Astana IT University

Final Project Report
Intel Image Classification with Deep Learning

Abdikhamidova Aigerim, Sergalieva Aisara
19.11.2024
Applied Machine Learning

Supervisor: Aiymbay Sunggat

Table of Contents

1. Introduction

Project Overview

This project explores the implementation of an image classification task using a deep learning approach. The primary goal is to classify images into six distinct categories: buildings, forests, glaciers, mountains, seas, and streets. The workflow involves detailed preprocessing of the dataset, building and optimizing convolutional neural network (CNN) models, and deploying the final improved model using Streamlit for user interaction.

This project provides an end-to-end pipeline for image classification, from dataset preparation and model training to deployment. Through this process, we aim to enhance the model's accuracy and generalizability to unseen data.

Objective

The objective of this project is twofold:

1. To develop an efficient and high-performing deep learning model for image classification tasks.
2. To deploy the trained model in a user-friendly environment using Streamlit, enabling real-time testing by end users.

The outcomes of this project include an analysis of the model's performance metrics, insights into the challenges of image classification, and the successful deployment of the improved model for public use.

2. Dataset Overview

Source

- **Dataset**: Intel Image Classification Dataset
- **Dataset Size**:
    - Train: 11,230 images across six categories.
    - Validation: 2,804 images across six categories.
    - Test: 3,000 images across six categories.

Characteristics

- The dataset comprises images with a consistent resolution of 150x150 pixels.

- Images are evenly distributed across six classes: buildings, forests, glaciers, mountains, seas, and streets.
- The dataset contains both training and testing folders, ensuring a structured and systematic evaluation.

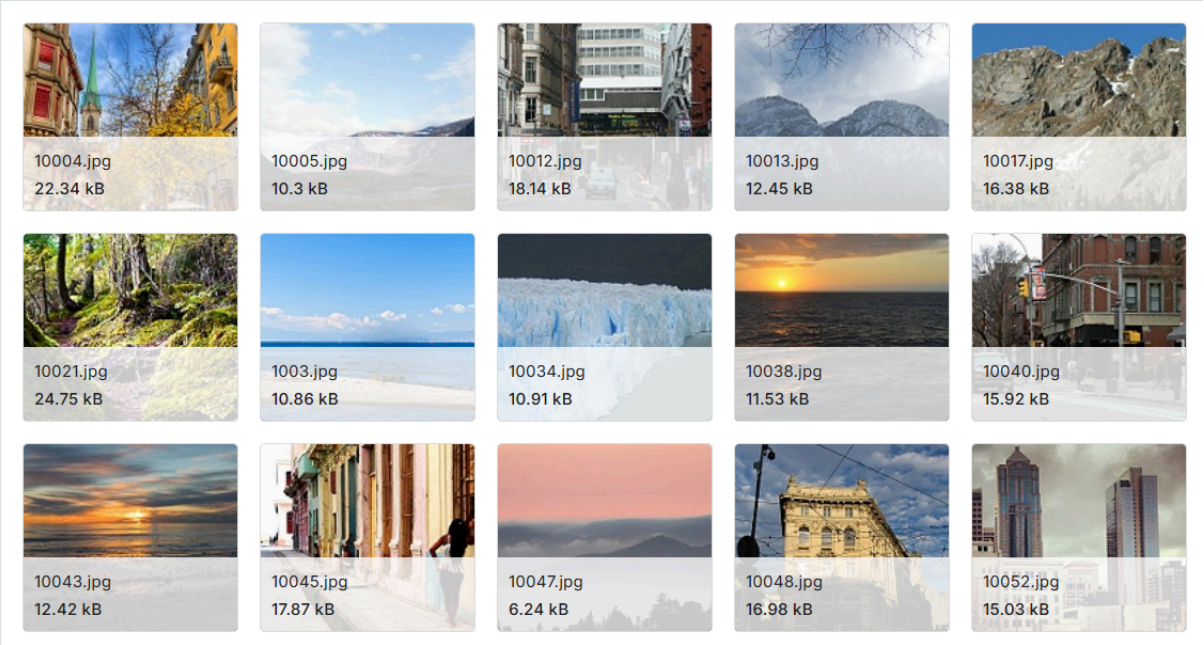
Dataset Preparation

- **Normalization**: Pixel values are normalized to the [0,1] range using a rescaling factor of 1.0/255.
- **Augmentation**: To increase model robustness, the training data was augmented with transformations such as random rotations, horizontal flips, and zoom adjustments. This helped prevent overfitting and improved generalization.

