DF BUSTER - A DEEPFAKE VIDEO DETECTION APP

A Mini Project Report

Submitted in partial fulfilment of the requirements for

the award of the degree of

MASTER OF COMPUTER APPLICATIONS

Submitted by

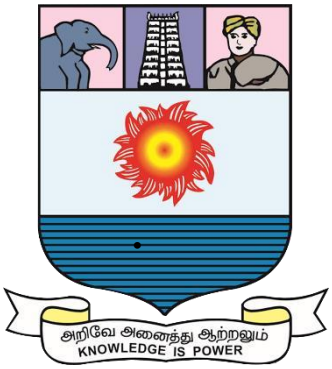
KANNAN M

(Reg. No: 24084015300111213)

Under the Guidance of

Dr. R.S. RAJESH B.E., M.E., Ph.D.

Professor



Department of Computer Science and Engineering

Manonmaniam Sundaranar University

Tirunelveli 627012

November 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MANONMANIAM SUNDARANAR UNIVERSITY

TIRUNELVELI - 12

BONAFIDE CERTIFICATE

This is to certify that the Mini Project report entitled **" DF BUSTER - A DEEPFAKE VIDEO DETECTION APP "** submitted in partial fulfilment of the degree of MASTER OF COMPUTER APPLICATIONS in the Department of Computer Science & Engineering, Manonmaniam Sundaranar University, done by **KANNAN M** Register No. **24084015300111213** is an authentic work carried out during the academic year 2025-26.

GUIDE **HEAD OF THE DEPARTMENT**

Submitted for the Viva – Voce Examination on: _____

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

I thank Lord Almighty, who blessed me with wonderful faculties, friends, and family members whose love and encouragement have given new significance to my accomplishment. I express my sincere thanks with love to my parents, who are adherent cheer leader and support throughout the project.

I whole-hearty thank **Dr. R. S. RAJESH M.E., Ph.D.,** Professor, Head of the Department of Computer Science and Engineering, Manonmaniam Sundaranar University for providing constructive criticism and suggestions for the successful completion of this project.

I thank **Dr. S.ANTELIN VIJILA M.E., Ph.D.,** Project Coordinator, Assistant Professor, Department of Computer Science and Engineering. Manonmaniam Sundaranar University, Tirunelveli.

I extend my gratitude to my guide **Dr. R. S. RAJESH M.E., Ph.D.,** Professor**,** Head of the Department of Computer Science and Engineering. Manonmaniam Sundaranar University for his valuable guidance and motivation throughout my project.

I thank all staff members of department for their kind advice and pleasing co-ordination throughout the course.

I also thank my friends who support with real encouragement and suggestions in each and every phase of my project.

DECLARATION

I hereby declare that the project work entitled **" DEEPFAKE BUSTER - A MOBILE BASED DEEPFAKE VIDEO DETECTION SYSTEM "** submitted to Department of Computer Science & Engineering , Manonmaniam Sundaranar University, Tirunelveli in partial fulfillment of the requirements for the award of the degree of Master of Computer Applications is a record of original project work done during the academic year 2025-2026 under the supervision and guidance of **Dr. R. S. RAJESH M.E., Ph.D.,** Professor, Head of the Department of Computer Science and Engineering, Manonmaniam Sundarnar University, Tirunelveli.

[M.KANNAN]

TABLE OF CONTENT

ABSTRACT

With the deepfake videos becoming more common and realistic, it's getting harder to tell what's real online, which can cause big problems like spreading fake news. This project, "**DeepFake Buster**" is a mobile app designed to help anyone easily check if a video is real or a deepfake. The goal is to empower regular people with a simple tool to spot manipulated media right on their phones, helping to fight back against misinformation. The app's core is a powerful, dual-model deep learning engine. This combines two distinct models: a custom-built Convolutional Neural Network (CNN) designed to spot specific deepfake artifacts, and the highly efficient MobileNetV2 architecture for more generalized analysis. Together, these models scan a video frame by frame to find the subtle imperfections that fake videos often contain. I built the app using Flutter so it works on both Android and iPhone, and used TensorFlow Lite to ensure the deep learning models run fast directly on the device. This on-device processing is crucial because it keeps everything completely private and secure. When you use the app, you get a simple, clear verdict ‘Real’ or ‘Fake’ along with a detailed report showing the confidence scores, so you can see how it made its decision. By putting this advanced deep learning technology in everyone's hands, DeepFake Buster is a practical tool that helps promote media literacy and makes our digital world a safer, more informed place.

CHAPTER 1 : INTRODUCTION

In today's digital world, it's getting incredibly hard to trust the videos we see online. A new kind of fake video, known as a **deepfake**, has become a major reason for this. A deepfake is a video that has been altered using powerful Artificial Intelligence. The most common type is a "face-swap," where someone's face is realistically placed onto another person's body, making it look like they said or did something they never actually did. Because these fakes look so real, they have become a serious problem and can be used to spread false news, create scams, or unfairly damage a person's reputation.While experts may have special tools to spot these fakes, the average person is usually left guessing. To solve this problem, this project introduces **"Deepfake Buster"**. It is a user-friendly mobile application designed to put the power of deepfake detection directly into anyone's hands. At its heart, the app uses a smart, two-part deep learning system to analyze videos. All the hard work happens right on your phone, which means your videos are kept completely private and the app works even without an internet connection. This report will walk through the entire process of creating Deepfake Buster, starting from how the data was prepared and the models were trained, all the way to building the final mobile app and proving that it works effectively.

Problem Statement

The biggest problem with deepfakes today is that while it's getting easier for people to create them, there are no simple or safe tools for the average person to detect them. This creates a dangerous gap where fake videos can spread without anyone being able to verify them. Most existing detection methods have major issues: they often require you to upload your personal video to a company's server, which is a huge privacy risk. On top of that, the powerful deep learning models needed to spot these fakes are typically too big and slow to run properly on a normal smartphone. Finally, even if a tool is available, it's often too complicated for someone without a technical background to use. Therefore, the central challenge this project solves is the lack of a secure, fast, and user-friendly mobile app that lets anyone check for deepfakes right on their own phone.

Objective

- Develop **'Deepfake Buster',** a fully functional mobile application for deepfake video detection.
- Ensure the app is **extremely user-friendly** and accessible to people with **no technical expertise.**
- Design the system to run **entirely on-device**, with **no external server dependency.**
- Guarantee **complete user privacy and data security** — personal videos never leave the smartphone.
- Enable **offline functionality**, allowing users to analyze videos **anytime, anywhere** without internet access.
- Provide the **general public with a safe, portable, and effective** deepfake detection tool.

CHAPTER 2 : LITERATURE SURVEY

This chapter provides a comprehensive review of the foundational concepts and existing technologies relevant to understand the work behind ‘Deepfake Buster’. it's helpful to look at what other researchers have already done. This section reviews important ideas and studies that influenced this project, from how deepfakes are made and detected to the specific technologies used to build the app.

2.1 Understanding Deepfakes and Detection Strategies

Deepfakes are fake videos created using artificial intelligence (AI), where existing images and videos are combined to create realistic but fake content. The results can be so convincing that they are often impossible to spot with the naked eye. This has become a serious problem because these videos can be used for malicious purposes like creating political unrest, spreading false information, or blackmailing people. To combat this, researchers have developed various detection strategies, which can be grouped into two main approaches.

Signature-Based and Inconsistency Analysis

This approach is like being a detective looking for specific clues or mistakes that the deepfake-making AI leaves behind. These clues are often so subtle that a person wouldn't notice them just by watching the video.

- **Biological Inconsistencies:** Early deepfake models often failed to replicate small but important human behaviors. Researchers developed methods to detect things like unnatural eye-blinking patterns, as many early deepfakes couldn't produce a blinking face. Other methods looked for inconsistent head movements or a lack of realistic pulse signals in the skin. However, as deepfake AIs have improved, they have learned to include these details, making these methods less reliable today.
- **Technical Artifacts:** This method looks for technical glitches in the video. For example, a detector might look for blurry boundaries where a fake face was pasted onto a body, a color mismatch between the fake face and the original skin, or visible parts of the original face flickering for a frame. Other advanced techniques analyze digital "fingerprints" left behind by the AI that created the fake. While these methods work well for older or lower-quality deepfakes, they aren't very useful in the long run because deepfake creators just update their AIs to stop making that specific mistake.

2.2 Using Deep Learning for Detection

To keep up with improving deepfake technology, most modern research has turned to deep learning, especially using **Convolutional Neural Networks (CNNs)**. A CNN is a type of deep learning model that is excellent at finding patterns in images. Instead of manually looking for clues, a CNN can be trained on thousands of real and fake images to learn the subtle differences on its own.

Several researchers have demonstrated the success of this approach. **Anuj Badale, Chaitanya Darekar, and others** showed that a combination of CNN and Dense layers could be used to classify deepfakes, achieving up to 91% accuracy in their experiments. Their work involved creating a dataset, preprocessing the images, and training a CNN model, a pipeline very similar to the one used in this project. Similarly, in a project by **Navdeep Singh Hada**, a combination of a ResNet50 (a powerful type of CNN) and an LSTM (for analyzing sequences of frames) was used to reach a high accuracy of 94.63%. These studies confirm that deep learning models are the current state-of-the-art for effectively identifying manipulated videos.

2.3 Challenges of On-Device Detection

While powerful, many deep learning models can be very large and slow, making them computationally expensive. This makes them difficult to run on a smartphone, which has limited processing power and memory compared to a powerful server. As authors **Gustavo B. Souza and others** point out, many proposed deepfake detection models are "not suitable for limited hardware devices". This is a major barrier to creating an accessible tool for the general public.

To solve this, researchers have explored creating more efficient models. One paper discusses using a "Width Extended" CNN (wCNN) which is designed to be less complex while still providing good results on mobile devices. Another key technology is TensorFlow Lite, which is a tool that optimizes large deep learning models to make them smaller and faster so they can run efficiently on a phone. The goal of using these technologies is to make detection possible "on-the-fly" without needing to upload videos to a server, which directly addresses the privacy and performance issues of other solutions.

2.4 Identification of Research Gap

After reviewing all the existing research, it's clear there's a big gap between the powerful deepfake detection tools created by scientists and what's actually available for everyday people to use. While researchers have made highly accurate models, these models are almost always designed to run on powerful servers, not on a normal smartphone.

This project identifies three specific problems with the current situation:

- **Lack of Privacy:** Most available solutions require you to upload your video to a third-party server for analysis. This is a huge privacy risk, as you lose control over your personal data.
- **Too Slow and Big for Phones:** The deep learning models that are best at spotting fakes are often very large and require a lot of processing power, making them too slow and clunky to run on a standard mobile device.
- **Too Complicated for Regular Users:** There isn't a simple, user-friendly mobile app that a non-technical person can easily download and use to check a video's authenticity.

2.5 Building a Complete System

The "Deepfake Buster" project was designed to solve the major problems with current deepfake detection by creating a complete system that is accurate, fast, private, and easy for anyone to use. The final piece of this puzzle is a user-friendly mobile application built using Flutter, a modern framework that allows the app to work on both Android and iOS from a single codebase. By combining this Flutter front-end with optimized TensorFlow Lite models, it's possible to create a practical tool where all the complex analysis happens directly on the user's phone. The process follows a clear workflow, similar to the system described by Navdeep Singh Hada: it splits a video into frames, detects and crops the faces, feeds them to a deep learning model for classification, and finally presents a simple "Fake or Real" verdict to the user. By building on successful deep learning research while specifically addressing the challenges of on-device performance and user privacy, the "Deepfake Buster" project bridges the gap between advanced academic tools and a practical, much-needed solution for the public.

CHAPTER 3 : SYSTEM SPECIFICATION

This chapter details the technical specifications that were required to build and run the 'Deepfake Buster' application. It covers the specific hardware and software needed for both the end-users who will run the app on their phones, and for the developers who created the project. Additionally, it provides a description of the core software technologies that were used to bring the application to life.

3.1 Hardware Specifications

a. **End-User Side (Minimum Recommended):**
   - **Operating System:** Android 11 (API level 29) / iOS 15.0
   - **Processor:** 64-bit octa-core processor (e.g., Qualcomm Snapdragon 7-series, Google Tensor, Apple A13 Bionic or newer)
   - **RAM:** 4 GB
   - **Storage:** 550 MB of available space for application installation.

b. **Development Side (Recommended):**
   - **Processor:** 8-core CPU (e.g., Intel Core i7 / AMD Ryzen 7 or better)
   - **RAM:** 16 GB or more
   - **Graphics Processing Unit (GPU):** NVIDIA GeForce RTX 3060 (8 GB VRAM) or higher, for efficient deep learning model training.
   - **Storage:** 512 GB Solid State Drive (SSD) or larger.

