

Infosys SpringBoard Virtual Internship Program



DermalScan: AI Facial Skin Aging Detection App

Submitted by:
Suganth S S

Under the guidance of Mentor:
Mr. Praveen

Abstract

This project focuses on preparing a facial skin-aging dataset for an AI model that classifies conditions such as dark spots, wrinkles, puffy eyes, and clear skin. The images were organized, preprocessed, and enhanced using augmentation techniques to improve data quality and diversity. These steps provide a strong foundation for effective model training in the next module.

Table of Contents

1. Introduction
 2. Problem Statement
 3. Objectives
 4. Module-wise Implementation
 - Module 1: Dataset Setup & Image Labeling
 - Module 2: Preprocessing & Augmentation
 5. Tools & Technologies
 6. Conclusion
-

1. Introduction

AI DermalScan is a deep learning-based system designed to classify facial aging features from images. The project leverages convolutional neural networks, image preprocessing, and face detection to build an end-to-end solution capable of analyzing facial skin conditions.

2. Problem Statement

3. Objectives

- Build a dataset categorized into facial aging features.
 - Preprocess and augment images for optimal model performance.
 - Train an EfficientNetB0-based classifier for aging sign detection.
 - Implement face detection using Haar Cascades.
 - Develop a user-friendly web interface for image upload and output visualization.
 - Build backend integration for seamless inference.
 - Allow exporting annotated images and logs.
-

4. Module-wise Implementation

Module 1: Dataset Setup & Image Labeling

In this module, the dataset required for the AI DermalScan model was prepared and analyzed. The following tasks were completed:

- Created the main dataset folder with four class categories:
clear skin, dark spots, puffy eyes, wrinkles
- Verified that all images were correctly placed, readable, and properly labeled
- Counted the number of images in each class to understand dataset distribution
- Generated a simple table showing the image count per category
- Plotted a bar chart to visually compare the number of images across classes

These steps ensured that the dataset is structured and ready for preprocessing in Module 2.

Python Code Snippet for Image Count:

```
for cls in classes:
    count = len(os.listdir(os.path.join(dataset_path, cls)))
    data.append([cls, count])
```

Python Code Snippet for Dataset Visualization:

```
df.plot(kind="bar", x="Class", y="Image Count")  
  
plt.title("Dataset Class Distribution")  
  
plt.xlabel("Classes")  
  
plt.ylabel("Number of Images")  
  
plt.show()
```

