TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

HIMALAYA COLLEGE OF ENGINEERING

[CT-654]

A

PROJECT REPORT

ON

AI GENERATED FACE DETECTION

SUBMITTED BY:

KALYAN SUBEDI [HCE078BCT018]

ROSHNA SHRESTHA [HCE078BCT031]

RUJIT SHRESTHA [HCE078BCT032]

SANDHYA ADHIKARI [HCE078BCT038]

A PROJECT REPORT SUBMITTED TO DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR BACHELOR'S DEGREE IN COMPUTER ENGINEERING

Department of Electronics and Computer Engineering

Lalitpur, Nepal

March, 2025

AI GENERATED FACE DETECTION

Submitted by:

Kalyan Subedi [HCE078BCT018]

Roshna Shrestha [HCE078BCT031]

Rujit Shrestha [HCE078BCT032]

Sandhya Adhikari [HCE078BCT038]

"A Project Report Submitted in Partial Fulfillment of the Bachelor's Degree in Computer Engineering"

Supervisor:

Er. Sushant Pandey

Department of Electronics and Computer Engineering

Himalaya College of Engineering

Tribhuvan University

Lalitpur , Nepal

March, 2025

ACKNOWLEDGEMENT

First and foremost, we are profoundly grateful to our project coordinator Er. Ramesh Tamang, and Er. Narayan Adhikari, whose support, expert guidance, and insightful feedback have been invaluable. We would like to extend our sincere and heartfelt thanks to our supervisor, Er. Sushant Pandey, for his continuous guidance and encouragement throughout the project. Additionally, we express our gratitude to our Head of Department (HOD), Er. Ashok GM, and Deputy Head of Department (DHOD), Er. Devendra Kathayat, for providing us with essential resources and constant support, which equipped us with the knowledge and skills required for this project.

We would also like to express our gratitude to Himalaya College of Engineering for providing us with necessary resources and a conductive environment for learning and innovation. The academic foundation and opportunities provided by the institute have been pivotal in enabling us to delve into the realm of programming world and enhance our skills.

Group Members

Kalyan Subedi (HCE078BCT018)

Roshna Shrestha (HCE078BCT031)

Rujit Shrestha (HCE078BCT032)

Sandhya Adhikari (HCE078BCT038)

ABSTRACT

This research addresses the challenge of detecting AI-generated deepfakes, particularly realistic ones, to preserve digital authenticity and security. A detection system based on DenseNet121 architecture, utilizing transfer learning and fine-tuning on a balanced dataset of 65,000 images, achieves 99.42% validation accuracy with only 7 training epochs. Preprocessing techniques like data augmentation, normalization, and adaptive training strategies with checkpointing enhance feature extraction and model generalization. The web application, built with React for the frontend and Flask for the backend, allows image uploads, cropping, and predictions with confidence scores.

The model performed well on an independent test set of 13,000 images, with only 65 false negatives and 11 false positives. This study underscores the role of deep learning in combating digital manipulation, providing practical solutions for security, media validation, and identity verification. Future work will expand to detect new image categories and analyze videos, while considering the ethical implications of generative technologies to foster trust in digital content.

Keywords: AI-face detection, DenseNet121, synthesized media, transfer learning, web classifier.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLE

LIST OF ABBREVIATIONS

ADAM:	Adaptive moment activation
BLEU:	Bilingual Evaluation Understudy
CGFI:	Computer Generated Face Image
CNN:	Convolutional Neural Network
FFW:	Fake Face in the Wild
GAN:	Generative Adversarial Networks
GNN:	Graph Neural Network
JPEG:	Joint Photographic Expert Group
LBP:	Local Binary Problems
LSTM:	Long Short-term Memory
NMT:	Neural Machine Translation
PNG:	Portable Network Graphics
RNN:	Recurrent Neural Networks
VGG19:	Visual Geometry Group

CHAPTER 1 INTRODUCTION

1.1 Background

In an age characterized by the pervasive dissemination of visual content across digital platforms, the authenticity and credibility of images have become increasingly paramount [1]. While this enhances engagement and enriches user experience, it also brings significant challenges. The ease of creating and sharing images, combined with the advent of powerful AI-driven tools, has made it increasingly difficult to differentiate between authentic and fabricated visual content.

Deep learning is particularly important in the field of artificial intelligence, which builds a neural network that mimics the thinking pattern of the human, builds a neural network that responds intelligently like the human brain, or even surpasses it [2]. With recent advances in deep learning, it is now possible to seamlessly generate manipulated images morphing, Snap-chat, Computer Generated Face Image (CGFI), Generative Adversarial Networks (GAN) and Face2Face [3].

DenseNet121 is a popular convolutional neural network (CNN) architecture designed for image recognition tasks. It is part of the DenseNet (Dense Convolutional Network) family, which introduces dense connectivity between layers to improve information flow and reduce redundancy.

We have witnessed significant advancements in image-generation AI models in recent years, including the impressive transformer auto-regressive model DALL-E 2, Mid journey, Imagen 3 developed by Open-AI. While these models have various applications, including entertainment, advertising, and architecture design, their use raises important ethical questions due to the potential misuse of AI-generated images. As such, we aim to build a machine learning model that can differentiate between AI-generated images and real human face images to promote ethical practices in their use and contribute to this significant area of study.

1.2 Problem Statement

The rise of highly realistic AI-generated faces, powered by technologies like GANs, has made it increasingly difficult to distinguish between real and synthetic faces. This poses significant challenges, including the spread of misinformation, identity manipulation, and a growing loss of trust in digital media. Current face detection methods often lack the accuracy and interpretability needed to effectively address these issues, particularly in critical areas such as media, identity verification, and legal proceedings. This highlights the urgent need for robust, precise, and transparent AI-driven face detection solutions.

1.3 Objectives

The objectives of our projects are:

- Develop a DenseNet121-based AI model to distinguish real from AI-generated faces.
- Deploy via an intuitive web application using React for the frontend and Flask for the backend.

1.4 Scope and Applications

- Develop AI models for real-time detection of real vs. synthetic images, ensuring the authenticity of content across diverse platforms and environments.
- Apply the model to authenticate visual content and moderate media, effectively addressing fake or misleading visuals across various platforms.
- Use the model in security systems and identity verification applications to prevent fraud, identity theft, and ensure digital trust.

1.5 Report Organization

The diagram illustrates the structure of a project report, starting with Chapter 1, which introduces the project by discussing its background, objectives, and scope. Chapter 2 provides a literature review, summarizing prior research and highlighting gaps that the project aims to address. Chapter 3 focuses on requirement analysis, detailing both functional and non-functional requirements essential for the system. Chapter 4 outlines the system design, presenting technical workflows and key diagrams to describe the system's structure and components. Chapter 5 explains the methodology, describing the tools, techniques, and processes used to achieve the project's objectives. Chapter 6 presents the project schedule, breaking down tasks into phases with a timeline for completion. Finally, Chapter 7 concludes with the expected outcomes, outlining the deliveries and how they align with the project goals.

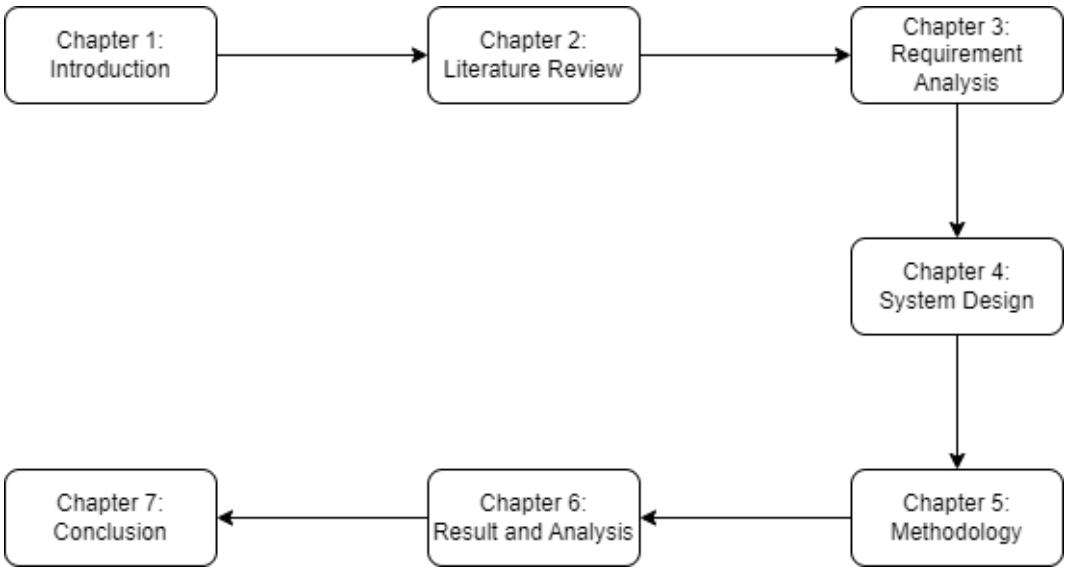


Figure 1.1 Report Organization

CHAPTER 2 LITERATURE REVIEW

2.1 Related Work

The rapid increase in AI generated images has made image authenticity detection crucial. The advent of advanced image manipulation techniques has given rise to a critical need for robust mechanisms that can distinguish between authentic and synthetic visual content [1]. As the boundary between authentic and artificial visual content has become increasingly thinner, this technological evolution presents significant challenges across various sectors, from journalism, social media platforms and cybersecurity. To overcome these challenges several methods and techniques have been introduced.

Deep neural networks are increasingly used on mobile devices, where computational resources are limited [4]. The paper focuses on CondenseNet: an efficient convolution network that facilitates feature reuse in a neural network via dense connectivity and learned group convolutions. During its test time, the model could be implemented using standard group convolutions that allowed for efficient computation practice that proved DenseNet to be far ahead of its competitors during the time.

A study done on generalization of fake face detection methods utilizing more than 53,000 images from videos and CGI generation [3] created dataset named Fake Face in the Wild (FFW). This paper shed light on the deficiency of the detection algorithms when unknown data was present. The accuracy of the various LBP and CNN based architectures like AlexNet, VGG19, ResNet50 were analyzed and tabulated. The performance was categorized into two sets, i.e. known fake face and unknown fake face. The CNN based methods were found to be 98% accurate compared to 96% of LBP in known fake faces but the tides turned around when the second test was performed.

Because of the wide application of deep learning, there are more neural network structures in image recognition technology nowadays, but there are various

differences in the accuracy of image recognition because of the various differences in network structures. For this reason, it is especially important to use different neural network structures for different forms of image data [2]. This paper focuses on exploring the differences between LSTM networks, residual networks, and CNN networks in terms of the accuracy of cartoon character recognition. After acquiring 14 different cartoon character images and removing duplicates to obtain the original data, data enhancement was performed on the preliminary data. The images were rotated to complete the pre-processing of the data, which solved the problem of using different code forms for different forms of data importing into the neural network; the LSTM network, CNN network and CNN network with added residual function were used to recognize the pre-processed data. The experiments show that the CNN network structure with residual function can achieve higher accuracy compared to LSTM.

“Deep learning in Image classification: A survey Report” [5] provides an insight into the rapidly growing field of deep learning. In this work, several popular neural networks such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Graph Neural Networks (GNN) and a small overview of their application in image classification in discussed. The paper also focuses on the challenge of deep learning in image classification. It is highlighted that while CNNs are natural option for input data comprising of clear spatial structures, RNNs excel when the data is sequentially aligned. Apple’s Siri and Google Assistant are prime examples of RNN used in sequential ordered data. GNNs, unlike other neural networks, don’t require data to be represented in Euclidean space and can handle inclusive data generated from non-Euclidean domains. So, GNN is one of the emerging tools to resolve problems like image classification.