3.2 Software Specifications

a. **End-User Side:**
   - A standard, unmodified installation of Android or iOS.
   - Permissions granted by the user for accessing local file storage.

b. **Development Side:**
   - **Operating System:** Windows 11, macOS Ventura, or Ubuntu 22.04 LTS.
   - **Model Development Environment:**
     - Python 3.10.x
     - TensorFlow 2.15.x, Keras 2.15.x
     - OpenCV-Python 4.8.x
     - Scikit-learn, NumPy, Pandas, Matplotlib

- **Application Development Environment:**
    - Flutter SDK 3.19.x
    - Dart SDK 3.4.x
    - Android Studio "Iguana" (2023.2.1) or Visual Studio Code with Flutter and Dart extensions.
    - Xcode 15+ (for iOS development).

3.3 Software Description

The development of the "Deepfake Buster" system leverages a modern, dual-stack toolchain, strategically dividing the workload between a robust Python-based environment for machine learning and a high-performance Dart-based framework for mobile application development. Each software component was selected for its specific capabilities in contributing to the project's objectives of accuracy, efficiency, and usability.

i) Model Development and Training Stack (Python Ecosystem)

- **Python:** Serving as the foundational language for the entire machine learning pipeline, Python (version 3.10.x) was chosen for its clean syntax and, more importantly, its unparalleled ecosystem of scientific computing and AI libraries. Its role was central to every phase of model development, from initial data ingestion and preprocessing to model architecture design, training, and final evaluation.
- **TensorFlow:** As the core deep learning framework, TensorFlow (version 2.15.x) provided the end-to-end platform for all neural network operations. Its powerful backend, capable of leveraging GPU acceleration via CUDA, was essential for efficiently training the deep CNN models on the large video dataset. TensorFlow's computation graph abstraction allowed for the construction and optimization of complex model architectures.
- **Keras:** Utilized as a high-level API within TensorFlow, Keras was instrumental in accelerating the model development process. Its user-friendly, modular approach was used to define the layers of both the custom CNN (Model 1) and the transfer learning architecture based on MobileNetV2 (Model 2). This allowed for rapid prototyping, experimentation, and fine-tuning of network parameters.

- **OpenCV (Open Source Computer Vision Library):** This indispensable computer vision library was the workhorse for all video and image manipulation tasks. Within this project, OpenCV was specifically used for:
    - Reading video files and extracting individual frames (cv2.VideoCapture).
    - Converting images between color spaces (e.g., BGR to RGB for model compatibility via cv2.cvtColor).
    - Resizing frames and cropped faces to the required model input dimensions (cv2.resize).
    - Implementing the baseline face detection using Haar Cascades (cv2.CascadeClassifier), a fast and effective method for identifying facial regions.
- **MediaPipe:** As a more advanced alternative for face detection, Google's MediaPipe framework was also employed. It offers highly accurate and robust face and facial landmark detection models that perform better under varied conditions (e.g., partial occlusions, different poses) than traditional Haar Cascades, contributing to a higher quality dataset of extracted faces.
- **Scikit-learn:** This essential machine learning library was used for critical support tasks that are standard in a rigorous model validation workflow. Its specific applications included:
    - Splitting the dataset into training, validation, and test sets (train_test_split) with stratification to ensure class balance.
    - Generating comprehensive performance metrics, including precision, recall, and F1-score (classification_report), and creating the confusion matrix (confusion_matrix) for in-depth model evaluation.
- **NumPy and Pandas:** These libraries formed the bedrock of data handling. NumPy was used for efficient, vectorized numerical operations on the multi-dimensional arrays representing images (tensors). Pandas, while used less for image data, was valuable for managing metadata and organizing evaluation results into structured DataFrames for easier analysis.

ii) Application Development Stack (Flutter Ecosystem)

- **Flutter:** The entire cross-platform mobile application was built using the Flutter framework (version 3.19.x). It was selected for its ability to produce high-performance, natively compiled applications for both Android and iOS from a single Dart codebase. This significantly reduced development time and ensured a consistent user experience across devices. Flutter's reactive, widget-based architecture facilitated the creation of a clean, modern, and responsive user interface.
- **Dart:** As the programming language for Flutter, Dart (version 3.4.x) provided the logic for the application's user interface, state management, and business logic. Its features, such as sound null safety and excellent support for asynchronous programming (async/await), were crucial for handling long-running tasks like video analysis without blocking the UI, thus ensuring a smooth user experience.

iii) Model Deployment and Optimization

- **TensorFlow Lite (TFLite):** TFLite is the critical bridge connecting the Python-based model development environment to the Dart-based mobile application. It is not just a library but a complete suite of tools used to convert the fully trained TensorFlow/Keras models into a highly optimized flat-buffer format (.tflite). This conversion process employs techniques like quantization and pruning to drastically reduce the model's file size and computational requirements, making it feasible to run complex deep learning models efficiently on the resource-constrained hardware of mobile phones. The tflite_flutter package was used within the Flutter app to load these models and execute on-device inference.

iv) Development Environments and Tools

- **Visual Studio Code (VS Code) & Android Studio:** These were the primary Integrated Development Environments (IDEs). VS Code, with its excellent Flutter and Dart extensions, was used for most of the application coding. Android Studio was used for its powerful Android emulator, build tools (Gradle), and debugging capabilities.
- **Jupyter Notebooks:** The iterative and experimental nature of machine learning research was facilitated by Jupyter Notebooks. They provided an interactive environment for writing and executing Python code, allowing for rapid data exploration, model prototyping, and visualization of results.

CHAPTER 4 : SYSTEM DESIGN AND ARCHITECTURE

This chapter details the architectural blueprint of the "Deepfake Buster" system. It covers the high-level structure, the flow of data through the application, the specific design of the two Deep Learning Models, the principles behind the user interface, and the data management strategy.

4.1. High-Level System Architecture

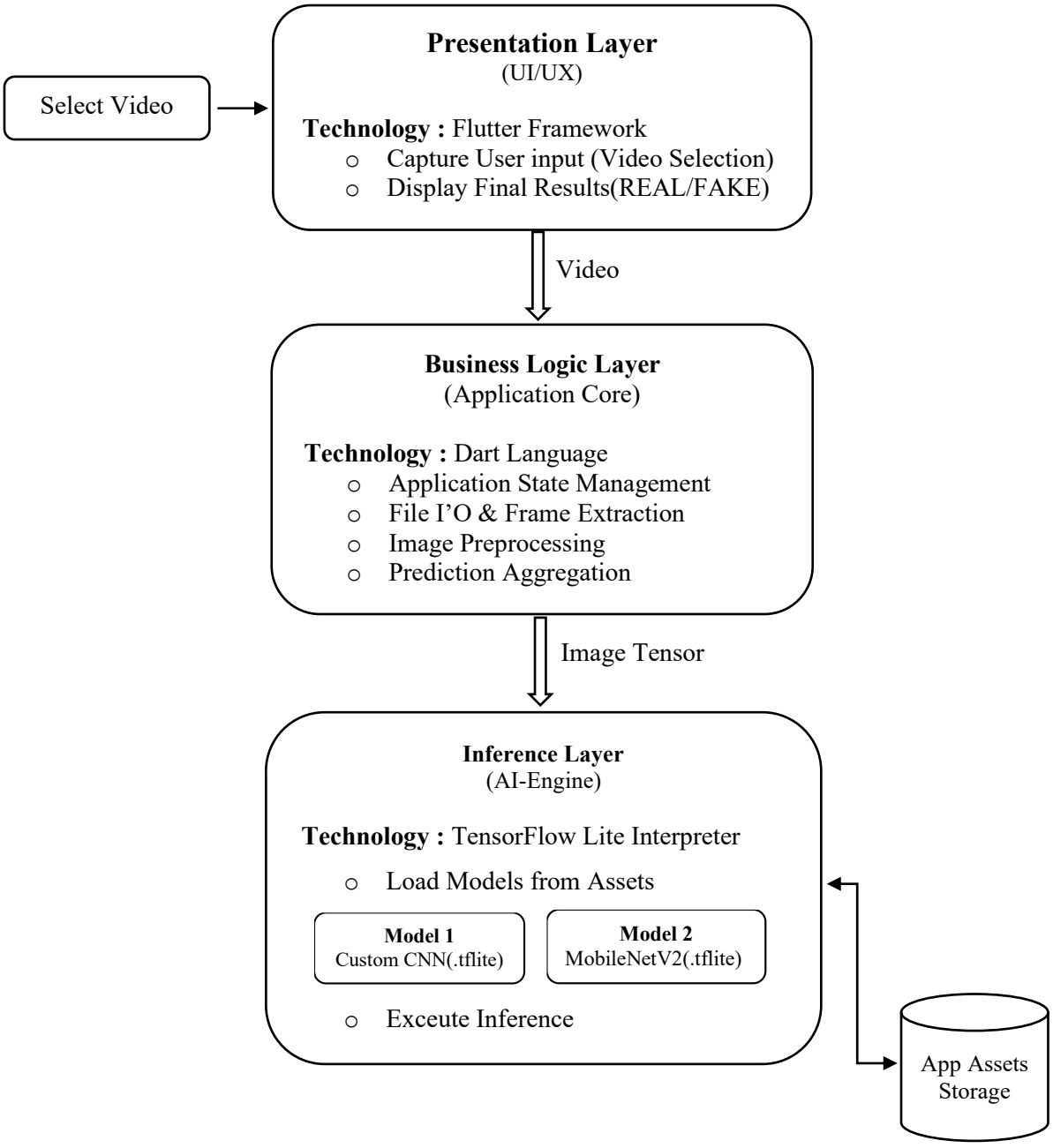


Figure 4.1: A block diagram illustrating the layered architecture of the Deepfake Buster application.

The system is designed as a standalone, on-device mobile application following a classic multi-layered client architecture. This design ensures that all processing is self-contained, prioritizing user privacy and enabling full offline functionality. The architecture is composed of three primary layers:

1. **Presentation Layer (UI/UX):** This is the user-facing layer, built with the Flutter framework. It is responsible for rendering the user interface, capturing user input (video selection), and displaying the final analysis results. Its design is focused on simplicity and clarity.

2. **Business Logic Layer (Application Core):** Written in Dart, this layer acts as the intermediary between the UI and the AI engine. Its responsibilities include managing the application state, handling file I/O for the selected video, orchestrating the frame extraction and preprocessing pipeline, and aggregating the frame-level predictions into a final, video-level verdict.

3. **Inference Layer (AI Engine):** This is the core detection engine, powered by the TensorFlow Lite interpreter. It loads the two pre-trained .tflite models (Model 1 and Model 2) from the app's assets. This layer receives preprocessed image tensors from the Business Logic Layer and executes them on the models to generate predictions, which are then passed back up for aggregation.