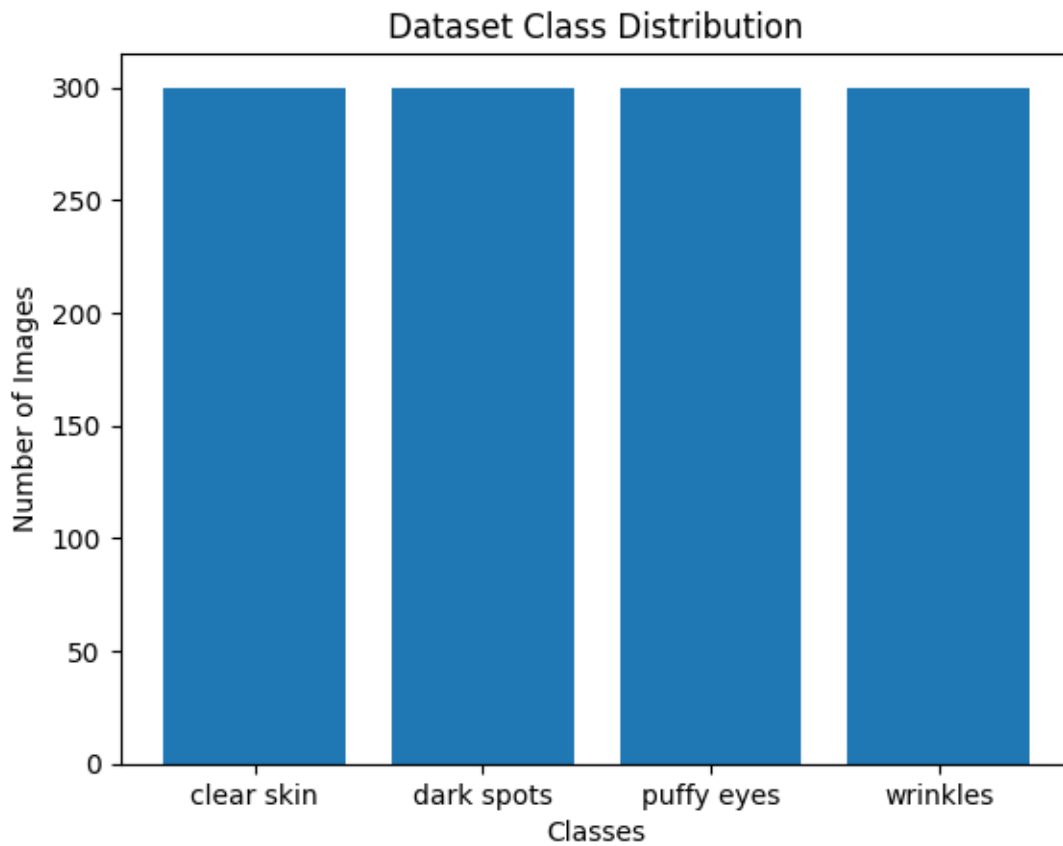


Fig 1 : Class-Wise Image Distribution in the Dataset

This bar chart illustrates the number of images available in each category: clear skin, dark spots, puffy eyes, and wrinkles. The distribution helps determine dataset balance and guides augmentation strategies for the following module.

Module 2: Image Preprocessing & Augmentation

In this module, the prepared dataset was preprocessed and augmented to improve model training quality.

The following tasks were completed:

- Resized all images to 224×224 to match model input requirements
- Normalized pixel values to the 0–1 range for efficient training
- Applied multiple augmentation techniques to enrich dataset variety, including:
 - Rotation
 - Horizontal flipping
 - Zoom transformation
 - Brightness variation
 - Shifting & Shearing
- Generated augmented samples to visually verify preprocessing quality

These steps ensured that the dataset becomes more diverse, reducing overfitting and improving model generalization in the next module.

Python Code Snippet Used for Augmentation:

```
datagen = ImageDataGenerator(  
    rescale=1.0 / 255.0,  
    rotation_range=25,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    brightness_range=(0.6, 1.4)  
)
```

Visualization of Augmentation Results:

This figure shows the original input image alongside four augmented versions generated using rotation, zoom, brightness adjustments, and spatial transformations. These augmentations help increase dataset diversity and improve model generalization during training.

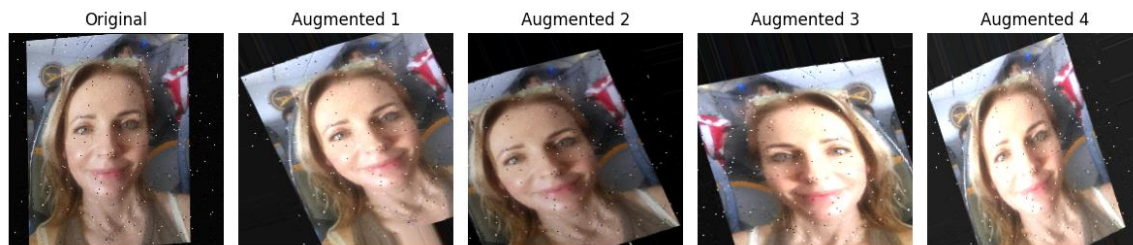


Fig 2: Original Image and Augmented Variants

Module 3: Model Training & Evaluation

In this module, a deep learning model was trained and evaluated to classify facial skin-aging conditions using transfer learning.

The following tasks were completed:

- Split the dataset dynamically into training (70%), validation (20%), and test (10%) sets using a two-stage splitting strategy
- Applied EfficientNet-specific preprocessing to ensure compatibility with pretrained ImageNet weights
- Built a classification model using EfficientNetB0 as the base architecture
- Trained the model in two phases:
 - o Initial training with frozen base layers
 - o Fine-tuning of top layers with a reduced learning rate
- Used optimization callbacks to improve training stability, including:
 - o EarlyStopping
 - o ModelCheckpoint
 - o ReduceLROnPlateau
- Evaluated the model using training, validation, and test accuracy metrics

These steps ensured effective learning, reduced overfitting, and improved model generalization.

