SIGNATURE FRAUD DETECTION USING DEEP LEARNING WITH COMBINED CNN AND SIFT

P. HARI BHARADWAJ 22951A1221

K. ARAVIND 22951A1228

P. PRADHAM KUSHAL REDDY 23955A1207

SIGNATURE FRAUD DETECTION USING DEEP LEARNING WITH COMBINED CNN AND SIFT

A Project Report
submitted in partial fulfilment of
the requirement for the award of the degree of

Bachelor of Technology in
Information Technology
by

P. Hari Bharadwaj - 22951A1221
K. Aravind - 22951A1228
P. Pradham Kushal Reddy - 23955A1207

Department of Information Technology
INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal, Hyderabad – 500 043, Telangana

May, 2025

DECLARATION

We certify that

a. The work contained in this report is original and has been done by us under the guidance of our supervisor(s).

b. The work has not been submitted to any other Institute for any degree or diploma.

c. We have followed the guidelines provided by the Institute in preparing the report.

d. We have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever we have used materials (data, theoretical analysis, figures, and text) from other sources, we have given due credit to them by citing them in the text of the report and giving their details in the references. Further, we have taken permission from the copyright owners of the sources, whenever necessary.

Place:	**Signature of the Student**
Date:	**22951A1221**
	22951A1228
	23955A1207

CERTIFICATE

This is to certify that the project report **Signature Fraud detection using deep learning with combined CNN and SIFT** by **Mr. P. Hari Bharadwaj, Mr. K. Aravind, Mr. P. Pradham Kushal Reddy,** to the Institute of Aeronautical Engineering, Hyderabad in partial fulfillment of the requirements for the award of the Degree Bachelor of Technology in **Information Technology** is a bonafide record of work carried out by them under my guidance and supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute for the award of any Degree.

Supervisor	**Head of Department**
Mrs. K. Venkata Ramana Devi	**Dr. M Purushotham Reddy**

Date:

APPROVAL SHEET

This project report entitled **Signature Fraud detection using deep learning with combined CNN and SIFT** by **Mr. P. Hari Bharadwaj, Mr. K. Aravind, Mr. P. Pradham Kushal Reddy,** is approved for the award of the Degree Bachelor of Technology in **Information Technology.**

Examiner **Supervisor**

Mrs. K. Venkata Ramana Devi

Principal

Dr. L V Narasimha Prasad

Date:

Place: Dundigal

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We thank our college management and **Sri. M. Rajashekar Reddy,** Chairman, for providing us with the necessary infrastructure to carry out the project work.

We express our sincere thanks to **Dr. L V Narasimha Prasad,** beloved Principal who has been a great source of information for our work and to **Dr. M Purushotham Reddy,** Professor and Head, Department of Information Technology, for extending his support to carry out this project work.

We are especially thankful to our internal supervisor **Mrs. K. Venkata Ramana Devi**, Assistant Professor, Department of Information Technology, for his internal support and professionalism who helped us in shaping the project into a successful one. A special thanks to our beloved Project Coordinator **Mr. U. Sivaji,** Associate Professor, Department of Information Technology

We take this opportunity to express our thanks to one and all who directly or indirectly helped us in bringing this effort to present form.

P. Hari Bharadwaj 22951A1221

K. Aravind 22951A1228

P. Pradham Kushal Reddy 23955A1207

ABSTRACT

Secure authentication plays a major role in ensuring the identity of a person. Everyone has an individual and unique signature, which is used as their personal identification in all their legal transactions. Even though a lot of things got digitalized, traditional signatures are still used in a lot of places, such as check payments and government offices, and they still rely on a human manually verifying them. So, forgery detection plays a key role in reducing these kinds of overhead. Manual verification is not only difficult to check if two signatures are the same but also very time-consuming. Pandemic further made people do tasks digitally, which also included uploading their own signatures digitally. This increases the urgency of implementing a system to identify and verify the user’s signature. This paper proposes a method to pre-process the signature to make verification simple as well as use methods like the Convolution Neural Network (CNN), Scale-Invariant Feature Transform (SIFT) to identify forged signatures and compare the results obtained with various parameters.

Keywords: Signature Verification, Forgery Detection, Convolutional Neural Network (CNN), SIFT, Deep Learning, Feature Fusion, Dual CNN Architecture, Image Preprocessing.

CONTENTS

Project Description		I
Declaration		II
Certificate		III
Approval Sheet		IV
Acknowledgement		V
Abstract		VI
Contents		VII
List of Figures		IX
Chapter 1	**Introduction**	1
	1.1 Objective	2
	1.2 Scope of study	3
	1.3 Introduction	3
	1.4 Existing System	4
	1.4.1 Limitations of Existing System	6
	1.5 Proposed System	7
	1.5.1 Advantages of Proposed System	10
Chapter 2	**Literature Survey**	12
	2.1 Requirement Specifications	17
	2.1.1 Hardware Requirements	17
	2.1.2 Software Requirements	17
	2.1.3 Functional Requirement	17
	2.1.4 Non-Functional Requirements	18
Chapter 3	**Design**	19
	3.1 UML Diagrams	19
	3.1.1 Use Case Diagram	20
	3.1.2 Class Diagram	21
	3.1.3 Sequence Diagram	22
Chapter 4	**Methodology and Implementation**	23
	4.1 Modules	23
	4.2 Module Descriptions	23
	4.2.1 Data Collection and Preprocessing Module	23

	4.2.2 Model Training and Validation Module	23
	4.2.3 Model Evaluation Module	24
	4.3 Python Implementation	24
	4.3.1 What is Python	24
	4.3.2 Advantages of Python	25
	4.4 Source Code	25
Chapter 5	**Testing**	33
	5.1 Types of Testing	33
	5.1.1 Unit Testing	33
	5.1.2 Integration Testing	34
	5.1.3 Functional Testing	34
	5.1.4 System Testing	35
	5.1.5 White box Testing	35
	5.1.6 Black box Testing	36
	5.1.7 Acceptance Testing	36
	5.2 System Test Strategy	37
Chapter 6	**Result**	38
Chapter 7	**Conclusion**	46
Chapter 8	**Future Scope**	47
References		49

LIST OF FIGURES

Figure	**Title**	**Page**
1.5.1	CNN Architecture	8
1.5.2	Keypoint Detection using SIFT Algorithm	9
3.1.1	Use Case Diagram	20
3.1.2	Class Diagram	21
3.1.3	Sequence Diagram	22
6.1	Loading and importing required python classes	39
6.2	Unzip Dataset and Extract genuine and forged images	39
6.3	Extracting CNN and SIFT features from images	40
6.4	Training and saving model	40
6.5	Models saved in model_output	41
6.6	Extracting CNN and SIFT features in Flask	41
6.7	Compare Images	42
6.8	Signatures Loaded	42
6.9	Model definition total params and trainable params	43
6.10	Epochs	43
6.11	Upload Two Images	44
6.12	Reference Image	44
6.13	Test Signature Image	45
6.14	Result	45

CHAPTER 1
INTRODUCTION

In an increasingly digital world, the need for secure and reliable identity verification systems is greater than ever. Despite advancements in biometric technologies like fingerprint recognition, iris scanning, and facial recognition, handwritten signatures remain one of the most widely accepted and legally recognized methods of authentication. Signatures are not only culturally ingrained but are also simple to use and require no additional hardware. However, their widespread use also makes them a prime target for forgery and fraudulent activities, especially in sectors such as banking, education, government, and legal documentation.

To address these challenges, this project proposes an automated and intelligent approach for signature verification using a hybrid model that combines Convolutional Neural Networks (CNN) and Scale-Invariant Feature Transform (SIFT). By fusing CNN and SIFT features, the proposed system captures both global structures and local keypoints, resulting in a highly robust model capable of detecting even subtle forgeries.

The project involves preprocessing signature images, extracting key features using both methods, combining the features through a fusion strategy, and training a classifier to distinguish between genuine and forged signatures. This hybrid approach not only improves the accuracy and robustness of signature verification systems but also lays the foundation for scalable real-world applications in identity verification. It demonstrates the potential of combining traditional computer vision techniques with modern deep learning models to build more resilient biometric systems.

1.1 Objective

This project, "Signature Fraud Detection using Deep Learning with Combined SIFT and CNN," aims to achieve the following objectives:

- **Development of a Hybrid Deep Learning Model:** Implemented an advanced architecture that combines Scale-Invariant Feature Transform (SIFT) for robust keypoint extraction with Convolutional Neural Networks (CNNs) for learning intricate patterns from signature images.

- **Dataset Acquisition and Preprocessing:** Utilized CEDAR dataset consisting of genuine and forged signature images. The dataset is pre-processed to enhance model performance by standardizing input dimensions, normalizing pixel values, and potentially augmenting data to improve generalization.

- **Performance Evaluation:** Assess the performance of the implemented model using standard classification metrics such as Accuracy, Precision, Recall, F1-score, and potentially the Area Under the ROC Curve (AUC). This will allow for a quantitative comparison of the model's ability to detect fraudulent signatures.

- **Development of a User Interface (UI) Application:** Implemented a Flask-based web application to provide a user-friendly interface. This application will allow users to upload two signature images for comparison and receive an output indicating the likelihood of the test signature being fraudulent.

- **Practical Applications:** Explore the practical applications of the developed signature fraud detection system, particularly in enhancing security for financial institutions, legal document verification, and preventing identity theft.

1.2 Scope of study

The scope of this study encompasses the training and evaluation of a hybrid SIFT-CNN deep learning model using a dataset of genuine and forged signature images.

Performance will be evaluated using established classification metrics, alongside qualitative assessments of the system's ability to correctly identify genuine and forged signatures under various conditions. Additionally, the development of a Flask-based UI will demonstrate the practical usability of the system, enabling users to compare two signature images and receive a prediction.

The study will also consider the real-world implications of the developed model, particularly in enhancing security measures and automating the process of signature verification in sectors vulnerable to fraud.

1.3 Introduction

Signature verification is a critical biometric technique used for personal identification and authentication, widely employed in financial transactions, legal documents, and access control systems. The task involves determining whether a questioned signature is genuine or a forgery. While human visual inspection has been the traditional method, it is subjective, time-consuming, and prone to error, especially with skilled forgeries. This necessitates the development of automated and reliable signature fraud detection systems.

Automated signature fraud detection presents several challenges:

- **Intra-personal Variability:** An individual's genuine signature naturally varies over time and with different writing conditions.
- **Inter-personal Similarity (Skilled Forgeries):** Forgers often attempt to meticulously replicate genuine signatures, making them visually very similar.
- **Data Scarcity:** Obtaining large, diverse, and well-annotated datasets of genuine and forged signatures can be difficult.

This project specifically addresses these challenges by implementing a hybrid deep learning model that combines the strengths of Scale-Invariant Feature Transform (SIFT) and Convolutional Neural Networks (CNNs).

- SIFT is renowned for its ability to extract distinctive local features that are invariant to scale, rotation, and illumination changes, making it effective for capturing stable keypoints in signature strokes.
- CNNs excel at automatically learning hierarchical feature representations from raw pixel data, capturing complex patterns and global characteristics of the signatures that might be missed by handcrafted feature extractors.

By integrating SIFT features with features learned by a CNN, this project aims to create a more robust and accurate system. SIFT can provide crucial local information about stroke details and junctions, while the CNN can learn broader contextual patterns and subtle distinctions indicative of genuine or forged signatures.

To train and evaluate the model, a dataset comprising both genuine signatures and various types of forgeries (e.g., random, simple, skilled) is crucial. This allows the model to learn the nuances that differentiate authentic signatures from imitations. The development of a Flask-based user interface will further serve as a practical demonstration, allowing users to upload two signature images for comparison and receive an assessment from the model.

Ultimately, the ability to accurately and efficiently detect signature fraud has significant implications for enhancing security, reducing financial losses, and improving trust in various authentication processes. By advancing signature verification technology, this project aims to contribute to the development of more intelligent and reliable systems for combating fraud.

1.4 Existing System

Automated signature verification and fraud detection have been active research areas for decades, with various approaches developed to distinguish between genuine and forged signatures. These systems can be broadly categorized based on the features extracted and the classification techniques employed.

Key existing approaches include:

- **Manual Visual Inspection:** The most traditional method, relying on human experts to scrutinize signatures. While experts can be highly accurate, this process is subjective, time-consuming, and not scalable for high-volume applications.
- **Handcrafted Feature-Based Systems (Offline):**
    - **Global Features:** These systems extract features representing the overall characteristics of the signature, such as aspect ratio, height, width, slant, number of connected components, and projection profiles.
    - **Local Features:** These focus on finer details, like critical points, stroke curvature, edge points, and grid-based features.
    - **Classifiers:** Extracted features are then fed into classical machine learning classifiers like Support Vector Machines (SVMs), k-Nearest Neighbors (k-NN), Hidden Markov Models (HMMs), or Random Forests.
- **Local Invariant Feature Descriptors:**
    - **SURF, ORB:** Techniques like Speeded UP Robust Features (SURF) have been used to extract distinctive keypoints from signature images. These keypoints are then matched between a reference and a questioned signature, or their descriptors are used as features for a classifier. This approach offers robustness to minor variations, scaling, and rotation.
- **Pure Deep Learning Approaches (CNNs):**
    - **Direct Classification:** CNNs are trained end-to-end to learn discriminative features directly from the raw pixel data of signature images to classify them as genuine or forged.
    - **Siamese Networks/Triplet Networks:** These architectures are commonly used for signature verification. They learn a feature embedding where signatures from the same person are close in the feature space, and signatures from different people (or a genuine vs. a forgery) are far apart. The CNN acts as the feature extractor within these networks.
    - **Transfer Learning:** Similar to other image tasks, CNNs pre-trained on large image datasets (like ImageNet) are often used as a starting point, with fine-tuning performed on signature datasets to leverage learned general image features.

- **Online Signature Verification Systems:** These systems capture dynamic information during the signing process (e.g., pen pressure, speed, acceleration, stroke order) using specialized hardware. While often more accurate due to richer information, they are not applicable to verifying static, scanned signatures, which is the focus of our project.

Evaluation of these systems typically involves metrics like False Acceptance Rate (FAR), False Rejection Rate (FRR), Equal Error Rate (EER), and overall Accuracy. Datasets like CEDAR, GPDS, MCYT, and the Brazilian PUCPR signature dataset are commonly used for training and benchmarking.

While these existing systems have achieved considerable success, they also possess certain limitation.

1.5 Limitations of the Existing System

- **Vulnerability to Skilled Forgeries:** Many systems, especially those relying solely on global features or simpler local features, can be deceived by skilled forgeries that closely mimic the genuine signature's overall shape and key characteristics.
- **High Intra-Class Variability:** Genuine signatures from the same individual can exhibit significant variations due to factors like mood, writing conditions, or time. This makes it challenging to define a tight boundary for genuine samples.
- **Sensitivity of Handcrafted Features:** Systems relying on handcrafted features require significant domain expertise for feature selection and engineering. These features might not be robust enough to capture all the subtle nuances differentiating genuine signatures from sophisticated forgeries.
- **Generalization Issues:** Models trained on one specific dataset or a limited set of writers may not generalize well to signatures with different styles, written with different pens, or on different types of paper.
- **Dependency on Large and Diverse Datasets:** Deep learning models, particularly pure CNN approaches, require substantial amounts of labeled genuine and forged signature data for effective training. Acquiring high-quality, diverse forged signature data is often a significant challenge.