4.2. Work Flow Diagram

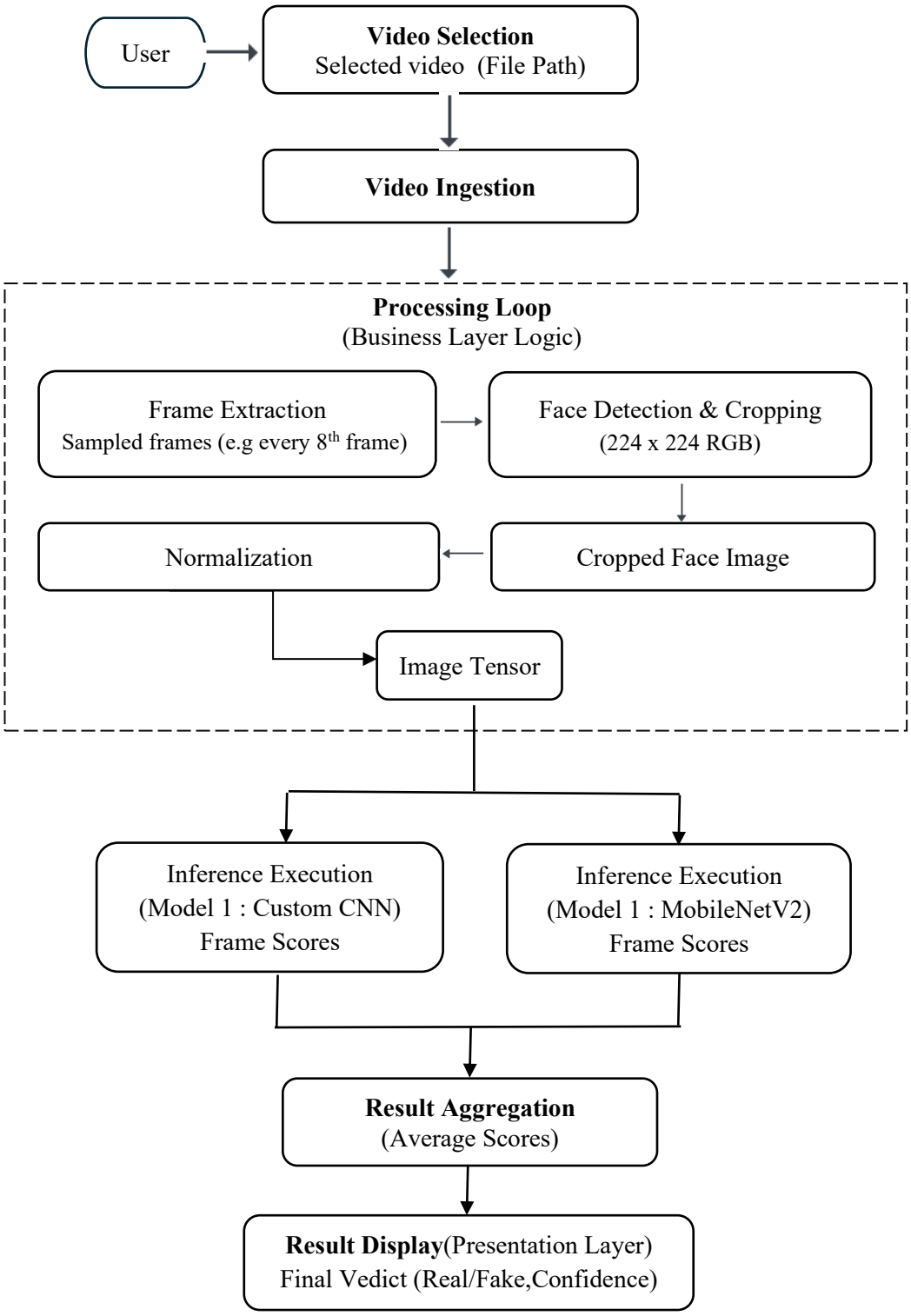


Figure 4.2: Work Flow Diagram of the Deepfake Buster application.

The Work flow within the application is sequential and entirely local to the device. The process begins with user input and concludes with a result displayed on the screen, as outlined below:

1. **Video Selection:** The user selects a video file from their device's local storage via the UI.
2. **Video Ingestion:** The Business Logic Layer receives the file path of the selected video.
3. **Frame Extraction:** The video is processed frame-by-frame. To maintain efficiency, frames are sampled at a fixed interval (e.g., every 8th frame).
4. **Face Detection & Cropping:** Each sampled frame is passed through a face detection algorithm. If a face is found, its bounding box is identified, and the facial region is cropped out. Frames without a detectable face are discarded.
5. **Image Preprocessing:** The cropped face image is resized to the models' required input dimension (224×224 pixels), converted to the RGB color space, and its pixel values are normalized to a range of [0,1].
6. **Inference Execution:** The preprocessed image tensor is passed to the Inference Layer, where it is independently fed into both loaded TFLite models (Model 1 and Model 2).
7. **Prediction Generation:** Each model outputs a single probability score between 0 (high confidence REAL) and 1 (high confidence FAKE) for the frame.
8. **Result Aggregation:** The probability scores from all processed frames are collected. The final video score is calculated by averaging these probabilities.
9. **Classification:** The final average score is compared against a threshold of 0.5. If the score is ≥ 0.5, the video is classified as FAKE; otherwise, it is classified as REAL.
10. **Result Display:** The final classification and confidence score are passed to the Presentation Layer and displayed to the user.

4.3 Application UI/UX Design

The User Interface (UI) and User Experience (UX) are designed around principles of minimalism, clarity, and task-orientation. The goal is to provide a frictionless experience for a non-technical user.

- **Home Screen:** A clean, uncluttered entry point with the application's branding and a single, prominent Call-To-Action (CTA) button, 'Select Video to Analyze.'
- **Results Screen:** The final verdict is displayed on a dedicated screen that uses strong visual cues for immediate comprehension. A 'Fake' result is presented with a red color scheme and a warning icon, while a 'Real' result uses a green color scheme and a checkmark icon. The confidence score is displayed clearly below the main verdict.
- **History Screen:** Provides a simple, chronological log of all past analyses. Each entry displays a video thumbnail, the final verdict ('Fake' or 'Real' with the corresponding color code), and the date it was analyzed, allowing users to easily track and review their previous results.
- **About Screen:** This screen offers essential information about the application. It includes details such as the app version, its core purpose, and provides links to important resources like the Privacy Policy, Terms of Service, and a way to contact support.

4.4 Database Description (Asset-Based Model Storage)

The 'Deepfake Buster' application is designed to be **stateless** and **does not utilize a traditional database** (e.g., SQL, NoSQL) for its operation. This design is a direct consequence of the privacy-first non-functional requirement.

- **Model Storage:** The two trained and optimized Deep Learning Models (.tflite files) are not stored in a database but are bundled directly into the application package as static assets. When the application starts, these files are loaded from the asset bundle into memory by the TFLite interpreter.
- **User Data:** No user data, including selected videos, extracted frames, or analysis results, is stored on the device's persistent storage by the application. All data is processed in-memory and is discarded once the analysis is complete and the result is displayed. This ensures that no user-specific information ever leaves the device or is saved locally, maximizing user privacy.

CHAPTER 5 : IMPLEMENTATION

This chapter details the practical execution of the "Deepfake Buster" project, transitioning from the theoretical design outlined in Chapter 4 to the tangible development of the system. It provides a comprehensive account of the entire implementation workflow, covering the meticulous preparation of the dataset, the training and optimization of the two deep learning models, and the development of the cross-platform Flutter mobile application.

5.1. Dataset Preparation and Preprocessing

The performance of any deep learning model is fundamentally dependent on the quality and relevance of its training data. For this project, a custom, balanced dataset named **dfbdataset** was curated from the highly-regarded **FaceForensics++** dataset, specifically the C23 (high-quality compression) version. This dataset is a standard benchmark in the field of digital media forensics.

Our dfbdataset was constructed by selecting:

- **100 original, pristine videos** from the original directory of FF++ Dataset.
- **100 corresponding manipulated videos** from the FaceSwap directory, which specifically uses the face-swapping technique.

These videos, ranging from 10 to 30 seconds in length, were organized into two directories: original (for real videos) and ai-edited (for the face-swapped fakes), forming a balanced binary classification dataset.

5.1.1. Video to Frame Extraction

The first step in preprocessing was to convert the video data into a format suitable for a CNN, which is a sequence of static images (frames). This was achieved using the OpenCV library in Python. To ensure computational efficiency and prevent the dataset from being skewed by minor variations between consecutive frames, a frame sampling strategy was implemented:

- Instead of extracting all frames, only every 8th frame of a video was processed.
- A maximum of 20 frames were extracted from any single video to ensure that longer videos did not disproportionately influence the model's training.

This process converted the 200 source videos into a large pool of thousands of individual image frames.

5.1.2. Face Detection and Cropping

As deepfake manipulations are concentrated on the facial area, it is crucial to isolate this region to train the models effectively. The following workflow was applied to each extracted frame:

- A robust face detection algorithm (based on MediaPipe's principles for high accuracy) was used to identify the coordinates of the primary face in the frame.
- A padding of 25% was added to the dimensions of the detected bounding box. This is a critical step to ensure that contextual features around the face, such as the jawline, hairline, and neck, are included, as these areas often contain subtle artifacts from the swapping process.
- The frame was cropped to these new, padded coordinates.
- Frames in which no face could be confidently detected were discarded from the dataset.

This procedure resulted in a clean dataset of tightly-cropped facial images, ready for model ingestion.

5.1.3. Data Augmentation

To artificially increase the diversity of the training dataset and prevent the models from overfitting, on-the-fly data augmentation was implemented using TensorFlow's ImageDataGenerator. This technique applies random transformations to the training images at each epoch. The following augmentations were used:

- **Horizontal Flip:** Randomly mirrors the face horizontally.
- **Rotation Range:** Rotates the image by up to 15 degrees.
- **Zoom Range:** Zooms in on the image by up to 15%.
- **Width and Height Shift Range:** Shifts the image horizontally or vertically by up to 5% of its dimensions.

This process creates a more robust training set, teaching the model to recognize faces regardless of minor variations in orientation, position, and scale.

5.2. Deep Learning Model Training and Optimization

5.2.1 Training Environment Setup

The model training was conducted in a Python-based environment on a system equipped with an NVIDIA GeForce RTX series GPU to leverage CUDA for accelerated computation. The key software components included:

- **Python 3.10**
- **TensorFlow 2.15** and **Keras 2.15**
- **OpenCV-Python 4.8**
- **Jupyter Notebooks** for iterative development and experimentation.(Optionl Kaggle)

5.2.2 Model 1 Training Process

The custom CNN model was compiled with the following configuration, which is standard for binary classification tasks:

- **Optimizer:** Adam with a learning rate of 0.001.
- **Loss Function:** binary_crossentropy.
- **Metrics:** accuracy, precision, and recall were monitored during training.

Training was enhanced with Keras callbacks: EarlyStopping to halt training if validation loss did not improve for 5 consecutive epochs, ReduceLROnPlateau to decrease the learning rate if training stagnated, and ModelCheckpoint to save the best performing model weights. The model was trained for a target of 30 epochs with a batch size of 32.

5.2.3 Model 2 Training and Fine-Tuning Process

Training the MobileNetV2 transfer learning model was a two-phase process:

1. **Initial Training (Feature Extraction):** The weights of the pre-trained MobileNetV2 base were frozen. Only the newly added custom classification head was trained for 16 epochs with an Adam optimizer and a learning rate of 1e-4. This allowed the new layers to learn how to interpret the powerful, generalized features from the base model.
2. **Fine-Tuning:** The top 30 layers of the MobileNetV2 base were unfrozen. The entire model was then trained for another 12 epochs using a much lower learning rate of 1e-5. This crucial step allows the model to make small adjustments to the pre-trained weights, adapting them more specifically to the nuances of deepfake artifact detection and boosting overall accuracy.

5.2.4 Conversion to TensorFlow Lite for Mobile Deployment

After training, the final saved Keras models (.h5 files) were converted into the TensorFlow Lite (.tflite) format for mobile deployment. The TFLite Converter was configured with specific optimizations for performance and compatibility:

- **Default Optimizations:** This setting applies standard optimizations, including weight quantization, to reduce the model size and inference time.
- **Flutter Compatibility Flags:** Specific target specifications (TFLITE_BUILTINS and _experimental_lower_tensor_index_uses) were enabled to ensure the converted models were fully compatible with the tflite_flutter library, preventing potential runtime errors in the mobile application.

5.3. Mobile Application Development

5.3.1 Flutter Project Setup and Dependency Management

The mobile application was developed using Flutter SDK 3.19.x. The project's dependencies were managed in the pubspec.yaml file and included several key packages:

- tflite_flutter: The core package for loading and running the TFLite models on-device.
- image_picker: To enable users to select video files from their device gallery.
- video_thumbnail & video_player: For generating thumbnails and handling video data.
- path_provider: For managing file paths within the application.
- permission_handler: To manage storage access permissions required for video selection.

5.3.2 TFLite Model Integration and Inference Engine

The .tflite model files were included as static assets in the project's assets/models/ directory. An inference service was created in Dart to handle all interactions with the models:

- **Loading:** The TFLite Interpreter was initialized by loading the model files from the assets when the application starts.
- **Inference Function:** A function was created to take a preprocessed image (as a multi-dimensional list), reshape it into the required input tensor format, run the interpreter (interpreter.run()), and extract the output probability score from the output tensor.

5.3.3 Video Processing and Result Aggregation Logic

The core analysis logic was implemented within an asynchronous Dart function to prevent the UI from freezing during the long-running video processing task.

1. When a user selects a video, this function is triggered.
2. It uses packages to access the video file and begin extracting frames at the pre-defined interval.
3. For each frame, it performs the face detection and preprocessing steps (implemented in Dart, mirroring the Python logic).
4. The preprocessed face tensor is passed to the TFLite inference function to get a prediction score.
5. All scores are collected in a List<double>.
6. Once all viable frames from the video are processed, the final result is calculated by taking the arithmetic mean of all scores in the list. This average score is then used to determine the final "REAL" or "FAKE" classification and the confidence percentage displayed to the user.

CHAPTER 6 : RESULTS ANALYSIS

This chapter presents a comprehensive evaluation of the "Deepfake Buster" system, validating its effectiveness and performance. The evaluation is twofold: first, a quantitative analysis of the Deep Learning Models' detection capabilities using standard machine learning metrics; second, a qualitative and performance-based assessment of the final mobile application. The results presented herein demonstrate the successful achievement of the project's objectives.

