper_128.onnx model for identifying manipulated content. In addition to these primary models, the study compares the performance of additional generation and detection models to provide a nuanced understanding of their capabilities.

The web-based implementation employs a Tkinter-based frontend for user interaction, seamlessly integrating the backend functionalities. The project evaluates the generation models based on training data, pre-processing techniques, and user feedback, while the detection models are scrutinized for accuracy, false positive/negative rates, and robustness against diverse deepfake techniques.The comparative analysis extends to explore alternative generation models, including other models, and detection models, each with its unique characteristics. The results of the study offer insights into the strengths and limitations of different models, shedding light on their real-world applicability.

Furthermore, the report discusses the technical challenges encountered during implementation and proposes potential future enhancements for both generation and detection methodologies. The conclusion summarizes the key findings and their implications, providing a valuable resource for researchers, developers, and policymakers grappling with the evolving landscape of deepfake technology.This project contributes to the ongoing discourse surrounding deepfakes by offering a detailed examination of various models, their performances, and the challenges associated with their implementation within a user-friendly web-based framework.

INTRODUCTION

In recent years, the rise of deepfake technology has ushered in a new era of both creative possibilities and ethical concerns. Deepfakes, or artificially generated media that convincingly depict events or individuals, have found applications in entertainment, and social media, and even pose potential threats to misinformation and privacy. As the prevalence of deepfakes continues to grow, the need for robust tools for both their generation and detection becomes increasingly paramount.

This project delves into the realm of deepfakes, offering a comprehensive exploration of both sides of the spectrum. On one front, we examine the intricacies of deepfake generation, employing advanced models such as StyleGAN2, DeepFaceLab, and CycleGAN. Each model brings its unique approach to the synthesis of realistic and diverse images, catering to the evolving landscape of deepfake creation. Simultaneously, our endeavor extends to the critical domain of deepfake detection, wherein we scrutinize the efficacy of models like inswapper_128.onnx, MesoNet, Capsule-Forensics, and FaceForensics++. These detection models leverage innovative techniques, ranging from micro-expression analysis to capsule networks, to discern authentic content from manipulated media.

The primary objective of this project is to offer a nuanced comparative analysis of various deepfake generation and detection models. By juxtaposing established models with cutting-edge counterparts, we aim to unravel the strengths, weaknesses, and distinctive features inherent in each approach. This exploration not only contributes to the understanding of state-of-the-art deepfake technologies but also serves as a foundation for advancing the development of more resilient and sophisticated tools in the ongoing battle between creation and detection. Through a web-based framework, we present an accessible interface that integrates these models, fostering a practical understanding of their performance and usability. As we navigate through the intricacies of deepfake technology, this project seeks to shed light on the evolving landscape of synthetic media, offering insights that contribute to the ongoing discourse on its implications, applications, and the imperative need for effective safeguards.

1 Deepfake Generation Models

1.1 GAN(Generative Adversarial Networks)

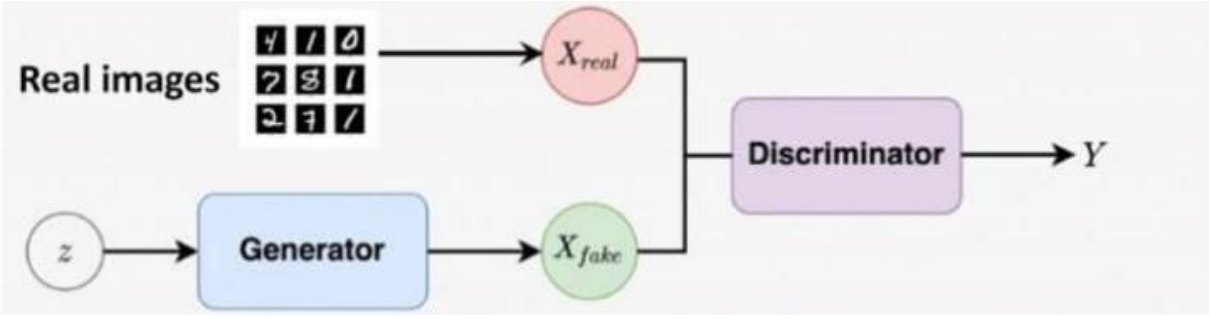


Figure 1 General GAN model structure

A Generative Adversarial Network (GAN) is a powerful class of neural networks used for unsupervised learning.

Generative: GANs learn a generative model, which describes how data is generated in terms of a probabilistic model.

Adversarial: The term "adversarial" means setting one thing up against another. In the context of GANs, the generative results are compared with actual images in the dataset. A mechanism called a discriminator is used to distinguish between real and fake images.

Networks: GANs use deep neural networks as artificial intelligence (AI) algorithms for training.

A GAN consists of two neural networks:
Generator: It produces synthetic data (such as images, text, or audio).

Discriminator: It tries to distinguish between the synthetic data generated by the generator and real data from a training set. These two networks engage in a competitive interplay:The generator aims to create realistic samples that fool the discriminator approximately half the time. The discriminator gets better at distinguishing real from fake data as it learns from both the generator's output and the actual data. Gans have revolutionized generative modelling and found applications in image synthesis, style transfer, and text-to-image synthesis. They can generate lifelike content across various domains, including creating photorealistic images indistinguishable from real photos. The main concept for this is used for converting low resolution into a high resolution and converting black and white into an RGB one.

1.2 GFP-GAN

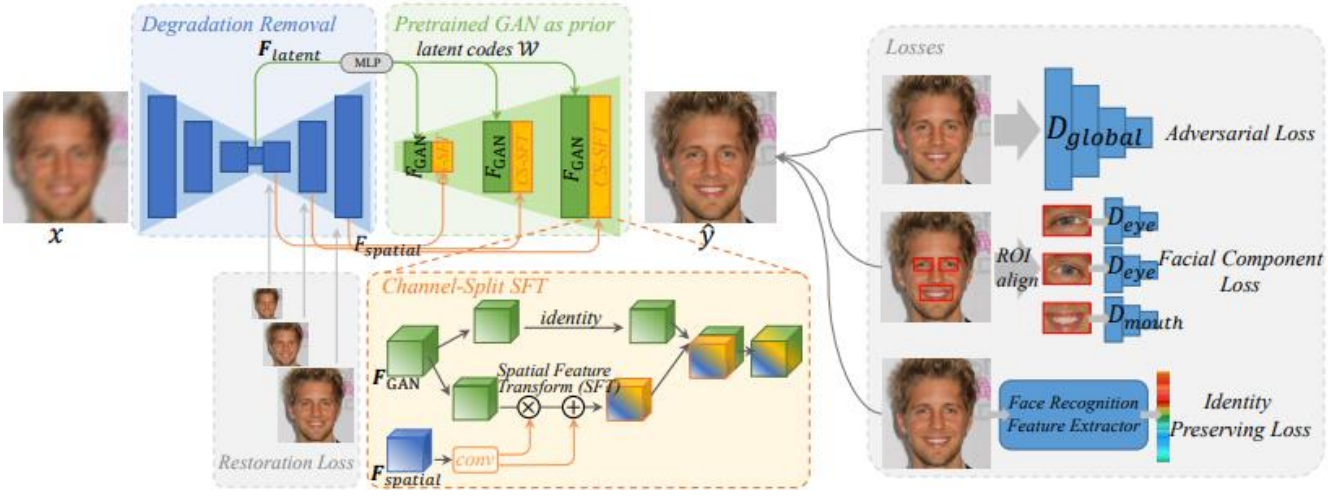


Figure 2 GFP-GAN model

GFP-GAN, this framework consists of a degradation removal module and a pre-trained face GAN as prior. They bridge them with latent code mapping and several channel-split spatial feature transform (CS-SFT) layers, which enable effective prior

incorporation and fidelity preservation. They also design several losses to enhance facial details and retain identity.

1.3 Training Data and Pre-processing

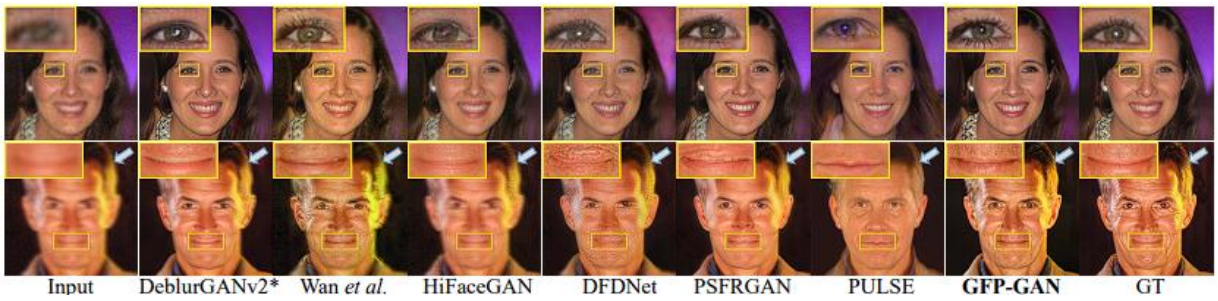


Figure 3 Comparison between state-of-the-art models

This dataset of 70,000 good-looking face images to train our model. We make the images smaller to fit our model. We also create some fake bad-looking images from the good ones, so that our model can learn how to fix them [1]. We do this by making the images blurry, noisy, smaller, and lower quality. We choose different levels of blurriness, noise, size, and quality for each image. We also changed the colours of the images a bit to make them more diverse.