- **Interpretability of Deep Models:** While CNNs can achieve high accuracy, their "black-box" nature makes it difficult to understand *why* a particular signature was classified as genuine or fraudulent, which can be a concern in critical applications.
- **Feature Fusion Challenges:** Systems attempting to combine different types of features (e.g., SIFT with other global features) might not always effectively integrate them, leading to suboptimal performance if not carefully designed.
- **Overfitting Risks:** With limited or imbalanced signature datasets, models (especially complex deep learning ones) are prone to overfitting, performing well on training data but poorly on unseen signatures.

Overall, while existing systems have advanced the field of signature fraud detection, the proposed system, by combining the strengths of SIFT for robust local keypoint detection and CNNs for powerful hierarchical feature learning.

1.6 Proposed System

The proposed system leverages a hybrid deep learning and feature engineering approach to enhance the accuracy and reliability of automatic signature verification. By integrating a Convolutional Neural Network (CNN) for holistic visual feature extraction with Scale-Invariant Feature Transform (SIFT) for robust local keypoint analysis, this system addresses the challenges of distinguishing genuine signatures from sophisticated forgeries.

The outputs from both the CNN and SIFT processing branches are then fused and fed into a classifier to make a prediction on whether a signature is forged. Furthermore, for signatures classified as potentially genuine by the model, an additional direct SIFT keypoint matching step is performed between the reference and test signature.

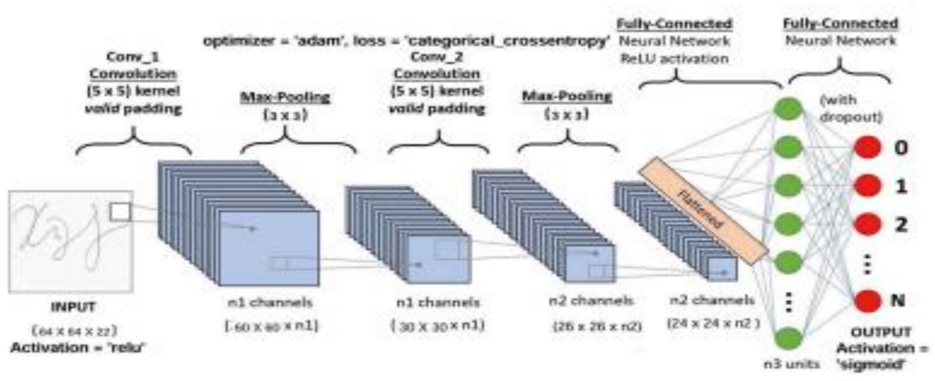

optimizer = 'adam', loss = 'categorical_crossentropy'
Conv_1 Convolution (5 x 5) kernel valid padding
Max-Pooling (2 x 2)
Conv_2 Convolution (5 x 5) kernel valid padding
Max-Pooling (2 x 2)
Fully-Connected Neural Network ReLU activation
Fully-Connected Neural Network
(with dropout)
Flattened
0
1
2
N
INPUT (64 x 64 x 22) Activation = 'relu'
n1 channels (60 x 60 x n1)
n1 channels (30 x 30 x n1)
n2 channels (26 x 26 x n2)
n2 channels (24 x 24 x n2)
n3 units
OUTPUT Activation = 'sigmoid'

Fig. 1.5.1 CNN Architecture

Fig 1.5.1 shows the CNN Architecture. Convolutional Neural Networks (CNNs) have tested no-hit in recent years at an outsized variety of image processing-based machine learning tasks. Several different strategies of playacting such tasks as shown in figure revolve around a method of feature extraction, during which hand-chosen options extracted from a picture fed into a classifier to make a classification call.

Such processes solely as sturdy because of the chosen options, which regularly take giant amounts of care and energy to construct. Against this, in CNN, the options fed into the ultimate linear classifier all learned from the dataset. A CNN consists of a variety of layers as shown in figure, beginning at the raw image pixels, that each performs an easy computation and feeds the result to the successive layer, with the ultimate result being fed to a linear classifier.

The layers computation area unit supports a variety of parameters that learned through the method of back propagation, during which for every parameter, the gradient of the classification loss with relation to that parameter is computed and therefore the parameter is updated to minimize the loss perform. The look of any signature verification system typically needs the answer of 5 sub-issues: data retrieval, pre-processing, feature extraction, identification method, and performance analysis.

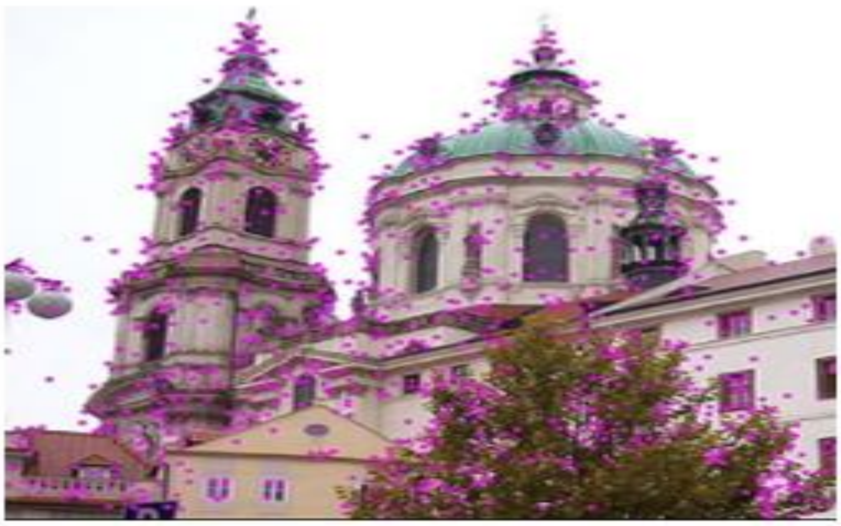


Fig. 1.5.2 Keypoint detection using SIFT Algorithm

Fig 1.5.2 shows keypoint detection using SIFT Algorithm, which is a computer vision algorithm to detect, describe, and match local features in images, invented by David Lowe in 1999. Applications include object recognition, robotic mapping and navigation, image stitching, 3D modeling, gesture recognition, video tracking, individual identification of wildlife and match moving.

The dataset used for training this model consists of 2,644 images, each classified into genuine and forged signatures. This variety helps the model learn nuanced relationships between genuine and forged, ultimately improving its ability to generalize across different image types.

In addition to the core hybrid architecture, the system's design incorporates established practices for model development and evaluation. The CNN component, while custom-designed for this task, is built to efficiently learn discriminating visual features directly from the signature dataset. Similarly, SIFT, as a well-established feature descriptor, inherently brings robust, pre-defined local feature extraction capabilities, analogous to leveraging learned representations.

To evaluate the effectiveness of the proposed model, standard classification performance metrics are utilized. The training script directly monitors and reports accuracy and loss on a validation set. From the binary classification output, other crucial metrics such as precision, recall, and F1-score could be derived to provide a comprehensive understanding of the model's performance in distinguishing genuine signatures from forgeries, although these are not explicitly calculated in the provided training script.

The ultimate goal of the proposed system is to provide a reliable and automated tool for verifying the authenticity of signatures. This is beneficial for applications such as document verification, fraud detection in financial transactions, and securing access control systems where signature validation is a critical component. The user interface, built with Flask, is designed to be intuitive, allowing users to easily upload a reference signature and a test signature.

1.6.1 Advantages of Proposed System

The proposed signature verification system offers several significant advantages that enhance its effectiveness and usability in various security and authentication applications:

- **Improved Verification Accuracy:** By employing a hybrid approach that combines a Convolutional Neural Network (CNN) for holistic visual pattern recognition and Scale-Invariant Feature Transform (SIFT) for robust local keypoint analysis, the system achieves a higher accuracy in distinguishing genuine signatures from forgeries. This improvement over single-method approaches allows for more reliable verification outcomes.

- **Enhanced Feature Representation for Signatures:** The dual-feature extraction strategy (CNN and SIFT) enables the model to capture a comprehensive set of characteristics from the signature. The CNN learns global stylistic patterns, while SIFT focuses on detailed, invariant local features, leading to a richer and more discriminative representation of the signature.

- **Robustness to Common Signature Variations:** The use of SIFT features, inherently designed to be robust against scale, rotation, and minor affine transformations, coupled with a model trained on diverse examples of genuine and forged signatures, improves the system's ability to handle natural variations in signing while still detecting forgery attempts.

- **Efficient Processing for Practical Use:** The integration of an optimized CNN architecture and efficient SIFT feature extraction allows the system to process signature images and deliver verification results relatively quickly. This feature is crucial for applications requiring timely authentication, such as document processing or transaction approvals.
- **Increased Objectivity and Consistency:** The automated nature of the proposed system minimizes the subjectivity and potential for human error associated with manual signature inspection. This ensures consistent and objective verification, leading to a more reliable and repeatable process.

- **User-Friendly Interface for Application:** The system includes a web-based interface (built with Flask) designed for ease of use, allowing users to simply upload a reference signature and a test signature to receive a verification result. This facilitates straightforward integration into workflows requiring signature checks.

- **Modular and Adaptable Architecture:** The system's architecture, with distinct modules for CNN processing, SIFT feature extraction, and a final decision logic, allows for future enhancements. This could include incorporating additional feature types, refining model components, or adapting to larger and more diverse signature datasets.

- **Quantitative Performance Monitoring:** The system utilizes standard classification metrics such as accuracy and loss during the training and validation phases. This focus on objective metrics allows for continuous improvement and data-driven refinement of the model.

CHAPTER 2

LITERATURE SURVEY

Ruben Tolosana, Ruben Vera-Rodriguez, Julian Fierrez introduced the DeepSignDB on-line handwritten signature database. Their work, likely detailed in an article such as " DeepSign: Deep On-Line Signature Verification" (2021) [1], focused on establishing the largest on-line signature database to date, comprising over 70,000 signatures from 1,526 users acquired via stylus and finger inputs across office and mobile scenarios using 8 different devices. The authors demonstrated that their adapted Time-Aligned Recurrent Neural Networks (TA-RNNs) approach, which innovatively combines Dynamic Time Warping with Recurrent Neural Networks to enhance robustness against forgeries, achieved state-of-the-art results on DeepSignDB, yielding EERs below 2.0% even for skilled forgeries when using only one training signature per user. This study provides a substantial public resource and a robust baseline, encouraging the research community to utilize DeepSignDB for fair comparisons with state-of-the-art methods, evaluating novel deep learning architectures, analyzing challenging finger-input scenarios, and potentially for neuromotor studies in e-learning and e-health.

Jivesh Poddara , Vinanti Parikha , Santosh Kumar Bhartia developed and evaluated a signature recognition and verification system. Their work, potentially titled " Offline Signature Recognition and Forgery Detection using Deep Learning" (2020) [2], focused on accurately identifying signature holders and detecting forgeries with high precision. The authors detailed a system where recognition is trained on Convolutional Neural Networks using a dataset of 1320 images, while forgery detection is trained on an individual's complete image set (around twenty-five pictures), with calculations performed at runtime to minimize classification errors. They highlighted the use of entirely different threshold values for feature matching on testing and training vectors, which significantly boosted the overall performance and efficiency. The study reported encouraging results, emphasizing the system's economy in real-time forgery detection and its critical importance for applications like enforcement, security management, and authenticating documents.

Sergey Zagoruyko, Nikos Komodakis demonstrated a method for learning a general similarity function for image patches directly from raw image pixels using Convolutional Neural Network (CNN) models. In their work, potentially titled " Learning to compare image patches via convolutional neural networks." (2015) [3], they explored several neural network architectures specifically designed for this task. The authors showed that these architectures achieved extremely good performance, significantly outperforming existing state-of-the-art methods on multiple problems and benchmark datasets. Their findings highlighted that 2-channel-based architectures were clearly superior in terms of results, while for siamese-based architectures, 2-stream multi-resolution models and SPP-based siamese networks proved to be extremely strong, consistently boosting performance and confirming the importance of multi-resolution information in patch comparison. This study suggests that further performance improvements could be gained by investigating methods to accelerate the evaluation of 2-channel networks and by utilizing larger training datasets.

Fahmy, M. M. presented an online handwritten signature verification system. Their paper, potentially titled " Online handwritten signature verification system based on DWT features extraction and neural network classification." (2010) [4], detailed a methodology that utilized Discrete Wavelet Transform (DWT) to decompose pen position and movement angle parameters into low-frequency (approximation) and high-frequency (detail) sub-bands, with a focus on using these features, particularly approximations, as inputs to a multi-matcher system composed of six neural networks. Through experiments on a database of 5 users (20 genuine and 20 skilled forgeries each), the authors demonstrated that selecting specific DWT features which enhance interpersonal variations and suppress intrapersonal ones significantly improved performance. This research highlights the efficacy of DWT sub-band analysis and careful feature selection for distinguishing genuine signatures from skilled forgeries in an online verification context.

Ass Prof. Manjula Devi P, Maaz Ahmed, Mohammad Faiz Alam, Ashay R employed a fully Siamese Neural Network (SNN) classification to develop an impeccable handwriting signature verification method. Their study, potentially titled " Signature Forgery Detection Using Cnn And Hog" (2024) [5], outlined a scheme involving data collection, preprocessing, CNN-based feature selection and extraction, classification, and triple loss evaluation. The authors detailed a data categorization strategy into anchor images, positive (genuine but slightly varied) images, and negative (forged) images. The system was trained using TensorFlow methods on an Ubuntu 18.04 CPU, leveraging the platform's speed, security, and readily available TensorFlow and Python packages for straightforward implementation of the CNN architecture. This research highlights the application of SNNs with a triplet loss function as an effective approach for robust signature verification.

Priti Bharti , Chitra Natarajan successfully developed a model capable of learning from signatures to determine their authenticity. Their study, potentially titled " Deep Learning-Based Signature Verification" (2023) [6], highlighted the application of this method in government offices requiring handwritten signature authentication. The authors utilized Convolutional Neural Networks (CNNs) for signature learning but noted that the structure of their fully connected layer was not yet optimal. They established a paradigm where, for each of the 30 users, two distinct classes—"Real" and "forgery"—were created, resulting in a model designed to predict among 60 classes. This research underscores the potential of CNNs for signature verification tasks in critical applications, while also acknowledging areas for architectural refinement.

Harshit Sharma presented an offline bank cheque signature verification scheme. Their paper, titled "Offline Bank Cheque Signature Verification using SIFT" (2019) [7], described a method that leverages the SIFT (Scale-Invariant Feature Transform) algorithm to utilize local features of signature images for verification. The authors detailed a matching process where an input signature image is compared against all stored signature images. A match is considered successful if the maximum number of key points from the input image aligns with a stored image.

Upasna Jindal, Surjeet Dalal based on a survey of existing literature, concluded that local features from signature images can be effectively evaluated for verification. Their analysis, potentially summarized in a review titled " Survey on Signature verification and recognition using SIFT and its variant" (2016) [8], indicated that SIFT (Scale-Invariant Feature Transform) and its variants are optimal for local feature extraction from various image types, including signatures. The authors observed that prevalent approaches in signature verification involve preprocessing the signature image and then extracting features based on the signature field area and key point descriptions to form feature vectors, primarily to mitigate forgery. This survey suggests that these novel local feature-based approaches demonstrate significant experimental value and effectiveness in the domain of signature verification.