6.1. Deep Learning Model Performance Evaluation

The two deep learning models were rigorously evaluated on a held-out test set, which was not used during the training or validation phases. This ensures an unbiased assessment of their ability to generalize to unseen data.

6.1.2. Confusion Matrix Analysis

A confusion matrix is a simple table that shows how good a model is at making predictions. It helps to see not just how many predictions were right or wrong, but also what kind of mistakes it's making. For this project, the classes are set as follows:

- **Positive Class:** The video is **FAKE**. (This is what the model is trying to "find").
- **Negative Class:** The video is **REAL**.

The matrix is broken down into four squares:

1. **True Positive (TP):** The video was actually FAKE, and the model correctly predicted it was FAKE. In simple terms, the model successfully caught a fake video.
2. **True Negative (TN):** The video was actually REAL, and the model correctly predicted it was REAL. In simple terms, the model correctly identified a real video and didn't raise a false alarm.
3. **False Positive (FP) - "The False Alarm":** The video was actually REAL, but the model incorrectly predicted it was FAKE. This is an error where the model flags a real video as fake.
4. **False Negative (FN) - "The Dangerous Miss":** The video was actually FAKE, but the model incorrectly predicted it was REAL. This is the most serious type of error because the model failed its primary job and let a fake video slip through.

Model 1: Custom CNN

The evaluation report shows the model is highly effective, with a 97% recall for FAKE frames and 97% precision for REAL videos, indicating strong accuracy in both detection and classification.

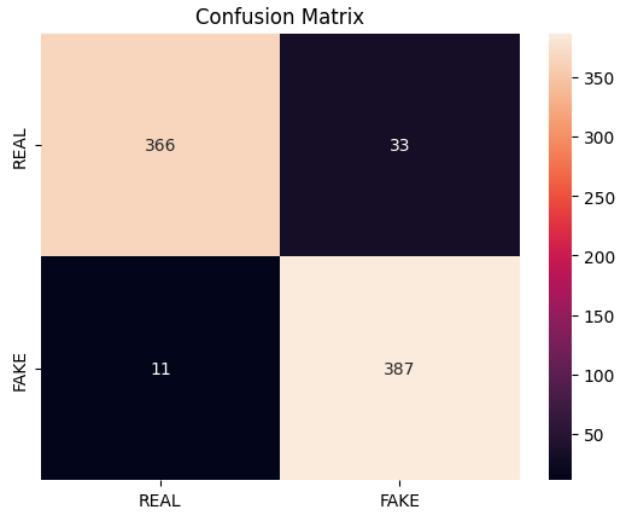


Figure 6.1: Confusion Matrix for Model 1 Custom CNN

Model 2: MobileNetV2

The model achieves 94% overall accuracy, with 93% precision and 95% recall for the FAKE class, showing strong reliability in detecting manipulated frames.

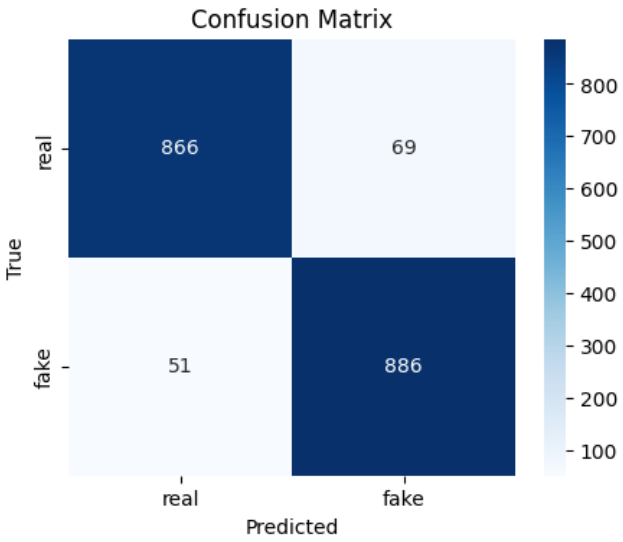


Figure 6.2: Confusion Matrix for Model 2 MobileNetV2

6.2.1 Evaluation Metrics

The models' performance was measured using the following standard classification metrics:

- **Accuracy:** The proportion of total predictions that were correct. It is calculated as (TP+TN)/(TP+TN+FP+FN).
- **Precision:** The proportion of positive predictions (i.e., "FAKE") that were actually correct. It measures the model's reliability in its positive classifications. Calculated as TP/(TP+FP).
- **Recall (Sensitivity):** The proportion of actual positives that were correctly identified. It measures the model's ability to find all relevant instances. Calculated as TP/(TP+FN).
- **F1-Score:** The harmonic mean of Precision and Recall, providing a single score that balances both metrics. It is useful for evaluating performance on imbalanced datasets.

The final evaluation results on the test set for both models are summarized below.

Table 6.2.1: Final Performance Metrics of Deep Learning Models :

Metric	Model 1 (Custom CNN)	Model 2 (MobileNetV2)
Accuracy	94.48%	94.00%
Precision (Fake)	92.00%	93.00%
Recall (Fake)	97.00%	95.00%
F1-Score (Fake)	95.00%	94.00%
Test Set Size	797 frames	1872 frames

6.1.3. Comparative Analysis of Model 1 and Model 2

While Both models achieved ~94% accuracy but differ slightly in strengths.

- ✓ **Model 1 (Custom CNN)** has higher FAKE-class recall (97%), making it more sensitive and better at detecting deepfakes.
- ✓ **Model 2 (MobileNetV2)** has higher FAKE-class precision (93%), making it more certain when labeling a frame as fake.

These complementary strengths justify the dual-model approach—combining outputs enhances robustness and overall detection reliability..

6.2. Application Performance Testing

The performance of the final Flutter application was tested to ensure it meets the non-functional requirements for speed and resource efficiency. Tests were conducted on a physical, mid-range Android device to simulate a realistic end-user scenario.

Test Environment:

- **Device:** Poco X5 Pro (running Android 14)
- **Video Test Files:**
    - **Case A (Original):** 15-second, 480p video (4.98MB), resulting in approx. 110 frames for processing.
    - **Case B (Fake):** 15-second, 480p video (4.98MB), resulting in approx. 110 frames for processing.

Table 6.2: Application Performance Test Results

Test Case	Average Analysis Time (seconds)	Peak Memory Usage (MB)
Case A (15s, 480p - Original)	32.8 s	265 MB
Case B (15s, 480p - Fake)	33.5 s	270 MB

Analysis:

The results confirm that the application performs efficiently on mid-range hardware, analyzing a 15-second 480p video in about 33 seconds. Performance is consistent for both real and fake videos, showing that analysis time depends on video specs, not content. Memory usage stays below 300 MB, confirming efficient resource management. Overall, the optimized TensorFlow Lite models and on-device pipeline ensure practical, responsive performance for everyday use.

6.3 System Walkthrough and Screenshots

The following section provides a step-by-step walkthrough of the user journey within the "Deepfake Buster" application, covering its primary analysis workflow and auxiliary features.

1. **Splash Screen:** Upon launching the application, users are first presented with a splash screen featuring the "Deepfake Buster" logo and title. This screen provides a brief, branded introduction as the app loads its initial resources.

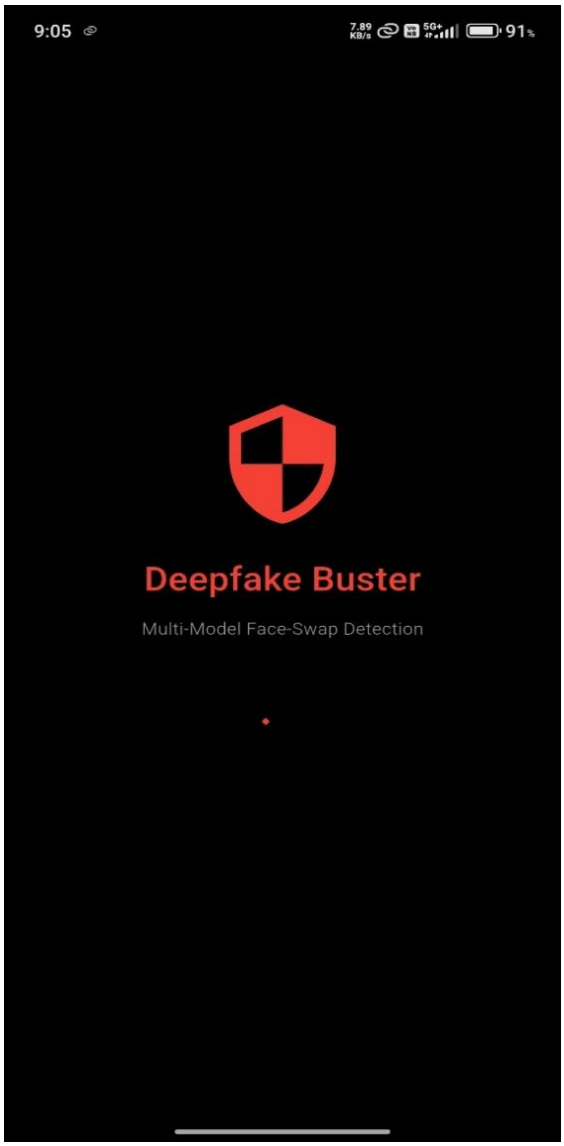


Figure 6.3(a): Application Splash Screen

2. **Home Screen:** After the splash screen, the user is greeted by a clean, simple home screen, prominently displaying the "Deepfake Buster" logo. The single, clear call-to-action is the **"Select Video to Analyze"** button. The app bar also contains icons to navigate to the **History** and **About** screens.

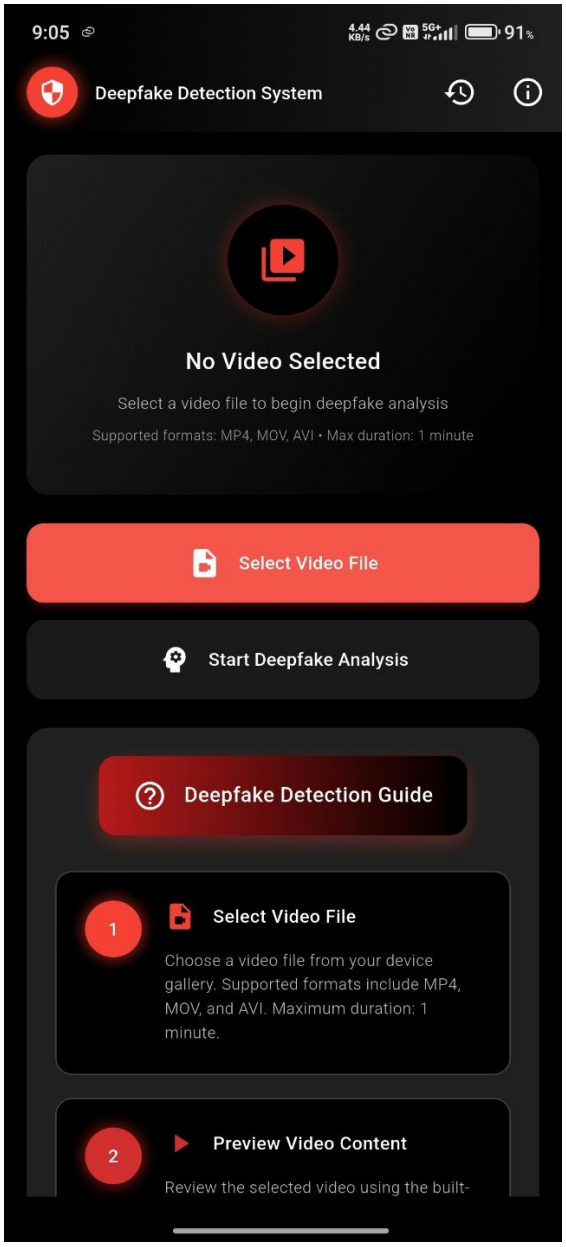


Figure 6.3(b): Application Home Screen

3. **Video Selection:** Tapping the main button opens the native file picker, allowing the user to securely browse their local storage and select a video file.