1.4 Data Set Selections

We construct one synthetic dataset and three different real datasets with distinct sources. All these datasets have no overlap with our training dataset. We provide a brief introduction here.

CelebA-Test is the synthetic dataset with 3,000 CelebA-HQ images from its testing partition. The generation way is the same as that during training.

LFW-Test. LFW contains low-quality images in the wild. We group all the first image for each identity in the validation partition, forming 1711 testing images.

CelebChild-Test contains 180 child faces of celebrities collected from the Internet. They are low-quality and many of them are black-and-white old photos.

WebPhoto-Test. We crawled 188 low-quality photos in real life from the Internet and extracted 407 faces to construct the WebPhoto testing dataset. These photos have diverse and complicated degradation. Some of them are old photos with very severe degradation in both details and colour.

1.5 Comparison

Methods	LPIPS↓	FID↓	NIQE ↓	Deg.↓	PSNR↑	SSIM↑
Input	0.4866	143.98	13.440	47.94	25.35	0.6848
DeblurGANv2* [40]	0.4001	52.69	4.917	39.64	**25.91**	**0.6952**
Wan *et al*. [61]	0.4826	67.58	5.356	43.00	24.71	0.6320
HiFaceGAN [67]	0.4770	66.09	4.916	42.18	24.92	0.6195
DFDNet [44]	0.4341	59.08	4.341	40.31	23.68	0.6622
PSFRGAN [6]	0.4240	47.59	5.123	39.69	24.71	0.6557
mGANprior [19]	0.4584	82.27	6.422	55.45	24.30	0.6758
PULSE [52]	0.4851	67.56	5.305	69.55	21.61	0.6200
GFP-GAN (ours)	**0.3646**	**42.62**	**4.077**	**34.60**	25.08	0.6777
GT	0	43.43	4.292	0	∞	1

Figure 4 Comparison between models

Red and blue indicate the best and the second-best performance. '*' denotes finetuning on our training set. Deg. represents the identity distance. Feature Extracted and comparison between different GANs like HiFaceGAN, DFDNet, PSFRGAN, Super-FAN, and Wan et al. GAN inversion methods for face restoration: PULSE and mGANprior are also included for comparison. We also compare GFP-GAN [1] with image restoration methods like RCAN, ESRGAN, and DeblurGANv2

1.6 Metrics used for comparison

These are the following metrics used by standard researchers worldwide

1. **FID (Fréchet Inception Distance)**:
   FID measures the similarity between the distribution of generated images and real images. It uses features extracted from a pretrained neural network (usually InceptionV3) to compute the distance. Lower FID values indicate better image quality and realism.
2. **NIQE (Naturalness Image Quality Evaluator)**:
   NIQE is a no-reference metric that assesses the naturalness and quality of an image. It estimates the amount of distortion or artifacts present in an image. Higher NIQE scores indicate worse image quality.
3. **PSNR (Peak Signal-to-Noise Ratio)**:
   PSNR measures the quality of a reconstructed image by comparing it to the original (ground truth) image. It computes the ratio of the maximum possible power of a signal to the power of the noise . Higher PSNR values indicate better image quality.
4. **SSIM (Structural Similarity Index)**:
   SSIM assesses the structural similarity between two images. It considers luminance, contrast, and structure. SSIM values range from -1 to 1, with 1 indicating perfect similarity.
5. **LPIPS (Learned Perceptual Image Patch Similarity)**:

LPIPS is a perceptual metric that quantifies the similarity between images. It uses a pre-trained deep neural network) to extract features. Smaller LPIPS values indicate greater perceptual similarity.

6. **ArcFace**:

Arc Face is a face recognition model that learns discriminative features for face identification. It computes angular distances (angles) between feature vectors. Smaller angular distances indicate closer identity to the ground truth.

1.7 Weakness

Figure 5 Grey Image from model

Training bias This method performs well on most dark-skinned faces and various population groups, as our method uses both the pre-trained GAN and input image features for modulation. Besides, we employ reconstruction loss and identity preserving loss to restrict the outputs to retain fidelity with inputs [1]. However, when input images are Gray-scale, the face colour may have a bias, as the inputs do not contain sufficient colour information. Thus, a diverse and balanced dataset is needed.

2 Deepfake Detection Model Overview

2.1 MTCNN(Multi-Task Cascaded Convolutional Neural Networks)

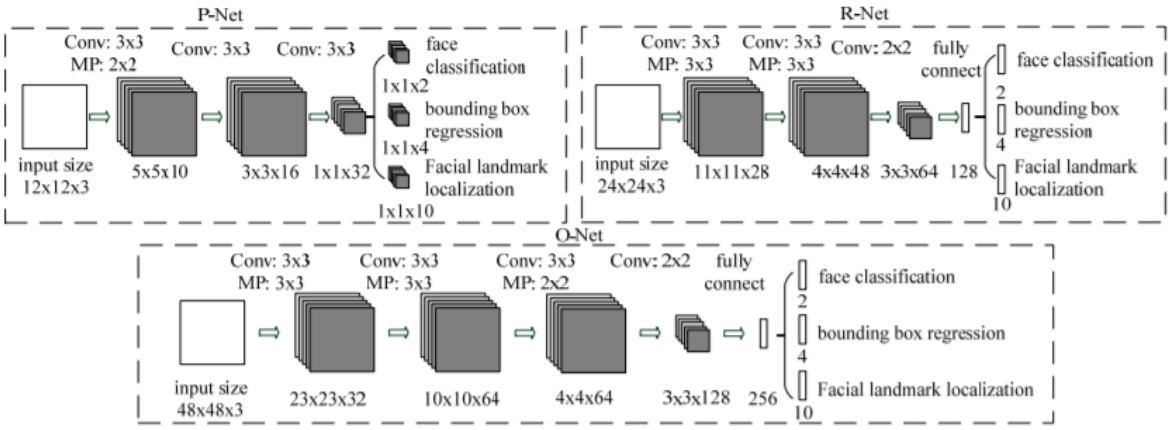


Figure 6 Mtcnn model architecture

MTCNN architecture: **P-Net, R-Net, O-Net**

The **P-Net**, short for Proposal Network, is designed to swiftly detect faces within 12x12 sized frames. Its primary objective is to provide rapid results in face detection. Operating as the initial stage in the face detection pipeline, the P-Net scans the input frames and efficiently filters out numerous non-face candidates, contributing to the the overall speed of the face detection process [2].

Following the P-Net, the **R-Net**, or Refined Network, is introduced with a more complex structure. It serves as the second stage in the face-detection process. All potential face candidates identified by the P-Net are forwarded to the R-Net for further scrutiny [2]. The R-Net's deeper architecture allows it to refine the selection made by the P-Net, effectively rejecting a substantial number of false positives. This deeper level of scrutiny enhances the accuracy of the face detection system.

Ultimately, the output network, abbreviated as **O-Net**, plays a crucial role in providing the final results. The O-Net is responsible for returning the bounding box , which outlines the detected face area, and the positions of facial landmarks. By combining the outputs of the P-Net and R-Net, the O-Net ensures a comprehensive and accurate representation of the detected faces, including both their spatial location and facial features. The sequential integration of these networks optimizes the face detection process, balancing speed and precision in identifying and delineating faces in input frames.

2.2 InceptionResnetV1

InceptionNet is a convolutional neural network (CNN) designed for image classification tasks and developed for the ImageNet Large Scale Visual RecognitionChallenge. InceptionNet is known for using inception modules, blocks of layers designed to learn a combination of local and global features from the input data. These modules are composed of smaller convolutional and pooling layers, which are combined to allow the network to learn spatial and temporal features from the input data

2.3 GradCAM

The Grad-CAM technique utilizes the gradients of the classification score concerning the final convolutional feature map, to identify the parts of an input image that most impact the classification score. The places where this gradient is large are exactly the places where the final score depends most on the data.