P. Bhuvaneswari, K. Melvin Christopher, V. Rishi Mahesh Raj, M. Ajithkumar underscored advancements in handwritten signature recognition, aiming to develop a deep learning-based system capable of discerning genuine from forged signatures. Their study, potentially titled "Signature Forgery Detection Using Deep Learning " (2024) [9], detailed a project involving the collection and preprocessing of a dataset, the implementation of the VGG16 model with transfer learning for signature recognition, and the assessment of the system's performance through diverse metrics using a merged dataset. The authors emphasized the primary finding that training on larger and more diverse datasets is significant for enhancing robustness and generalization capabilities. They also highlighted that the future development of a user-friendly interface would augment the system's accessibility and usability, thereby optimizing its efficacy and propelling advancements in automated signature recognition technology across various applications.

Jose A.P.L, B. Baptista, N. Lavado, M. Mendes investigated automatic verification methods for handwritten signatures on attendance sheets. Their work, possibly titled "Offline Handwritten Signature Verification Using Deep Learning" (2022) [10], explored two approaches: (1) an MLP classifier for validating the presence of handwritten marks, and (2) a CNN model for individual signature recognition and forgery detection. The authors found that while the optical mark classification (MLP) model achieved high accuracy for mark detection, it did not identify the author. The proposed CNN model demonstrated positive results for individual signature recognition even with limited genuine samples (as few as 10) and minimal information in the region of interest, dropout layers, and batch normalization. They noted that a limitation is the requirement for training examples of signatures to be recognized and suggested that incorporating local feature extraction or exploring other neural network architectures could enhance effectiveness. For real-world application, future work includes the development of a Graphical User Interface for ease of use by campus staff.

2.2 Requirements Specifications

2.2.1 Hardware Requirements:

• Processor	: Intel Core i5 or AMD Ryzen 5 (or higher)
• GPU	: NVIDIA GTX 1080 or higher with CUDA support
• RAM	: 16GB (Recommended)
• Storage	: 50GB of free SSD space
• Keyboard	: Standard keyboard
• Mouse	: Two or three-button mouse

2.2.2 Software Requirements:

• Operating System	: A 64-bit version of Windows 10, macOS, or Ubuntu Linux
• Programming Language	: Python 3.7 or higher
• Deep Learning Frameworks	: TensorFlow 2.17, Keras 3.4.1
• Libraries	: OpenCV,Numpy, Flask, scikit-learn, tqdm
• Development Environment	: Google Colab or VS Code

2.2.3 Functional Requirements:

- Load and Pre-process Signature Image Dataset (CEDAR)
- Extract Keypoints and Descriptors using SIFT
- Train Convolutional Neural Network on processed signature data
- Combine SIFT features with CNN for enhanced classification
- Classify signature as Genuine or Forged
- Display prediction result in user-friendly interface (optional)
- Visualize SIFT keypoints and CNN activation maps

2.2.4 Non-Functional Requirements:

- Scalability
- Reliability
- Performance
- Usability
- Maintainability
- Accessibility
- Portability

CHAPTER 3
DESIGN

3.1 UML Diagrams

UML stands for Unified Modeling Language or SFL (Standards for Facilities). UML is a general-purpose object-orientated modeling language that has become popular in the field of software engineering and is based on standard principles. Control is done, and it is also created by, that particular group. The Object Management Group.

The intention is that UML models become a universal standard for the creation of object-centric software diagrams. In its current form UML is comprised of two major components: finally, we will create a Meta model and a notation. of this, we will Model our content Meta and the notation. In the future, UML can also be supported by a method or mechanism that is either related to or connected with UML by some means.

Unified Modeling Language is standard language for modeling unambiguously replacing the descriptions which are visual, explicit and also documentation ones of software through the use of the artifacts. The application of the modeling includes business modeling as well as non-software systems.
The UML reflects a chosen set of best engineering practices that have been effectively employed to model of an extensive software system.

3.1.1 Use Case Diagram

In the Unified Modelling Language (UML), a use case diagram is a behavioural diagram, which stands for and comes from the analysis involving and of use cases. Its main job is to draw a diagram which symbolizes how a system performs various tasks. It iconizes actors, their use cases and depicts the cause-effect relationships between those use cases. The primary value of the use case diagram is that it displays the operation of system actors for which functions are composed. By depicting actorsroles in the system as shown in Fig. 3.1.1, the model could be seen below.

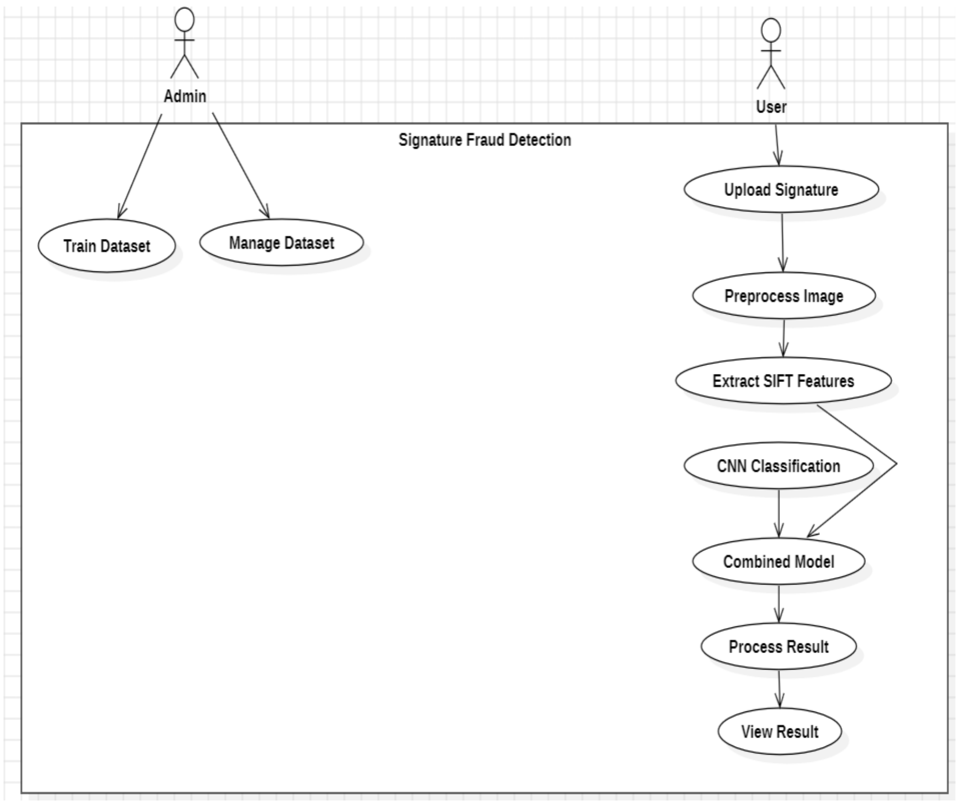
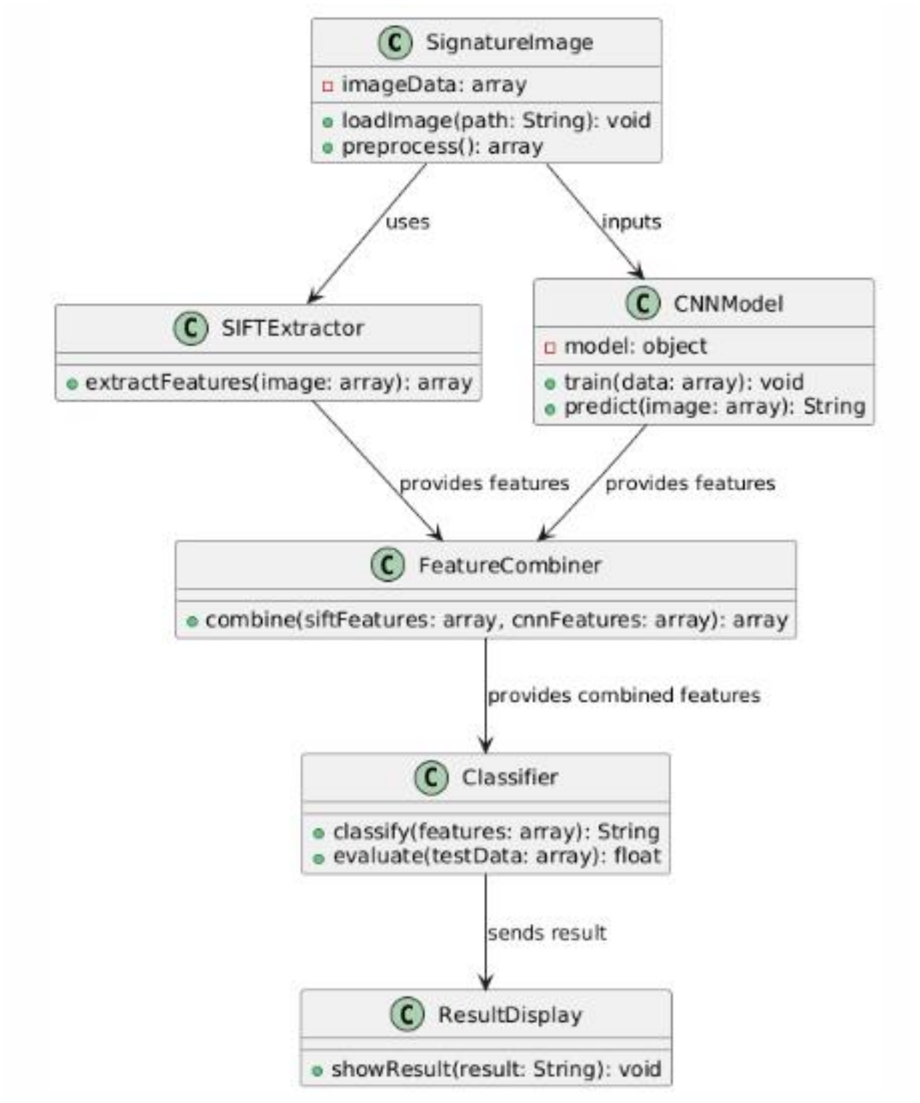


Figure 3.1.1 Use Case Diagram

3.1.2 Class Diagram

In software engineering, a class diagram in which the unifier model language (UML) is a sort of static structural diagram that is used in describing the structure of a system by showing the ins system classes and their attributes, the operations (or methods) and the relations among them. Through this, maintains the information content class hierarchy of the shown figure in Fig. 3.1.2.



SignatureImage
imageData: array
loadImage(path: String): void
preprocess(): array
uses
inputs
SIFTExtractor
extractFeatures(image: array): array
CNNModel
model: object
train(data: array): void
predict(image: array): String
provides features
provides features
FeatureCombiner
combine(siftFeatures: array, cnnFeatures: array): array
provides combined features
Classifier
classify(features: array): String
evaluate(testData: array): float
sends result
ResultDisplay
showResult(result: String): void

Figure 3.1.2 Class Diagram

3.1.3 Sequence Diagram:

A sequence diagram among the diagrams that use the Unified Modeling Language (UML) in the interaction diagram category shows how processes run in order and with whom. It is an XBee stack communication protocol which is represented in Fig 3.1.3. Timing diagrams are also referred to as Event DAG, event sequences, and scenario diagrams.

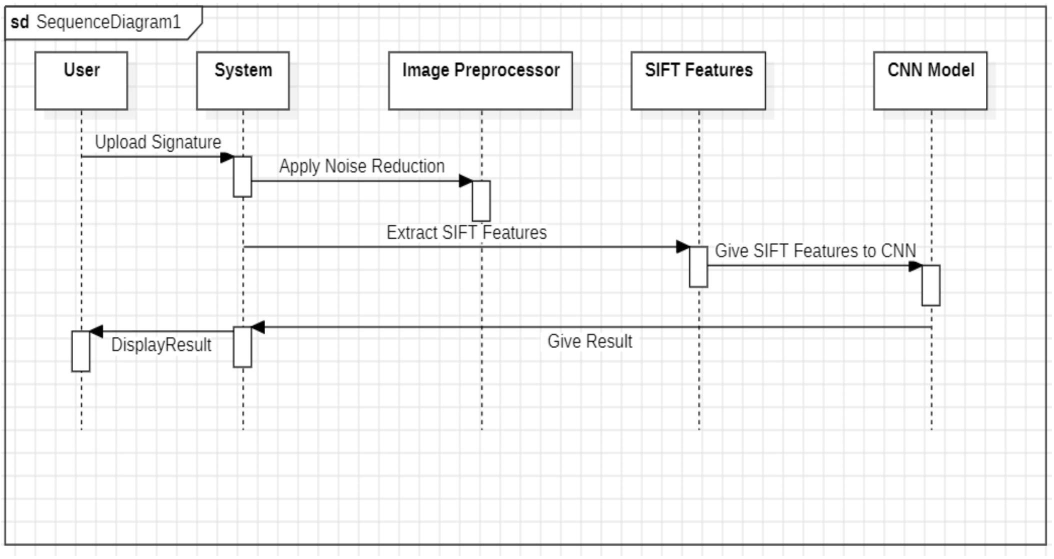


Figure 3.1.3 Sequence Diagram

CHAPTER 4

METHODOLOGY AND IMPLEMENTATION

4.1 Modules:

- Install TensorFlow
- Install NumPy
- Install OpenCv
- Install Flask
- Install Pickle

4.2 Module Descriptions

4.2.1 Data Collection and Preprocessing Module:

- **Data Gathering:** The system expects a dataset file. This archive should contain signature images organized into genuine and forged subdirectories, which are then extracted for processing.

- **Data Preprocessing:** In Image Preprocessing, Images are loaded, converted to RGB, resized to a standard 128x128 pixels, and pixel values are normalized for the CNN input. Grayscale versions are used for SIFT feature extraction, where a fixed number of SIFT descriptors are extracted and then scaled.

4.2.2 Model Training and Validation Module:

- **Training Setup:** A custom dual-input neural network is trained. This model combines a CNN branch (for image features) and a separate branch for SIFT features to classify signatures as genuine or forged.

- **Validation and Hyperparameter Tuning:** The model is validated on a test set split from the original data. While hyperparameters like learning rate, epochs, and batch size are predefined, the training uses EarlyStopping and ReduceLROnPlateau for dynamic adjustments rather than Grid Search.

4.2.3 Model Evaluation Module:

- **Performance Metrics Calculation:** During training, the model's loss (binary cross-entropy) and accuracy are monitored for both training and validation sets. After training, the model is evaluated on the test set using model.evaluate(), and the final Test Accuracy and Test Loss are logged.

4.3 Python Implementation

4.3.1 What is Python

Python is a versatile, high-level programming language used extensively in various domains, including artificial intelligence and machine learning. It supports both Object-Oriented and Procedural programming paradigms, making it a preferred choice for academic and industrial projects. Python's concise syntax allows for shorter, more readable code compared to languages like Java or C++. Major tech companies like Google, Facebook, and Instagram leverage Python for their machine learning models and web frameworks.