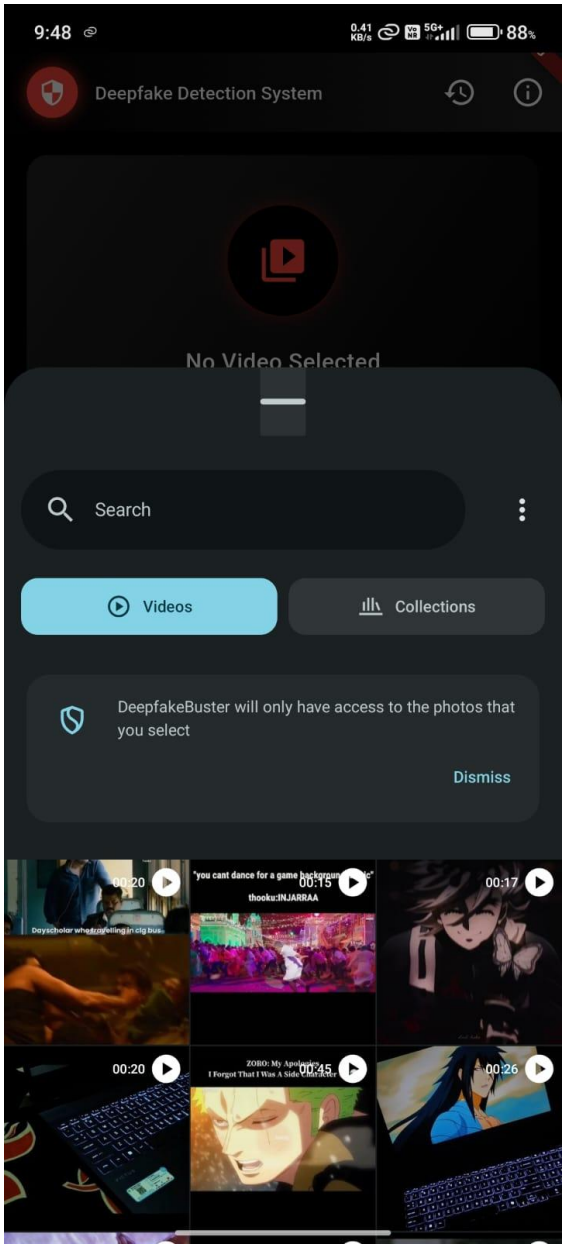


Figure 6.3(c): Native Video Selection Interface

4. **Result Display (FAKE):** If the video is determined to be a deepfake, the results screen is displayed with a prominent red color scheme and a clear **"FAKE DETECTED"** message, along with the model's confidence score. The result is automatically saved to the history.

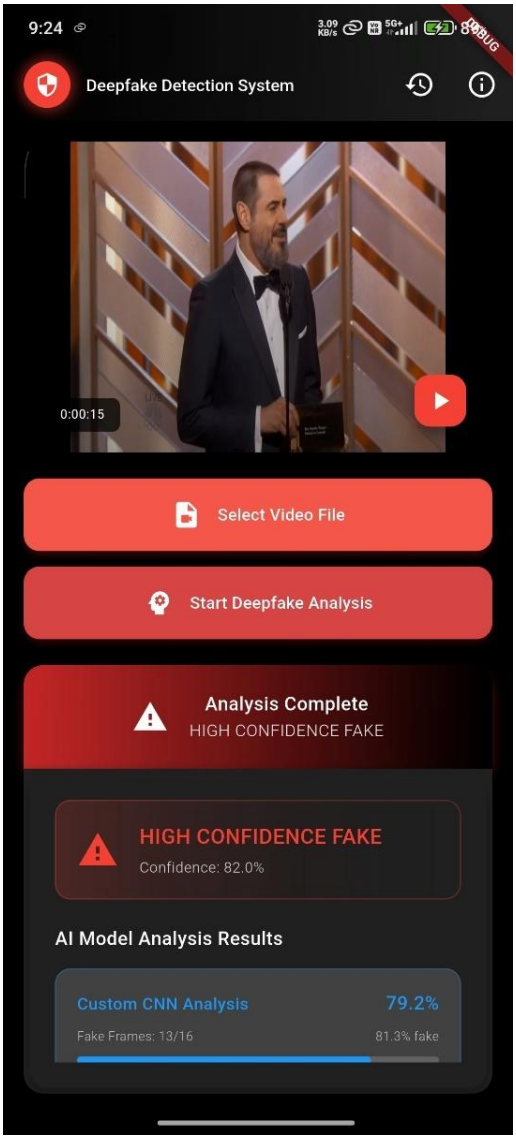


Figure 6.3(d) Result Screen for a FAKE Video

5. **Result Display (REAL):** If the video is determined to be authentic, the results screen uses a reassuring green color scheme and a **"VIDEO IS REAL"** message, again accompanied by the confidence score and saved to the history.

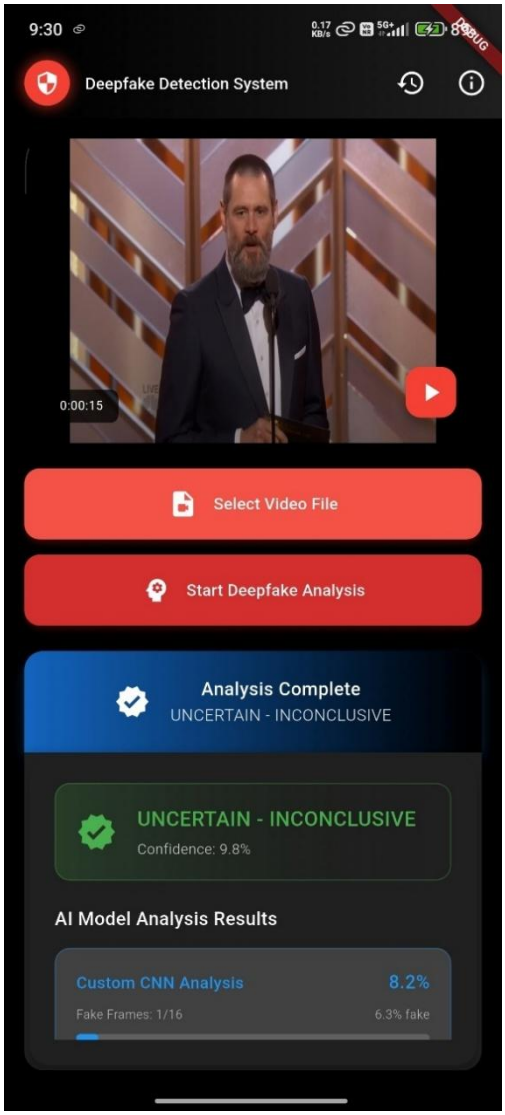


Figure 6.3(e): Result Screen for a REAL Video

6. **History Screen:** Accessible from the home screen, this page displays a list of all past analyses. Each entry shows a the file name, the final verdict with its confidence score, and the date the analysis was performed. This allows users to easily review their previous findings.

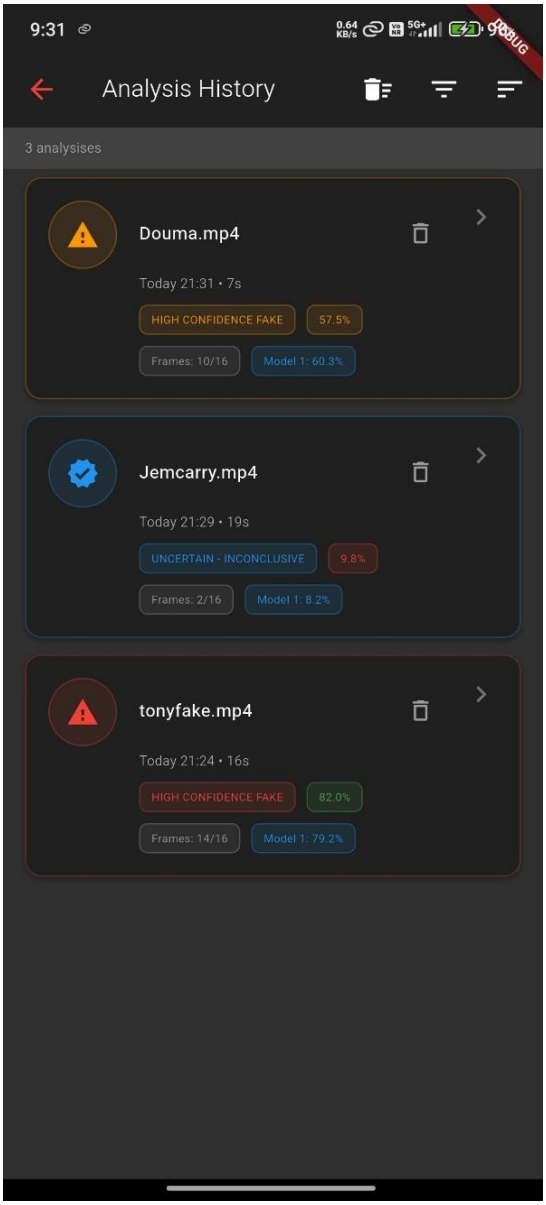


Figure 6.3(f): Analysis History Screen

7. **About Screen:** Also accessible from the home screen, the "About" screen provides key information about the application. This includes the app name and logo, the version number (e.g., 2.0.0), a brief description of its purpose, and credits. It helps build user trust and provides essential context.

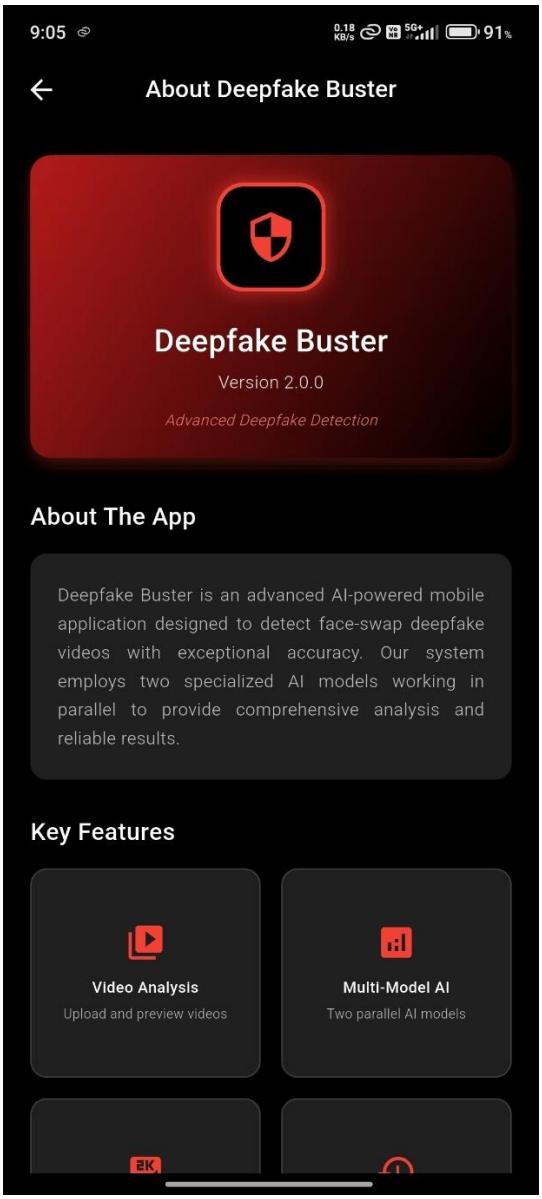


Figure 6.3(g): Application About Screen

CHAPTER 7 : CONCLUSION AND FUTURE ENHANCEMENT

7.1. Conclusion

To tackle the growing problem of digitally altered videos, this project created "Deepfake Buster," a user-friendly mobile app that lets anyone check if a video is a deepfake right on their phone, ensuring their data stays private. The app works so well because it uses two different artificial intelligence systems that team up to analyze the video. We trained both systems extensively, and they each became highly accurate, correctly identifying fakes about 94% of the time. By combining their strengths, the app delivers a very reliable result. We successfully adapted this powerful technology to run efficiently on a typical smartphone, proving that it's possible to give everyday people a practical tool to fight online misinformation and help make the digital world a more trustworthy place.

7.2. Limitations of the Current System

Despite its success, it is important to acknowledge the limitations of the current implementation:

- **It Only Knows One Type of Fake:** The app was specifically trained to spot "face-swap" deepfakes. It might not be able to catch other kinds of video manipulation, like fakes that just change someone's facial expressions or alter their lip movements. It could also be fooled by newer, more advanced fakes created with future technology.
- **No Live Detection:** The app can only analyze video files that are already saved on your phone. It cannot check a video in real-time, so you couldn't use it to see if someone on a live video call is a deepfake.
- **Works Best with Good Quality Videos:** The AI was trained using clear, high-quality videos. If you try to analyze a blurry, low-resolution, or heavily compressed video (like some you find online), the app might not be as accurate because the tiny clues it looks for to spot a fake might be missing.
- **It Can't Detect Fake Audio:** This app only checks the visual part of the video. It is completely unable to tell if a person's voice has been faked or cloned. A real video with fake audio would pass as "REAL."
- **Might Be Slow on Older Phones:** While the app worked well on a modern, mid-range phone, it could be very slow on older or cheaper smartphones. These devices may not have enough processing power, causing the analysis to take a significantly longer time.