Model Performance Comparison Table:

Model	Training Accuracy	Validation Accuracy
EfficientNetB0	90.36%	95.63%
MobileNetV2	94.69%	87.92%

Python Code Snippet Used for Model Training:

```
base = EfficientNetB0(weights="imagenet", include_top=False,
input_shape=(224, 224, 3))

base.trainable = False
```

Python code snippet used for callback during model training:

```
from tensorflow.keras.callbacks import EarlyStopping,
ModelCheckpoint, ReduceLROnPlateau

callbacks = [

    EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True),

    ModelCheckpoint("best_dermalscan_model.h5",
monitor='val_loss', save_best_only=True),

    ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3,
min_lr=1e-6)

]
```

Model Performance Visualization:

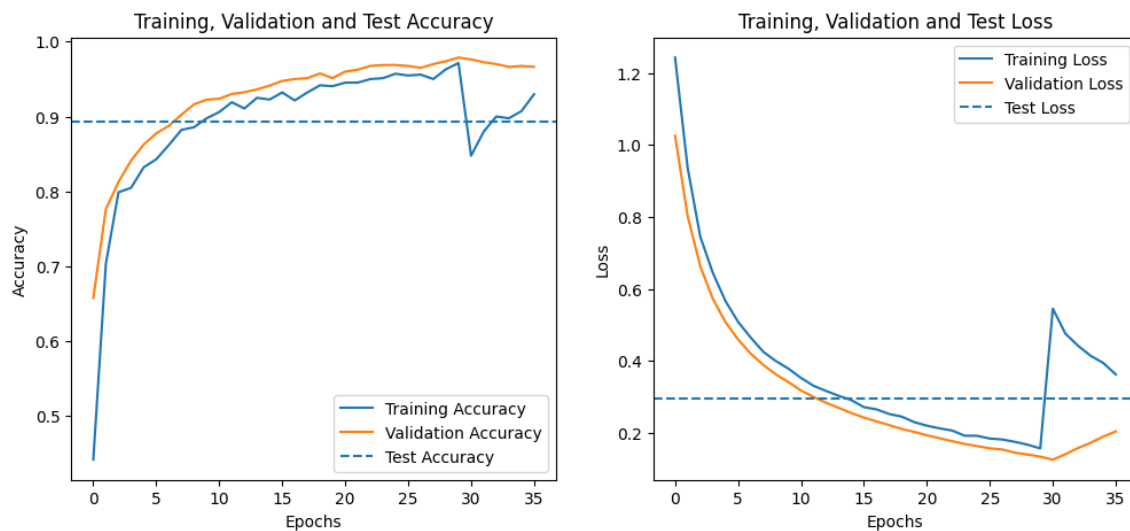


Fig 3: Training, Validation and Test Accuracy & Loss Curves

This figure illustrates the training, validation, and test accuracy and loss curves across epochs. The plots demonstrate stable convergence and improved performance after fine-tuning the model.

Module 4 – Face Detection & Skin Condition Prediction

In this module, face detection and inference pipeline was implemented to evaluate the trained skin-condition classification model on real images. The system detects the face region using OpenCV Haar Cascade and predicts the skin condition using the trained EfficientNet model.

The following tasks were completed:

- Implemented face detection using OpenCV Haar Cascade classifier
 - Loaded trained CNN model (EfficientNet) for inference
 - Cropped detected face region and applied preprocessing for prediction
 - Generated prediction probabilities for all four classes
– *clear skin, dark spots, puffy eyes, wrinkles*
 - Displayed predicted class with confidence score (%)
 - Estimated approximate age based on class type
 - Drew bounding box over face and visually displayed results
 - Rendered output image with class | confidence | age neatly above bounding box
-

Code Snippet Used for Prediction:

```
face = image[y:y+h, x:x+w]

face_resized = cv2.resize(face, (224, 224))

face_array = img_to_array(face_resized)

face_array = np.expand_dims(face_array, axis=0)

pred = model.predict(face_array)

label = class_labels[np.argmax(pred)]

confidence = np.max(pred) * 100
```

Output Visualization:

The output shows detected face with bounding box and prediction result printed above the face.