In this project, Python serves as the backbone for deep learning model development (signature verification), web application deployment, and image processing. Its powerful libraries, such as TensorFlow and Flask, provide essential functionality to implement a signature verification system. Key applications of Python used in this project include:

- Deep Learning with TensorFlow and Keras is used for training and deploying the dual-input (CNN and SIFT features) model designed for signature verification.
- Web Development using Flask is used to build an interface for users to upload reference and test signature images, receive a verification result (genuine, forged, or mismatch).

4.3.2 Advantages of Python

Python offers several benefits that make it ideal for this project:

- **Extensive Libraries:** Python’s large ecosystem of libraries like TensorFlow (for deep learning), Flask (for web development), NumPy (for numerical computation), OpenCV (for image processing), and Scikit-learn (for machine learning utilities like data splitting and scaling) simplifies the development process. These libraries provide pre-built solutions, significantly reducing the effort to implement complex functionalities for the signature verification system from scratch.
- **Simplicity and Productivity:** Python’s simple syntax allows developers to write clear and concise code, which is evident in the structure of files. This enhances productivity, especially in projects involving model definition and web application logic.
- **Flexibility:** Python allows easy integration of different types of tasks, such as deep learning model training, feature extraction (SIFT), and web serving, all within a cohesive environment. Its compatibility with TensorFlow and OpenCV ensures smooth implementation of the signature verification pipeline.
- **Embeddability:** While not explicitly demonstrated as embedding Python into another language in this specific project, Python's general characteristic of embeddability allows its scripts and functionalities to be potentially integrated into larger systems if needed.

4.3.3 Source Code

```
import os
import cv2
import numpy as np
import pickle
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense,
concatenate, Dropout
```

```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import zipfile
import shutil
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')
logger = logging.getLogger(__name__)

APP_ROOT = os.path.dirname(os.path.abspath(__file__))
ZIP_FILE_PATH = os.path.join(APP_ROOT, 'dataset.zip')
EXTRACTION_PATH = os.path.join(APP_ROOT, 'extracted_dataset_local')
MODELS_OUTPUT_DIR = os.path.join(APP_ROOT, 'models_output')

IMG_WIDTH, IMG_HEIGHT = 128, 128
SIFT_MAX_FEATURES = 64

def check_gpu():
    gpus = tf.config.experimental.list_physical_devices('GPU')
    if gpus:
        try:
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)
            logger.info(f"GPUs available: {len(gpus)}")
        except RuntimeError as e:
            logger.error(f"Error setting up GPU: {e}")
    else:
        logger.info("No GPUs available, using CPU.")

def unzip_dataset():
    if not os.path.exists(ZIP_FILE_PATH):
        logger.error(f"Dataset zip file not found at: {ZIP_FILE_PATH}")
        raise FileNotFoundError(f"Dataset zip file not found at: {ZIP_FILE_PATH}")

    if os.path.exists(EXTRACTION_PATH):
        logger.info(f"Removing existing extraction directory: {EXTRACTION_PATH}")
        shutil.rmtree(EXTRACTION_PATH)
    os.makedirs(EXTRACTION_PATH, exist_ok=True)

    logger.info(f"Attempting to unzip {ZIP_FILE_PATH} to {EXTRACTION_PATH}...")
    try:
        with zipfile.ZipFile(ZIP_FILE_PATH, 'r') as zip_ref:
            zip_ref.extractall(EXTRACTION_PATH)
        logger.info(f"Successfully unzipped dataset to {EXTRACTION_PATH}")

        extracted_items = os.listdir(EXTRACTION_PATH)
        if len(extracted_items) == 1 and os.path.isdir(os.path.join(EXTRACTION_PATH,
extracted_items[0])):
            dataset_root = os.path.join(EXTRACTION_PATH, extracted_items[0])
        else:
            dataset_root = EXTRACTION_PATH

        genuine_p = os.path.join(dataset_root, 'genuine')
        forged_p = os.path.join(dataset_root, 'forged')

        if not os.path.exists(genuine_p) or not os.path.exists(forged_p):
            logger.error(f"Error: 'genuine' ({genuine_p}) or 'forged' ({forged_p}) not found
post-unzip.")
            logger.error(f"Contents of {EXTRACTION_PATH}:
{os.listdir(EXTRACTION_PATH)}")
            if os.path.exists(dataset_root):
                logger.error(f"Contents of {dataset_root}: {os.listdir(dataset_root)}")
            raise FileNotFoundError("Could not locate 'genuine' and 'forged' subdirectories.")
        return genuine_p, forged_p
    except Exception as e:
        logger.error(f"An error occurred during unzipping or path setup: {e}",
exc_info=True)
        raise

def load_and_preprocess_image_cnn(image_path):
    try:
        img = cv2.imread(image_path)
        if img is None:
            logger.warning(f"CNN Preprocessing: Could not read image {image_path}.
Skipping.")
            return None
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (IMG_WIDTH, IMG_HEIGHT))
        img = img.astype('float32') / 255.0
        return img
    except Exception as e:
        logger.error(f"Error processing {image_path} for CNN: {e}", exc_info=True)
        return None

def extract_sift_features_for_model(image_path):
    try:
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        if img is None:
            logger.warning(f"SIFT Features: Could not read grayscale image {image_path}.
Returning zeros.")
            return np.zeros(SIFT_MAX_FEATURES * 128)

        sift = cv2.SIFT_create(nfeatures=SIFT_MAX_FEATURES)
        _, descriptors = sift.detectAndCompute(img, None)

        if descriptors is None or len(descriptors) == 0:
            return np.zeros(SIFT_MAX_FEATURES * 128)

        if len(descriptors) < SIFT_MAX_FEATURES:
            padding = np.zeros((SIFT_MAX_FEATURES - len(descriptors), 128))
            descriptors = np.vstack((descriptors, padding))
        elif len(descriptors) > SIFT_MAX_FEATURES:
            descriptors = descriptors[:SIFT_MAX_FEATURES]
        return descriptors.flatten()
    except Exception as e:
        logger.error(f"Error extracting SIFT from {image_path}: {e}", exc_info=True)
        return np.zeros(SIFT_MAX_FEATURES * 128)

def load_data(genuine_path, forged_path):
    cnn_images, sift_features_list, labels = [], [], []
    logger.info("Loading genuine signatures...")
    for filename in os.listdir(genuine_path):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.tif', '.tiff')):
            img_path = os.path.join(genuine_path, filename)
            cnn_img = load_and_preprocess_image_cnn(img_path)
            sift_feat = extract_sift_features_for_model(img_path)
            if cnn_img is not None:
                cnn_images.append(cnn_img)
                sift_features_list.append(sift_feat)
                labels.append(0) # 0 for genuine
    logger.info(f"Loaded {len(cnn_images)} genuine signatures.")
    genuine_count = len(cnn_images)

    logger.info("Loading forged signatures...")
    for filename in os.listdir(forged_path):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.tif', '.tiff')):
            img_path = os.path.join(forged_path, filename)
            cnn_img = load_and_preprocess_image_cnn(img_path)
            sift_feat = extract_sift_features_for_model(img_path)
            if cnn_img is not None:
                cnn_images.append(cnn_img)
                sift_features_list.append(sift_feat)
                labels.append(1) # 1 for forged
    logger.info(f"Loaded {len(cnn_images) - genuine_count} forged signatures.")
    logger.info(f"Total samples: {len(cnn_images)}")

    if not cnn_images:
        raise ValueError("No images were loaded. Check dataset paths and image files after
unzipping.")
    return np.array(cnn_images), np.array(sift_features_list), np.array(labels)
```

```
def build_model(sift_input_shape_dim):
    cnn_input_layer = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3), name='cnn_input')
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(cnn_input_layer)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = Flatten()(x)
    cnn_output_branch = Dense(128, activation='relu')(x)

    sift_input_layer = Input(shape=(sift_input_shape_dim,), name='sift_input')
    y = Dense(128, activation='relu')(sift_input_layer)
    y = Dropout(0.3)(y)
    sift_output_branch = Dense(64, activation='relu')(y)

    combined_branches = concatenate([cnn_output_branch, sift_output_branch])
    z = Dense(128, activation='relu')(combined_branches)
    z = Dropout(0.5)(z)
    output_layer = Dense(1, activation='sigmoid', name='output')(z)

    final_model = Model(inputs=[cnn_input_layer, sift_input_layer], outputs=output_layer)
    final_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    final_model.summary(print_fn=logger.info)
    return final_model

def train_and_save_model():
    check_gpu()
    genuine_path, forged_path = unzip_dataset()
    cnn_images, sift_features, labels = load_data(genuine_path, forged_path)

    X_cnn_train, X_cnn_test, \
    X_sift_train, X_sift_test, \
    y_train, y_test = train_test_split(
        cnn_images, sift_features, labels,
        test_size=0.2, random_state=42, stratify=labels
    )
    logger.info(f"Training samples: {len(X_cnn_train)}, Testing samples:
{len(X_cnn_test)}")

    sift_scaler = StandardScaler()
    X_sift_train_scaled = sift_scaler.fit_transform(X_sift_train)
    X_sift_test_scaled = sift_scaler.transform(X_sift_test)

    sift_input_dim = X_sift_train_scaled.shape[1]
    model = build_model(sift_input_dim)

    callbacks_list = [
        EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True,
verbose=1),
        ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6,
verbose=1)
    ]

    logger.info("Starting model training...")
    history = model.fit(
        [X_cnn_train, X_sift_train_scaled], y_train,
        validation_data=([X_cnn_test, X_sift_test_scaled], y_test),
        epochs=50, # Adjust as needed
        batch_size=32,
        callbacks=callbacks_list,
        verbose=1
    )

    loss, accuracy = model.evaluate([X_cnn_test, X_sift_test_scaled], y_test, verbose=0)
    logger.info(f"\nTest Accuracy: {accuracy*100:.2f}%")
    logger.info(f"Test Loss: {loss:.4f}")
    os.makedirs(MODELS_OUTPUT_DIR, exist_ok=True)
    model_save_path = os.path.join(MODELS_OUTPUT_DIR,
'signature_combined_model.h5') # Or .keras
    scaler_save_path = os.path.join(MODELS_OUTPUT_DIR, 'sift_feature_scaler.pkl')

    model.save(model_save_path)
    logger.info(f"Model saved to {model_save_path}")
    with open(scaler_save_path, 'wb') as f:
        pickle.dump(sift_scaler, f)
    logger.info(f"SIFT scaler saved to {scaler_save_path}")
    logger.info("Training complete. Model and scaler saved.")

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

CHAPTER 5
TESTING

Testing is a critical phase in software development, aimed at identifying defects and ensuring that the software aligns with specified requirements and user expectations. In the context of this emotion recognition project, we implement a systematic testing process designed to uncover gaps or faults, focusing on key specifications such as component evaluation, subassembly modeling, and overall system performance. By employing various types of testing, we can assess the reliability and response time of the application, ensuring a high-quality product that meets user needs.

5.1 Types of Testing

5.1.1 Unit Testing

Unit testing serves as the foundational phase in our software development lifecycle. It focuses on verifying the functionality of individual components or functions within the application. In the context of this project, unit tests specifically target functions related to image preprocessing (such as resizing signature images, normalization, any binarization or noise reduction steps).

Each unit is assessed in isolation, allowing developers to detect bugs at an early stage, thus preventing more significant issues later in the development process. Utilizing frameworks like `pytest` or `unittest`, we can automate these tests, leading to faster feedback and easier maintenance.

By writing unit tests alongside the development process, we encourage better coding practices and modular design. This results in code that is not only easier to read but also simpler to modify. A comprehensive suite of unit tests guarantees reliability and simplifies ongoing development as new requirements emerge.

5.1.2 Integration Testing

Once individual components of the Signature Fraud Detection system have been validated, integration testing becomes essential to examine how these modules interact with one another. This testing phase ensures seamless data flow and correct interplay between different parts of the application, crucial for the overall functionality.
In this project, integration tests focus on the pipeline from image input (e.g., uploaded via Flask) through preprocessing, SIFT feature extraction, and CNN feature extraction.

The feature fusion module's interaction with both SIFT and CNN feature outputs. The communication between the backend processing logic (SIFT, CNN, fusion, classification) and the Flask UI components (e.g., ensuring data is correctly passed from image upload to the model and results are displayed back to the user). How the system handles the comparison of two signature images (a reference and a test signature) through the entire processing chain.

Simulating scenarios where users upload signature images and the system processes them helps identify potential interface issues between the SIFT module, the CNN model, the fusion logic, and the final decision-making component. By conducting these tests, we ensure that the application is robust and capable of delivering accurate fraud detection results.

5.1.3 Functional Testing

Functional testing is critical for verifying that the Signature Fraud Detection application meets specified requirements and performs as expected from the user's perspective. This type of testing emphasizes validating the application’s features and functionalities under various scenarios, using the Flask UI as the primary interaction point.

In our project, functional tests evaluate the system's ability to correctly classify a pair of signatures (reference and test) as "Genuine" or "Forged" (or provide a similarity score) based on diverse test datasets containing known genuine and forged signatures.

By identifying discrepancies between expected and actual results during functional testing, we can make necessary adjustments before the final deployment. This process not only enhances reliability but also boosts user confidence in the signature fraud detection application.

5.1.4 System Testing

System testing provides a comprehensive evaluation of the integrated Signature Fraud Detection application, assessing its performance, functionality, and stability in an environment that mimics real-world conditions. This phase encompasses the entire workflow, from a user uploading two signature images via the Flask UI, through preprocessing, SIFT and CNN feature extraction, fusion, comparison, and the final fraud/genuine prediction displayed to the user.

During system testing, we evaluate the end-to-end processing time for signature comparison. The system's behaviour under various load conditions, the robustness of the system when encountering different types of signature images, error handling and recovery mechanisms, the overall usability and user experience of the Flask interface.

This phase is vital for uncovering systemic issues, performance bottlenecks, or usability challenges that may not have been evident during earlier testing stages. Insights gained from system testing ensure that the application is robust and ready for demonstration or deployment.

5.1.5 White Box Testing

White box testing focuses on the internal logic and structure of the application. This approach allows testers to examine the code's pathways and decision points, which is particularly beneficial for uncovering logical errors.

In the context of this project, white box testing analyses the specific algorithms used in the SIFT feature extraction and matching, the architecture and weights (if inspecting a trained model) of the CNN model to ensure it has learned meaningful patterns or to debug unexpected behaviour, the logic within the feature fusion component to ensure SIFT and CNN features are being combined as intended.

By assessing performance metrics such as execution speed and resource usage, testers can recommend enhancements to algorithms, improving both the reliability and overall performance of the system.

5.1.6 Black Box Testing

Black box testing evaluates the application based solely on its outputs in response to various inputs, without knowledge of the internal code structure. This method emphasizes testing the application’s functionality from an end-user perspective.

In our project, black box tests involve, Uploading various pairs of genuine and forged signature images through the Flask UI. Assessing the accuracy of the system's "Genuine" / "Forged" classification (or similarity score) for these pairs. Testing the UI for intuitive navigation, clear instructions for uploading images, and understandable presentation of results.

By simulating real-world scenarios and user interactions, this approach helps identify discrepancies between expected and actual results, ensuring that the application meets user needs and is ready for practical deployment in diverse environments.

5.1.7 Acceptance Testing

Acceptance testing validates the system's effectiveness in meeting user expectations and specified requirements. This phase involves real users assessing the application to determine its usability and performance in real-world contexts.

In our project, acceptance testing is crucial for evaluating if the system's fraud detection accuracy is acceptable for its intended, ensuring the output (Genuine/Forged prediction) is clear and understandable.

This phase is essential for fine-tuning the model before deployment, ensuring that it meets the requirements for practical applications. By engaging potential end-users, we can refine the system further based on their insights and experiences.

5.2 System Testing Strategy

The testing strategy for the Signature Fraud Detection project is designed to ensure both accuracy and user-friendliness. It encompasses multiple phases:

- **Unit Testing:** Verifying the functionality of individual components, such as image preprocessing modules, SIFT feature extraction functions, CNN model layers/blocks, feature fusion logic, and Flask helper functions.
- **Integration Testing:** Ensuring that all components (image input, preprocessing, SIFT extraction, CNN extraction, feature fusion, classification, and Flask UI) work cohesively, focusing on the data flow and interaction between them when comparing two signature images.
- **Functional Testing:** Assessing the system's ability to accurately classify pairs of signatures as "Genuine" or "Forged" using the Flask UI, based on diverse test datasets, and ensuring outputs align with expected results.
- **System Testing:** Evaluating the entire end-to-end workflow, from a user uploading two signature images via the Flask application to obtaining the final fraud prediction, under various conditions to test robustness, performance, and usability.
- **White Box Testing:** Examining the internal code logic of the SIFT algorithms, CNN architecture, and feature fusion methods to identify logical errors and optimization opportunities.
- **Black Box Testing:** Evaluating the Flask application's functionality from an end-user perspective by providing various signature inputs and assessing the correctness of the fraud detection output without knowledge of internal workings.
- **Acceptance Testing:** Validating the system's effectiveness through real-world usage scenarios (potentially with mock users) to ensure it meets expectations and is suitable for its intended purpose of demonstrating signature fraud detection.

The overall objective of this testing strategy is to validate that the emotion recognition system meets performance standards, user expectations, and is prepared for real-world applications. By employing a comprehensive testing approach, we aim to deliver a reliable and efficient emotion recognition solution.

CHAPTER 6
RESULT

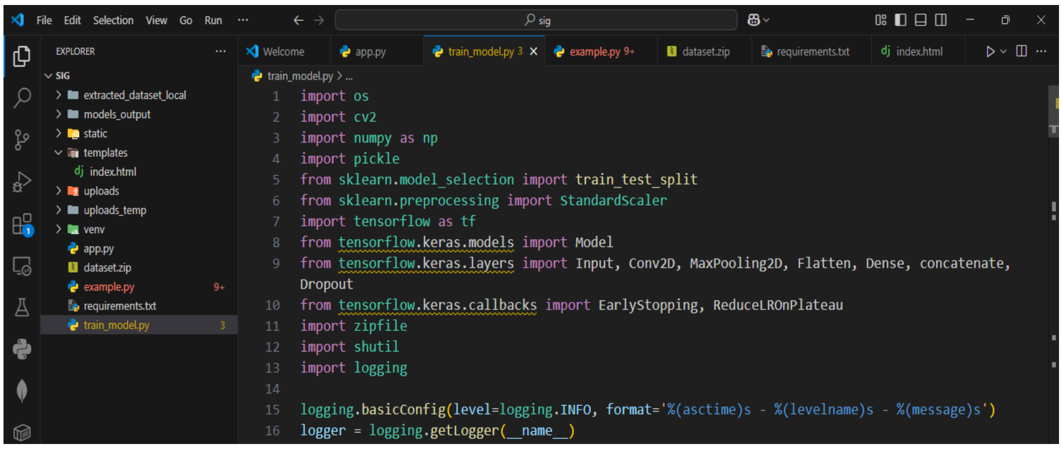
In the domain of biometric security and fraud detection, deep learning techniques have showcased remarkable advancements in accurately verifying the authenticity of human signatures. By utilizing convolutional neural networks (CNNs) for learning intricate visual patterns and classical computer vision algorithms for robust feature extraction, significant improvements have been achieved in distinguishing between genuine and forged signatures. This project leverages a hybrid model that combines a CNN with the Scale-Invariant Feature Transform (SIFT) algorithm to create a comprehensive and effective signature verification system.

The dataset used in this study is the well-established CEDAR dataset, which contains a large collection of genuine and forged signatures from numerous individuals. Each individual provides several genuine samples, along with skilled forgeries created by other people, allowing the model to learn the subtle variations that differentiate an authentic signature from a counterfeit one. Feature extraction is performed using a two-pronged approach: a CNN is trained to extract deep, learned features related to stroke style, pressure, and texture, while SIFT is used to extract robust keypoint descriptors that are invariant to minor changes in scale and rotation. The system then compares the feature vectors of two input signatures to determine their similarity.

Several advanced optimization techniques were employed to fine-tune the model’s performance. This loss function is ideal for this task as it works to minimize the distance between feature vectors of genuine signature pairs while maximizing the distance between genuine and forged pairs. The training process involved systematic evaluation using metrics such as accuracy, precision, recall, and the F1-score to measure the model's effectiveness in correctly classifying signatures.

The project was executed using Python in Visual Studio Code, with Flask providing the backend and user interface for signature uploads and result display. The training process for the deep learning model took more than 6 hours, a duration attributable to the computational demands of the combined CNN-SIFT model and the size of the CEDAR dataset. Screenshots of the Flask UI and the model's backend logic are provided below to illustrate the various steps involved.

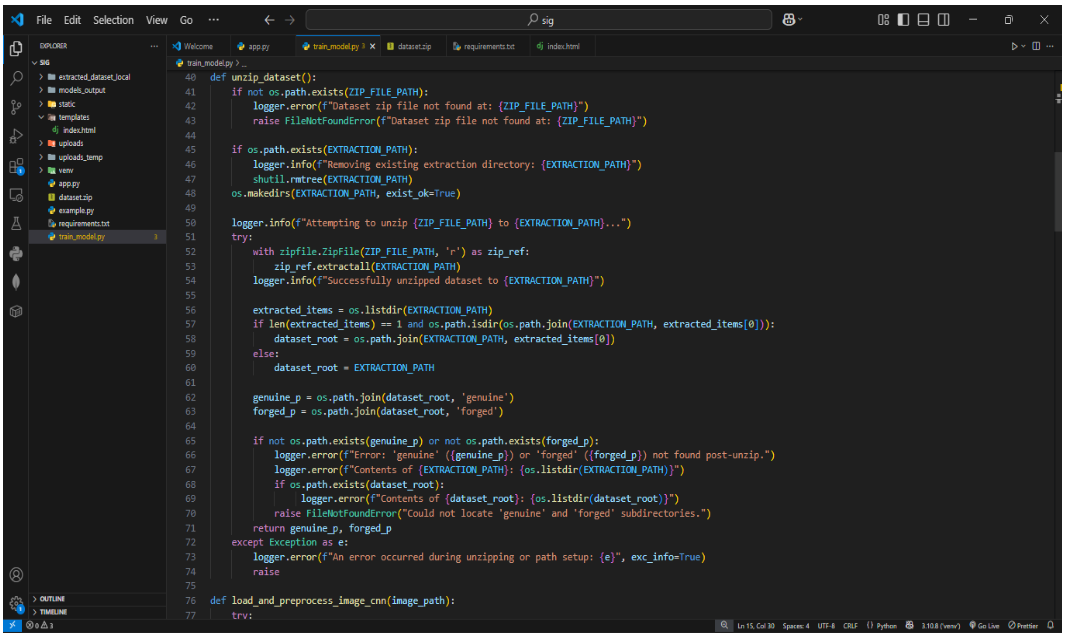
In VS Code, training all the algorithms took more than 6 hours. The following figures display the python code with all the steps involved.