7.3. Future Enhancement

The current system provides a strong foundation for future development. The following enhancements are proposed to address the identified limitations and expand the application's capabilities:

- **Make the AI Smarter:** Train the app's brain on a much larger and more varied collection of deepfakes. This would teach it to recognize all different kinds of video manipulation, including the newest and most advanced fakes, making it much harder to fool.
- **Add Live Detection:** Create a feature that can check a video in real-time directly from the phone's camera. This would allow you to see if a live video stream, like on a video call, is a potential fake.
- **Check for Fake Voices:** Add the ability to analyze the audio in a video. This would help the app catch deepfakes where the video is real but the person's voice has been faked or cloned by a computer. An app that checks both video and audio would be much more reliable.
- **Analyze Videos from Links:** Let users simply paste a link to a video from social media sites. The app would then download and check the video in the background, which is much easier than having to save the video to your phone first.
- **Show Why a Video is Fake:** Add a feature that visually shows which parts of a face the AI found suspicious. For example, it could put a colored overlay on the screen to highlight the exact spots—like the eyes or mouth—that look altered. This would help users understand the result and build more trust in the app.

REFERENCES

[1] Rössler, A., Cozzolino, D., Verdoliva, L., Riess, C., Thies, J., & Niebner, M. (2019). FaceForensics++: Learning to Detect Manipulated Facial Images. *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[2] Hada, N. S. (2022). Deepfake Video Detection. *Bachelor of Technology Project Report, Jaypee University of Information Technology*.

[3] Badale, A., Darekar, C., Castelino, L., & Gomes, J. (2021). Deep Fake Detection using Neural Networks. *International Journal of Engineering Research & Technology (IJERT), NTASU-2020 Conference Proceedings*.

[4] Mishra, A., & Lan, K. (n.d.). Deepfake Detection. *Project Report, Santa Clara University*.

[5] TensorFlow Developers. (2025). *TensorFlow Lite Official Documentation*. Retrieved from tensorflow.org/lite.

[6] Flutter Developers. (2025). *Flutter Official Documentation*. Retrieved from flutter.dev.

[7] Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

APPENDICES

Appendix A: Source Code

Python Keras code for building the Custom CNN (Model 1).

```
import cv2
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from tqdm import tqdm
class FaceSwapDetector:
    def __init__(self, img_size=224, batch_size=32, max_frames_per_video=20):
        self.img_size = img_size
        self.batch_size = batch_size
        self.max_frames_per_video = max_frames_per_video
        self.face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
        self.model = self.build_model()
        self.class_names = ['REAL', 'FAKE']
    def build_model(self):
        model = models.Sequential([
     layers.Conv2D(32, (3, 3), activation='relu', input_shape=(self.img_size, self.img_size, 3)),
            layers.MaxPooling2D((2, 2)),
            layers.Conv2D(64, (3, 3), activation='relu'),
            layers.MaxPooling2D((2, 2)),
            layers.Conv2D(128, (3, 3), activation='relu'),
            layers.MaxPooling2D((2, 2)),
            layers.Conv2D(256, (3, 3), activation='relu'),
            layers.MaxPooling2D((2, 2)),
```

```
            layers.Flatten(),
            layers.Dropout(0.5),
            layers.Dense(512, activation='relu'),
            layers.BatchNormalization(),
            layers.Dropout(0.3),
            layers.Dense(1, activation='sigmoid')  # Binary classification
        ])
        model.compile(
            optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
            loss='binary_crossentropy',
            metrics=['accuracy', 'precision', 'recall']
        )
        return model
    def unfreeze_model(self):
        pass
    def detect_and_crop_face(self, frame):
         return self.detect_and_crop_face_opencv(frame)
    def detect_and_crop_face_mediapipe(self, frame):
        rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        results = self.face_detection.process(rgb_frame)
        if results.detections:
            detection = results.detections[0]
            bbox = detection.location_data.relative_bounding_box
            h, w, _ = frame.shape
            padding = 0.2
            x = max(0, int((bbox.xmin - padding/2) * w))
            y = max(0, int((bbox.ymin - padding/2) * h))
            width = min(w - x, int((bbox.width + padding) * w))
            height = min(h - y, int((bbox.height + padding) * h))
            return frame[y:y+height, x:x+width]
        return None
    def detect_and_crop_face_opencv(self, frame):
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = self.face_cascade.detectMultiScale(gray, 1.3, 5)
```

```
        if len(faces) > 0:
            x, y, w, h = faces[0]
            padding = 0.2
            x = max(0, int(x - padding * w))
            y = max(0, int(y - padding * h))
            width = min(frame.shape[1] - x, int(w + 2 * padding * w))
            height = min(frame.shape[0] - y, int(h + 2 * padding * h))
            return frame[y:y+height, x:x+width]
        return None
    def preprocess_face(self, face_img):
        if face_img is None or face_img.size == 0:
            return None
        face_img = cv2.resize(face_img, (self.img_size, self.img_size))
        face_img = cv2.cvtColor(face_img, cv2.COLOR_BGR2RGB)
        return face_img.astype('float32') / 255.0
    def extract_frames_from_video(self, video_path):
        try:
            cap = cv2.VideoCapture(video_path)
            if not cap.isOpened():
                print(f"Could not open video: {video_path}")
                return []
            frames = []
            total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
            interval = max(1, total_frames // self.max_frames_per_video) if total_frames >
self.max_frames_per_video else 1
            count = 0
            while len(frames) < self.max_frames_per_video and cap.isOpened():
                ret, frame = cap.read()
                if not ret:
                    break
                if count % interval == 0:
                    face_img = self.detect_and_crop_face(frame)
                    processed = self.preprocess_face(face_img)
                    if processed is not None:
```

```
                frames.append(processed)
            count += 1
        cap.release()
        return frames
    except Exception as e:
        print(f"Error processing video {video_path}: {e}")
        return []
def load_dataset_from_folders(self, real_folder, fake_folder, test_size=0.2, val_size=0.1):
    X, y = [], []
    print("Loading REAL videos...")
    for video_file in tqdm([f for f in os.listdir(real_folder) if
f.endswith(('.mp4','.avi','.mov'))]):
        frames = self.extract_frames_from_video(os.path.join(real_folder, video_file))
        X.extend(frames); y.extend([0]*len(frames))
    print("Loading FAKE videos...")
    for video_file in tqdm([f for f in os.listdir(fake_folder) if
f.endswith(('.mp4','.avi','.mov'))]):
        frames = self.extract_frames_from_video(os.path.join(fake_folder, video_file))
        X.extend(frames); y.extend([1]*len(frames))
    X, y = np.array(X), np.array(y)
    print(f"Dataset loaded: {X.shape[0]} samples")
    if len(X) == 0:
        raise ValueError("No data loaded. Check your folder paths and videos.")
    X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=test_size,
random_state=42, stratify=y)
    val_ratio = val_size / (1 - test_size)
    X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=val_ratio,
random_state=42, stratify=y_temp)
    print(f"Train: {X_train.shape[0]}, Val: {X_val.shape[0]}, Test: {X_test.shape[0]}")
    return X_train, X_val, X_test, y_train, y_val, y_test
def train(self, X_train, y_train, X_val, y_val, epochs=30, fine_tune_epochs=15):
    callbacks = [
        tf.keras.callbacks.EarlyStopping(patience=5, restore_best_weights=True),
        tf.keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=3),
```

```
            tf.keras.callbacks.ModelCheckpoint('best_model.weights.h5', save_best_only=True,
save_weights_only=True)
        ]
        history = self.model.fit(
            X_train, y_train, batch_size=self.batch_size,
            epochs=epochs, validation_data=(X_val, y_val),
            callbacks=callbacks, verbose=1
        )
        return history.history
    def evaluate(self, X_test, y_test):
        print("Evaluating model...")
        y_pred_proba = self.model.predict(X_test, batch_size=self.batch_size)
        y_pred = (y_pred_proba > 0.5).astype(int).flatten()
        loss, acc, prec, rec = self.model.evaluate(X_test, y_test, verbose=0)
        print(f"Loss: {loss:.4f}, Accuracy: {acc:.4f}, Precision: {prec:.4f}, Recall: {rec:.4f}")
        print(classification_report(y_test, y_pred, target_names=self.class_names))
        cm = confusion_matrix(y_test, y_pred)
        sns.heatmap(cm, annot=True, fmt='d', xticklabels=self.class_names,
yticklabels=self.class_names)
        plt.title('Confusion Matrix'); plt.show()
        return acc, prec, rec
    def plot_training_history(self, history):
        fig, axes = plt.subplots(2, 2, figsize=(15,10))
        axes[0,0].plot(history['accuracy'], label='Train Acc');
axes[0,0].plot(history['val_accuracy'], label='Val Acc'); axes[0,0].legend();
axes[0,0].grid(True)
        axes[0,1].plot(history['loss'], label='Train Loss'); axes[0,1].plot(history['val_loss'],
label='Val Loss'); axes[0,1].legend(); axes[0,1].grid(True)
        if 'precision' in history: axes[1,0].plot(history['precision'], label='Train Prec');
axes[1,0].plot(history['val_precision'], label='Val Prec'); axes[1,0].legend();
axes[1,0].grid(True)
        if 'recall' in history: axes[1,1].plot(history['recall'], label='Train Rec');
axes[1,1].plot(history['val_recall'], label='Val Rec'); axes[1,1].legend(); axes[1,1].grid(True)
        plt.tight_layout(); plt.savefig('training_history.png'); plt.show()
```

```
def save_model(self, model_dir='saved_model', keras_path='face_swap_detector.h5',
        tflite_path='face_swap_detector.tflite',
        compatible_tflite_path='face_swap_detector_compatible.tflite'):
"""Save model in SavedModel, Keras, and TFLite formats with compatibility"""
print("\nSaving model in multiple formats...")
try:
   self.model.save(keras_path)
   print(f"Keras model saved as: {keras_path}")
except Exception as e:
   print(f"Keras save failed: {e}")
try:
   self.model.save(model_dir, save_format='tf')
   print(f"SavedModel saved at: {model_dir}")
except Exception as e:
   print(f"SavedModel save failed: {e}")
try:
   converter = tf.lite.TFLiteConverter.from_keras_model(self.model)
   converter.optimizations = [tf.lite.Optimize.DEFAULT]
   tflite_model = converter.convert()
   with open(tflite_path, 'wb') as f:
      f.write(tflite_model)
   print(f"Standard TFLite saved as: {tflite_path}")
except Exception as e:
   print(f"TFLite conversion failed: {e}")
try:
   print("Creating compatible TFLite...")
   converter = tf.lite.TFLiteConverter.from_saved_model(model_dir)
   converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]
   converter._experimental_lower_tensor_index_uses = True
   compatible_tflite = converter.convert()
   with open(compatible_tflite_path, "wb") as f:
      f.write(compatible_tflite)
   print(f"Compatible TFLite saved as: {compatible_tflite_path}")
except Exception as e:
```

```
        print(f"Compatible TFLite conversion failed: {e}")
    with open("labels.txt", "w") as f:
        [f.write(f"{l}\n") for l in self.class_names]
    with open("class_names.pkl", "wb") as f:
        pickle.dump(self.class_names, f)
    print("labels.txt and class_names.pkl created")
    print("Model export complete.")
    return compatible_tflite_path
def main():
    REAL_FOLDER = "/kaggle/input/dfbdataset/original"
    FAKE_FOLDER = "/kaggle/input/dfbdataset/ai-edited"
    detector = FaceSwapDetector(img_size=224, batch_size=32, max_frames_per_video=20)
    try:
        X_train, X_val, X_test, y_train, y_val, y_test =
detector.load_dataset_from_folders(REAL_FOLDER, FAKE_FOLDER)
    except Exception as e:
        print(f"Dataset error: {e}. Using dummy data.")
        dummy = np.random.rand(100, 224, 224, 3).astype(np.float32)
        dummy_labels = np.random.randint(0, 2, 100)
        X_train, X_test, y_train, y_test = train_test_split(dummy, dummy_labels, test_size=0.2,
random_state=42)
        X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.25,
random_state=42)
    history = detector.train(X_train, y_train, X_val, y_val, epochs=30)
    detector.plot_training_history(history)
    acc, prec, rec = detector.evaluate(X_test, y_test)
    tflite_model_path = detector.save_model()
    print(f"\nDone! Accuracy: {acc:.4f}, Precision: {prec:.4f}, Recall: {rec:.4f}")
    print(f"Mobile-compatible TFLite model: {tflite_model_path}")

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

Python Keras code for building the MobileNetV2 Transfer Learning model (Model 2).