The conference proceeding of 2024 International Conference on Contemporary Computing and Communications (Inc4) titled ”Detecting AI-generated images with CNN and Interpretation using Explainable AI” [1] encompasses a meticulous data collection process, preprocessing steps, the selection of the DenseNet121 model architecture, model training, and evaluation of the model's performance through a range of metrics, including accuracy (93.42), precision, recall, and the Area Under

the ROC Curve (AUC). To enhance the interpretation ability of the model, Gradient-weighted Class Activation Mapping (Grad-CAM) technique is utilized enabling visualization of the regions within images that influenced the model's decision-making.

“Neural Machine Translation using Adam Optimized Generative Adversarial Network” by [6] discusses the different methods of Neural Machine Translation (NMT) and their architectures. To overcome the problem of vocabulary remaining the same, attention-based NMT along with sub-word segmentation is introduced. An Adam-based Bi-directional GAN is employed to optimize the training process and to stabilize the GANs. The model is evaluated based on BLEU scores and is compared with the existing models. In this paper, a unique Bidirectional GAN for NMT is proposed to address this problem and stabilize the GAN training.

DenseNet-121 and its DenseNet variants are not only highly accurate but also remarkably efficient. For example, a DenseNet-BC with L (Represents the network depth, i.e., the total number of layers) = 100 and k (The growth rate; the number of feature maps each layer contributes) = 12 uses just 0.8 million parameters yet achieves error rates on CIFAR-10 (A 10-class image dataset developed by the Canadian Institute for Advanced Research) comparable to a 1001-layer pre-activation ResNet (Residual Network) that uses 10.2 million parameters. On ImageNet, DenseNet-121 reaches a top-1 error of 25.02% (23.61% with 10-crop testing), demonstrating that its dense connectivity yields superior gradient flow and feature reuse while maintaining a much lighter model compared to traditional architectures like ResNets and Inception. [7]

2.2 Related Theory

It is essential to understand the related theories that form the foundation of our work. The following section provides an overview of these relevant theories.

2.2.1 CNN Model

A Convolutional Neural Network (CNN) is a deep learning architecture specifically designed to process image data and detect visual patterns such as edges, shapes, and textures. In our project, we are using a particular type of CNN called DenseNet121. DenseNet121 improves upon traditional CNNs by connecting every layer to all subsequent layers. This unique feature improves the flow of information throughout the network, which not only reduces the number of parameters but also helps prevent overfitting. This results in a more efficient and accurate model for detecting features in images.

Traditional Convolutional Neural Networks (CNNs) have been the backbone of many computer vision applications over the past decade. These networks typically feature a series of convolutional layers that extract hierarchical features from images, interspersed with pooling layers to reduce spatial dimensions and eventually followed by fully connected layers for classification. Models such as LeNet, AlexNet, VGG, and ResNet have showcased the power of CNNs for tasks ranging from image recognition to object detection.

Core Characteristics of Traditional CNNs:

- **Layer-by-Layer Processing:** Information is processed sequentially from one layer to the next. Each layer learns to extract increasingly complex features.
- **Pooling Operations:** Max pooling or average pooling is used to reduce the dimensionality, which can sometimes lead to a loss of fine-grained information.
- **Fully Connected Layers:** The network eventually flattens the feature maps for classification, which can lead to a large number of parameters and risk of overfitting.

- **Gradient Flow Issues:** In deeper CNNs, problems like vanishing gradients can emerge, making it difficult for the network to learn effectively without additional architectural tweaks or techniques such as batch normalization.

2.2.2 DenseNet121

DenseNet121 is a convolutional neural network architecture designed to improve information flow and feature reuse by densely connecting each layer to all subsequent layers. Unlike traditional CNNs, where each layer only passes information to the next one, DenseNet121 connects each layer directly with every other layer ahead of it. This reduces the number of parameters while maintaining model performance, helping to alleviate the vanishing gradient problem. DenseNet's dense connectivity also leads to more efficient training and allows for better gradient flow, which is beneficial in deep networks. [7]

Some of the terms used when describing DenseNet architecture are as follows::

Convolution: A process where filters slide over input data to detect features like edges and textures, forming the basis of CNNs.

Pooling: Reduces the size of feature maps by selecting the maximum or average value in regions, improving efficiency and reducing overfitting.

Batch Normalization: Normalizes layer inputs to speed up training and stabilize learning by controlling activation ranges.

Relu: An activation function that outputs the input if it's positive, or zero otherwise, introducing non-linearity to help the model learn complex patterns.

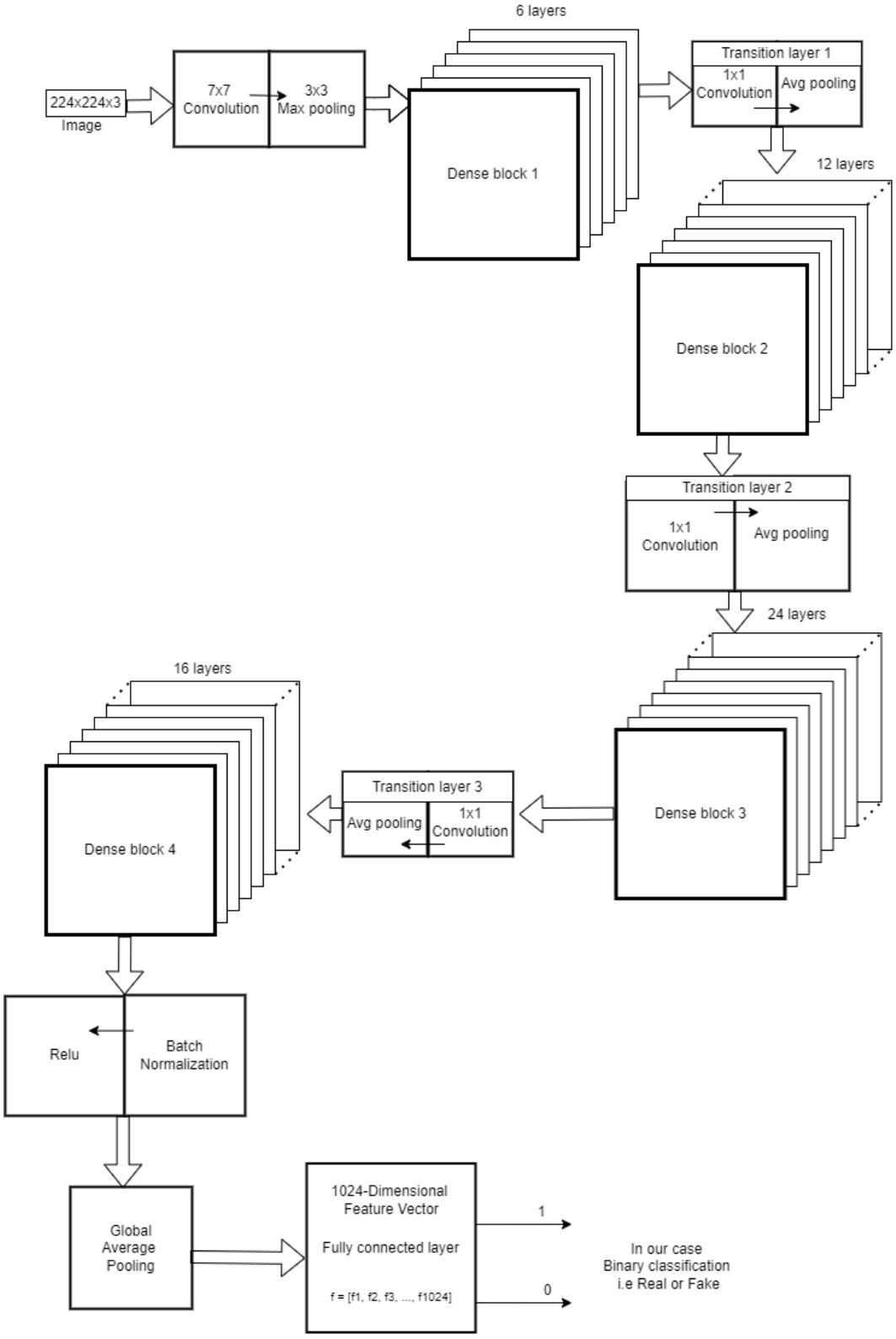


Figure 2.1 Dense Net Architecture

Input Layer

- The network starts by receiving an input image. Suppose the input size is 224x224x3 (a typical size for an RGB image).
- This input is passed on to the **Initial Convolutional Layer**.

Initial Convolutional Layer

- Convolution (7x7 filter) is applied with a stride of 2, which reduces the spatial dimensions of the image (224x224 becomes 112x112). This step extracts basic features like edges and textures.
- Batch Normalization and ReLU **Activation** follow.
- This layer is followed by **Max Pooling (3x3)** with a stride of 2, which further reduces the size (112x112 becomes 56x56).

Dense Block 1

Now the input enters the first Dense Block, which consists of **6 densely connected layers**. Here's what happens:

- Layer 1: Convolution (1x1 filter) → Batch Norm → ReLU → Convolution (3x3 filter).
- Layer 2: Receives inputs from Layer 1 + Initial Layer Output. It repeats the same conv operations.
- Layer 3: Receives inputs from Layer 1 + Layer 2 + Initial Output.
- This continues until Layer 6, which receives inputs from all previous layers.

  Output dimensions stay at 56x56 throughout.

Transition Layer 1

- The transition layer reduces the size to prevent too much computational burden. It includes:
  - 1x1 Convolution to reduce the feature map size.
  - Batch Normalization and ReLU.
  - Average Pooling **(2x2)**, which reduces the dimensions from **56x56** to **28x28**.

Dense Block 2

- This block has 12 densely connected layers.
- Each new layer in this block receives inputs from all previous layers in the block.
- The output remains 28x28 but with many more feature maps.

Transition Layer 2

- Similar to the previous transition, we reduce dimensions again using:
    - 1x1 Convolution.
    - Batch Norm and ReLU.
    - Average Pooling, reducing the size to **14x14**.

Dense Block 3

- This block has **24 densely connected layers**, following the same pattern.
- Each layer receives inputs from all preceding layers in this block.
- The output size remains **14x14**.

Transition Layer 3

- Similar process as before:
    - 1x1 Convolution reduces the feature map size.
    - Average Pooling, reducing size from **14x14** to **7x7**.

Dense Block 4

- The last block consists of **16 densely connected layers**.
- It follows the same pattern: each layer receives inputs from all previous layers in this block.
- The output remains **7x7**, but the number of feature maps continues to grow.

Global Average Pooling

- After the last dense block, we applied **Global Average Pooling**. This compresses the feature maps into a **1x1xK** output by averaging each 7x7 feature map across its entire spatial area.

- This step converts the **7x7** feature map into a **1D feature vector**.

Fully Connected Layer

- Finally, this feature vector is passed to a **Fully Connected (FC) layer**, which maps the features to the output classes.
- In the case of ImageNet, for example, this would result in a **1000**-dimensional output for the 1000 classes in the dataset.

This efficient architecture has led DenseNet-121 to achieve state-of-the-art performance across various image classification tasks, making it a suitable choice for applications such as detecting AI-generated fake faces. [8]

2.2.3 Traditional CNNs VS DenseNet-121

This section highlights how the architectural innovations of DenseNet-121 address some of the key limitations found in traditional CNNs, leading to improved efficiency, better gradient flow, and enhanced performance in image processing tasks.