```python
import os
import cv2
import numpy as np
import pickle
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, concatenate, Dropout
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import zipfile
import shutil
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)
```

Fig 6.1 Loading and importing required python classes

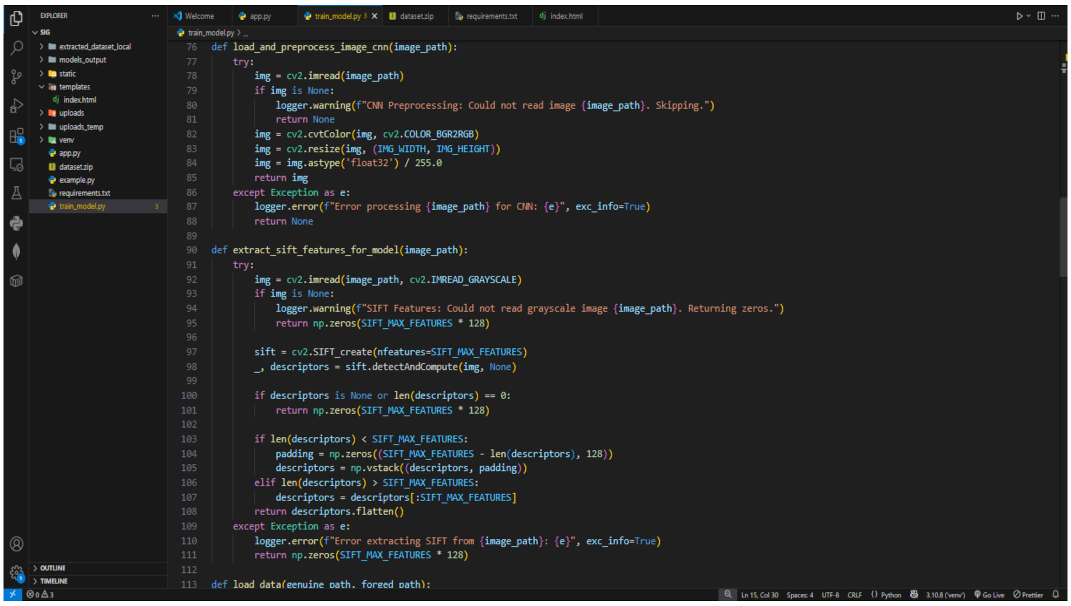
In the above fig 6.1, it demonstrates the code for importing the essential libraries and modules needed for signature fraud detection.



```python
def unzip_dataset():
    if not os.path.exists(ZIP_FILE_PATH):
        logger.error(f"Dataset zip file not found at: {ZIP_FILE_PATH}")
        raise FileNotFoundError(f"Dataset zip file not found at: {ZIP_FILE_PATH}")

    if os.path.exists(EXTRACTION_PATH):
        logger.info(f"Removing existing extraction directory: {EXTRACTION_PATH}")
        shutil.rmtree(EXTRACTION_PATH)
    os.makedirs(EXTRACTION_PATH, exist_ok=True)

    logger.info(f"Attempting to unzip {ZIP_FILE_PATH} to {EXTRACTION_PATH}...")
    try:
        with zipfile.ZipFile(ZIP_FILE_PATH, 'r') as zip_ref:
            zip_ref.extractall(EXTRACTION_PATH)
        logger.info(f"Successfully unzipped dataset to {EXTRACTION_PATH}")

        extracted_items = os.listdir(EXTRACTION_PATH)
        if len(extracted_items) == 1 and os.path.isdir(os.path.join(EXTRACTION_PATH, extracted_items[0])):
            dataset_root = os.path.join(EXTRACTION_PATH, extracted_items[0])
        else:
            dataset_root = EXTRACTION_PATH

        genuine_p = os.path.join(dataset_root, 'genuine')
        forged_p = os.path.join(dataset_root, 'forged')

        if not os.path.exists(genuine_p) or not os.path.exists(forged_p):
            logger.error(f"Error: 'genuine' ({genuine_p}) or 'forged' ({forged_p}) not found post-unzip.")
            logger.error(f"Contents of {EXTRACTION_PATH}: {os.listdir(EXTRACTION_PATH)}")
            if os.path.exists(dataset_root):
                logger.error(f"Contents of {dataset_root}: {os.listdir(dataset_root)}")
            raise FileNotFoundError("Could not locate 'genuine' and 'forged' subdirectories.")
        return genuine_p, forged_p
    except Exception as e:
        logger.error(f"An error occurred during unzipping or path setup: {e}", exc_info=True)
        raise

def load_and_preprocess_image_cnn(image_path):
    try:
```

Fig. 6.2 Unzip Dataset and Extract genuine and forged images

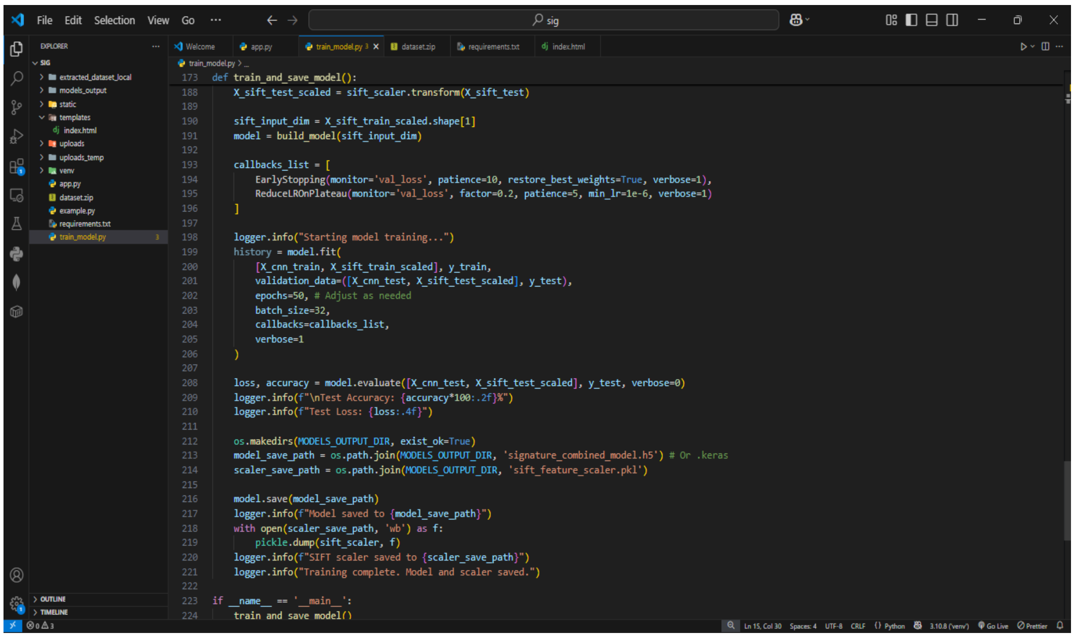
The above fig.6.2, outlines the process of unzipping dataset and extracting of genuine and forged images.