Code Example for Dataset Preparation

```
train_datagen = ImageDataGenerator(rescale=1.0/255, validation_split=0.2)
train_generator = train_datagen.flow_from_directory(
    '/kaggle/input/intel-image-classification/seg_train/seg_train',
    target_size=(150, 150),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)
```

Visual Overview



```
Found 11230 images belonging to 6 classes.
Found 2804 images belonging to 6 classes.
Found 3000 images belonging to 6 classes.
```

3. Data Preprocessing

Key Preprocessing Steps

1. **Normalization**: The raw pixel values of the images were scaled to a range of 0 to 1 to ensure faster convergence during training.
2. **Data Splitting**: The dataset was split into training, validation, and test subsets to evaluate model performance comprehensively.
3. **Augmentation**: Augmentations included random zooming, flipping, and rotation to diversify the training data and reduce overfitting.

The processed data was loaded into the TensorFlow/Keras pipeline using the *ImageDataGenerator* class, enabling real-time data feeding during model training.

4. Model Training

Base Model

The base model was a simple CNN with three convolutional layers and ReLU activations. This model was trained with the following parameters:

- **Optimizer**: Adam (learning rate: 0.001)
- **Loss Function**: Categorical Crossentropy
- **Evaluation Metrics**: Accuracy, Precision, Recall, F1-score, and AUC.

Training Results for Base Model

- **Training Accuracy**: 63.27%
- **Validation Accuracy**: 55.99%
- **Validation AUC**: 91.29%

Training Observations

- The base model showed moderate performance but struggled with complex features in the images. Training and validation loss curves suggested minor overfitting.

Improved Model

To address the shortcomings of the base model, enhancements were made:

1. Added more convolutional layers to capture higher-level features.
2. Integrated batch normalization to stabilize training and improve generalization.
3. Applied dropout to reduce overfitting by randomly deactivating neurons during training.

Training Results for Improved Model

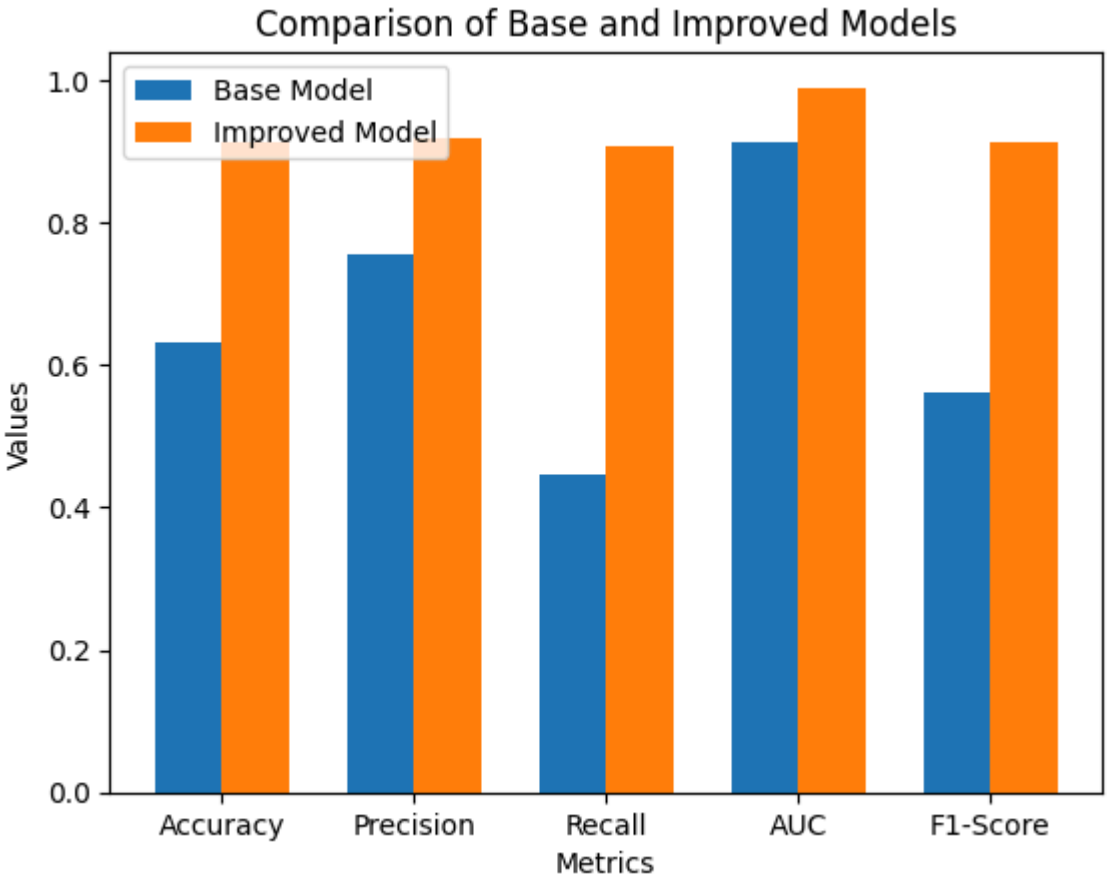
- **Training Accuracy**: 91.30%
- **Validation Accuracy**: 91.77%
- **Test Accuracy**: 91.30%
- **Test F1-Score**: 91.23%
- **Test AUC**: 98.92%