Disadvantages of Traditional CNNs

- **Limited Feature Reuse:**
  Traditional CNNs typically do not have a mechanism for early layer features to be directly available to later layers. This can lead to redundant computations and a less efficient representation of features.
- **Vanishing Gradient Problem:**
  In very deep networks, gradients may diminish as they propagate backward through the layers. This can slow down or even stall the learning process, making it challenging to train very deep architecture without additional strategies.
- **High Parameter Count:**
  The use of fully connected layers and less efficient layer connectivity often

leads to a large number of parameters. This can increase the risk of overfitting, especially on smaller datasets, and results in higher computational and memory requirements.

- **Inefficient Information Flow:**
  With information being processed sequentially without direct shortcuts between non-adjacent layers, valuable features may degrade or be lost as the network depth increases, potentially reducing overall performance.

Advantages of DenseNet-121

- **Enhanced Feature Reuse:**
  DenseNet-121's architecture allows each layer to access the feature maps of all previous layers directly. This reuse of features leads to more informative representations and reduces the need for redundant computations.

- **Improved Gradient Flow:**
  The dense connections ensure that gradients can propagate more smoothly throughout the network, effectively mitigating the vanishing gradient problem. This enables the network to be both deep and trainable.

- **Parameter Efficiency:**
  By leveraging feature reuse, DenseNet-121 achieves competitive or even superior performance with a smaller number of parameters compared to traditional CNNs. This results in a more compact model that is computationally efficient.

- **Better Information Propagation:**
  The dense connectivity not only improves gradient flow but also facilitates better propagation of low-level features to higher layers. This can lead to improved performance on image recognition tasks, as the network can leverage both fine and coarse features effectively.

- **Reduced Overfitting:**
  The efficient use of parameters and enhanced feature propagation can also

contribute to a reduction in overfitting, making DenseNet-121 robust even when trained on relatively small datasets.

2.2.4 Adam Optimizer

Adam Optimizer (Adaptive Moment Estimation) is an optimization algorithm in deep learning, which adjusts the learning rate for each parameter individually, helping the model converge faster and more efficiently. [9]

Gradient Calculation Using Cross-Entropy Loss:

The gradient of a loss function with respect to the model's weights tells us how to adjust the weights to reduce the loss. The gradient is calculated using backpropagation.

For our project, the loss function used is Cross-Entropy Loss. Here's how it works:

- **Cross-Entropy Loss** measures the difference between the predicted probability distribution (from the model's output) and the true class labels (which are either 0 or 1 for binary classification).

$$Loss = -(y.log(\hat{y}) + (1 - y).log\ (1 - \hat{y}) \quad (1)$$

Where:

y is the actual label (0 or 1).

$\hat{y}$ is the predicted probability of the positive class (real or 1).

For binary classification, the predicted value is the probability that the class is 1.

$\log(\hat{y})$ is the log of the predicted probability for the positive class.

$\log\ (1 - \hat{y})$ is the log of the predicted probability for the negative class.

- The loss function gives a higher penalty if the model's predicted probabilities are far from the actual class labels, and a lower penalty if the predicted probabilities are closer to the true labels.

During backpropagation, the gradient of the Cross-Entropy Loss is computed with respect to each model parameter (weight). The gradient tells the optimizer how to update the weight to reduce the loss.

Input:

It takes the gradient (g_t) of the loss function with respect to the model's parameters at each step. This gradient is calculated for each input image passed through the DenseNet121 model.

First Moment Estimate (m_t)

Adam calculates the first moment estimate, which is the moving average of the gradients. This helps smooth out noisy gradient updates:

$$M_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \quad (2)$$

Here,

M_t is the first moment at time step t, and β_1 is a hyper parameter which controls the decay rate of this moving average.

Second moment estimate (v_t):

It is the moving average of the squared gradients, which helps to adapt the learning rate for each parameter:

$$V_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \quad (3)$$

V_t, the second moment estimate and β_2, hyper parameter, controls the decay rate for the squared gradients.

Bias Correction: Bias is corrected using the following formulas:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad (4)$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2{}^t} \quad (5)$$

Parameter Update

Finally, the weight of the model is updated using the corrected first and second moments, with the following formula:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \quad (6)$$

Where,

θ_t is the updated model parameter at time step t.

α is the learning rate (typically set to 0.001)

ϵ is a small constant to prevent division by zero.

This process is repeated for each batch of input images, adjusting the model's parameters to minimize the loss function over multiple epochs. The use of Adam Optimizer ensures that the model adapts to each image's features and converges faster, optimizing DenseNet121's performance in detecting real vs. AI-generated faces.

CHAPTER 3 REQUIREMENT ANALYSIS

3.1 Functional Requirements

The system should provide the following core functionalities:

- **Upload Image**: The user can upload an image containing a face to be analyzed.
- **Crop Image**: User should be able to crop the uploaded image.
- **Start Prediction:** The user can click on the predict button when they want the image to be analyzed.
- **AI Detection**: Once the image is uploaded, the system will process the image and will generate an output whether the face is AI-generated or real and confidence percentage.
- **View Result**: The system will display the result generated by the ML model.
- **Image Format**: The system should accept image inputs in common formats such as JPEG and PNG.

3.2 Non-Functional Requirements

These requirements define how well the system performs its core functions:

- The system should process an image and return a prediction with minimum delay.
- The model should be optimized to handle inference with minimal latency.
- The AI detection model should have higher accuracy.
- The system should be robust to different types of faces, lighting conditions, and backgrounds.

- The user interface should be intuitive and easy to use, allowing users to upload an image and receive results with minimal effort.

3.3 Technical Requirements

These are the technical specifications that define the architecture, tools, and technologies the system will use.

Programming Languages and Frameworks

- **Backend**: Flask (for backend logic)
- **Web Framework**: React for front end
- **Machine Learning Framework**: PyTorch (for training and running the DenseNet121 model)
- **Version Control**: Git (for source code management)

Hardware Requirements

- **NVIDIA RTX 3050**: Accelerates parallel processing with its dedicated tensor core.
- **VRAM**: 4GB enables efficient handling of facial image data.
- **RAM**: 16GB supports smooth concurrent processing and model training.
- **Processor**: Intel i5 11th Gen delivers balanced performance.

Data Requirements

- **Training Data**: A dataset of real and AI-generated faces, with proper labels (real or AI-generated).
- **Test Data**: A separate dataset for evaluating the model's performance.

CHAPTER 4 SYSTEM DESIGN

This chapter focuses on the system design, detailing the key components and how they interact within the fake face detection system. It provides an overview of the system's structure, showing how data flows between the user and the system, as well as the internal processes that enable the system to function effectively. The design presented offers a clear understanding of how the system operates and supports the detection of AI-generated fake faces

4.1 Use Case Diagram

The diagram below illustrates the interactions between the user and the system's main functions.

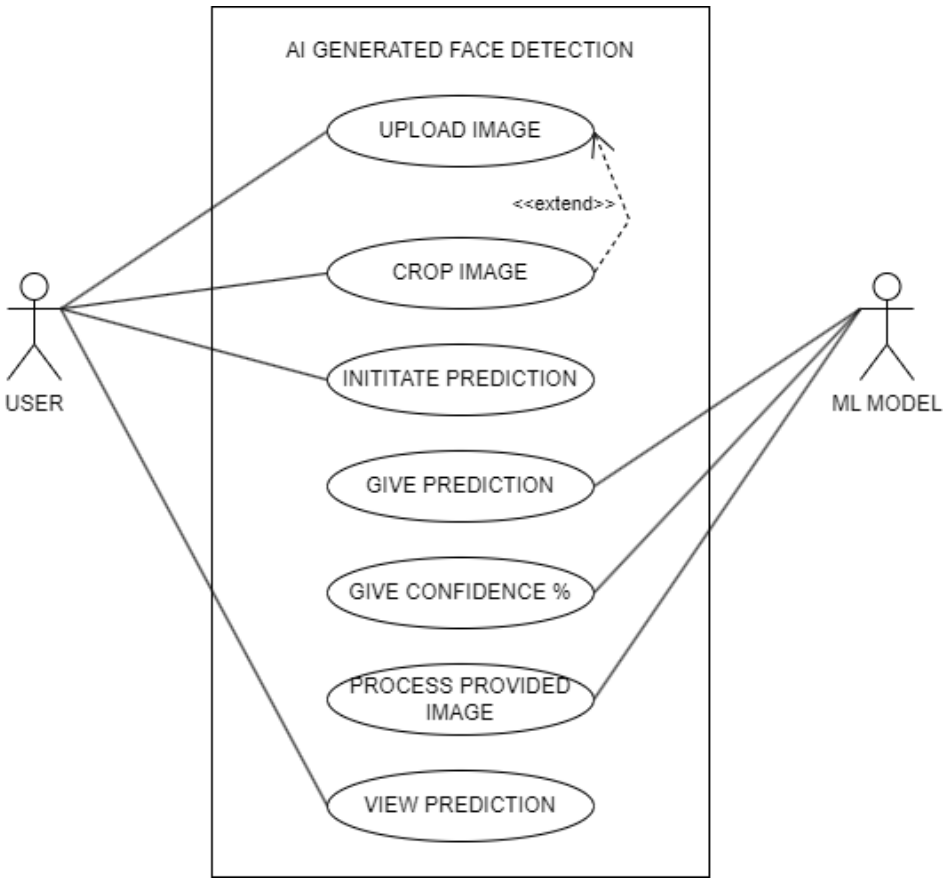


Figure 4.1: Use Case Diagram

4.2 Data Flow Diagram

The diagram below shows the flow of data between the user and the AI generated fake face detection system.

4.2.1 Level 0 DFD

This Level 0 Data Flow Diagram (DFD) provides a high-level overview of the interaction between the **User** and the **Fake Face Detector** system.

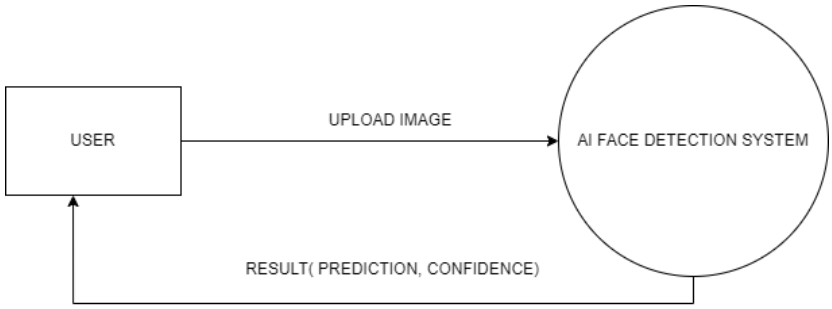


Figure 4.2 Level 0 DFD

4.2.2 Level 1 DFD

The Level 1 DFD expands on the system's internal processes.