```python
def load_and_preprocess_image_cnn(image_path):
    try:
        img = cv2.imread(image_path)
        if img is None:
            logger.warning(f"CNN Preprocessing: Could not read image {image_path}. Skipping.")
            return None
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (IMG_WIDTH, IMG_HEIGHT))
        img = img.astype('float32') / 255.0
        return img
    except Exception as e:
        logger.error(f"Error processing {image_path} for CNN: {e}", exc_info=True)
        return None

def extract_sift_features_for_model(image_path):
    try:
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        if img is None:
            logger.warning(f"SIFT Features: Could not read grayscale image {image_path}. Returning zeros.")
            return np.zeros(SIFT_MAX_FEATURES * 128)

        sift = cv2.SIFT_create(nfeatures=SIFT_MAX_FEATURES)
        _, descriptors = sift.detectAndCompute(img, None)

        if descriptors is None or len(descriptors) == 0:
            return np.zeros(SIFT_MAX_FEATURES * 128)

        if len(descriptors) < SIFT_MAX_FEATURES:
            padding = np.zeros((SIFT_MAX_FEATURES - len(descriptors), 128))
            descriptors = np.vstack((descriptors, padding))
        elif len(descriptors) > SIFT_MAX_FEATURES:
            descriptors = descriptors[:SIFT_MAX_FEATURES]
        return descriptors.flatten()
    except Exception as e:
        logger.error(f"Error extracting SIFT from {image_path}: {e}", exc_info=True)
        return np.zeros(SIFT_MAX_FEATURES * 128)

def load_data(genuine_path, forged_path):
```

Fig. 6.3 Extracting CNN and SIFT features from images

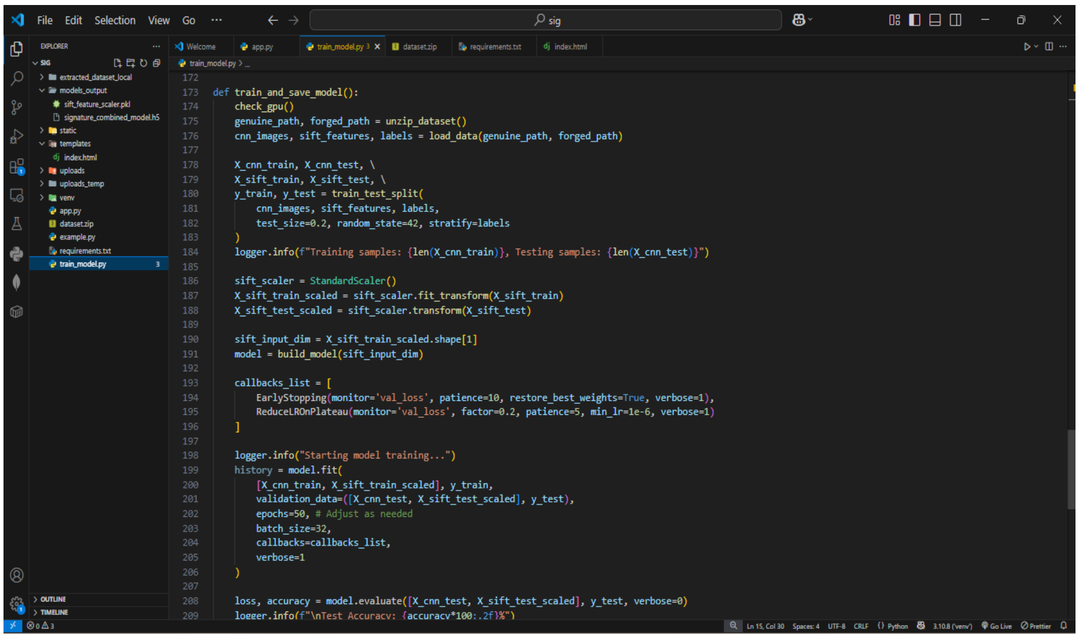
The above fig. 6.3, illustrates the functions which are extracting CNN and SIFT features from the images.



```python
def train_and_save_model():
    X_sift_test_scaled = sift_scaler.transform(X_sift_test)

    sift_input_dim = X_sift_train_scaled.shape[1]
    model = build_model(sift_input_dim)

    callbacks_list = [
        EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True, verbose=1),
        ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6, verbose=1)
    ]

    logger.info("Starting model training...")
    history = model.fit(
        [X_cnn_train, X_sift_train_scaled], y_train,
        validation_data=([X_cnn_test, X_sift_test_scaled], y_test),
        epochs=50, # Adjust as needed
        batch_size=32,
        callbacks=callbacks_list,
        verbose=1
    )

    loss, accuracy = model.evaluate([X_cnn_test, X_sift_test_scaled], y_test, verbose=0)
    logger.info(f"\nTest Accuracy: {accuracy*100:.2f}%")
    logger.info(f"Test Loss: {loss:.4f}")

    os.makedirs(MODELS_OUTPUT_DIR, exist_ok=True)
    model_save_path = os.path.join(MODELS_OUTPUT_DIR, 'signature_combined_model.h5') # Or .keras
    scaler_save_path = os.path.join(MODELS_OUTPUT_DIR, 'sift_feature_scaler.pkl')

    model.save(model_save_path)
    logger.info(f"Model saved to {model_save_path}")
    with open(scaler_save_path, 'wb') as f:
        pickle.dump(sift_scaler, f)
    logger.info(f"SIFT scaler saved to {scaler_save_path}")
    logger.info("Training complete. Model and scaler saved.")

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

Fig. 6.4 Training and saving model

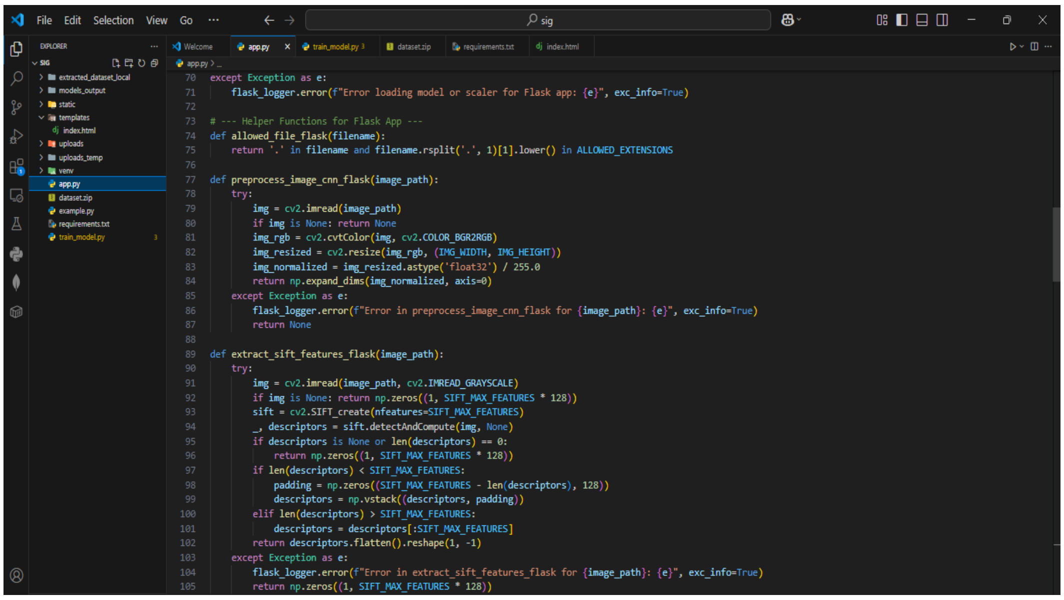
Fig 6.4, the process shown here is training and saving the model in a models folder.



```python
def train_and_save_model():
    check_gpu()
    genuine_path, forged_path = unzip_dataset()
    cnn_images, sift_features, labels = load_data(genuine_path, forged_path)

    X_cnn_train, X_cnn_test, \
    X_sift_train, X_sift_test, \
    y_train, y_test = train_test_split(
        cnn_images, sift_features, labels,
        test_size=0.2, random_state=42, stratify=labels
    )
    logger.info(f"Training samples: {len(X_cnn_train)}, Testing samples: {len(X_cnn_test)}")

    sift_scaler = StandardScaler()
    X_sift_train_scaled = sift_scaler.fit_transform(X_sift_train)
    X_sift_test_scaled = sift_scaler.transform(X_sift_test)

    sift_input_dim = X_sift_train_scaled.shape[1]
    model = build_model(sift_input_dim)

    callbacks_list = [
        EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True, verbose=1),
        ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=1e-6, verbose=1)
    ]

    logger.info("Starting model training...")
    history = model.fit(
        [X_cnn_train, X_sift_train_scaled], y_train,
        validation_data=([X_cnn_test, X_sift_test_scaled], y_test),
        epochs=50, # Adjust as needed
        batch_size=32,
        callbacks=callbacks_list,
        verbose=1
    )

    loss, accuracy = model.evaluate([X_cnn_test, X_sift_test_scaled], y_test, verbose=0)
    logger.info(f"\nTest Accuracy: {accuracy*100:.2f}%")
```

Fig. 6.5 Models saved in model_output

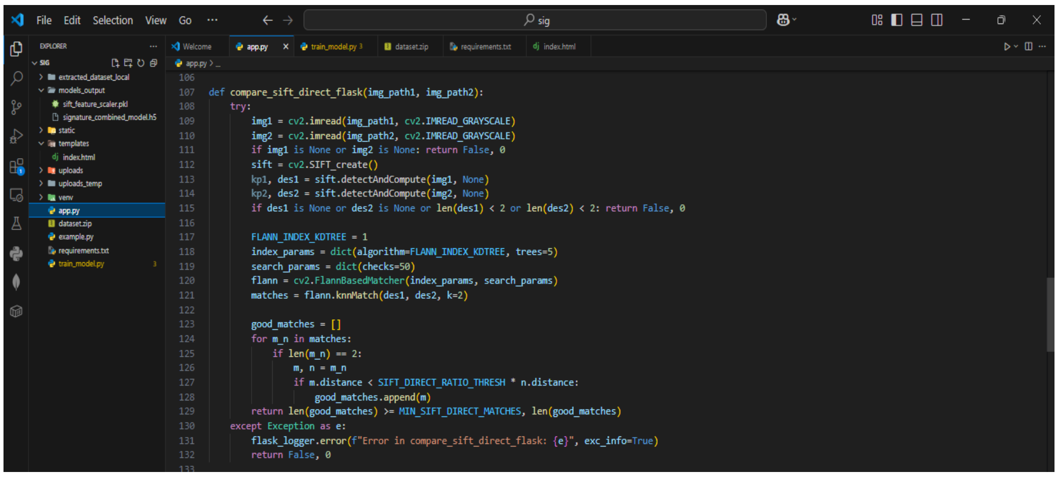
Fig 6.5 The training dataset has been loaded successfully and the models sift_features_scalar.pkl, signature_combined_model.h5 are saved in mode_output folder.



```python
    except Exception as e:
        flask_logger.error(f"Error loading model or scaler for Flask app: {e}", exc_info=True)

# --- Helper Functions for Flask App ---
def allowed_file_flask(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

def preprocess_image_cnn_flask(image_path):
    try:
        img = cv2.imread(image_path)
        if img is None: return None
        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img_resized = cv2.resize(img_rgb, (IMG_WIDTH, IMG_HEIGHT))
        img_normalized = img_resized.astype('float32') / 255.0
        return np.expand_dims(img_normalized, axis=0)
    except Exception as e:
        flask_logger.error(f"Error in preprocess_image_cnn_flask for {image_path}: {e}", exc_info=True)
        return None

def extract_sift_features_flask(image_path):
    try:
        img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
        if img is None: return np.zeros((1, SIFT_MAX_FEATURES * 128))
        sift = cv2.SIFT_create(nfeatures=SIFT_MAX_FEATURES)
        _, descriptors = sift.detectAndCompute(img, None)
        if descriptors is None or len(descriptors) == 0:
            return np.zeros((1, SIFT_MAX_FEATURES * 128))
        if len(descriptors) < SIFT_MAX_FEATURES:
            padding = np.zeros((SIFT_MAX_FEATURES - len(descriptors), 128))
            descriptors = np.vstack((descriptors, padding))
        elif len(descriptors) > SIFT_MAX_FEATURES:
            descriptors = descriptors[:SIFT_MAX_FEATURES]
        return descriptors.flatten().reshape(1, -1)
    except Exception as e:
        flask_logger.error(f"Error in extract_sift_features_flask for {image_path}: {e}", exc_info=True)
        return np.zeros((1, SIFT_MAX_FEATURES * 128))
```

Fig. 6.6 Extracting CNN and SIFT features in Flask

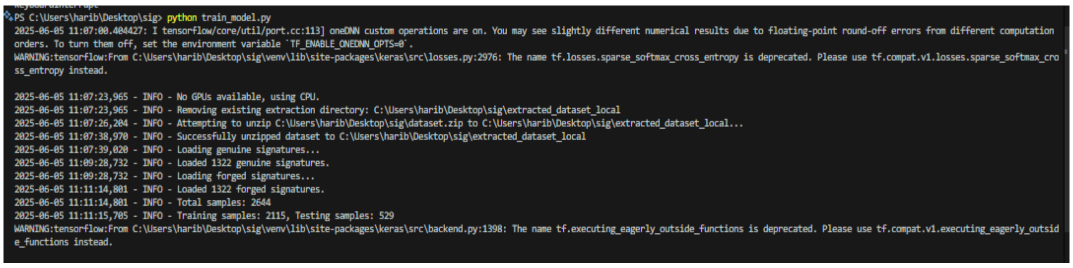
In fig 6.6 the features are extracted and loaded in the flask application.



```python
def compare_sift_direct_flask(img_path1, img_path2):
    try:
        img1 = cv2.imread(img_path1, cv2.IMREAD_GRAYSCALE)
        img2 = cv2.imread(img_path2, cv2.IMREAD_GRAYSCALE)
        if img1 is None or img2 is None: return False, 0
        sift = cv2.SIFT_create()
        kp1, des1 = sift.detectAndCompute(img1, None)
        kp2, des2 = sift.detectAndCompute(img2, None)
        if des1 is None or des2 is None or len(des1) < 2 or len(des2) < 2: return False, 0

        FLANN_INDEX_KDTREE = 1
        index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
        search_params = dict(checks=50)
        flann = cv2.FlannBasedMatcher(index_params, search_params)
        matches = flann.knnMatch(des1, des2, k=2)

        good_matches = []
        for m_n in matches:
            if len(m_n) == 2:
                m, n = m_n
                if m.distance < SIFT_DIRECT_RATIO_THRESH * n.distance:
                    good_matches.append(m)
        return len(good_matches) >= MIN_SIFT_DIRECT_MATCHES, len(good_matches)
    except Exception as e:
        flask_logger.error(f"Error in compare_sift_direct_flask: {e}", exc_info=True)
        return False, 0
```