Fig 4: Face Detection with Skin Condition Classification and Age Estimation

This visual output confirms that the model inference pipeline is functioning correctly and successfully identifies skin conditions from input images.

Module 5 – Web UI for Image Upload and Visualization

In this module, a web-based user interface was developed to enable users to upload facial images and visualize skin-condition prediction results interactively. The focus of this module was to create a responsive frontend that allows seamless image upload, processing, and result visualization without noticeable delay.

The following tasks were completed:

- Developed a responsive frontend using HTML and CSS
- Implemented an image upload interface to accept facial images from users
- Integrated the frontend with the backend inference pipeline
- Displayed annotated output image containing detected faces and bounding boxes
- Visualized predicted skin condition labels along with class probability (%)
- Ensured proper alignment of bounding boxes and text annotations
- Optimized UI rendering to avoid lag during image upload and result display

Code Snippet Used for Visualization Logic:

```

```

```
<table>

  <tr>

    <th>Face</th>

    <th>Class</th>

    <th>Confidence</th>

  </tr>

</table>
```

Output Visualization:

The web interface allows users to upload an image and instantly view the annotated result image along with prediction details. Bounding boxes and confidence values are clearly displayed for each detected face, ensuring easy interpretation of results.

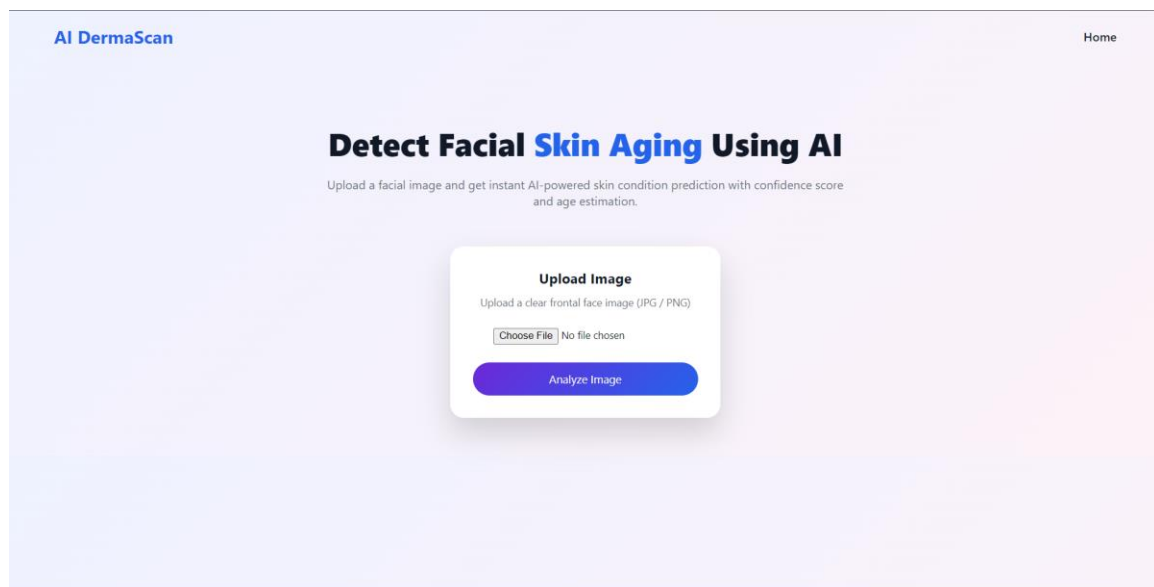


Fig 5.1: Web UI for Image Upload and Skin Condition Visualization

This module confirms the successful implementation of a clean, responsive, and user-friendly interface suitable for real-time demonstrations and evaluation.

Module 6 – Backend Pipeline for Model Inference

In this module, a robust backend inference pipeline was implemented to process uploaded images and generate skin-condition predictions efficiently. The focus was on modularizing

preprocessing and inference logic, ensuring smooth integration with the web-based user interface.