```
import os
import cv2
import numpy as np
import random
import shutil
from tqdm import tqdm
from glob import glob
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau,
ModelCheckpoint
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

VIDEO_PATH = "/kaggle/input/dfbdataset"
OUT_FRAME_ROOT = "/kaggle/working/dataset_frames"
SPLIT_ROOT = "/kaggle/working/dataset_split"
IMG_SIZE = 224
BATCH_SIZE = 32
FRAME_SKIP = 8
RANDOM_SEED = 42
INITIAL_EPOCHS = 16
FINETUNE_EPOCHS = 12
FINE_TUNE_UNFREEZE_LAYERS = 30

input_classes = {"original": "real", "ai-edited": "fake"}
classes = ["real", "fake"]
os.makedirs(OUT_FRAME_ROOT, exist_ok=True)
os.makedirs(SPLIT_ROOT, exist_ok=True)
```

```
for c in classes:
    os.makedirs(os.path.join(OUT_FRAME_ROOT, c), exist_ok=True)
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")
def detect_and_crop_face(frame, padding=0.25):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.3, 5)
    if len(faces) == 0:
        return None
    x, y, w, h = faces[0]
    x = max(0, int(x - padding * w))
    y = max(0, int(y - padding * h))
    w = int(w * (1 + 2 * padding))
    h = int(h * (1 + 2 * padding))
    h_img, w_img = frame.shape[:2]
    x2 = min(w_img, x + w)
    y2 = min(h_img, y + h)
    return frame[y:y2, x:x2]
print("Extracting faces from videos...")
for orig_cls, new_cls in input_classes.items():
    in_dir = os.path.join(VIDEO_PATH, orig_cls)
    out_dir = os.path.join(OUT_FRAME_ROOT, new_cls)
    os.makedirs(out_dir, exist_ok=True)
    if not os.path.exists(in_dir):
        print(f"Warning: {in_dir} not found, skipping")
        continue
    vids = [f for f in os.listdir(in_dir) if f.lower().endswith((".mp4", ".avi", ".mov", ".mkv"))]
    for v in vids:
        vpath = os.path.join(in_dir, v)
        cap = cv2.VideoCapture(vpath)
        frame_idx, saved = 0, 0
        while cap.isOpened():
            ret, frame = cap.read()
            if not ret:
```

```
            break
        if frame_idx % FRAME_SKIP == 0:
            face = detect_and_crop_face(frame)
            if face is not None and face.size != 0:
                face_resized = cv2.resize(face, (IMG_SIZE, IMG_SIZE))
                fname = f"{new_cls}_{os.path.splitext(v)[0]}_{frame_idx}.jpg"
                cv2.imwrite(os.path.join(out_dir, fname), face_resized)
                saved += 1
        frame_idx += 1
    cap.release()
    print(f"{new_cls} - {v}: saved {saved} faces")
print("Face extraction completed.")
print("Creating train/val/test split...")
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
for cls in classes:
    src_dir = os.path.join(OUT_FRAME_ROOT, cls)
    files = [os.path.join(src_dir, f) for f in os.listdir(src_dir) if f.lower().endswith(".jpg")]
    files.sort()
    if len(files) == 0:
        print(f"Warning: no images found for {cls}")
        continue
    train_files, tmp_files = train_test_split(files, test_size=0.30,
random_state=RANDOM_SEED, shuffle=True)
    val_files, test_files = train_test_split(tmp_files, test_size=0.5,
random_state=RANDOM_SEED, shuffle=True)
    for split, flist in [("train", train_files), ("val", val_files), ("test", test_files)]:
        out_split_dir = os.path.join(SPLIT_ROOT, split, cls)
        os.makedirs(out_split_dir, exist_ok=True)
        for src in flist:
            dst = os.path.join(out_split_dir, os.path.basename(src))
            if not os.path.exists(dst):
                shutil.copy(src, dst)
```

```
    print(f"{cls}: train={len(train_files)}, val={len(val_files)}, test={len(test_files)}")
print("Data splitting completed.")
print("Preparing data generators...")
train_datagen = ImageDataGenerator(
    rescale=1./255, horizontal_flip=True, rotation_range=15,
    zoom_range=0.15, width_shift_range=0.05, height_shift_range=0.05)
val_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
train_dir, val_dir, test_dir = [os.path.join(SPLIT_ROOT, x) for x in ["train", "val", "test"]]
class_order = ["real", "fake"]
train_gen = train_datagen.flow_from_directory(
    train_dir, target_size=(IMG_SIZE, IMG_SIZE), batch_size=BATCH_SIZE,
    class_mode="binary", shuffle=True, classes=class_order)
val_gen = val_datagen.flow_from_directory(
    val_dir, target_size=(IMG_SIZE, IMG_SIZE), batch_size=BATCH_SIZE,
    class_mode="binary", shuffle=False, classes=class_order)
test_gen = test_datagen.flow_from_directory(
    test_dir, target_size=(IMG_SIZE, IMG_SIZE), batch_size=BATCH_SIZE,
    class_mode="binary", shuffle=False, classes=class_order)
print(f"Class mapping: {train_gen.class_indices}")
print(f"Train={train_gen.samples}, Val={val_gen.samples}, Test={test_gen.samples}")
print("Building MobileNetV2 model...")
base_model = MobileNetV2(weights="imagenet", include_top=False,
input_shape=(IMG_SIZE, IMG_SIZE, 3))
base_model.trainable = False
x = base_model.output
x = layers.GlobalAveragePooling2D()(x)
x = layers.Dropout(0.4)(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dropout(0.3)(x)
output = layers.Dense(1, activation="sigmoid")(x)
model = models.Model(inputs=base_model.input, outputs=output)
model.compile(optimizer=tf.keras.optimizers.Adam(1e-4), loss="binary_crossentropy",
metrics=["accuracy"])
```

```
model.summary()
callbacks = [
    EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True),
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=2, verbose=1),
    ModelCheckpoint("mobilenet_initial.h5", save_best_only=True, monitor="val_loss")  ]
steps_per_epoch = max(1, train_gen.samples // BATCH_SIZE)
validation_steps = max(1, val_gen.samples // BATCH_SIZE)
history_init = model.fit(
    train_gen, steps_per_epoch=steps_per_epoch,
    validation_data=val_gen, validation_steps=validation_steps,
    epochs=INITIAL_EPOCHS, callbacks=callbacks
)
base_model.trainable = True
total_layers = len(base_model.layers)
for i, layer in enumerate(base_model.layers):
    if i < total_layers - FINE_TUNE_UNFREEZE_LAYERS:
        layer.trainable = False
    else:
        layer.trainable = True
model.compile(optimizer=tf.keras.optimizers.Adam(1e-5), loss="binary_crossentropy",
metrics=["accuracy"])
callbacks_ft = [
    EarlyStopping(monitor="val_loss", patience=4, restore_best_weights=True),
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=2, verbose=1),
    ModelCheckpoint("mobilenet_finetuned.h5", save_best_only=True, monitor="val_loss")
]
history_ft = model.fit(
    train_gen, steps_per_epoch=steps_per_epoch,
    validation_data=val_gen, validation_steps=validation_steps,
    epochs=FINETUNE_EPOCHS, callbacks=callbacks_ft)
model.save("mobilenet_final.h5")
print("Saved mobilenet_final.h5")
test_loss, test_acc = model.evaluate(test_gen, verbose=1)
```

```
print(f"Test Loss={test_loss:.4f}, Test Accuracy={test_acc:.4f}")
y_true = test_gen.classes
y_prob = model.predict(test_gen, verbose=1)
y_pred = (y_prob > 0.5).astype("int32").reshape(-1)
print("\nClassification Report:")
print(classification_report(y_true, y_pred, target_names=list(test_gen.class_indices.keys())))
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=list(test_gen.class_indices.keys()),
            yticklabels=list(test_gen.class_indices.keys()))
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()

print("Exporting Flutter-compatible TFLite model...")
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS]
converter._experimental_lower_tensor_index_uses = True
tflite_model = converter.convert()
tflite_path = "deepfake_mobilenet_compatible.tflite"
with open(tflite_path, "wb") as f:
    f.write(tflite_model)
print(f"Saved {tflite_path} (compatible with Flutter tflite_flutter)")
```

Core Dart function for loading the TFLite model and running inference in the Flutter application.

```
import 'dart:io';
import 'dart:typed_data';
import 'package:tflite_flutter/tflite_flutter.dart';
import 'package:image/image.dart' as img;
import 'package:path_provider/path_provider.dart';
import 'package:video_thumbnail/video_thumbnail.dart';
class ModelService {
  Interpreter? _model1;
  Interpreter? _model2;
  bool _modelsInitialized = false;
  Function(double)? _progressCallback;
  bool _cancelFlag = false;
  int framesUsedModel1 = 0;
  int framesUsedModel2 = 0;
  late ModelConfig _model1Config;
  late ModelConfig _model2Config;
  Map<String, dynamic>? _lastAnalysisResult;
  List<String> _analysisHistory = [];
  bool get areModelsInitialized => _modelsInitialized;
  Map<String, dynamic>? get lastAnalysisResult => _lastAnalysisResult;
  List<String> get analysisHistory => _analysisHistory;
  Future<void> initializeModels() async {
    try {
      print('Loading Deep Learning Models...');
      _model1 = await Interpreter.fromAsset(
        'assets/models/face_swap_detector_compatible.tflite',
        options: InterpreterOptions()..threads = 4,
      );
      print('Model 1 (FaceSwapDetector) loaded successfully');
      _model2 = await Interpreter.fromAsset(
        'assets/models/deepfake_mobilenet_compatible.tflite',
```

```
      options: InterpreterOptions()..threads = 4,
    );
    print('Model 2 (MobileNetV2) loaded successfully');
    _model1Config = _configureModel(_model1!);
    _model2Config = _configureModel(_model2!);
    print('Model 1 Configuration:');
    _modelsInitialized = true;
    print('Models initialized successfully');
    await testModelWithSample();
  } catch (e) {
    print('Error loading models: $e');
    rethrow;
  }
}
ModelConfig _configureModel(Interpreter model) {
  final inputTensor = model.getInputTensors().first;
  final outputTensor = model.getOutputTensors().first;
  final preprocessing = PreprocessingType.simpleNormalization;
  return ModelConfig(
    inputShape: List<int>.from(inputTensor.shape),
    inputType: inputTensor.type,
    outputShape: List<int>.from(outputTensor.shape),
    outputType: outputTensor.type,
    preprocessingType: preprocessing,
    inputSize: inputTensor.shape.length > 1 ? inputTensor.shape[1] : 224,
  );
}
void setProgressCallback(Function(double) callback) {
  _progressCallback = callback;
}
void cancelAnalysis() {
  _cancelFlag = true;
}
Future<Map<String, dynamic>> getVideoInfo(String videoPath) async {
```

```
  try {
    final videoFile = File(videoPath);
    final stat = await videoFile.stat();
    final duration = await _getVideoDuration(videoPath);
    return {
      'file_path': videoPath,
      'file_name': videoPath.split('/').last,
      'file_size': stat.size,
      'file_size_mb': (stat.size / (1024 * 1024)).toStringAsFixed(2),
      'duration_seconds': duration.inSeconds,
      'last_modified': stat.modified,
    };
  } catch (e) {
    print('Error getting video info: $e');
    return {
      'file_path': videoPath,
      'file_name': videoPath.split('/').last,
      'file_size': 0,
      'file_size_mb': '0',
      'duration_seconds': 0,
      'last_modified': DateTime.now(),
    };
  }
}
Future<Map<String, dynamic>> analyzeVideo(String videoPath) async {
  _cancelFlag = false;
  framesUsedModel1 = 0;
  framesUsedModel2 = 0;
  try {
    print('Starting video analysis: $videoPath');
    final videoInfo = await getVideoInfo(videoPath);
    final frames = await _extractFrames(videoPath);
    if (_cancelFlag) throw Exception('Analysis cancelled');
    if (frames.isEmpty) throw Exception('No frames extracted from video');
```

```
    print('Extracted ${frames.length} frames from video');
    final model1Results = await _analyzeFramesWithModel(
      frames,
      _model1!,
      _model1Config,
      'Model1',
    );
    final model2Results = await _analyzeFramesWithModel(
      frames,
      _model2!,
      _model2Config,
      'Model2',
    );
    if (_cancelFlag) throw Exception('Analysis cancelled');
    final combinedResult = _combineResults(
      model1Results,
      model2Results,
      videoInfo,
    );
    _lastAnalysisResult = combinedResult;
    _addToHistory(combinedResult, videoInfo);
    _printAnalysisResults(combinedResult);
    return combinedResult;
  } catch (e) {
    print('Video analysis failed: $e');
    rethrow;
  }
}
Future<ModelResults> _analyzeFramesWithModel(
  List<img.Image> frames,
  Interpreter model,
  ModelConfig config,
  String modelName,
) async {
```