Fig. 6.7 Compare Images

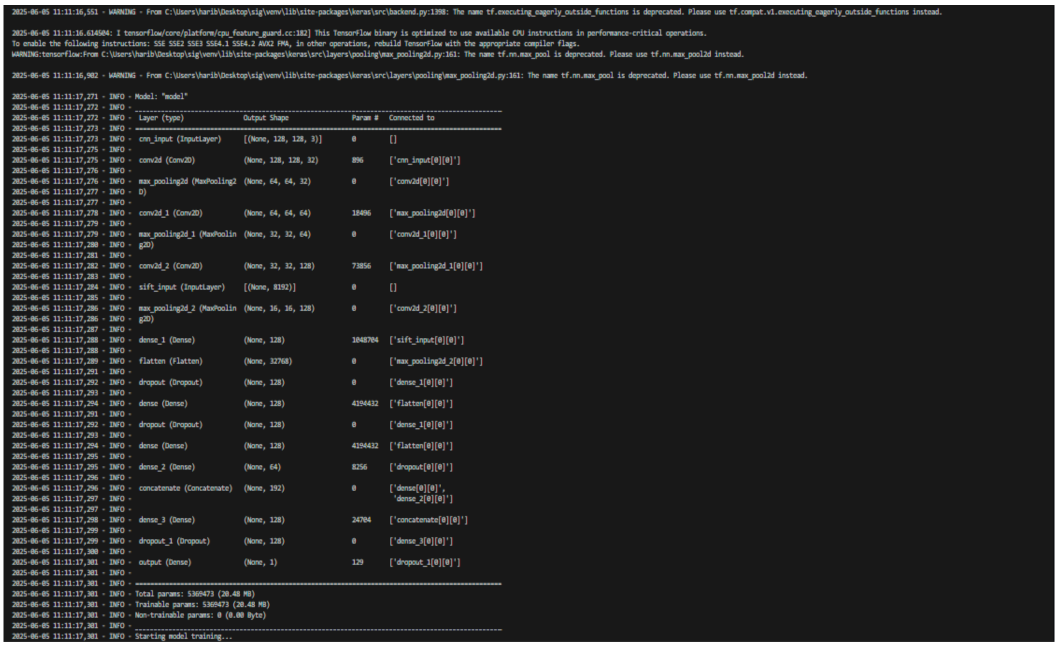
The fig 6.7 the images are compared by the image paths and good mmatches are stored in an array.



```
PS C:\Users\harib\Desktop\sig> python train_model.py
2025-06-05 11:07:00.404427: I tensorflow/core/util/port.cc:113] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation
orders. To turn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
WARNING:tensorflow:From C:\Users\harib\Desktop\sig\venv\lib\site-packages\keras\src\losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cro
ss_entropy instead.

2025-06-05 11:07:23,965 - INFO - No GPUs available, using CPU.
2025-06-05 11:07:23,965 - INFO - Removing existing extraction directory: C:\Users\harib\Desktop\sig\extracted_dataset_local
2025-06-05 11:07:26,204 - INFO - Attempting to unzip C:\Users\harib\Desktop\sig\dataset.zip to C:\Users\harib\Desktop\sig\extracted_dataset_local...
2025-06-05 11:07:38,970 - INFO - Successfully unzipped dataset to C:\Users\harib\Desktop\sig\extracted_dataset_local
2025-06-05 11:07:39,020 - INFO - Loading genuine signatures...
2025-06-05 11:09:28,732 - INFO - Loaded 1322 genuine signatures.
2025-06-05 11:09:28,732 - INFO - Loading forged signatures...
2025-06-05 11:11:14,801 - INFO - Loaded 1322 forged signatures.
2025-06-05 11:11:14,801 - INFO - Total samples: 2644
2025-06-05 11:11:15,705 - INFO - Training samples: 2115, Testing samples: 529
WARNING:tensorflow:From C:\Users\harib\Desktop\sig\venv\lib\site-packages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_outsid
e_functions instead.
```

Fig. 6.8 Signatures Loaded

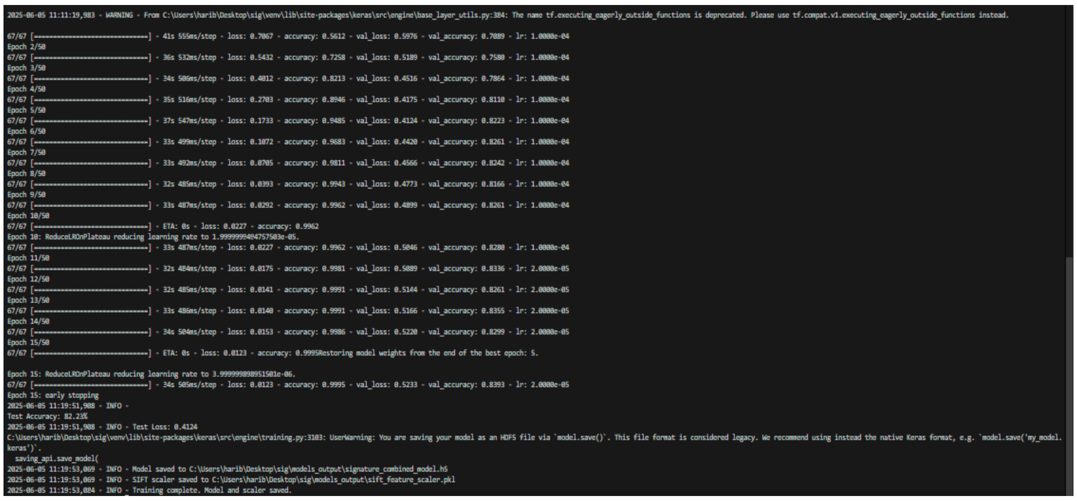
The dataset has been zipped and signature images which are genuine and forged are successfully loaded, as shown in the fig. 6.8.



```
2025-06-05 11:11:16,551 - WARNING - From C:\Users\harib\Desktop\sig\venv\lib\site-packages\keras\src\backend.py:1398: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_outside_functions instead.

2025-06-05 11:11:16.614504: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
WARNING:tensorflow:From C:\Users\harib\Desktop\sig\venv\lib\site-packages\keras\src\layers\pooling\max_pooling2d.py:161: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instead.

2025-06-05 11:11:16,982 - WARNING - From C:\Users\harib\Desktop\sig\venv\lib\site-packages\keras\src\layers\pooling\max_pooling2d.py:161: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instead.

2025-06-05 11:11:17,271 - INFO - Model: "model"
2025-06-05 11:11:17,272 - INFO - __________________________________________________________________________________________________
2025-06-05 11:11:17,272 - INFO -  Layer (type)                Output Shape                 Param #   Connected to
2025-06-05 11:11:17,273 - INFO - ==================================================================================================
2025-06-05 11:11:17,273 - INFO -  cnn_input (InputLayer)      [(None, 128, 128, 3)]        0         []
2025-06-05 11:11:17,275 - INFO -
2025-06-05 11:11:17,275 - INFO -  conv2d (Conv2D)             (None, 128, 128, 32)         896       ['cnn_input[0][0]']
2025-06-05 11:11:17,276 - INFO -
2025-06-05 11:11:17,276 - INFO -  max_pooling2d (MaxPooling2  (None, 64, 64, 32)           0         ['conv2d[0][0]']
2025-06-05 11:11:17,277 - INFO -  D)
2025-06-05 11:11:17,277 - INFO -
2025-06-05 11:11:17,278 - INFO -  conv2d_1 (Conv2D)           (None, 64, 64, 64)           18496     ['max_pooling2d[0][0]']
2025-06-05 11:11:17,279 - INFO -
2025-06-05 11:11:17,279 - INFO -  max_pooling2d_1 (MaxPoolin  (None, 32, 32, 64)           0         ['conv2d_1[0][0]']
2025-06-05 11:11:17,280 - INFO -  g2D)
2025-06-05 11:11:17,281 - INFO -
2025-06-05 11:11:17,282 - INFO -  conv2d_2 (Conv2D)           (None, 32, 32, 128)          73856     ['max_pooling2d_1[0][0]']
2025-06-05 11:11:17,283 - INFO -
2025-06-05 11:11:17,284 - INFO -  sift_input (InputLayer)     [(None, 8192)]               0         []
2025-06-05 11:11:17,285 - INFO -
2025-06-05 11:11:17,286 - INFO -  max_pooling2d_2 (MaxPoolin  (None, 16, 16, 128)          0         ['conv2d_2[0][0]']
2025-06-05 11:11:17,286 - INFO -  g2D)
2025-06-05 11:11:17,287 - INFO -
2025-06-05 11:11:17,288 - INFO -  dense_1 (Dense)             (None, 128)                  1048704   ['sift_input[0][0]']
2025-06-05 11:11:17,288 - INFO -
2025-06-05 11:11:17,289 - INFO -  flatten (Flatten)           (None, 32768)                0         ['max_pooling2d_2[0][0]']
2025-06-05 11:11:17,291 - INFO -
2025-06-05 11:11:17,292 - INFO -  dropout (Dropout)           (None, 128)                  0         ['dense_1[0][0]']
2025-06-05 11:11:17,293 - INFO -
2025-06-05 11:11:17,294 - INFO -  dense (Dense)               (None, 128)                  4194432   ['flatten[0][0]']
2025-06-05 11:11:17,291 - INFO -
2025-06-05 11:11:17,292 - INFO -  dropout (Dropout)           (None, 128)                  0         ['dense_1[0][0]']
2025-06-05 11:11:17,293 - INFO -
2025-06-05 11:11:17,294 - INFO -  dense (Dense)               (None, 128)                  4194432   ['flatten[0][0]']
2025-06-05 11:11:17,295 - INFO -
2025-06-05 11:11:17,295 - INFO -  dense_2 (Dense)             (None, 64)                   8256      ['dropout[0][0]']
2025-06-05 11:11:17,296 - INFO -
2025-06-05 11:11:17,296 - INFO -  concatenate (Concatenate)   (None, 192)                  0         ['dense[0][0]',
2025-06-05 11:11:17,297 - INFO -                                                                     'dense_2[0][0]']
2025-06-05 11:11:17,297 - INFO -
2025-06-05 11:11:17,298 - INFO -  dense_3 (Dense)             (None, 128)                  24704     ['concatenate[0][0]']
2025-06-05 11:11:17,299 - INFO -
2025-06-05 11:11:17,299 - INFO -  dropout_1 (Dropout)         (None, 128)                  0         ['dense_3[0][0]']
2025-06-05 11:11:17,300 - INFO -
2025-06-05 11:11:17,301 - INFO -  output (Dense)              (None, 1)                    129       ['dropout_1[0][0]']
2025-06-05 11:11:17,301 - INFO -
2025-06-05 11:11:17,301 - INFO - ==================================================================================================
2025-06-05 11:11:17,301 - INFO - Total params: 5369473 (20.48 MB)
2025-06-05 11:11:17,301 - INFO - Trainable params: 5369473 (20.48 MB)
2025-06-05 11:11:17,301 - INFO - Non-trainable params: 0 (0.00 Byte)
2025-06-05 11:11:17,301 - INFO - __________________________________________________________________________________________________
2025-06-05 11:11:17,301 - INFO - Starting model training...
```

Fig. 6.9 Model definition total params and trainable params

Fig 6.9 provides details about the model's total and trainable parameters, which have been calculated and verified during the project.



```
2025-06-05 11:11:19,983 - WARNING - From C:\Users\harib\Desktop\sig\venv\lib\site-packages\keras\src\engine\base_layer_utils.py:384: The name tf.executing_eagerly_outside_functions is deprecated. Please use tf.compat.v1.executing_eagerly_outside_functions instead.

67/67 [==============================] - 41s 555ms/step - loss: 0.7067 - accuracy: 0.5612 - val_loss: 0.5976 - val_accuracy: 0.7089 - lr: 1.0000e-04
Epoch 2/50
67/67 [==============================] - 36s 532ms/step - loss: 0.5432 - accuracy: 0.7258 - val_loss: 0.5189 - val_accuracy: 0.7580 - lr: 1.0000e-04
Epoch 3/50
67/67 [==============================] - 34s 506ms/step - loss: 0.4012 - accuracy: 0.8213 - val_loss: 0.4516 - val_accuracy: 0.7864 - lr: 1.0000e-04
Epoch 4/50
67/67 [==============================] - 35s 516ms/step - loss: 0.2703 - accuracy: 0.8946 - val_loss: 0.4175 - val_accuracy: 0.8110 - lr: 1.0000e-04
Epoch 5/50
67/67 [==============================] - 37s 547ms/step - loss: 0.1733 - accuracy: 0.9485 - val_loss: 0.4124 - val_accuracy: 0.8223 - lr: 1.0000e-04
Epoch 6/50
67/67 [==============================] - 33s 499ms/step - loss: 0.1072 - accuracy: 0.9683 - val_loss: 0.4420 - val_accuracy: 0.8261 - lr: 1.0000e-04
Epoch 7/50
67/67 [==============================] - 33s 492ms/step - loss: 0.0705 - accuracy: 0.9811 - val_loss: 0.4566 - val_accuracy: 0.8242 - lr: 1.0000e-04
Epoch 8/50
67/67 [==============================] - 32s 485ms/step - loss: 0.0393 - accuracy: 0.9943 - val_loss: 0.4773 - val_accuracy: 0.8166 - lr: 1.0000e-04
Epoch 9/50
67/67 [==============================] - 33s 487ms/step - loss: 0.0292 - accuracy: 0.9962 - val_loss: 0.4899 - val_accuracy: 0.8261 - lr: 1.0000e-04
Epoch 10/50
67/67 [==============================] - ETA: 0s - loss: 0.0227 - accuracy: 0.9962
Epoch 10: ReduceLROnPlateau reducing learning rate to 1.9999999494757503e-05.
67/67 [==============================] - 33s 487ms/step - loss: 0.0227 - accuracy: 0.9962 - val_loss: 0.5046 - val_accuracy: 0.8280 - lr: 1.0000e-04
Epoch 11/50
67/67 [==============================] - 32s 484ms/step - loss: 0.0175 - accuracy: 0.9981 - val_loss: 0.5089 - val_accuracy: 0.8336 - lr: 2.0000e-05
Epoch 12/50
67/67 [==============================] - 32s 485ms/step - loss: 0.0141 - accuracy: 0.9991 - val_loss: 0.5144 - val_accuracy: 0.8261 - lr: 2.0000e-05
Epoch 13/50
67/67 [==============================] - 33s 486ms/step - loss: 0.0140 - accuracy: 0.9991 - val_loss: 0.5166 - val_accuracy: 0.8355 - lr: 2.0000e-05
Epoch 14/50
67/67 [==============================] - 34s 504ms/step - loss: 0.0153 - accuracy: 0.9986 - val_loss: 0.5220 - val_accuracy: 0.8299 - lr: 2.0000e-05
Epoch 15/50
67/67 [==============================] - ETA: 0s - loss: 0.0123 - accuracy: 0.9995Restoring model weights from the end of the best epoch: 5.

Epoch 15: ReduceLROnPlateau reducing learning rate to 3.999999898951501e-06.
67/67 [==============================] - 34s 505ms/step - loss: 0.0123 - accuracy: 0.9995 - val_loss: 0.5233 - val_accuracy: 0.8393 - lr: 2.0000e-05
Epoch 15: early stopping
2025-06-05 11:19:51,988 - INFO -
Test Accuracy: 82.23%
2025-06-05 11:19:51,988 - INFO - Test Loss: 0.4124
C:\Users\harib\Desktop\sig\venv\lib\site-packages\keras\src\engine\training.py:3103: UserWarning: You are saving your model as an HDF5 file via `model.save()`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.
keras')`.
  saving_api.save_model(
2025-06-05 11:19:53,069 - INFO - Model saved to C:\Users\harib\Desktop\sig\models_output\signature_combined_model.h5
2025-06-05 11:19:53,069 - INFO - SIFT scaler saved to C:\Users\harib\Desktop\sig\models_output\sift_feature_scaler.pkl
2025-06-05 11:19:53,084 - INFO - Training complete. Model and scaler saved.
```

Fig. 6.10 Epochs

Fig 6.10 model was fine-tuned over 15 epochs, and the trained model was successfully saved, as shown in the figure.