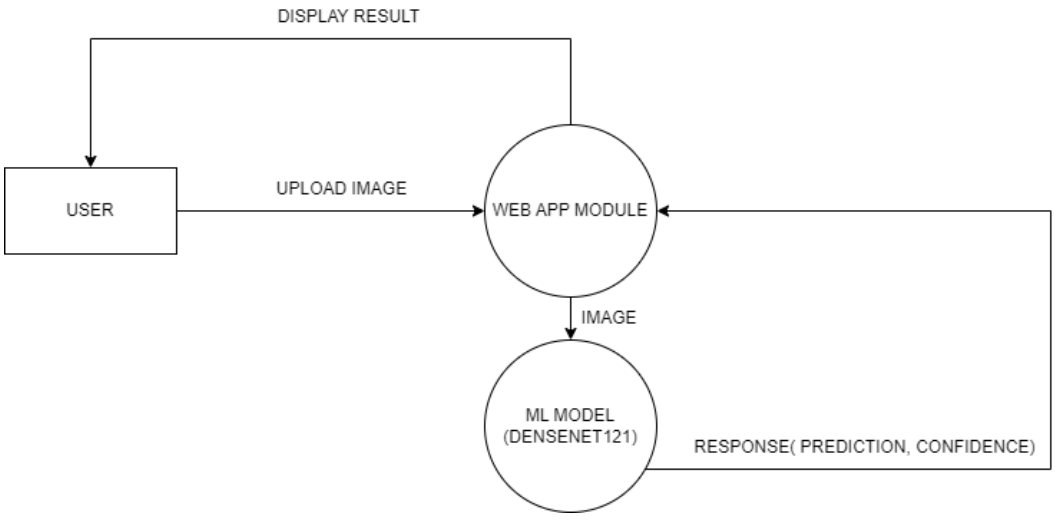


Figure 4.3 Level 1 DFD

4.3 Sequence diagram

This sequence diagram illustrates how a user uploads an image to the web application, optionally crops it, and then requests a prediction from the inference model. Specifically, the user's actions trigger the web app to display an image preview and handle cropping; once finalized, the cropped image is sent to the inference model for processing. After the model determines whether the face is real or fake (along with a confidence score), the result is returned to the web app, which displays the outcome to the user.

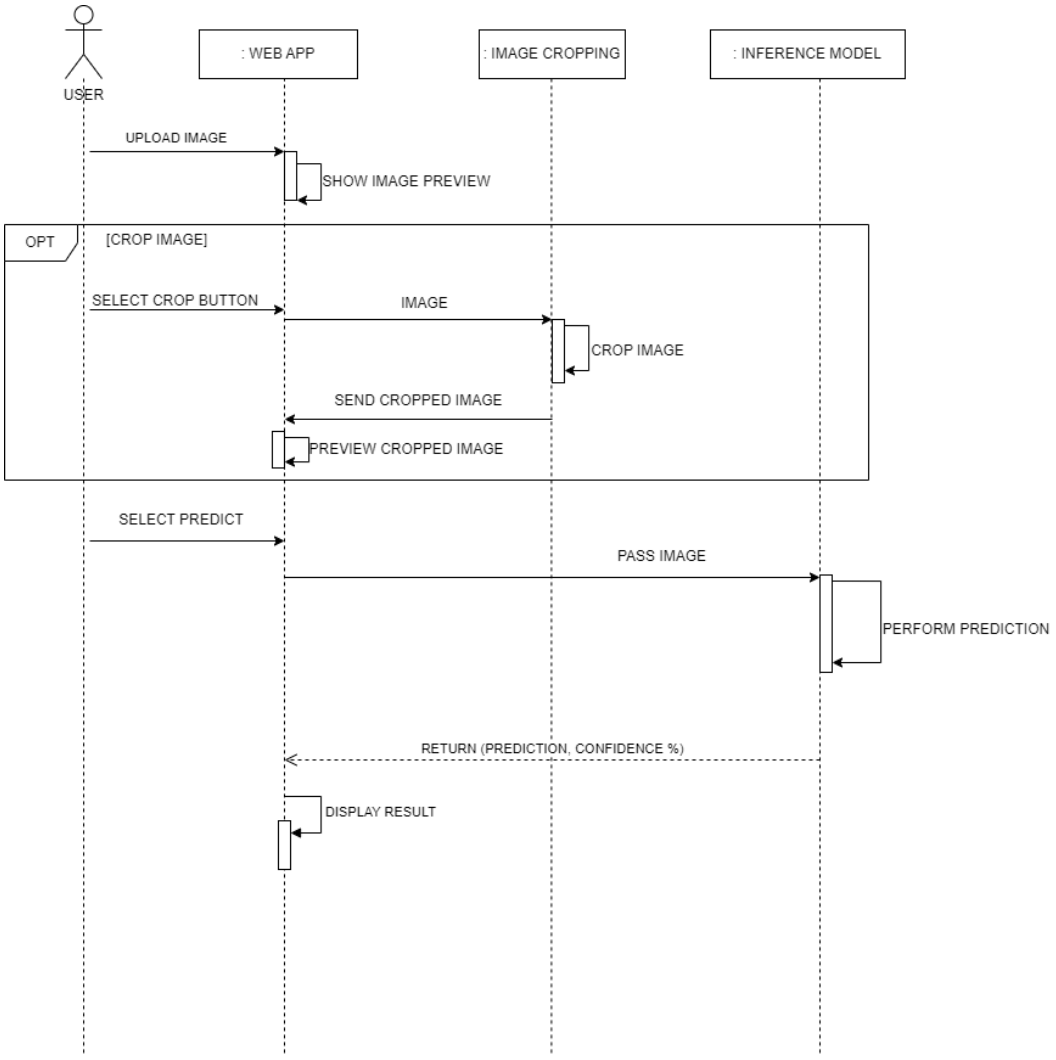


Figure 4.4 Sequence Diagram

CHAPTER 5 METHODOLOGY

This section outlines the process involved in the development of the system, including the related theories that underpin it. The steps in the methodology can be visualized in the figure below.

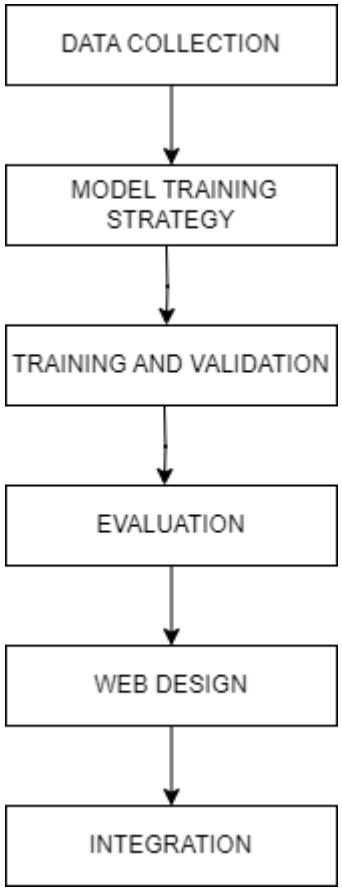


Figure 5.1 System block diagram of the system

Our methodology follows a structured process: data collection from various sources, training and fine-tuning the DenseNet-121 model on the dataset, evaluating its performance, designing a user-friendly web interface, and finally integrating the trained model into the web application. This approach ensures seamless functionality across all stages of the project.

5.1 System Architecture

This section outlines the architecture of the system, covering both the machine learning model training process and the website design.

5.1.1 Data collection

A substantial amount of data was collected from various online sources, including Kaggle, Git repositories, and Google Drive. In total, we gathered 65,000 images—32,500 real images and 32,500 fake images. For effective model training, we adopted an 80-20 split, allocating 80% of the images for training and reserving the remaining 20% for validation and evaluation. To streamline access and simplify code implementation, the images were stored locally on our hard drive, with the entire dataset totaling approximately 65 GB.

5.1.2 Model Training Strategy

Since our model was trained locally on our laptop, we encountered the challenge of managing limited hardware resources. Running a large number of epochs for long durations wasn't feasible due to the risk of damaging the hardware, as extended periods of intensive computation can strain the system. Additionally, if an error occurred after completing a long training session, retraining the model would consume significant time and resources. To address these issues, we made several modifications to our training code to provide more control and flexibility. Specifically, we designed the code to allow us to run a smaller, manageable number of epochs—starting with just one. After each epoch, we would evaluate the model's performance and make necessary adjustments. This approach made it easier to identify and fix errors quickly since we didn't have to wait through long training sessions. If something went wrong, the debugging process was more efficient. Furthermore, instead of starting from scratch each time, we implemented a system where the model was saved after each epoch and updated with the latest progress rather than being overwritten entirely. This ensured that when we reran the training process, we were building upon previous work rather than wasting time retraining from the beginning. This methodology provided us with a greater degree of freedom,

enabling us to code, modify, and debug the model more effectively. The specific techniques and code modifications that enabled this process are explained in further detail below.

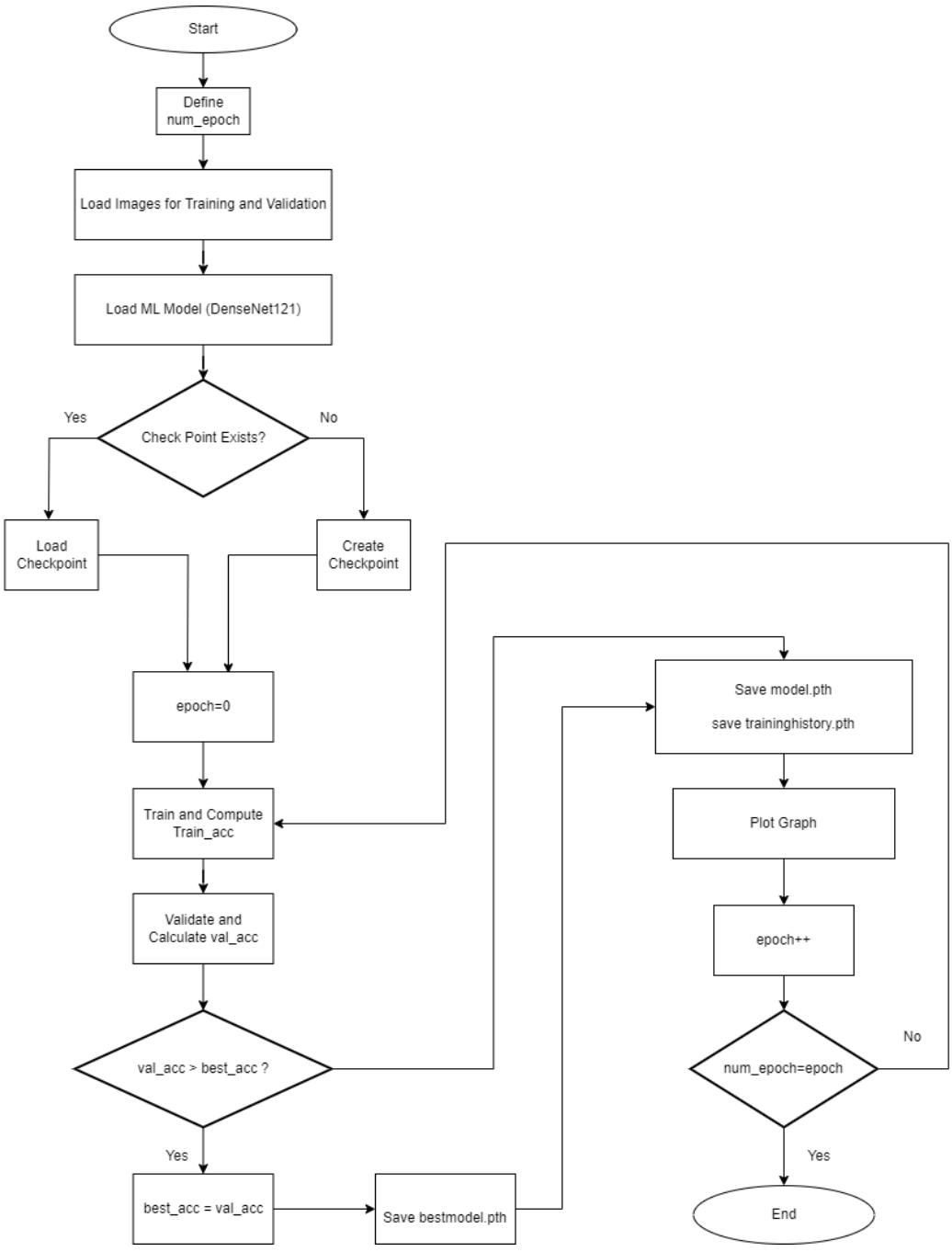


Figure 5.2 Training flowchart

The above figure 5.2 shows the high-level flow of the training ml model. Each of the blocks are explained in brief below.