1. **Feedforward Pass:**
    - The input image is fed through the neural network, and its activations are computed at each layer of the network.
    - The final classification score is obtained from the output layer.
2. **Compute Gradient:**

- The gradient of the target class score with respect to the feature maps of the last convolutional layer is calculated. This gradient represents how much the final score would change concerning changes in each element of the feature maps.

3. **Global Average Pooling (GAP):**
   - Global Average Pooling is applied to the gradients obtained from the previous step. This step helps to capture the importance of each feature map by taking the average of the gradient values in each channel.
4. **Weighted Sum of Feature Maps:**
   - The weighted sum of the feature maps is computed, where the weights are given by the corresponding values from the Global Average Pooling step. This effectively produces a weighted combination of the feature maps, emphasizing the importance of each spatial location.
5. **ReLU Activation:**
   - The obtained map is passed through a ReLU activation to remove any negative values, as negative contributions would not contribute positively to the importance of a region in the context of the target class.
6. **Upsample:**
   - The resulting heatmap is upsampled to match the dimensions of the original input image. This provides a spatial correspondence between the heatmap and the input image.
7. **Overlay on Input Image:**
   - The heatmap is then overlaid onto the input image, highlighting the regions that are most influential in the network's decision for the target class.

3 CODE WITH EXPLANATION

3.1 Detection

```
import torch
import torch.nn.functional as F
from facenet_pytorch import MTCNN, InceptionResnetV1
from PIL import Image
import cv2
from pytorch_grad_cam import GradCAM
from pytorch_grad_cam.utils.model_targets import ClassifierOutputTarget
from pytorch_grad_cam.utils.image import show_cam_on_image
import gradio as gr
DEVICE = 'cuda:0' if torch.cuda.is_available() else 'cpu'
print(DEVICE)
mtcnn = MTCNN(
    select_largest=False,
    post_process=False,
    device=DEVICE
).to(DEVICE).eval()
```

Explanation

Import all necessary libraries for this code to function and this code checks if a CUDA-enabled GPU is available and sets the device accordingly. It then initializes an MTCNN model with specific settings, such as returning all detected faces, disabling post-processing, and setting the device to either GPU or CPU. The model is then moved to the selected device and set to evaluation mode. Finally, the chosen device is printed.

```
model = InceptionResnetV1(
    pretrained="vggface2",
    classify=True,
    num_classes=1,
    device=DEVICE
)

checkpoint = torch.load("../models/resnetinceptionv1_epoch_32.pth", map_location=torch.device('cpu'))
model.load_state_dict(checkpoint['model_state_dict'])
model.to(DEVICE)
model.eval()
```

Explanation

The code initiates an InceptionResnetV1 model configured for binary classification, utilizing pre-trained weights from the VGGFace2 dataset. It proceeds to load the model state from a stored checkpoint file (`resnetinceptionv1_epoch_32.pth`). Following this, the model is transferred to a designated device (either GPU or CPU) and set to evaluation mode. This systematic approach is a standard procedure for deploying pre-trained models onto specific devices, ensuring seamless compatibility and optimal performance during inference. The checkpoint file encapsulates the trained model's parameters, facilitating the restoration of its state.

```
def predict(input_image:Image.Image):
    face = mtcnn(input_image)
    if face is None:
        raise Exception('No face detected')
    face = face.unsqueeze(0)
    face = F.interpolate(face, size=(256, 256), mode='bilinear', align_corners=False)
    prev_face = face.squeeze(0).permute(1, 2, 0).cpu().detach().int().numpy()
    prev_face = prev_face.astype('uint8')

    face = face.to(DEVICE)
    face = face.to(torch.float32)
    face = face / 255.0
    face_image_to_plot = face.squeeze(0).permute(1, 2, 0).cpu().detach().int().numpy()
```

```
    target_layers=[model.block8.branch1[-1]]
    use_cuda = True if torch.cuda.is_available() else False
    cam = GradCAM(model=model, target_layers=target_layers, use_cuda=use_cuda)
    targets = [ClassifierOutputTarget(0)]

    grayscale_cam = cam(input_tensor=face, targets=targets, eigen_smooth=True)
    grayscale_cam = grayscale_cam[0, :]
    visualization = show_cam_on_image(face_image_to_plot, grayscale_cam, use_rgb=True)
    face_with_mask = cv2.addWeighted(prev_face, 1, visualization, 0.5, 0)

    with torch.no_grad():
        output = torch.sigmoid(model(face).squeeze(0))
        prediction = "real" if output.item() < 0.5 else "fake"

        real_prediction = 1 - output.item()
        fake_prediction = output.item()

        confidences = {
            'real': real_prediction,
            'fake': fake_prediction
        }
    print(face_with_mask)
    return confidences, face_with_mask
```

Explanation

The `predict` function begins by using the MTCNN model to detect and extract a face from the input image. It checks for successful detection and raises an exception if no face is found. The face is then preprocessed, resized, and converted to a format suitable for visualization. Subsequently, it employs GradCAM, utilizing a pre-trained InceptionResnetV1 model to generate a heatmap highlighting relevant regions for classification. The function returns confidence scores for "real" and "fake" predictions, along with a visual representation of the face overlaid with the GradCAM heatmap. The original face image is blended with the heatmap to enhance interpretability. The confidence scores provide insights into the model's prediction certainty and the visual output aids in understanding the model's focus during classification.

```
interface = gr.Interface(
    fn=predict,
    inputs=[
        gr.Image(label="Input Image", type="pil")
    ],
    outputs=[
        gr.Label(label="Class"),
        gr.Image(label="Face with Explainability", type="pil")
```

```
    ],
).launch()
```

Explanation

- The code creates a Gradio interface using `gr.Interface`.
- It associates the interface with the `predict` function for image predictions.
- The interface includes an image upload component labeled "Input Image" for user input.
- Outputs consist of a predicted class label and a visual representation of the input face with an explainability heatmap.
- Users can interact with the launched interface to upload images and observe model predictions along with visual explanations.
- Gradio simplifies the process of building interactive interfaces for machine learning models.

3.2 Generator

```
import cv2
from typing import Any, List
import threading
from insightface.app.common import Face
import os
import insightface
import numpy
from GFPGAN.gfpgan.utils import GFPGANer
import tkinter as tk
from tkinter import ttk, filedialog
from PIL import Image, ImageTk

Face = Face
print(Face)
execution_providers = ['CPUExecutionProvider']
Frame = numpy.ndarray[Any,Any]
```

Explanation

Necessary imports are made in this segment

```
def resolve_relative_path(path: str):
    notebook_dir = os.getcwd()
    print(temp := os.path.abspath(os.path.join(notebook_dir, path)))
    return temp
```

Explanation

This function is used to get the absolute path when relative path is given

```
THREAD_SEMAPHORE = threading.Semaphore()
THREAD_LOCK = threading.Lock()
FACE_ENHANCER=None

def get_face_enhancer() -> Any:
    global FACE_ENHANCER
    with THREAD_LOCK:
        if FACE_ENHANCER is None:
            model_path = resolve_relative_path('./models/GFPGANv1.4.pth')
            FACE_ENHANCER = GFPGANer(model_path=model_path, upscale=1, device='cpu')

    return FACE_ENHANCER
def enhance_face(target_face: Face, temp_frame: Frame) -> Frame:
    start_x, start_y, end_x, end_y = map(int, target_face['bbox'])
    padding_x = int((end_x - start_x) * 0.5)
    padding_y = int((end_y - start_y) * 0.5)
    start_x = max(0, start_x - padding_x)
    start_y = max(0, start_y - padding_y)
    end_x = max(0, end_x + padding_x)
    end_y = max(0, end_y + padding_y)
    temp_face = temp_frame[start_y:end_y, start_x:end_x]
    if temp_face.size:
        with THREAD_SEMAPHORE:
            _, _, temp_face = get_face_enhancer().enhance(
                temp_face,
                paste_back=True
            )
        temp_frame[start_y:end_y, start_x:end_x] = temp_face
    return temp_frame
```

Explanation

Global variables are defined for a thread semaphore (THREAD_SEMAPHORE), a thread lock (THREAD_LOCK), and a global variable (FACE_ENHANCER) initialized to None. This function returns the global variable **FACE_ENHANCER**, which is an instance of the **GFPGANer** class.It uses a thread lock (**THREAD_LOCK**) to ensure that the initialization of **FACE_ENHANCER** is thread-safe.If **FACE_ENHANCER** is **None**, it initializes it with a new instance of **GFPGANer** using a model located at the specified path. This function takes a **target_face** (presumably a bounding box of a face) and a **temp_frame** (presumably an

image frame) as input.It extracts a region of interest (ROI) from the **temp_frame** corresponding to the bounding box of the **target_face**.It then uses the **get_face_enhancer** function to obtain the face enhancer (**GFPGANer**) and enhances the extracted face region. The enhanced face is then pasted back into the original **temp_frame** at the same location.The resulting **temp_frame** is returned

```
from typing import Optional
FACE_ANALYSER = None
similar_face_distance: Optional[float] = None
def get_face_analyser() -> Any:
    global FACE_ANALYSER
    with THREAD_LOCK:
        if FACE_ANALYSER is None:
            FACE_ANALYSER = insightface.app.FaceAnalysis(
                name='buffalo_l', providers=execution_providers)
            FACE_ANALYSER.prepare(ctx_id=0)
    return FACE_ANALYSER
```