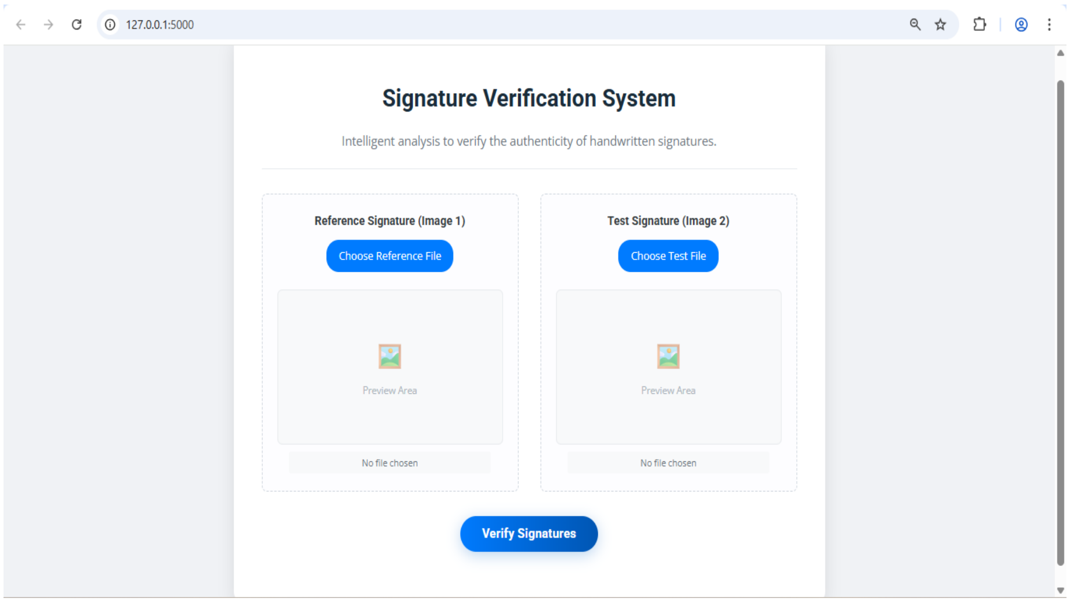


Fig. 6.11 Upload Two Images

Fig 6.11 is the UI of the project and tells to upload two images.

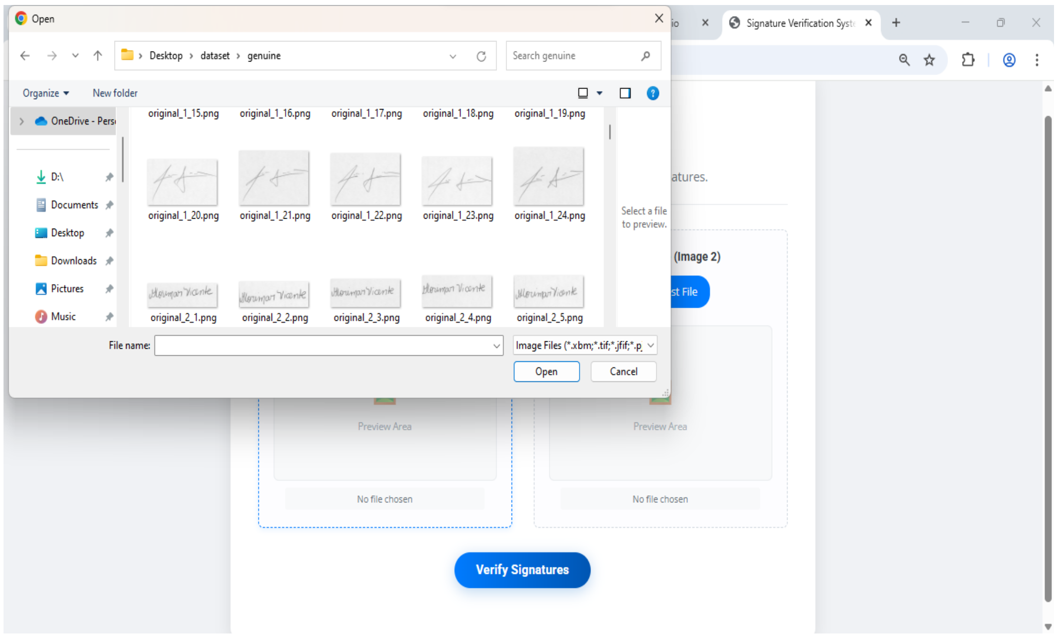


Fig. 6.12 Reference Image

Fig 6.12 tells us about uploading the first reference image from the dataset.

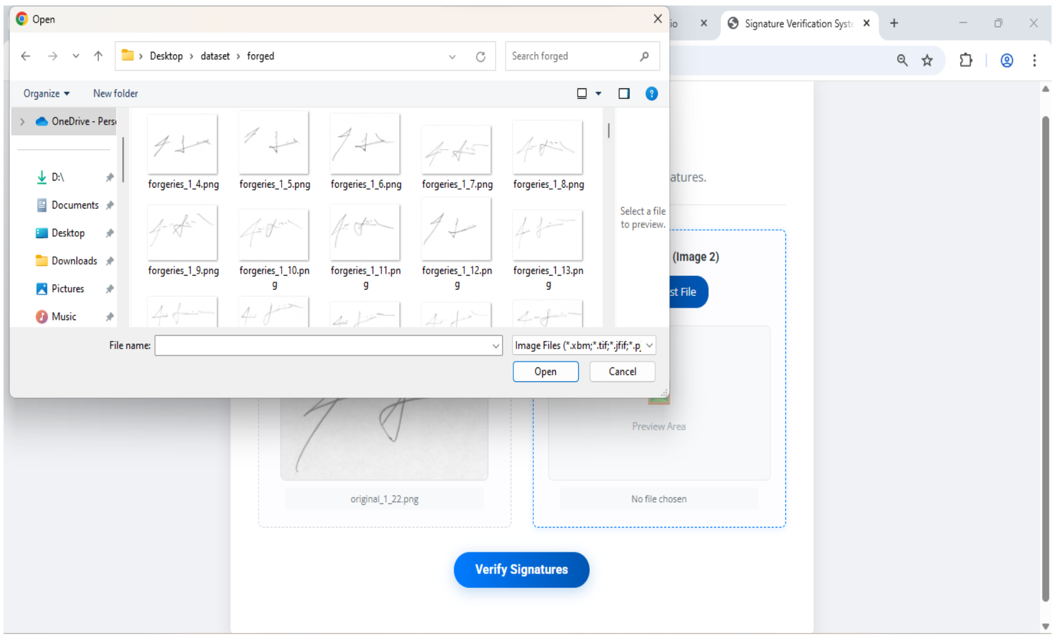
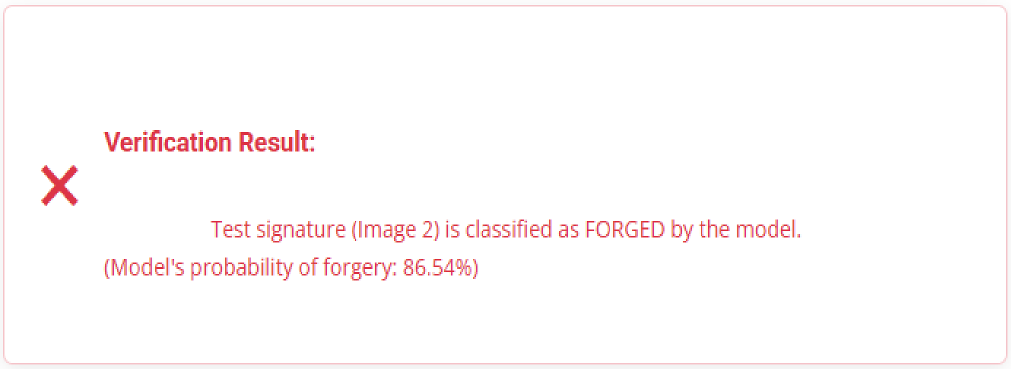


Fig. 6.13 Test Signature Image

Fig 6.13 tells about uploading a test signature image for giving the result.



Verification Result:

Test signature (Image 2) is classified as FORGED by the model.
(Model's probability of forgery: 86.54%)

Fig. 6.14 Result

Fig 6.14 tells us that the signature uploaded is classified as Forged by the model with a confidence score.

CHAPTER 7
CONCLUSION

The developed Signature Fraud Detection system, utilizing a hybrid approach of deep learning (CNN) and traditional computer vision techniques (SIFT), marks a significant step towards enhancing the security and reliability of signature-based authentication. By synergistically combining the robust local feature extraction capabilities of SIFT with the powerful hierarchical pattern recognition of a Convolutional Neural Network (CNN), the system demonstrates its capability to effectively differentiate between genuine and fraudulent signatures.

In terms of practical implementation, the integration of this SIFT-CNN model into a web application developed using Flask allows users to easily upload a reference signature and a test signature for real-time comparison. The Flask interface provides a user-friendly platform for interacting with the fraud detection engine, making the technology accessible for demonstration and potential real-world evaluation. This user-centric design ensures that the system is not only functional in its analytical capabilities but also practical to use.

The successful development and testing of the SIFT-CNN model underscore the growing potential of hybrid intelligent systems in automating complex tasks like signature fraud detection. While the system has shown promising results in its ability to analyze and compare signatures, certain limitations, such as handling highly skilled forgeries that perfectly mimic genuine stroke dynamics or extreme intra-personal variations in genuine signatures, remain challenging.

In conclusion, this project demonstrates the transformative potential of combining classical feature engineering with deep learning for robust signature fraud detection. By leveraging the distinct strengths of both SIFT and CNNs, the system offers a more comprehensive approach to analyzing signature authenticity. The user-friendly Flask web interface ensures that this technology can be easily demonstrated and understood, highlighting its value as a versatile tool for research, development, and potential deployment in security-sensitive applications.

CHAPTER 8
FUTURE SCOPE

The future potential for signature fraud detection using hybrid deep learning and computer vision techniques is substantial, with several key areas of exploration expected to further enhance the system's capabilities, accuracy, and applicability. As AI and machine learning continue to evolve, these advancements will significantly impact both academic research and practical security implementations. The future scope for this project includes the following key areas:

- **Enhanced Hybrid Model Architectures:** Future research will likely focus on refining the synergy between handcrafted features like SIFT and deep learning models like CNNs. Exploring more sophisticated feature fusion techniques beyond simple concatenation, such as attention mechanisms that weigh the importance of SIFT versus CNN features dynamically, or gated fusion mechanisms.

- **Incorporation of Online/Dynamic Signature Features:** While this project focuses on offline (static image) signatures, future systems could explore ways to incorporate pseudo-dynamic features extracted from static images (such as stroke thickness variation, inferred speed/pressure points) or even build hybrid systems that can process both offline images and limited dynamic data if available from certain capture devices.

- **Explainability and Interpretability (XAI):** As signature fraud detection systems make critical decisions, understanding *why* a signature is flagged as fraudulent is crucial. Implementing XAI techniques (e.g., LIME, SHAP, Grad-CAM for the CNN part) to visualize which parts of the signature or which features (SIFT keypoints or CNN activations) contributed most to the fraud/genuine decision.

- **Real-Time and On-Device Fraud Detection:** With advancements in model optimization and edge computing hardware future systems could be optimized for near real-time fraud detection, suitable for point-of-transaction verification. Deploying lightweight versions of the SIFT-CNN model on mobile devices or embedded systems for localized, secure signature verification without constant cloud connectivity.

- **Continuous Learning and Adversarial Robustness:** Developing models that can incrementally learn from new genuine signatures and new types of forgeries encountered over time, without complete retraining, to adapt to evolving fraud tactics.

- **Writer-Specific Thresholding and Adaptation:** Instead of a global threshold for genuine/forged, future systems could adapt verification thresholds on a per-user basis, learning the typical range of variation for each individual's genuine signatures to reduce False Rejections and improve sensitivity to forgeries for that specific writer.

- **Integration with Larger Security Ecosystems:** Combining signature fraud detection with other biometric modalities (e.g., fingerprint, face recognition) or behavioural biometrics for multi-factor authentication, creating more robust security systems. Integrating with forensic analysis tools to provide supporting evidence in fraud investigations.

- **Ethical and Legal Considerations:** Addressing data privacy concerns related to storing and processing signature biometrics. Ensuring fairness and minimizing bias in the models, particularly if trained on datasets that are not demographically representative.

The future of automated signature fraud detection using advanced techniques like combined SIFT and CNN is promising. Continuous advancements are expected to push the boundaries of accuracy, robustness, and trustworthiness, leading to more secure and adaptable systems that can be widely adopted across financial, legal, and governmental sectors to combat fraud effectively.

REFERENCES

[1] Tolosana, R., Vera-Rodriguez, R., Fierrez, J., and Ortega-Garcia, J. "DeepSign: Deep On-Line Signature Verification." IEEE Transactions on Biometrics, Behavior, and Identity Science, vol. 3, no. 2, 2021, pp. 229–239. doi:10.1109/TBIOM.2021.3054533.

[2] Poddar, J., Parikh, V., and Bharti, S. K. "Offline Signature Recognition and Forgery Detection Using Deep Learning." Procedia Computer Science, vol. 170, 2020, pp. 610–617. doi:10.1016/j.procs.2020.03.133.

[3] Zagoruyko, S., and Komodakis, N. "Learning to Compare Image Patches via Convolutional Neural Networks." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 4353–4361. doi:10.1109/CVPR.2015.7299064.

[4] Fahmy, M. M. M. "Online Handwritten Signature Verification System Based on DWT Features Extraction and Neural Network Classification." Ain Shams Engineering Journal, vol. 1, no. 1, 2010, pp. 59–70. doi:10.1016/j.asej.2010.09.007.

[5] P. Manjula Devi, M. Ahmed, M. F. Alam, and R. Ashay, "Signature Forgery Detection Using CNN and HOG," International Research Journal of Modernization in Engineering Technology and Science, vol. 6, no. 4, pp. 11006–11010, Apr. 2024. doi:10.56726/IRJMETS54585.

[6] Bharti, P., and Natarajan, C. "Deep Learning-Based Signature Verification." International Journal of Advance Research and Innovative Ideas in Education, vol. 9, no. 4, pp. 720–724, 2023.

[7] Sharma, H. "Offline Bank Cheque Signature Verification Using SIFT." International Journal of Research in Engineering, Science and Management (IJRESM), vol. 2, no. 10, 2019, pp. 255–259.

[8] Jindal, U., and Dalal, S. "Survey on Signature Verification and Recognition Using SIFT and Its Variant." International Journal of Recent Research Aspects, Special Issue: Conscientious and Unimpeachable Technologies, 3.2 (2016): 26–29.

[9] Bhuvaneswari, P., Christopher, K. M., Raj, V. R. M., and Ajithkumar, M. "Signature Forgery Detection Using Deep Learning." International Research Journal of Engineering and Technology (IRJET), vol. 11, no. 4, 2024, pp. 209–213.

[10] Lopes, J. A. P., Baptista, B., Lavado, N., and Mendes, M. "Offline Handwritten Signature Verification Using Deep Neural Networks." Energies, vol. 15, no. 20, 2022, pp. 1–15. doi:10.3390/en15207611.