Specifying number of epochs (num_epoch):

The number of epochs was specified directly in the code, ensuring that the machine learning model trained for only the predetermined number of epochs. This approach helped maintain sufficient intervals between training sessions, preventing our hardware from overheating or sustaining damage.

Loading and Preprocessing images:

The images were downloaded locally and underwent preprocessing to ensure the data was suitable for model training. This preprocessing step was essential to improve the quality and variability of the input data, helping the model generalize better during training. The preprocessing pipeline includes the following steps:

- **Resize**: Images are resized to a standard resolution of 224x224 pixels.
- **Random Horizontal Flip**: A random horizontal flip is applied to introduce variability.
- **Random Rotation**: The images are randomly rotated by up to 10 degrees to simulate different orientations.
- **Random Crop with Padding**: A random crop is applied with 4 pixels of padding, helping the model focus on different parts of the image.
- **Color Jitter**: Random adjustments are made to the brightness, contrast, saturation, and hue to add variety to the dataset's appearance.
- **Random Vertical Flip**: A random vertical flip is applied to further vary the input.
- **Gaussian Blur**: A Gaussian blur is applied with a kernel size of 5 and a random sigma value between 0.1 and 2.0, simulating different blur effects.
- **ToTensor**: The image is converted into a tensor, making it compatible with PyTorch.

- **Normalize**: The images are normalized using the mean and standard deviation values common for pre-trained models like those on ImageNet, ensuring the input data aligns with the expected distribution.

Initializing and Customizing DenseNet-121:

The ML model was initialized with pretrained weights from IMAGENET1K_V1, which had been trained to recognize a wide range of features, including those related to human faces. For our use case, we only needed these pretrained features as a foundation, allowing us to fine-tune the model on our specific dataset.

To maximize our project's benefits, we customized the DenseNet-121 architecture by selectively freezing and unfreezing its layers. Here's a breakdown of the approach:

- **Preserving Pretrained Features:**
  We froze the early layers of the model (all layers except the last five in the convolutional block). This ensures that the robust features learned from the ImageNet dataset remain unchanged during training, providing a strong foundational understanding of general image characteristics.

- **Fine-Tuning for Specifics:**
  The last five layers of the convolutional part were left unfrozen. Unfreezing these layers allows them to adjust and fine-tune their parameters according to the specific patterns and nuances present in our dataset.

- **Customizing the Classifier:**
  Similarly, the classifier layers were also unfrozen. This ensures that the final decision-making part of the model can be tailored to our specific task, improving prediction accuracy for our project.

- **Optimizer and Learning Rate Scheduler:**
  The model uses **Adam optimizer** to update the weights, with a learning rate of 0.001 and weight decay of 1e-4.

- Loss Function: Cross-entropy loss (nn.CrossEntropyLoss()) is used to measure the error between the predicted outputs and the true labels.
- Learning Rate Scheduler: We incorporated **ReduceLROnPlateau** to adjust the learning rate when the validation loss plateaus. This scheduler reduces the learning rate by a factor of 0.1 after 2 epochs without improvement, ensuring the optimizer can converge efficiently and prevent overfitting during training.

Checking training history:

The program first checks for the existence of any previous training history or checkpoints. If found, it loads the saved model and resumes training from the last checkpoint, continuing from where it was paused. If no history or checkpoint is available, a new model is initialized, and training starts from scratch.

Training Loop:

- **Model Training:** The model was trained by loading images in batches of 32, with each batch being processed through the following steps until all images are used.
    - Forward Pass: Each batch of data is passed through the model, where it processes the input (images) using its learned parameters (weights) to generate predictions. This step simulates how the model processes and "thinks" about the data to produce outputs.
    - Loss Calculation: After generating predictions, the model calculates the loss by comparing its predictions to the true labels using a loss function (e.g., cross-entropy for classification). The loss quantifies the error and provides feedback on how much the model needs to adjust its parameters.
    - Backward Pass: During this step, backpropagation computes the gradients of the loss with respect to the model's parameters. This helps the model understand how each parameter (weight) contributed to the error and how to adjust them for future predictions.

- Optimizer Step: The optimizer step is where the model's parameters are updated based on the gradients calculated during the backward pass. The optimizer (such as Adam in our case) adjusts the model's weights to minimize the loss, making small changes to improve predictions. It uses an optimization algorithm to compute the necessary updates. In our code, mixed precision training is employed using GradScaler and autocast, which helps speed up computations and reduce memory usage by using lower-precision (FP16) arithmetic for some operations while maintaining model accuracy. The optimizer step works by zeroing out previous gradients, applying the scaled gradients, and updating the model's parameters to progressively reduce error and improve performance.

- **Validation**: After each epoch of training, the model is evaluated on a separate validation dataset. Validation was done on 64 batches per epoch to speed up the validation process. This step helps assess how well the model generalizes to unseen data. The loss and accuracy are calculated for the validation data to monitor performance and detect overfitting or underfitting. During validation, the model performs a forward pass on the validation dataset without updating its weights, meaning no backpropagation occurs. The loss is computed in the same way as in training, but gradients are not calculated, as the focus is purely on performance evaluation.

- **Model and History Saving**: The current model's state is saved after every epoch. This allows for checkpointing the training process. This ensures that the model can be resumed from the last state saved. The following information is saved as checkpoint during the saving of the model (in our program the model checkpoint is saved in model.pth file) .

  Model Saving: The following components are saved when saving model in model.pth file.

- **Current Epoch:** Records the exact epoch number at which the checkpoint is created, enabling a seamless continuation of training from that point onward.
- **Model Parameters (Weights and Biases):** Captures the learned parameters of the model, ensuring that the precise state of the network is preserved.
- **Optimizer State:** Stores information about the optimizer's internal state (e.g., momentum, running averages), allowing it to resume updates without re-initialization.
- **Loss Value:** Logs the most recent loss, providing a snapshot of the model's performance at checkpoint creation.
- **Learning Rate Scheduler State:** Maintains the configuration of any learning rate adjustments, ensuring consistent learning rate progression upon resuming training.
- **Best Validation Accuracy:** Retains the highest validation accuracy achieved so far, facilitating comparisons and preventing overwriting of the best-performing model.

  Training History: The following components are saved when saving training history saved in history.pth file.

- **Losses and Accuracies (Training & Validation):** A comprehensive log of these metrics across all epochs is saved separately. This historical record is invaluable for:
- **Performance Tracking:** Monitoring improvements or potential overfitting/underfitting over time.
- **Visualization:** Plotting the training and validation accuracy and loss curves to diagnose model behavior and guide further optimization.

- **Best Model Saving (best_model.pth)**: The best-performing model, based on validation accuracy, is saved by comparing the accuracy of the previously saved best model with the accuracy of the model after training

for nth epoch. This ensures that the model with the best generalization capability is preserved for deployment or further use. If for example a model achieves its highest validation accuracy of 99% in the second epoch, but the validation accuracy drops to 98% in the third and fourth epochs, the best model is not updated. Instead, only the current model (as defined in model.py) is updated and saved as a checkpoint. The best model is only updated if the validation accuracy surpasses the previous highest, for example, reaching 99.1% in the fifth epoch. This ensures that only the most optimal version of the model is preserved. The information saved for best model is the same as for the normal model used for checkpoint explained in the above point.

- **Plotting Graphs**: After each epoch, the loss and accuracy metrics are plotted to visualize the model's learning process over time with the help of information saved in history.pth file. This helps in diagnosing any issues (e.g., overfitting or underfitting) and tracking improvements in the model's performance over epoch.

- **Looping decision:** If the number of epochs the model has been trained for is less than the specified number of epochs (num_epochs), the entire training loop is repeated until the number of epochs reaches the specified num_epochs.

5.1.3 Training And Validation

Leveraging the training strategy outlined earlier, we initiated model training on a dataset of 65,000 images over 7 epochs—52,000 images for training and 13,000 for validation. An 80% (real) to 20% (fake) distribution was maintained to ensure robust performance on unseen data. Training was done at 32 batches per epoch while validation was done at 64 batches per epoch. This setup resulted in approximately 1,625 training batches per epoch and about 204 validation batches per epoch. Each epoch took around 15 minutes to complete and was immediately followed by a validation session. Training resumed for subsequent epochs once the

validation results were reviewed. A detailed discussion of these outcomes is provided in Chapter 6: Results and Analysis.

5.1.4 Evaluation

After completing the training process, we proceeded to evaluate the model using a separate evaluation script that also generated a confusion matrix. The model was assessed with a validation set of 13,000 images—6,500 real and 6,500 fake. Since these images were not used during training, they provided an unbiased evaluation of the model's performance. The confusion matrix visually represents the results, and a detailed discussion of the evaluation outcomes is presented in Chapter 6: Results and Analysis.

5.1.5 Web Design

A website was developed to provide an improved interface for interacting with the inference model. Built with React and several supporting libraries, the platform offers a clean and intuitive experience that allows users to upload images, crop them, and view prediction results seamlessly.

The backend for the website was built using Flask, a lightweight and flexible Python web framework that seamlessly interfaces with our machine learning model for detecting fake versus real faces. Flask was an ideal choice for our project due to its simplicity, rapid development capabilities, and ease of integration with Python-based tools and libraries.

Core Components:

- **ImageUploader Component:**
  This is the central component that orchestrates image handling. It integrates various functionalities:
    - **Image Upload:** Utilizing the react-dropzone library, users can drag and drop images directly onto the interface. Additionally, the

component listens for clipboard paste events, allowing for immediate image upload.

- **Preview Display:** Once an image is uploaded, the component generates a preview using the browser's URL.createObjectURL API. This immediate feedback reassures users that their image has been successfully captured.

- **Image Cropping:** A dedicated cropping module (triggered via a modal) allows users to adjust the image before submission. This component is designed to overlay seamlessly on the main interface and provides a user-friendly cropping tool.

- **CustomButton and CropButton:**
  These modular buttons enhance user interaction by clearly indicating the actions available (submitting an image for prediction or opening the crop tool). Their consistent styling and placement contribute to an overall cohesive look and feel.

- **Loader and Animated Feedback:**
  The design incorporates engaging loading animations using libraries such as DotLottieReact. These animations not only serve to entertain but also provide visual feedback during longer operations like model inference.

Component Communication and Data Flow:

React's state management hooks (e.g., useState, useCallback) are used extensively to manage data flow between components:

- **State Variables:**
  Critical pieces of state include the uploaded file, preview URL, cropped image data, prediction outcomes, error messages, and loading indicators.

- **Event Handling:**
  Functions managing drag-and-drop events, clipboard inputs, and cropping

updates are memoized with useCallback to prevent unnecessary re-renders. This enhances overall performance and ensures a fluid user experience.

- **Data Preparation:**
  When the user submits an image (either the original or a cropped version), the file data is packaged using the FormData API. This conversion is done with care, particularly when transforming cropped image data (a base64 data URL) into a Blob and then into a File object, ensuring compatibility with the backend API.

Visual Layout and Structure:

The visual structure is designed to guide the user naturally through the process of uploading, editing, and submitting an image:

- **Centralized Content:**
  The main content is centered on the page with generous margins and padding. This centralization directs the user's focus on the primary task—image upload and preview—while still allowing for supplementary elements like error messages and loading animations.