Explanation

This function returns the global variable **FACE_ANALYSER**, which is an instance of the **insightface.app.FaceAnalysis** class . It uses a thread lock (**THREAD_LOCK**) to ensure that the initialization of **FACE_ANALYSER** is thread-safe.If **FACE_ANALYSER** is **None**, it initializes it with a new instance of **FaceAnalysis** from the InsightFace library, using the specified parameters (name, providers,ctx_id).The **prepare** method is called on the **FACE_ANALYSER** instance, which likely involves preparing the model for face analysis. The resulting **FACE_ANALYSER** instance is returned.

```
def get_many_faces(frame: Frame):
    try:
        return get_face_analyser().get(frame)
    except ValueError:
        return None
def find_similar_face(frame: Frame, reference_face: Face) -> Optional[Face]:
    many_faces = get_many_faces(frame)
    if many_faces:
        for face in many_faces:
            if hasattr(face, 'normed_embedding') and hasattr(reference_face, 'normed_embedding'):
                distance = numpy.sum(numpy.square(face.normed_embedding -
reference_face.normed_embedding))
                if distance < similar_face_distance:
                    return face
    return None
```

Explanation

The **`find_similar_face`** function aims to identify a face in a given frame that is similar to a reference face. It starts by obtaining multiple faces from the input frame using the **`get_many_faces`** function. If there are faces detected (**`many_faces`** is not empty), the function iterates through each face and checks if both the current face and the reference face have a **`normed_embedding`** attribute. The **`normed_embedding`** likely represents a normalized embedding vector for the face obtained through a face analysis model. The function then calculates the Euclidean distance between the embeddings of the current face and the reference face using the numpy library. If the calculated distance is less than the predefined **`similar_face_distance`**, it implies a similarity, and the current face is returned. If no similar face is found among the detected faces, the function returns **`None`**. The **`get_many_faces`** function is assumed to be responsible for extracting and returning a list of faces from the input frame, but its implementation details are not provided in the provided code snippet.

```
def process_Frame(src_face:Face,refer_face:Face,frame:Frame):
    many_faces = get_many_faces(frame)
    if many_faces:
        for target_face in many_faces:
            frame = enhance_face(target_face,frame)
    return frame
```

Explanation

The **process_Frame** function appears to be designed to process an input frame by enhancing the appearance of faces detected within it. The function takes three parameters: **src_face**, **refer_face**, and **frame**, which are assumed to represent face objects and an image frame, respectively.First, it calls the **get_many_faces** function to detect and retrieve a list of faces (**many_faces**) present in the input **frame**. If there are faces detected (i.e., **many_faces** is not empty), the function iterates through each detected face (**target_face**) and calls the **enhance_face** function. The purpose of the **enhance_face** function is to enhance the appearance of the specified face region in the **frame**. This enhancement could involve various image processing techniques, possibly using a face enhancement model such as GFPGAN. The enhanced face is then pasted back into the original frame.In summary, the **process_Frame** function processes a given frame by detecting faces, iterating over each detected face, and enhancing its appearance in the frame using the **enhance_face** function. The final processed frame, with enhanced faces, is returned. Note that the specific implementation details of the **get_many_faces** and **enhance_face** functions are assumed to be defined elsewhere in the code.

```
def get_one_face(frame: Frame, position: int = 0):
    many_faces = get_many_faces(frame)
    if many_faces:
        try:
```

```
            return many_faces[position]
        except IndexError:
            return many_faces[-1]
    return None
```

Explaination

The **get_one_face** function is designed to retrieve a single face from an input frame. It takes two parameters: **frame**, which represents the image frame, and **position**, an optional parameter with a default value of 0, indicating the position of the desired face in the list of detected faces.The function starts by calling the **get_many_faces** function to detect and obtain a list of faces (**many_faces**) present in the input **frame**. If there are faces detected (i.e., **many_faces** is not empty), the function attempts to access the face at the specified **position** within the list.If the face at the specified **position** exists in the list, it is returned.If the specified **position** is out of bounds (i.e., an **IndexError** occurs), the function returns the last face in the list (**many_faces[-1]**), effectively providing a fallback when the specified position exceeds the number of detected faces.If no faces are detected in the input frame, the function returns **None**.

```
FACE_SWAPPER = None

def get_face_swapper():
    global FACE_SWAPPER
    with THREAD_LOCK:
    if FACE_SWAPPER is None:
        execution_providers = ['CPUExecutionProvider']
        model_path = resolve_relative_path('./models/inswapper_128.onnx')
        FACE_SWAPPER = insightface.model_zoo.get_model(model_path, providers=execution_providers)
    return FACE_SWAPPER
```

Explanation

The code defines a global variable `FACE_SWAPPER` and a function `get_face_swapper` to obtain an instance of a face-swapping model. The function utilizes a thread lock (`THREAD_LOCK`) to ensure thread safety during the initialization process. If the global variable `FACE_SWAPPER` is `None`, indicating that the face-swapper has not been instantiated, the function proceeds to initialize it. The face-swapper is created using the InsightFace library's `get_model` function, which loads a pre-trained model from a specified path (`./models/inswapper_128.onnx`). Additionally, it specifies execution providers, such as 'CPUExecutionProvider', for the model. Once the face-swapper is initialized, it is stored in the global variable `FACE_SWAPPER`, and subsequent calls to `get_face_swapper` return the existing instance. This design ensures that only one instance of the face-swapper is created, promoting efficiency and thread safety in a multi-threaded environment.

```
def swap_face(source_face: Face, target_face: Face, temp_frame: Frame):
```

```
return get_face_swapper().get(temp_frame, target_face, source_face, paste_back=True)
```

Explanation

The **`swap_face`** function utilizes a face-swapping model to replace the face in a target region of an input frame **(`temp_frame`)** with another face. The function takes three parameters: **`source_face`**, **`target_face`**, and **`temp_frame`**, representing the source face to be swapped, the target face region in the frame, and the input frame, respectively . The core functionality of face swapping is achieved by calling the **`get_face_swapper()`** function, which provides an instance of the face-swapping model. The `get` method of the face-swapper is then invoked with the specified parameters:
`temp_frame`: The input frame where the face swapping will be performed.
`target_face`: The target face, which indicates the region in the **`temp_frame`** where the face will be replaced.**`source_face`:** The source face, which is the face that will be used for the replacement. The **`paste_back=True`** argument suggests that the replacement face should be seamlessly pasted back into the specified target region in the frame . The overall result is a frame **(`temp_frame`)** with the face in the target region replaced by the source face, effectively achieving the face-swapping operation. The face-swapping model is retrieved using the **`get_face_swapper`** function, ensuring that the model is initialized and shared across multiple calls to the **`swap_face`** function for efficiency and thread safety.

```
def main(src_path:str, target_path:str):
    src_Frame = src_path
    tar_frame = target_path
    print(src_Frame)
    print(tar_frame)
    processed_image_src = process_Frame(None,None,cv2.imread(src_Frame))
    processed_image_tar = process_Frame(None,None,cv2.imread(tar_frame))
    pro_tar = r"./output/pro_tar.jpg"
    pro_src = r"./output/pro_src.jpg"
    cv2.imwrite(pro_src,processed_image_src)
    cv2.imwrite(pro_tar,processed_image_tar)
    src_face = get_one_face(cv2.imread(pro_src))
    pro_tar = cv2.imread("./output/pro_tar.jpg")
    i=  0
    many_faces = get_many_faces(pro_tar)
    for tar_face in many_faces:
        pro_tar= swap_face(src_face,tar_face,pro_tar)
        file_name = f"./first-op-m.jpg"
        i+=1
        pro_img = process_Frame(None,None,pro_tar)
        cv2.imwrite(file_name,pro_img)
    print("complete")
```

Explanation

The paths to the source and target images are stored in variables **src_Frame** and **tar_frame**, respectively.The paths to the source and target images are printed to the console.The source and target images are read using OpenCV's **cv2.imread** function, and the **process_Frame** function is called to enhance the faces in both images. The processed images are stored in **processed_image_src** and **processed_image_tar**.The paths for saving the processed source and target images are defined, and the processed images are saved using **cv2.imwrite**. The source face (**src_face**) is obtained by calling **get_one_face** on the processed source image.The target image is read again, and faces in the target image are detected using **get_many_faces**.A loop iterates over each target face, performing face swapping using the **swap_face** function.The resulting frame is further processed, and the final images are saved with file names like **first-op-0.jpg**, **first-op-1.jpg**, and so on.