The following tasks were completed:

- Modularized image preprocessing and inference functions for reusability
- Loaded the trained EfficientNet model for real-time prediction
- Integrated face detection and bounding box extraction into the pipeline
- Passed cropped face regions through the model for classification
- Generated class probabilities for each detected face
- Logged prediction details including face ID, class label, confidence score, and bounding box coordinates
- Returned inference results to the frontend for visualization
- Performed end-to-end testing with the web UI

Code Snippet Used for Inference Pipeline:

```
face_array = preprocess_input(face_array)

preds = model.predict(face_array)

label = CLASS_LABELS[np.argmax(preds)]

confidence = np.max(preds) * 100
```

Pipeline Output and Testing:

The backend successfully processes uploaded images and returns structured prediction results to the frontend. Each request follows a seamless input-to-output flow, from image upload to annotated result generation.

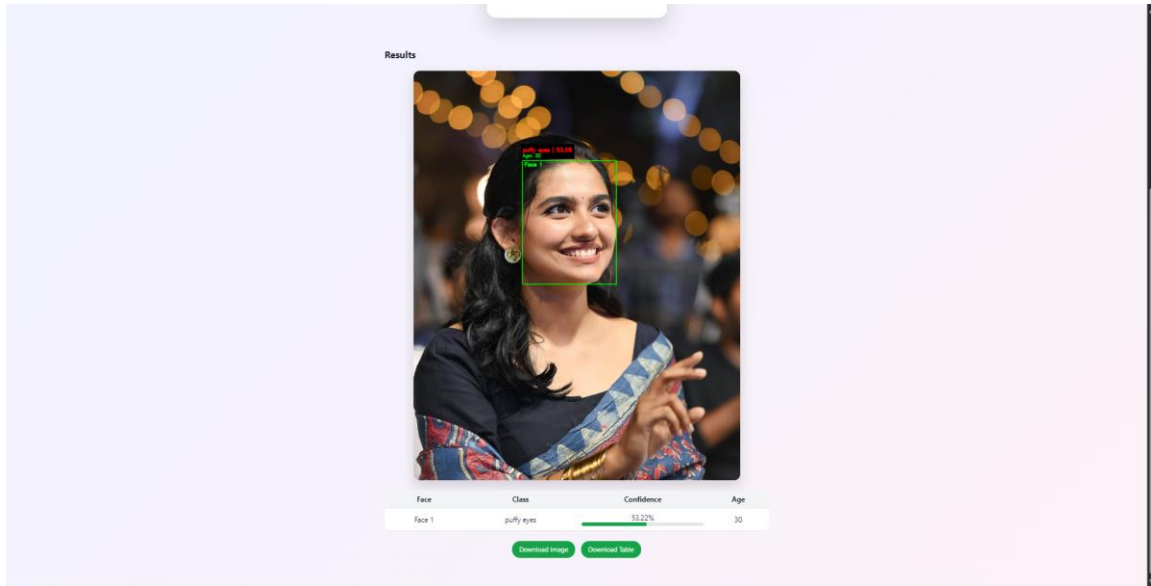


Fig 6: Backend Inference Pipeline Integrated with Web UI

Performance testing confirmed that the system completes inference within the required time limit (≤ 5 seconds per image), ensuring smooth user experience.

5. Tools & Technologies

Programming Languages:

- Python
- HTML
- CSS

Libraries & Frameworks:

- TensorFlow / Keras – deep learning model development and inference
- OpenCV – face detection, bounding box drawing, and image processing
- NumPy – numerical operations and array handling
- Pandas – tabular data handling and result logging
- Matplotlib – visualization of dataset distribution and training metrics
- Pillow (PIL) – image loading and preprocessing
- SciPy – support for image transformations and augmentation
- Flask – backend web framework for model inference and request handling

Development & Deployment Tools:

- Visual Studio Code – backend and frontend development
- Jupyter Notebook – model training, experimentation, and analysis
- GitHub – version control and project repository management

Platforms:

- Local Machine (CPU/GPU-based training and inference)
- Web Browser – frontend UI testing and interaction

6. Conclusion