- **Responsive Design:**
  CSS flexbox and grid techniques ensure that the layout adapts gracefully to various screen sizes, whether on desktops, tablets, or mobile devices. The dynamic resizing of containers and images ensures that the interface remains usable and aesthetically pleasing across devices.

Interactive Elements:

- **Drag-and-Drop Area:**
  A distinct drop zone, styled with dashed borders and clear iconography, visually indicates where users can drop images. The area’s styling (rounded corners, subtle shadows) provides both visual appeal and tactile feedback.

- **Image Preview Area:**
  Once an image is uploaded, it is presented in a preview container. The design ensures that the preview is easily viewable without overwhelming the rest of the interface. The container includes options to remove the image if needed.

- **Cropping Modal:**
  The cropping tool is implemented as an overlay modal. It uses a minimalistic design with clear controls for adjusting the crop region. The modal's appearance is designed to be non-intrusive; it temporarily takes focus while dimming the background to reduce distractions.

Visual Feedback and Animations:

- **Loading Animations:**
  During operations like prediction, a full-screen overlay with a looping animation (implemented with DotLottieReact) informs users that processing is ongoing. The animation is smooth and modern, contributing to a professional look.

- **Error Indicators:**
  Error messages are displayed prominently in red text and are accompanied by subtle animations or icons to draw attention. This immediate feedback helps users quickly identify and rectify issues, such as uploading an invalid file type.

- **Transitions and Effects:**
  CSS transitions are used to animate modal openings, button hover states, and image transformations. These subtle effects enhance the overall user experience without detracting from functionality.

Efficient Image Handling:

- **Lazy Loading:**
  The interface only loads image data when necessary. For instance, image

previews are generated on demand using URL.createObjectURL, which minimizes initial load times.

- **Data Conversion:**
  When a user crops an image, the resulting data URL is promptly converted into a Blob. This conversion is optimized for speed and memory usage, ensuring that the file is ready for upload without unnecessary delays.

- **Optimized Rendering:**
  React's virtual DOM and efficient reconciliation process ensure that only the parts of the UI that have changed are re-rendered. This optimization is critical when handling state updates triggered by image uploads or error messages.

Asynchronous Operations and User Feedback:

- **Non-Blocking API Requests:**
  Image submissions are handled asynchronously using the fetch API with async/await syntax. This design keeps the UI responsive during network operations.

- **Loading Overlays:**
  A full-screen loading overlay provides visual feedback during API calls. This overlay is lightweight and designed to disappear as soon as the prediction results are returned, ensuring that users are not left waiting without visual cues.

- **Debouncing User Actions:**
  Functions handling user interactions—such as rapid clicking of upload or submit buttons—are optimized with debouncing techniques. This prevents multiple rapid submissions and helps maintain a stable, predictable user experience.

5.1.6 Integration

The integration between our website and inference model (best_model.pth) is designed to provide a seamless and responsive user experience:

- **Image Submission from React:**
    - The React component allows users to upload or paste an image, with the option to crop it before submission.
    - Once ready, the image is packaged into a FormData object and sent via a POST request to the Flask server's /predict endpoint.
- **Flask Endpoint Processing:**
    - The Flask server, with CORS enabled via Flask-CORS, accepts the request and reads the image directly from the byte stream without the need to save it to disk.
    - The image is processed using PIL and transformed (resized, normalized) using torchvision before being passed to the DenseNet121 model.
- **Model Inference and Response:**
    - The model performs inference to determine whether the face is "real" or "fake" and calculates the confidence level using softmax.
    - These results are then encapsulated in a JSON response and sent back to the React front end.
- **Displaying Results in React:**
    - Upon receiving the JSON response, the React component updates the user interface to display the prediction result and confidence level in a clear and user-friendly format.
    - Error handling is built into both the front end and back end, ensuring that users are informed of any issues during the image processing or prediction stages.

This tightly integrated setup ensures that the process—from image upload and pre-processing, through model inference, to result display—is both efficient and intuitive for the user.

Figure 5.3 illustrates the high-level workflow of the system, highlighting how user images flow from the React front end to the Flask backend for preprocessing before being fed into the DenseNet-121 model (loaded with the best_model.pth weights). Once the model processes the preprocessed image and generates its prediction and confidence score, these results are passed back through the Flask backend and finally displayed to the user on the front end. This setup ensures that all critical steps—from image input and transformation to final inference—are clearly delineated and efficiently connected.

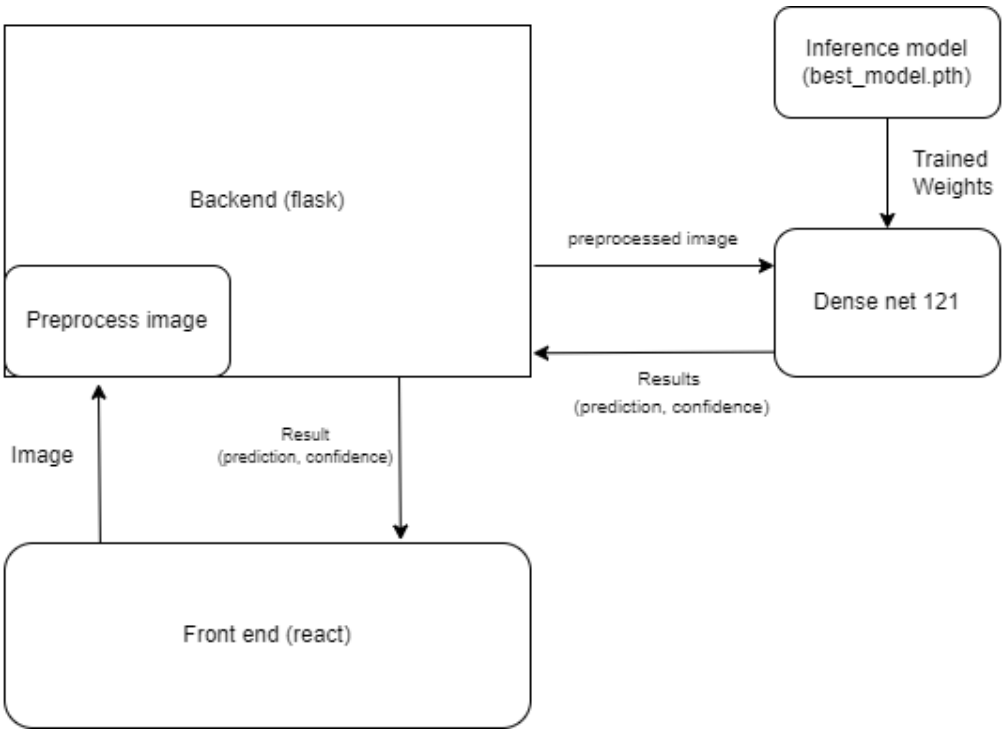


Figure 5.3 Integration Block Diagram

CHAPTER 6 RESULT AND ANALYSIS

This chapter presents the findings of our project by outlining the performance of our AI model alongside the functionality of our website. We examine key performance metrics from the training, validation, and evaluation phases, and provide a clear overview of how the website facilitates user interaction. The analysis highlights both the strengths and limitations of our approach, setting the stage for potential future improvements.

6.1 User Interface

The following results showcase the various animations, components, and modals employed in the UI.

The figure 6.1 below shows the user interface of our website, featuring a clean and minimalist design that ensures an intuitive and easy-to-navigate experience for users.

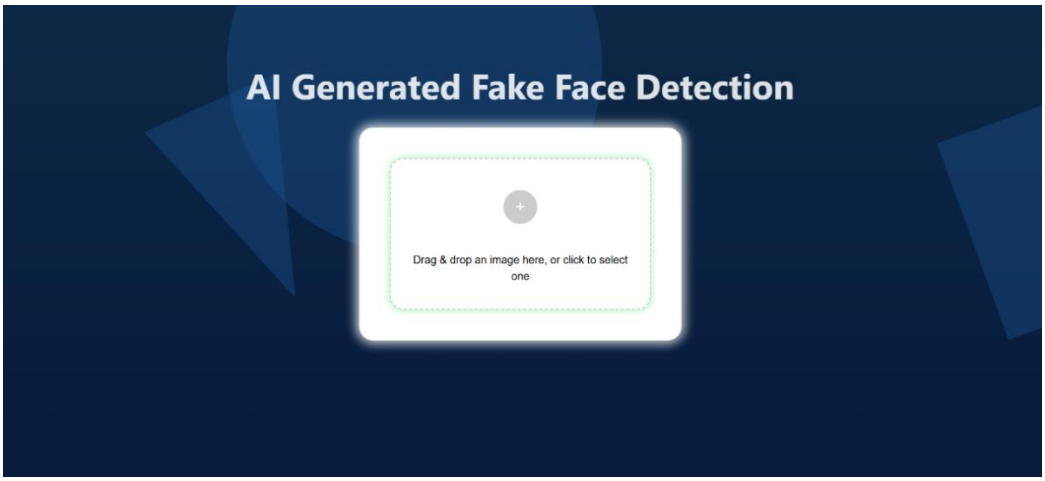


Figure 6.1 User Interface

When an image is uploaded to the website, two buttons Predict and Crop appear, as demonstrated in Figure 6.2. These buttons enable users to either make a prediction or crop the image before proceeding.

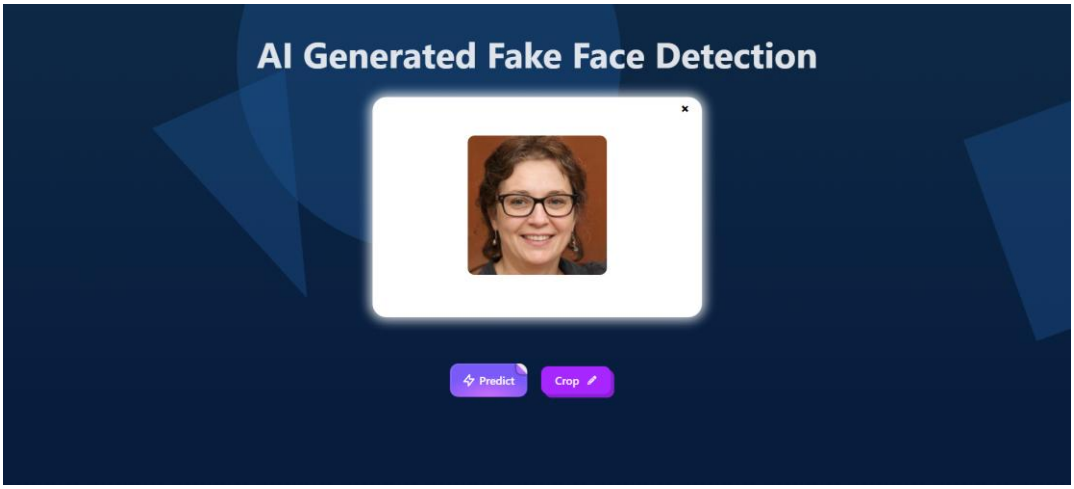


Figure 6.2 UI when image is uploaded

When the user clicks the Crop button, a modal window opens, allowing them to crop the uploaded image. Figure 6.3 demonstrates the result of this functionality.