3.3 UI

```
class ImageSubmitApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Image Submission")
        self.root.configure(bg="#2C3E50", padx=20, pady=20)  # Set background color and padding

        self.image_1 = None
        self.image_2 = None

        ttk.Label(root, text="Source", background="#2C3E50", foreground="white", font=('Helvetica',
12)).grid(row=0, column=0, pady=10, padx=10, sticky='e')
        ttk.Label(root, text="Target", background="#2C3E50", foreground="white", font=('Helvetica',
12)).grid(row=0, column=1, pady=10, padx=0, sticky='e')
        ttk.Label(root, text="Result", background="#2C3E50", foreground="white", font=('Helvetica',
12)).grid(row=0, column=6, pady=10, padx=10, sticky='e')

        self.image_label_1 = ttk.Label(root, background="#2C3E50", cursor="hand2")
        self.image_label_1.grid(row=1, column=0, pady=10, padx=10, sticky='w')

        self.image_label_2 = ttk.Label(root, background="#2C3E50", cursor="hand2")
        self.image_label_2.grid(row=1, column=1, pady=10, padx=10, sticky='w')

        self.deepfake_label = ttk.Label(root, text="Deepfaked Image", background="#2C3E50",
foreground="white", font=('Helvetica', 12))
        self.deepfake_label.grid(row=1, column=6, pady=10, padx=10, sticky='w')

        ttk.Button(root, text="Browse Image 1", command=self.browse_image_1,
style="TButton").grid(row=2, column=0, pady=10, padx=10, sticky='w')
        ttk.Button(root, text="Browse Image 2", command=self.browse_image_2,
style="TButton").grid(row=2, column=2, pady=10, padx=10, sticky='w')
```

```
        ttk.Button(root, text="Submit", command=self.submit_images, style="TButton").grid(row=3,
column=1, pady=20, padx=10, sticky='nsew')

        for i in range(7):
            root.columnconfigure(i, weight=1)
        for i in range(4):
            root.rowconfigure(i, weight=1)

    def browse_image_1(self):
        file_path = filedialog.askopenfilename(filetypes=[("Image files", "*.png;*.jpg;*.jpeg;*.gif;*.webp")])
        if file_path:
            self.image_1 = file_path
            self.display_image_1(Image.open(file_path), self.image_label_1)

    def browse_image_2(self):
        file_path = filedialog.askopenfilename(filetypes=[("Image files", "*.png;*.jpg;*.jpeg;*.gif")])
        if file_path:
            self.image_2 = file_path
            self.display_image_2(Image.open(file_path), self.image_label_2)

    def submit_images(self):
        if self.image_1 and self.image_2:
            deepfaked_image = deepfake_function(self.image_1, self.image_2)
            self.display_deepfaked_image(Image.open(deepfaked_image), self.deepfake_label)
            tk.messagebox.showinfo(title="Completion", message="Completed 👍")
        else:
            tk.messagebox.showerror(title="Error", message="Please select both Image 1 and Image 2.")

    def display_image_1(self, image, label):
        self.display_image(image, label, column=0)

    def display_image_2(self, image, label):
        self.display_image(image, label, column=2)

    def display_deepfaked_image(self, image, label):
        self.display_image(image, label, column=7)

    def display_image(self, image, label, column):
        aspect_ratio = image.width / image.height
        new_width = 200
        new_height = int(new_width / aspect_ratio)
        image = image.resize((new_width, new_height), Image.ANTIALIAS)

        img = ImageTk.PhotoImage(image)

        label.configure(image=img)
        label.image = img
        label.grid(row=1, column=column, pady=10, padx=10, sticky='w')
```

```
    def on_hover_enter(self, widget):
        widget.configure(background="#6C5B7B")

    def on_hover_leave(self, widget):
        widget.configure(background="#2C3E50")

def deepfake_function(image_path_1, image_path_2):
    main(image_path_1, image_path_2)
    return "./first-op-m.jpg"

if __name__ == "__main__":
    root = tk.Tk()

    style = ttk.Style()
    style.theme_use("clam")
    style.configure("TButton", background="#3498DB", foreground="white", padding=5)

    app = ImageSubmitApp(root)
    root.mainloop()
```

Explanation

Here the buttons are mapped to the corresponding function's

4 OUTPUTS

4.1 Using Tkinter

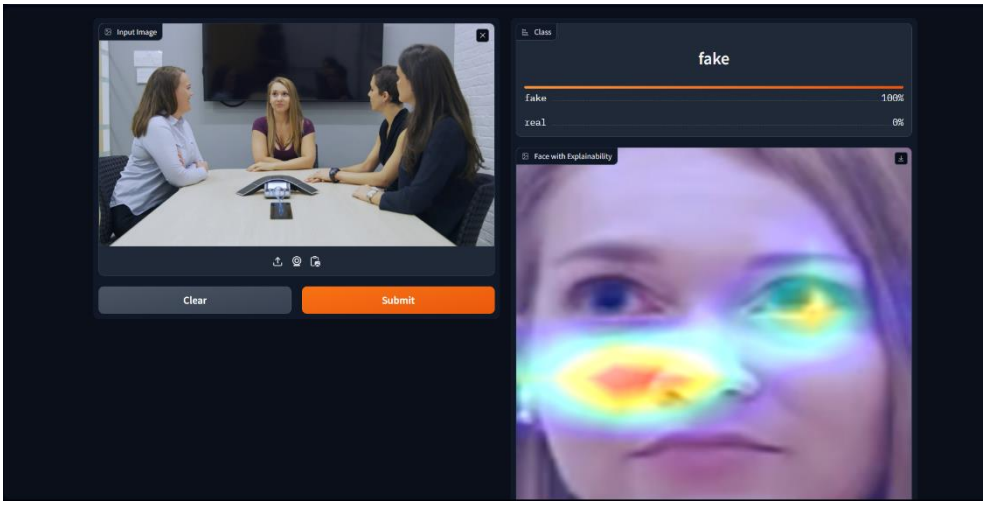


Figure 7 Output from detection

Before

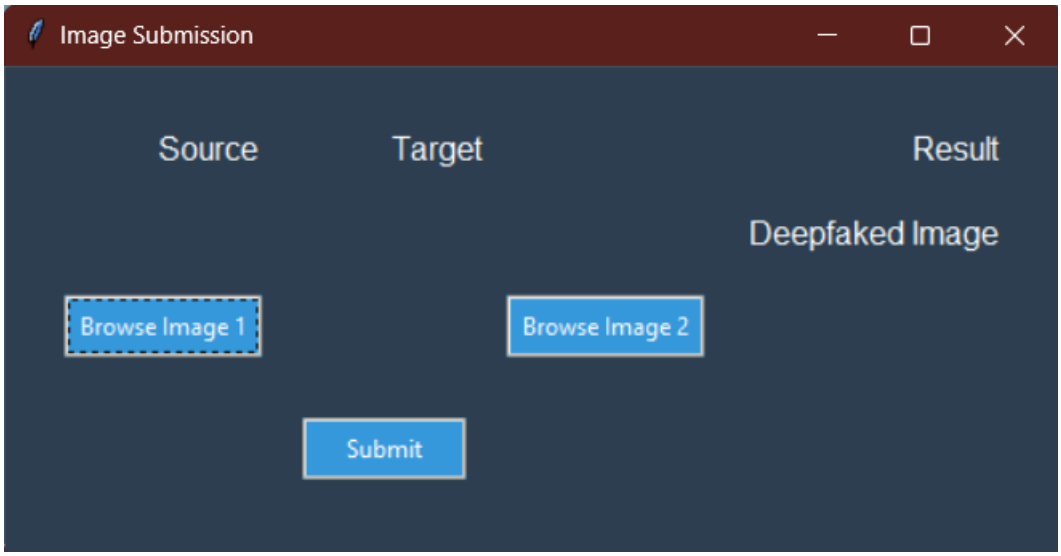


Figure 8 Output before image selection

After

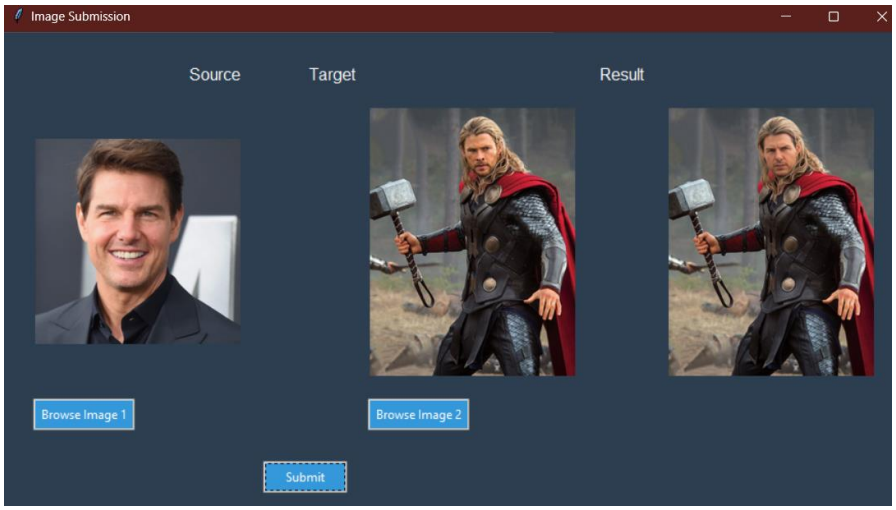


Figure 9 Result from Tkinter

The difference between real and fake images produced by this Algorithm by simple histogram plots