```
  final confidences = <double>[];
  int processedFrames = 0;
  for (int i = 0; i < frames.length; i++) {
   if (_cancelFlag) break;
   try {
    final confidence = await _analyzeSingleFrame(frames[i], model, config);
    if (confidence != null) {
     confidences.add(confidence);
     processedFrames++;
    }
    if (_progressCallback != null) {
     final overallProgress = (i + 1) / frames.length;
     _progressCallback!(overallProgress);
    }
    if (i % 5 == 0) {
     print(
    '$modelName - Frame ${i + 1}/${frames.length}: ${confidence?.toStringAsFixed(4) ??
"N/A"}',
      );
    }
   } catch (e) {
    print('$modelName frame ${i + 1} error: $e');
   }
  }
  return ModelResults(
   confidences: confidences,
   processedFrames: processedFrames,
   modelName: modelName,
  );
 }
 Future<double?> _analyzeSingleFrame(
  img.Image frame,
  Interpreter model,
  ModelConfig config,
```

```
) async {
  try {
    final inputBuffer = _preprocessFrame(frame, config);
    if (inputBuffer == null) throw Exception('Preprocessing failed');
    final input = _createInputTensor(inputBuffer, config);
    final output = _createOutputTensor(config);
    model.run(input, output);
    return _interpretModelOutput(output, config);
  } catch (e) {
    print('Frame analysis error: $e');
    return null;
  }
}
Float32List _createInputTensor(List<double> buffer, ModelConfig config) {
  return Float32List.fromList(buffer);
}
dynamic _createOutputTensor(ModelConfig config) {
  final shape = config.outputShape;
  int total = 1;
  for (var s in shape) total *= s;
  if (config.outputType == TensorType.float32) {
    return List<double>.filled(total, 0.0);
  } else if (config.outputType == TensorType.int32) {
    return List<int>.filled(total, 0);
  } else {
    return List<double>.filled(total, 0.0);
  }
dynamic _preprocessFrame(img.Image frame, ModelConfig config) {
  final resized = img.copyResize(
    frame,
    width: config.inputSize,
    height: config.inputSize,
  );
  return _preprocessToFloat32(resized, config);
```

```
}
List<double> _preprocessToFloat32(img.Image frame, ModelConfig config) {
  final buffer = List<double>.filled(
    config.inputSize * config.inputSize * 3,
    0.0,
  );
  int index = 0;
  for (int y = 0; y < config.inputSize; y++) {
    for (int x = 0; x < config.inputSize; x++) {
      final pixel = frame.getPixel(x, y);
      double r = img.getRed(pixel).toDouble();
      double g = img.getGreen(pixel).toDouble();
      double b = img.getBlue(pixel).toDouble();
      switch (config.preprocessingType) {
        case PreprocessingType.simpleNormalization:
          r /= 255.0;
          g /= 255.0;
          b /= 255.0;
          break;
        case PreprocessingType.imagenetStyle:
          r = (r / 255.0 - 0.485) / 0.229;
          g = (g / 255.0 - 0.456) / 0.224;
          b = (b / 255.0 - 0.406) / 0.225;
          break;
        case PreprocessingType.zeroCenter:
          r = (r - 127.5) / 127.5;
          g = (g - 127.5) / 127.5;
          b = (b - 127.5) / 127.5;
          break;
      }
      buffer[index++] = r;
      buffer[index++] = g;
      buffer[index++] = b;
    }
```

```
  }
  return buffer;
}
double _interpretModelOutput(dynamic output, ModelConfig config) {
  try {
    final flatOutput = _flattenOutput(output);
    if (flatOutput.isEmpty) return 0.0;
    switch (config.outputShape.length) {
      case 1:
        return flatOutput[0];
      case 2:
        if (config.outputShape[1] == 1) {
          return flatOutput[0];
        } else if (config.outputShape[1] == 2) {
          return flatOutput.length > 1 ? flatOutput[1] : flatOutput[0];
        } else {
          return flatOutput[0];
        }
      default:
        return flatOutput[0];
    }
  } catch (e) {
    print('Output interpretation error: $e');
    return 0.0;
  }
}
List<double> _flattenOutput(dynamic output) {
  final result = <double>[];
  void flatten(dynamic item) {
    if (item is double) {
      result.add(item);
    } else if (item is int) {
      result.add(item.toDouble());
    } else if (item is List) {
```

```
      for (var element in item) {
        flatten(element);
      }
    }
  }
  flatten(output);
  return result;
}
Map<String, dynamic> _combineResults(
  ModelResults model1Results,
  ModelResults model2Results,
  Map<String, dynamic> videoInfo,
) {
  framesUsedModel1 = model1Results.processedFrames;
  framesUsedModel2 = model2Results.processedFrames;
  final avgConfidence1 = model1Results.averageConfidence;
  final avgConfidence2 = model2Results.averageConfidence;
  final fakeFrames1 = model1Results.fakeFramesCount;
  final fakeFrames2 = model2Results.fakeFramesCount;
  final fakeRatio1 = model1Results.fakeRatio;
  final fakeRatio2 = model2Results.fakeRatio;
  final combinedConfidence = (avgConfidence1 * 0.6 + avgConfidence2 * 0.4);
  final isFake = _determineIfFake(
    avgConfidence1,
    avgConfidence2,
    fakeRatio1,
    fakeRatio2,
    combinedConfidence,
  );
  return {
    'is_fake': isFake,
    'confidence': combinedConfidence,
    'model1_confidence': avgConfidence1,
    'model2_confidence': avgConfidence2,
```

```
    'fake_frames_model1': fakeFrames1,
    'fake_frames_model2': fakeFrames2,
    'total_frames': model1Results.confidences.length,
    'processed_frames_model1': framesUsedModel1,
    'processed_frames_model2': framesUsedModel2,
    'fake_ratio_model1': fakeRatio1,
    'fake_ratio_model2': fakeRatio2,
    'video_info': videoInfo,
    'analysis_timestamp': DateTime.now(),
    'analysis_id': DateTime.now().millisecondsSinceEpoch.toString(),
  };
}
bool _determineIfFake(
  double confidence1,
  double confidence2,
  double fakeRatio1,
  double fakeRatio2,
  double combinedConfidence,
) {
  final highConfidenceFake = combinedConfidence > 0.7;
  final moderateConfidenceWithHighFakeRatio =
      combinedConfidence > 0.4 && (fakeRatio1 > 0.4 || fakeRatio2 > 0.4);
  final bothModelsAgree = confidence1 > 0.6 && confidence2 > 0.6;
  return highConfidenceFake ||
      moderateConfidenceWithHighFakeRatio ||
      bothModelsAgree;
}
void _addToHistory(
  Map<String, dynamic> result,
  Map<String, dynamic> videoInfo,
) {
```

```
  final entry =
      '${videoInfo['file_name']} - ${result['is_fake'] ? 'FAKE' : 'REAL'} '
      '(${(result['confidence'] * 100).toStringAsFixed(1)}%) - '
      '${DateTime.now().toString()}';
  _analysisHistory.add(entry);
  if (_analysisHistory.length > 10) {
    _analysisHistory.removeAt(0);
  }
}
Future<List<img.Image>> _extractFrames(String videoPath) async {
  try
    final frames = <img.Image>[];
    final videoFile = File(videoPath);
    if (!await videoFile.exists()) {
      throw Exception('Video file not found: $videoPath');
    }
    final duration = await _getVideoDuration(videoPath);
    final totalSeconds = duration.inSeconds;
    final numFrames = totalSeconds > 16 ? 16 : totalSeconds;
    if (numFrames == 0) throw Exception('Video is too short');
    print('Extracting $numFrame frames from ${totalSeconds}s video...');
    for (int i = 0; i < numFrames; i++) {
      if (_cancelFlag) break;
      final timeMs = (i * totalSeconds * 1000 ~/ numFrames);
      try {
        final uint8list = awaitVideoThumbnail.thumbnailData(
          video: videoPath,
          imageFormat: ImageFormat.PNG,
          maxWidth: 512,
          quality: 75,
          timeMs: timeMs,
        );
        if (uint8list != null)
          final image = img.decodeImage(uint8list);
```

```
        if (image != null) {
          frames.add(image);
        }
      }
      if (_progressCallback != null) {
        _progressCallback!((i + 1) / numFrames * 0.5);
      }
    } catch (e) {
      print('Error extracting frame at ${timeMs}ms: $e');
    }
  }
  return frames;
 } catch (e) {
  print('Frame extraction error: $e');
  rethrow;
 }
}
Future<Duration> _getVideoDuration(String videoPath) async {
 try {
  final videoFile = File(videoPath);
  final fileSize = await videoFile.length();
  const averageBitrate = 2000000;
  final estimatedSeconds = fileSize * 8 / averageBitrate;
  return Duration(seconds: estimatedSeconds.toInt().clamp(1, 60));
 } catch (e) {
  print('Could not determine video duration, using default: $e');
  return Duration(seconds: 30);
 }
}
Future<void> testModelWithSample() async {
 if (!_modelsInitialized) return;
 try {
  print('Testing models with sample input...');
```

```
    final sampleImage = img.Image(
      _model1Config.inputSize,
      _model1Config.inputSize,
    );
    for (int y = 0; y < _model1Config.inputSize; y++) {
      for (int x = 0; x < _model1Config.inputSize; x++) {
        sampleImage.setPixel(x, y, img.getColor(128, 128, 128));
      }
    }
    final result1 = await _analyzeSingleFrame(
      sampleImage,
      _model1!,
      _model1Config,
    );
    final result2 = await _analyzeSingleFrame(
      sampleImage,
      _model2!,
      _model2Config,
    );
    print('Test Results:');
    print('  - Model 1: ${result1?.toStringAsFixed(4) ?? "N/A"}');
    print('  - Model 2: ${result2?.toStringAsFixed(4) ?? "N/A"}');
  } catch (e) {
    print('Model test failed: $e');
  }
}
void clearHistory() {
  _analysisHistory.clear();
  _lastAnalysisResult = null;
}
Map<String, String> getFormattedResults() {
  if (_lastAnalysisResult == null) {
```

```
      return {
        'status': 'No analysis performed',
        'confidence': '0%',
       'details': 'Analyze a video to see results',
      };
    }
    final result = _lastAnalysisResult!;
    return {
      'status': result['is_fake'] ? 'FAKE DETECTED' : 'REAL VIDEO',
      'confidence': '${(result['confidence'] * 100).toStringAsFixed(1)}%',
      'details':
          'Model 1: ${(result['model1_confidence'] * 100).toStringAsFixed(1)}% | '
          'Model 2: ${(result['model2_confidence'] * 100).toStringAsFixed(1)}%',
      'frames_analyzed':
          '${result['processed_frames_model1'] + result['processed_frames_model2']} frames',
    };
  }
  void dispose() {
    _model1?.close();
    _model2?.close();
    _modelsInitialized = false;
  }
}
class ModelConfig {
  final List<int> inputShape;
  final TensorType inputType;
  final List<int> outputShape;
  final TensorType outputType;
  final PreprocessingType preprocessingType;
  final int inputSize;
  ModelConfig({
    required this.inputShape,
    required this.inputType,
```

```
    required this.outputShape,
    required this.outputType,
    required this.preprocessingType,
    required this.inputSize,
  });
}
class ModelResults {
  final List<double> confidences;
  final int processedFrames;
  final String modelName;
  ModelResults({
    required this.confidences,
    required this.processedFrames,
    required this.modelName,
  });
  double get averageConfidence {
    if (confidences.isEmpty) return 0.0;
    return confidences.reduce((a, b) => a + b) / confidences.length;
  }
  int get fakeFramesCount {
    return confidences.where((c) => c > 0.5).length;
  }
  double get fakeRatio {
    if (processedFrames == 0) return 0.0;
    return fakeFramesCount / processedFrames;
  }
}
enum PreprocessingType { simpleNormalization, imagenetStyle, zeroCenter }
```

Appendix B: Full Classification Reports

Model 1: Custom CNN

Classification Report:

	precision	recall	f1-score	support
REAL	0.97	0.92	0.94	399
FAKE	0.92	0.97	0.95	398
accuracy			0.94	797
macro avg	0.95	0.94	0.94	797
weighted avg	0.95	0.94	0.94	797

Model 2: MobileNetV2

Classification Report:

	precision	recall	f1-score	support
real	0.94	0.93	0.94	935
fake	0.93	0.95	0.94	937
accuracy			0.94	1872
macro avg	0.94	0.94	0.94	1872
weighted avg	0.94	0.94	0.94	1872