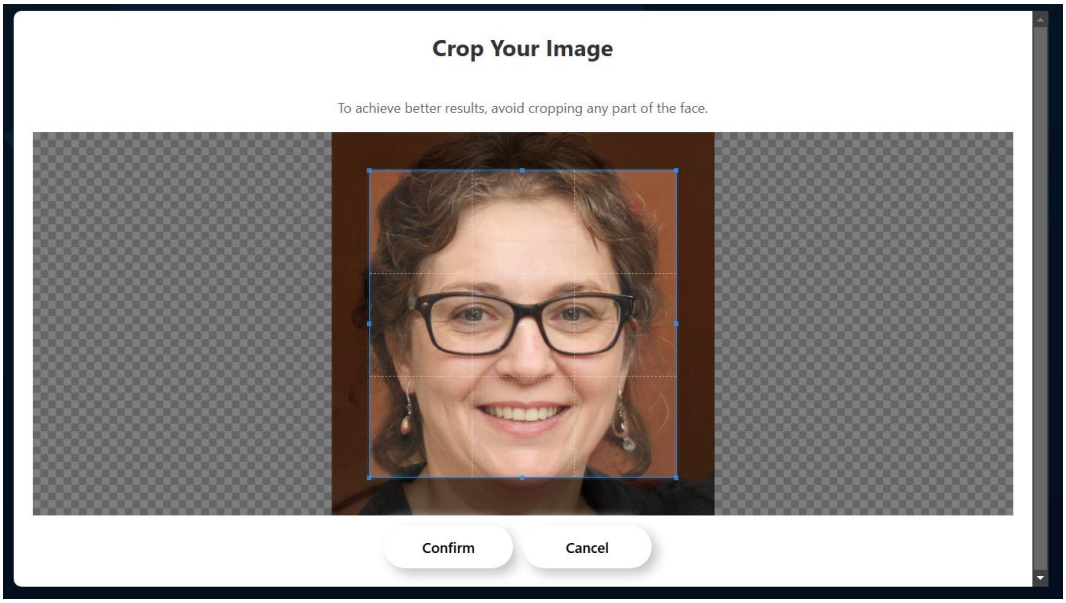


Figure 6.3 Crop UI

During the prediction process, a loading animation featuring a thinking emoji is displayed, as shown in Figure 6.4.

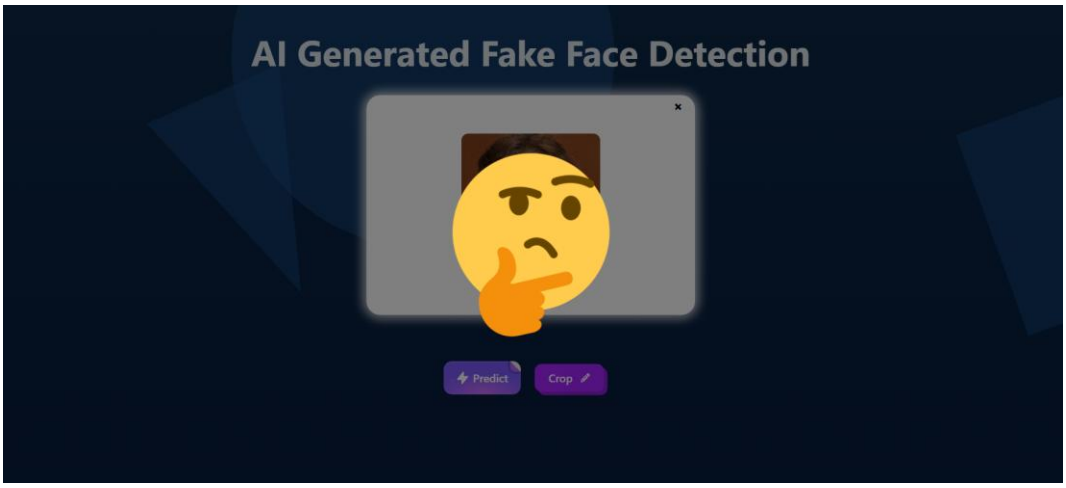


Figure 6.4 Loading Animation

Below is an image Figure 6.5 that illustrates the error handling process in action. The visual clearly demonstrates how the system detects and communicates errors to the user, providing informative feedback to ensure a smooth and reliable experience.

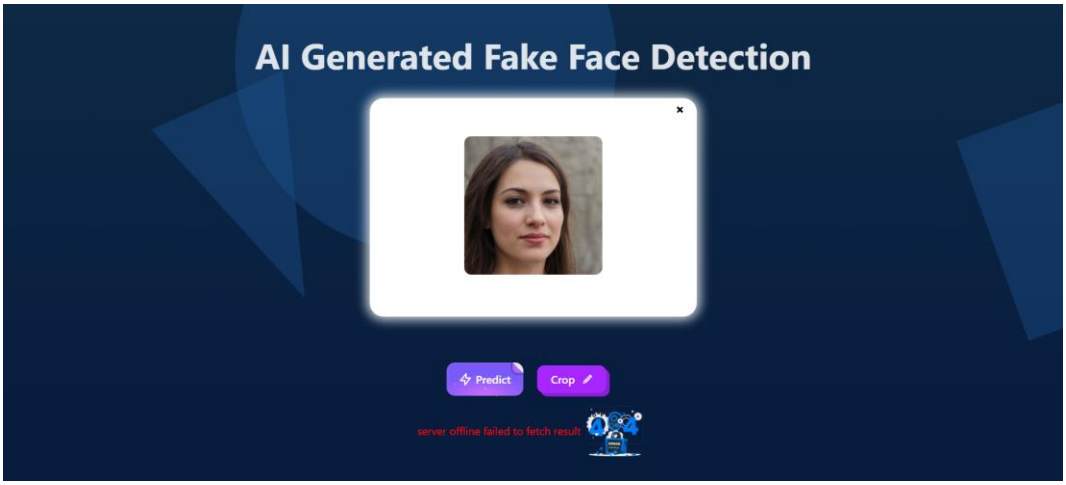


Figure 6.5 Error Handling

The results generated are displayed as follows shown in Figure 6.6

Figure 6.6 Result Displaying UI

6.2 Training and Validation Outcome

The results obtained after training the model for 7 epochs are discussed in this section. Since training and validation were performed consecutively for each epoch, the outcomes for both are presented and analyzed together for clarity and coherence. for a few seconds

In this section, we present the results obtained after training the model for seven epochs. Because each epoch included both a training phase and a subsequent validation phase, the outcomes for both processes are discussed together.

Table 6.1 Performance Summary

	Training		Validation	
Epoch	Loss	Accuracy (%)	Loss	Accuracy (%)
1	0.1443	94.4	0.0344	98.76
2	0.0775	97.23	0.0697	97.55
3	0.0603	97.91	0.0328	98.88
4	0.05	98.21	0.0231	99.21
5	0.0434	98.44	0.0187	99.42
6	0.0364	98.77	0.0222	99.3
7	0.0346	98.798	0.0189	99.361

Table 6.1 provides a comprehensive overview of the model's performance across seven epochs, presenting both the loss and accuracy for training and validation phases. From the data, it is evident that the model improves steadily with each epoch.

In the initial epoch, the training loss is 0.1443 with an accuracy of 94.4%, while validation shows a lower loss of 0.0344 and a high accuracy of 98.76%. As training progresses, the training loss consistently decreases, reaching 0.0346 by epoch 7 with a corresponding accuracy of 98.798%. Similarly, the validation loss follows a declining trend, dropping to 0.0189 by epoch 7, with validation accuracy peaking at 99.42%.

The table highlights a well-performing model, with minimal overfitting indicated by the close alignment between training and validation performance, as both exhibit low losses and high accuracy percentages over time. The decreasing loss values for both training and validation indicate that the model is learning effectively, while the high and increasing accuracy scores suggest robust generalization to unseen data. This table provides valuable insights into the model's training dynamics and validation success over the course of seven epochs.

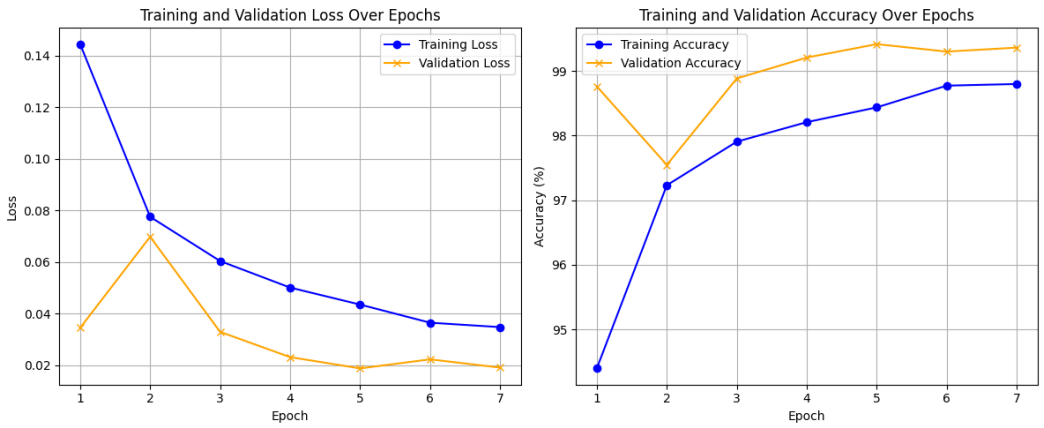


Figure 6.7 Performance graph

As can be seen in Figure 6.1, validation accuracy at epoch one is higher than training accuracy to start with, which is an indication of good generalization to begin with. Validation accuracy decreases during the second epoch, which is concerning.

Validation accuracy gets better by epoch three and peaks at epoch five, where the best model is saved. Although the training went on for a few more epochs, the validation accuracy dropped at epoch six, whereas the training accuracy kept on increasing. In epoch seven, validation accuracy improved slightly but not enough to beat the earlier best model saved at epoch five. We halted the training here to avoid overfitting as training accuracy was starting to get close to validation accuracy. The model that had the best validation accuracy of 99.42% at epoch five was selected for deployment as it also had a training accuracy of 98.44%. The model's higher validation accuracy than training accuracy at this point indicates that it is capable of generalizing well to new data and hence was the optimal model to select.

Since a lower validation accuracy than training accuracy is always a sign of overfitting, it can be observed that during the whole training process, the training accuracy never exceeded the validation accuracy, although the gap was very small at some points. That is, the model never overfitted at any point. So, the model has good generalization ability, and it is suitable to deal with unseen datasets.

6.3 Evaluation Outcome

During the evaluation phase, we tested the model using an unseen dataset consisting of approximately 13,000 images, with 6,500 real and 6,500 fake images. The model demonstrated excellent generalization, achieving an accuracy of 99.42% and loss of 0.0187 in distinguishing between real and fake images. Figure 6.2 displays the confusion matrix generated during the evaluation.

The confusion matrix shown in Figure 6.2 demonstrates a clear assessment of the model's performance. The model correctly classified 6,435 out of 6,500 real images as real, demonstrating its high precision in recognizing actual images. In the case of fake images, 6,489 out of 6,500 were classified as fake by the model, again demonstrating its capability in detecting false images. Despite these successes, however, the model was not entirely without error. The model incorrectly classified 11 fake images as real and 65 real images as fake. These misclassifications highlight

the areas where the model needs improvement, particularly in how it can distinguish subtle differences between real and fake images.