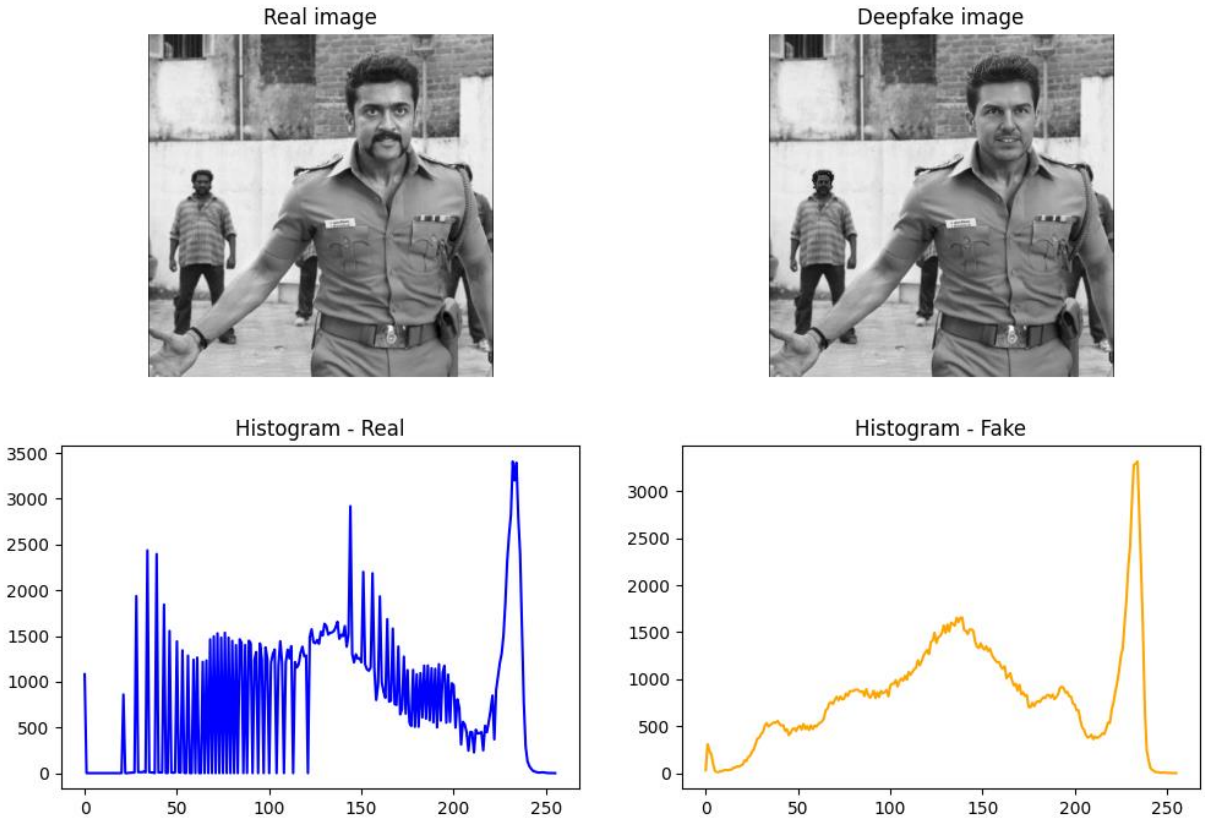


Figure 10 Histogram difference between real and fake- I

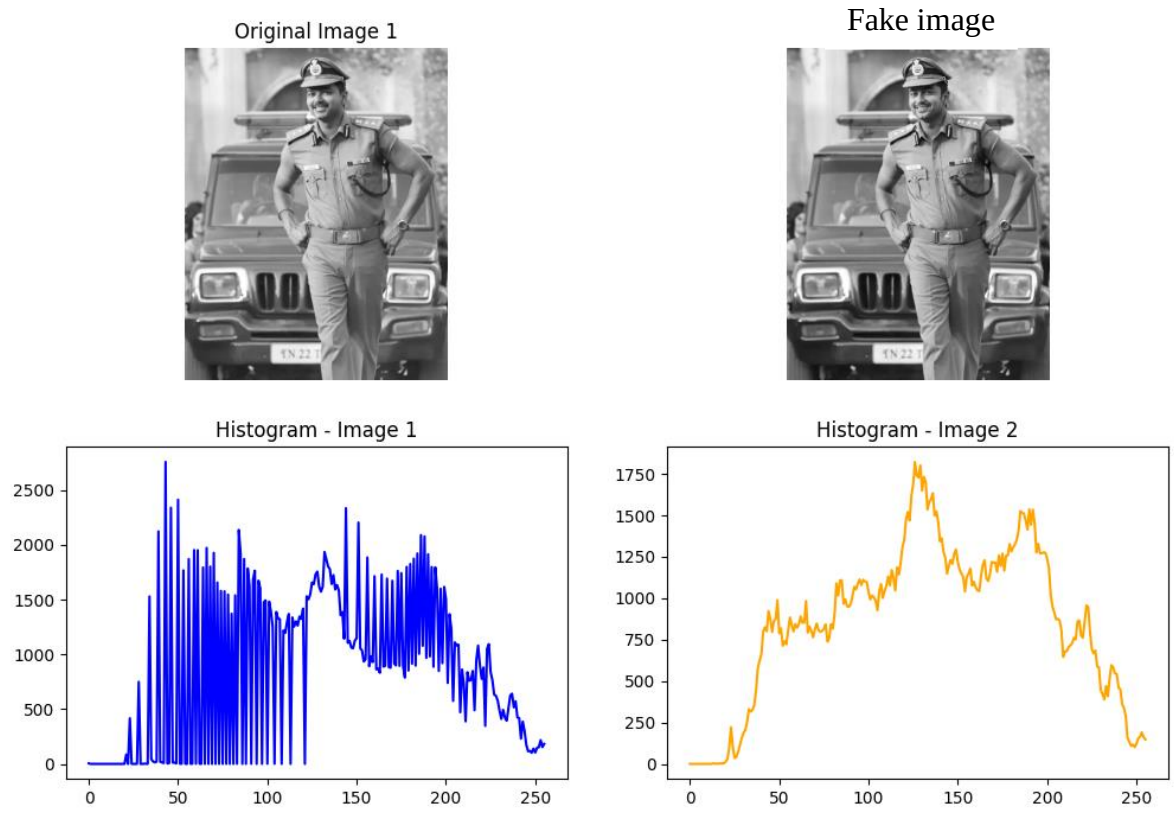


Figure 11 Histogram difference between real and fake - II

But fails to show the difference between high-resolution picture

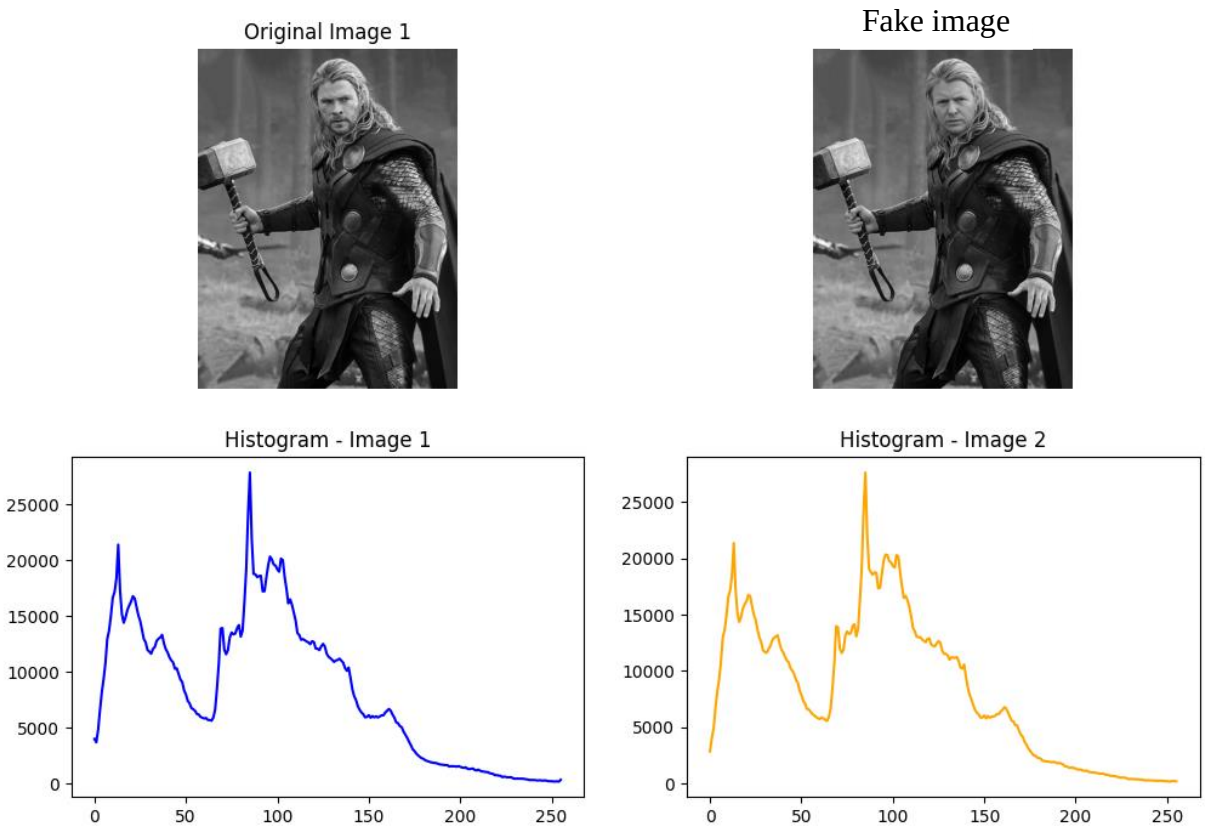


Figure 12 Histogram difference between real and fake - III

5.2 Using flask

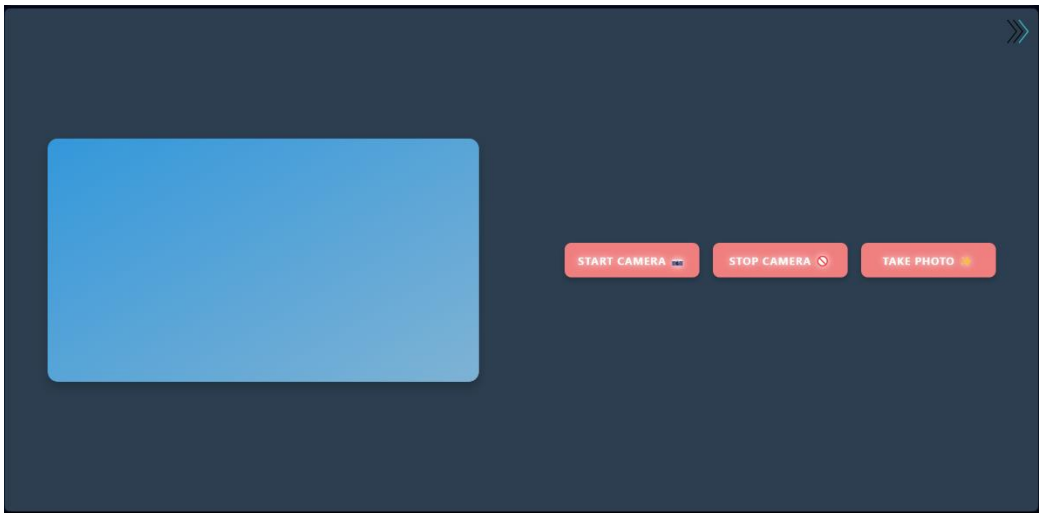


Figure 13 Camera application using webpage

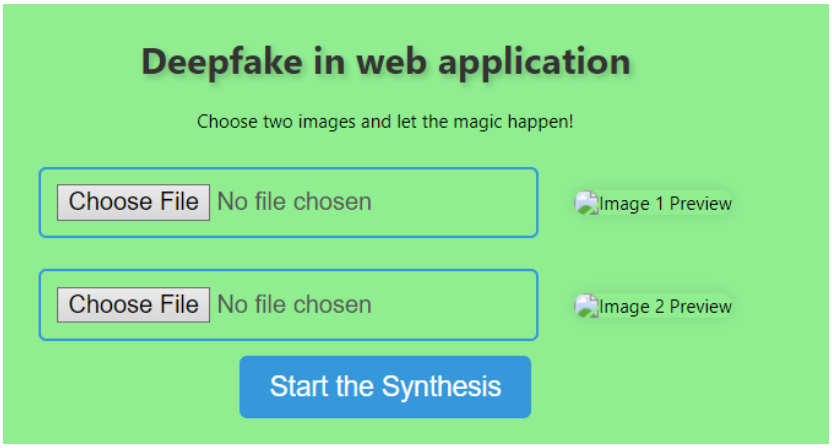


Figure 14 Web application for deepfake generation

Result's display

Figure 15 Application of the second webpage

6 Final codes

6.1 Flask code

```
from flask import Flask, render_template
from generator import main
import time
app = Flask(__name__)


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

```
@app.route('/upload', methods=['GET', 'POST'])
def upload():
    image1 = r"C:/Users/eshwa/Downloads/synthesized_image1.png"
    image2 = r"C:/Users/eshwa/Downloads/synthesized_image2.png"
    time.sleep(5)
    print("sucess")
    main(image1, image2)
    print("sucess2")
    return render_template('upload.html')


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

6.2 Html For base webpage

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Dazzling Image Synthesizer</title>
    <style>
      body {
        font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
        text-align: center;
        margin: 20px;
        background-color: lightgreen;
      }
      h1 {
        color: #333;
        text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.2);
      }
      input {
        margin: 10px;
        padding: 10px;
        border: 2px solid #3498db;
        border-radius: 5px;
        font-size: 16px;
        color: #555;
      }
      img {
        max-width: 100%;
        max-height: 300px;
        margin: 10px;
        box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
```

```
      border-radius: 8px;
    }
    button {
      background-color: #3498db;
      color: #fff;
      padding: 10px 20px;
      font-size: 18px;
      border: none;
      border-radius: 5px;
      cursor: pointer;
      transition: background-color 0.3s ease;
    }
    button:hover {
      background-color: #207cca;
    }
  </style>
</head>
<body>
  <h1>Deepfake in web application</h1>
  <p>Choose two images and let the magic happen!</p>
  <form action="/upload">
    <input type="file" id="image1Input" accept="image/*" />
    <img id="image1Preview" alt="Image 1 Preview" />
    <br />
    <input type="file" id="image2Input" accept="image/*" />
    <img id="image2Preview" alt="Image 2 Preview" />
    <br />
    <button onclick="downloadImages()">Start the Synthesis</button>
  </form>

  <script>
    function previewImage(inputId, previewId) {
      const input = document.getElementById(inputId)
      const preview = document.getElementById(previewId)
      const file = input.files[0]

      if (file) {
        const reader = new FileReader()
        reader.onload = function (e) {
          preview.src = e.target.result
        }
        reader.readAsDataURL(file)
      }
    }

    function downloadImages() {
      const image1Input = document.getElementById('image1Input')
      const image2Input = document.getElementById('image2Input')
```

```
      if (!image1Input.files[0] || !image2Input.files[0]) {
        alert('Please select both images.')
        return
      }

      const image1Url = URL.createObjectURL(image1Input.files[0])
      const image2Url = URL.createObjectURL(image2Input.files[0])