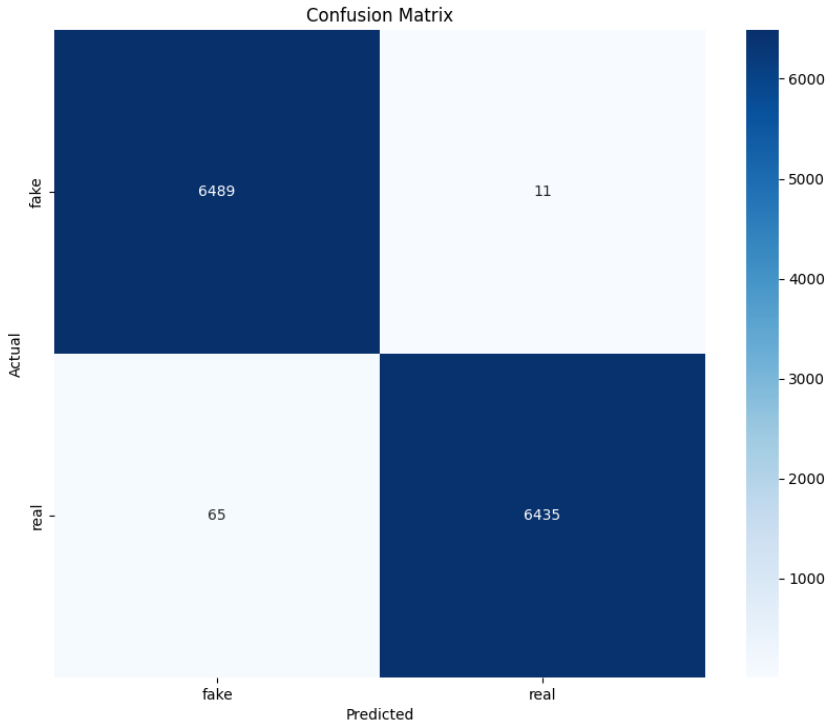


Figure 6.8 Confusion Matrix

The findings and analysis of this chapter reveal the high performance of the model and how well it generalizes to new data. In seven epochs, the model exhibited a steady improvement in both the training and validation phases, with only minor signs of overfitting noticed. Validation accuracy was highest at 99.42% in the fifth epoch, which was discovered to be the optimal model to deploy. A trial on a previously unseen dataset of 13,000 images also corroborated the success of the model, with an impressive accuracy rate of 99.42%. While instances of misclassification were noted, the general performance of the model demonstrates its feasibility for use in differentiating between authentic and counterfeit images.

6.4 Significance of Findings

Our investigation into AI-generated face detection yielded two notable findings that have direct implications for enhancing our model.

First, the model performed significantly better when using images cropped to display only the face. This improvement is attributed to the fact that our training data consisted primarily of cropped facial images. By focusing on the essential features of the face and minimizing background noise, the model was better able to detect subtle cues indicative of AI generation. This finding underscores the importance of dataset consistency and suggests that pre-processing techniques such as cropping can improve detection accuracy in practical applications.

Second, the model sometimes misclassified genuine images that had been digitally altered in Photoshop as fake. The alterations, although minor and often used for aesthetic improvements, introduced artifacts that resembled those created by AI generation. This observation highlights a vulnerability in the model: its sensitivity to small, benign changes can lead to false positives. Moving forward, refining the model to better distinguish between intentional photo edits and AI-generated modifications will be crucial.

Overall, these findings not only validate the effectiveness of our approach when aligned with the training conditions but also reveal areas for improvement. Adjusting the training dataset to include a more diverse range of image types and further refining the model's feature extraction process could enhance its robustness and reliability in real-world scenarios.

CHAPTER 7 CONCLUSION

7.1 Conclusion

Our AI system using DenseNet121 achieved a 99.42% validation accuracy in distinguishing authentic from AI-generated faces. Leveraging transfer learning and fine-tuning, the model effectively addresses synthetic media challenges. A user-friendly web app built with React and Flask enables near-real-time predictions.
The project underscores the need for responsible AI to reduce misinformation and identity forgery. Robust checkpointing and scalability ensure dependable performance in digital media authentication and cybersecurity.

7.2 Future Enhancement

Several potential future enhancements for the AI-generated face detection system include:

Generalization to Broad Image Detection:

Expand the model's scope to classify AI-generated content across diverse image types such as landscapes, objects, and artworks by leveraging advanced architectures like Vision Transformers (ViTs) to capture a broader range of features and improve robustness.

Browser Extensions:

Develop browser extensions that integrate the AI detection system into social media platforms to automatically flag synthetic content as suspicious, thereby alerting users in real time and promoting a safer online environment.

REFERENCES

[1] B. M. G, P. K. Rangarajan, A. r. S, M. Sukesh, A. D. M and J. Y, "Detecting AI-generated images with CNN and," 2024. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10649158&isnumber=10648990.

[2] Z. Guo, "Cartoon Figure Recognition with The Deep Residual Network," 2021. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9543197&isnumber=9543102.

[3] A. Khodabakhsh, R. Ramachandra, K. Raja, P. Wasnik and C. Busch, "Fake Face Detection Methods: Can They Be Generalized?," 2018. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8553251&isnumber=8488998.

[4] S. L. L. v. d. M. K. Q. W. Gao Huang, "CondenseNet: An Efficient DenseNet using Learned Group Convolutions," 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1711.09224.

[5] Z. He, "Deep Learning in Image Classification: A Survey Report," 2020. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9421942&isnumber=9421928.

[6] P. K. R. Ippatapu Venkata Srisurya[1], "Neural Machine Translation using Adam Optimised," 2023. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10084034&isnumber=10083503.

[7] G., Z. Liu, L. van der Maaten and K. Q. Weinberger, "Densely Connected Convolutional Networks," 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8099726/.

[8] "Using Densnet Model for Image Classification with PyTorch," [Online]. Available:

https://www.datatechnotes.com/2024/11/using-densnet-model-for-image.html

[9] D. P. K. a. J. Ba, "Adam: A Method for Stochastic Optimization," December 22, 2014. [Online]. Available: https://arxiv.org/abs/1412.6980#:~:text=We%20introduce%20Adam%2C%20an%20algorithm%20for%20first-order%20gradient-based,functions%2C%20based%20on%20adaptive%20estimates%20of%20lower-order%20moments..

APPENDICES

Appendix A: Code Listings

1) Image of training the first epoch.

2) Image of evaluation process.

3) Code snippet of image augmentation.

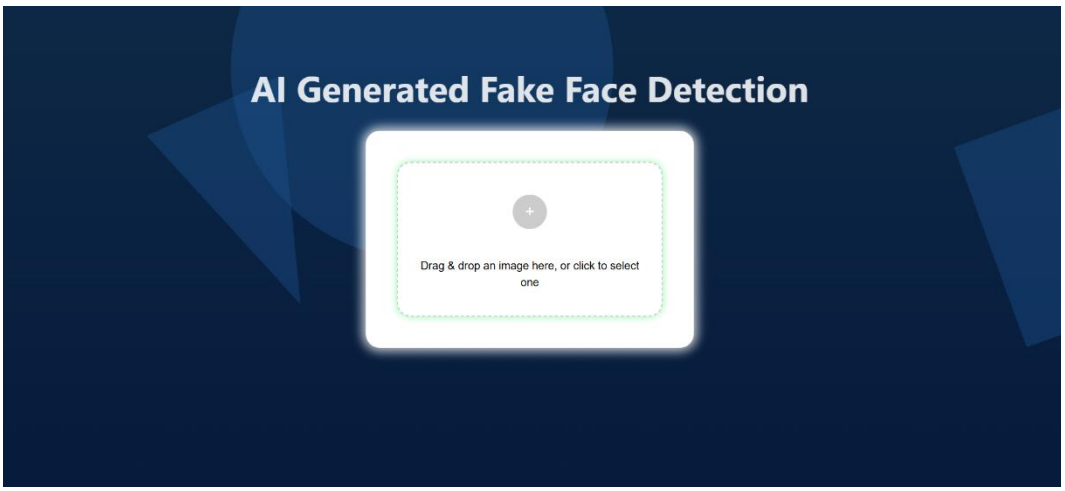
```
data_transforms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),  # Random horizontal flip
    transforms.RandomRotation(10),  # Random rotation
    transforms.RandomCrop(224, padding=4),  # Random cropping with padding
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2),  # Random color changes
    transforms.RandomVerticalFlip(),
    transforms.GaussianBlur(kernel_size=5, sigma=(0.1, 2.0)),  # Random Gaussian blur
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

4) Checkpoint saving code snippet

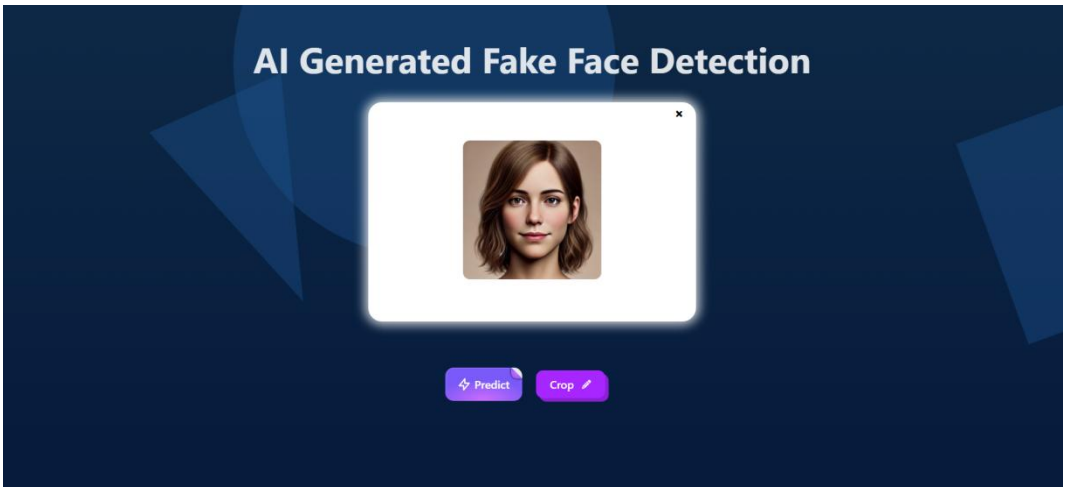
```
# Save checkpoint
torch.save({
    'epoch': epoch + 1,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': epoch_loss,
    'lr_scheduler_state_dict':scheduler.state_dict(),
    'best_accuracy': best_accuracy,
}, checkpoint_path)
```

Appendix B: Additional Visual Results

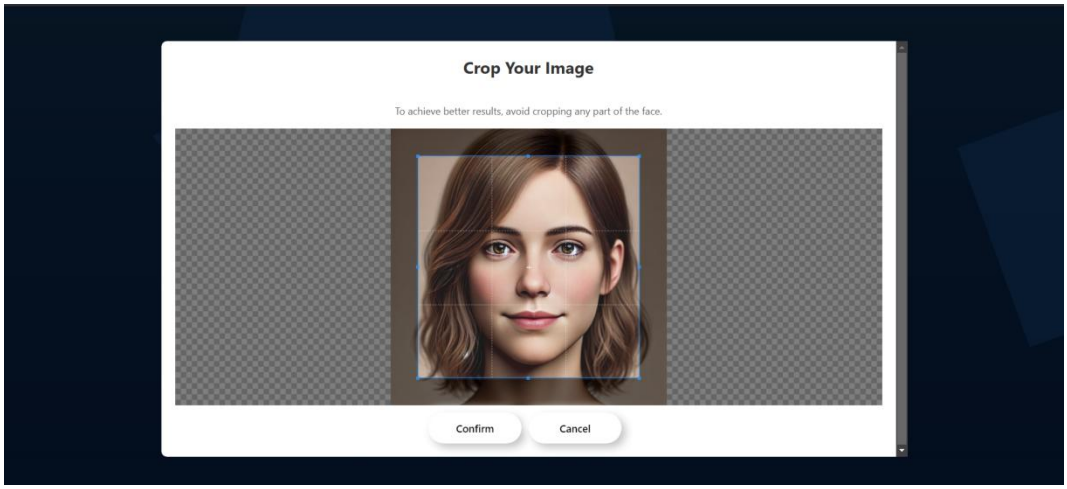
1) Web app UI (no image uploaded).



2) Web app UI (image uploaded).



3) Cropping image UI.



4) Final result.