      // Create links to download the images
      const link1 = document.createElement('a')
      link1.href = image1Url
      link1.download = 'synthesized_image1.png'
      link1.click()

      const link2 = document.createElement('a')
      link2.href = image2Url
      link2.download = 'synthesized_image2.png'
      link2.click()

      fetch('/upload', {
        method: 'POST'
      })
    }

    document
      .getElementById('image1Input')
      .addEventListener('change', function () {
        previewImage('image1Input', 'image1Preview')
      })

    document
      .getElementById('image2Input')
      .addEventListener('change', function () {
        previewImage('image2Input', 'image2Preview')
      })
  </script>
 </body>
</html>
```

6.3 For Upload HTML second webpage

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
<title>Deepfaked Image</title>
<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<style>
 $hoverEasing: cubic-bezier(0.23, 1, 0.32, 1);
 $returnEasing: cubic-bezier(0.445, 0.05, 0.55, 0.95);

 body {
  margin: 40px 0;
  font-family: 'Raleway';
  font-size: 14px;
  font-weight: 500;
  background-color: black;
  -webkit-font-smoothing: antialiased;
 }

 .title {
  font-family: 'Raleway';
  font-size: 24px;
  font-weight: 700;
  color: #5d4037;
  text-align: center;
 }

 p {
  line-height: 1.5em;
 }

 h1 + p,
 p + p {
  margin-top: 10px;
 }

 .container {
  padding: 40px 80px;
  display: flex;
  flex-wrap: wrap;
  justify-content: center;
 }

 .card-wrap {
  margin: 10px;
  transform: perspective(800px);
  transform-style: preserve-3d;
  cursor: pointer;
  //background-color: #fff;

  &:hover {
    .card-info {
```

```
      transform: translateY(0);
    }
    .card-info p {
      opacity: 1;
    }
    .card-info,
    .card-info p {
      transition: 0.6s $hoverEasing;
    }
    .card-info:after {
      transition: 5s $hoverEasing;
      opacity: 1;
      transform: translateY(0);
    }
    .card-bg {
      transition: 0.6s $hoverEasing, opacity 5s $hoverEasing;
      opacity: 0.8;
    }
    .card {
      transition: 0.6s $hoverEasing, box-shadow 2s $hoverEasing;
      box-shadow: rgba(white, 0.2) 0 0 40px 5px, rgba(white, 1) 0 0 0 1px,
        rgba(black, 0.66) 0 30px 60px 0, inset #333 0 0 0 5px,
        inset white 0 0 0 6px;
    }
  }
}

/* Customizing the scrollbar */

.card {
  position: relative;
  flex: 0 0 240px;
  width: 240px;
  height: 320px;
  background-color: #333;
  overflow: hidden;
  border-radius: 10px;
  box-shadow: rgba(black, 0.66) 0 30px 60px 0, inset #333 0 0 0 5px,
    inset rgba(white, 0.5) 0 0 0 6px;
  transition: 0.5s cubic-bezier(0.445, 0.05, 0.55, 1);
}

.card-bg {
  opacity: 0.5;
  position: absolute;
  top: -20px;
  left: -20px;
  width: 100%;
```

```
  height: 100%;
  padding: 20px;
  background-repeat: no-repeat;
  background-position: center;
  background-size: cover;
  transition: 1s cubic-bezier(0.445, 0.05, 0.55, 0.95), opacity 5s 1s cubic-bezier(0.445, 0.05, 0.55, 0.95);
  pointer-events: none;
 }

 .card-info {
  padding: 20px;
  position: absolute;
  bottom: 0;
  color: #fff;
  transform: translateY(40%);
  transition: 0.6s 1.6s cubic-bezier(0.215, 0.61, 0.355, 1);

  p {
   opacity: 0;
   text-shadow: rgba(black, 1) 0 2px 3px;
   transition: 0.6s 1.6s cubic-bezier(0.215, 0.61, 0.355, 1);
  }

  * {
   position: relative;
   z-index: 1;
  }

  &:after {
   content: '';
   position: absolute;
   top: 0;
   left: 0;
   z-index: 0;
   width: 100%;
   height: 100%;
   background-image: linear-gradient(
    to bottom,
    transparent 0%,
    rgba(#000, 0.6) 100%
   );
   background-blend-mode: overlay;
   opacity: 0;
   transform: translateY(100%);
   transition: 5s 1s cubic-bezier(0.445, 0.05, 0.55, 0.95);
  }
 }
```

```
    .card-info h1 {
      font-family: 'Playfair Display';
      font-size: 36px;
      font-weight: 700;
      text-shadow: rgba(black, 0.5) 0 10px 10px;
    }
    body {
      text-align: center;
      background-color: black;
    }

    #image-container {
      margin: 20px;
    }

    #creative-title {
      font-size: 24px;
      font-weight: bold;
      color: #333;
    }

    ::-webkit-scrollbar {
      width: 8px;
    }

    ::-webkit-scrollbar-thumb:hover {
      background-color: #555;
    }

    /* Handle */
    ::-webkit-scrollbar-thumb {
      background-color: #888;
      border-radius: 4px;
    }

    ::-webkit-scrollbar-track {
      background-color: #f1f1f1;
    }
  </style>
</head>
<body>
  <h1 class="title" style="color: white">Output's from deepfake</h1>

  <div id="app" class="container">
    <card data-image="../static/first-op-m.jpg">
      <h1 slot="header">Latest</h1>
      <p slot="content">
        Lorem ipsum dolor sit amet, consectetur adipisicing elit.
```

```
    </p>
  </card>
  <card data-image="../static/image-4.jpg">
    <h1 slot="header">Previous</h1>
    <p slot="content">Donald on Thor a fictional superhero</p>
  </card>
  <card data-image="../static/image-2.jpg">
    <h1 slot="header">Previous</h1>
    <p slot="content">Donald trump's face on rashmika.</p>
  </card>
  <card data-image="../static/image-3.jpg">
    <h1 slot="header">Previous</h1>
    <p slot="content">Donald trump's face on captain america</p>
  </card>
</div>
<div id="image-container">
  <img src="../static/first-op-m.jpg" alt="Deepfaked Image" />
  <div id="creative-title" style="color: white">Deepfaked Image</div>
</div>
<script>
  Vue.config.devtools = true

  Vue.component('card', {
    template: `
<div class="card-wrap"
  @mousemove="handleMouseMove"
  @mouseenter="handleMouseEnter"
  @mouseleave="handleMouseLeave"
  ref="card">
  <div class="card"
    :style="cardStyle">
    <div class="card-bg" :style="[cardBgTransform, cardBgImage]"></div>
    <div class="card-info">
      <slot name="header"></slot>
      <slot name="content"></slot>
    </div>
  </div>
</div>`,
    mounted() {
      this.width = this.$refs.card.offsetWidth
      this.height = this.$refs.card.offsetHeight
    },
    props: ['dataImage'],
    data: () => ({
      width: 0,
      height: 0,
      mouseX: 0,
      mouseY: 0,
```

```
    mouseLeaveDelay: null
  }),
  computed: {
    mousePX() {
      return this.mouseX / this.width
    },
    mousePY() {
      return this.mouseY / this.height
    },
    cardStyle() {
      const rX = this.mousePX * 30
      const rY = this.mousePY * -30
      return {
        transform: `rotateY(${rX}deg) rotateX(${rY}deg)`
      }
    },
    cardBgTransform() {
      const tX = this.mousePX * -40
      const tY = this.mousePY * -40
      return {
        transform: `translateX(${tX}px) translateY(${tY}px)`
      }
    },
    cardBgImage() {
      return {
        backgroundImage: `url(${this.dataImage})`
      }
    }
  },
  methods: {
    handleMouseMove(e) {
      this.mouseX = e.pageX - this.$refs.card.offsetLeft - this.width / 2
      this.mouseY = e.pageY - this.$refs.card.offsetTop - this.height / 2
    },
    handleMouseEnter() {
      clearTimeout(this.mouseLeaveDelay)
    },
    handleMouseLeave() {
      this.mouseLeaveDelay = setTimeout(() => {
        this.mouseX = 0
        this.mouseY = 0
      }, 1000)
    }
  }
})

const app = new Vue({
  el: '#app'
```

```
    })
  </script>
 </body>
</html>
```

Conclusion

In conclusion, our team embarked on a comprehensive exploration of deepfake detection and generation, employing advanced models to address the growing concerns surrounding synthetic media manipulation. To tackle the ever-evolving landscape of deepfakes, we meticulously crafted two [1]Python codes that not only enhance the identification of manipulated content but also delve into the intricacies of generating synthetic media. The implementation involved leveraging cutting-edge techniques and models in the field, emphasizing the importance of staying ahead of the curve in the ongoing battle against misinformation and deceptive content. Through our collaborative efforts, we not only contributed to the advancement of deepfake [2]detection mechanisms but also gained valuable insights into the nuances of deepfake generation. As the digital landscape continues to evolve, our work serves as a testament to the significance of proactive measures in ensuring the integrity and authenticity of multimedia content.

References

[1] “gan-model-reference,” [Online]. Available: https://arxiv.org/pdf/2101.04061v2.pdf.

[2] “mtcnn-model,” [Online]. Available: https://medium.com/@iselagradilla94/multi-task-cascaded-convolutional-networks-mtcnn-for-face-detection-and-facial-landmark-alignment-7c21e8007923.