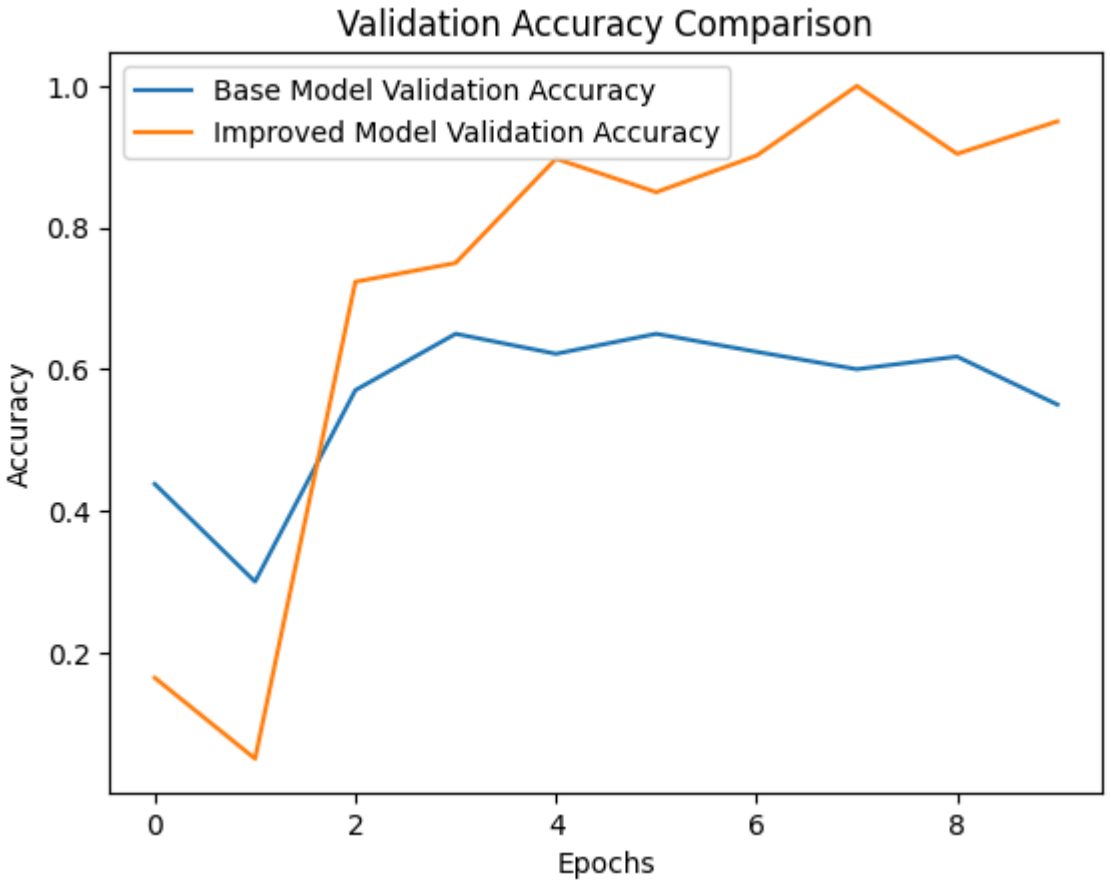
Comparison of Base and Improved Models

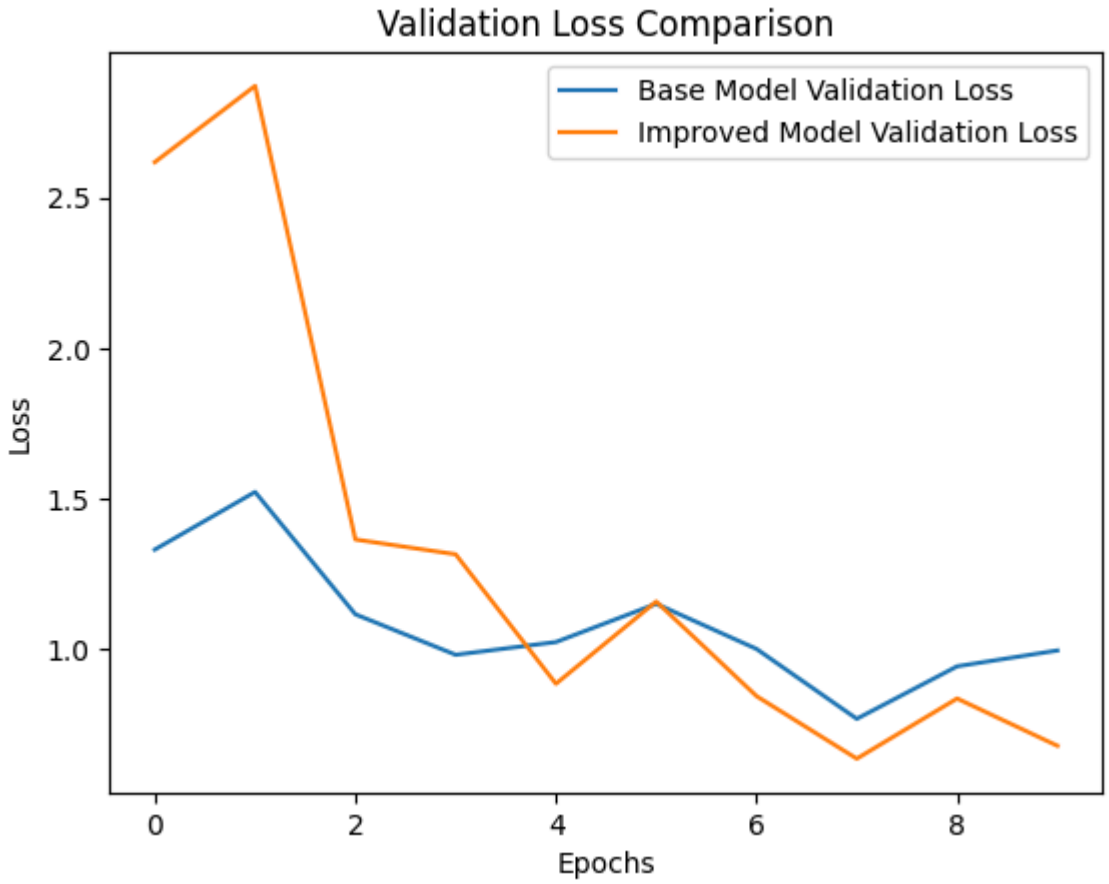
The improved model significantly outperformed the base model in all metrics. Validation accuracy improved by 36%, and AUC scores indicated better discriminatory ability.

Graphs

- **Training and Validation Accuracy**: Plotted for each epoch to visualize the learning process.
- **Training and Validation Loss**: Compared for the base and improved models, showing consistent improvements.


Comparison of Base and Improved Models
Base Model
Improved Model
Values
1.0
0.8
0.6
0.4
0.2
0.0
Accuracy
Precision
Recall
AUC
F1-Score
Metrics


Validation Accuracy Comparison
Base Model Validation Accuracy
Improved Model Validation Accuracy
Accuracy
Epochs
1.0
0.8
0.6
0.4
0.2
0
2
4
6
8



5. Deployment

Streamlit Integration

The trained improved model was deployed using Streamlit to provide an interactive interface for real-time predictions. Users can upload images and receive predictions for the classified categories with corresponding confidence scores.

Deployment Steps

1. **Model Saving**: The improved model was saved as an `.h5` file.
2. **Streamlit Application**:
    - The app displays an interface for image uploads.
    - Predictions are made using the deployed model and shown on the interface.
3. **Hosting**: The app can be hosted locally or on a cloud platform for accessibility.

Deployment Code

import streamlit as st

```
from tensorflow.keras.models import load_model

from PIL import Image


st.title("Image Classifier")

model = load_model('improved_model.h5')

uploaded_image = st.file_uploader("Upload an image for classification", type=["jpg", "png"])

if uploaded_image:

    image = Image.open(uploaded_image)

    st.image(image, caption="Uploaded Image.")

    st.write("Classifying...")

    # Prediction code here
```

Deployment Screenshot

A screenshot was captured of the deployed app, showing predictions for a sample image.

6. GitHub Repository

Repository Details

- **Link**: GitHub Repository
- **Contents**:
    - `preprocessing.py`: Code for data preprocessing.
    - `train_model.py`: Scripts for model training.
    - `app.py`: Streamlit app for model deployment.
    - README with:
        - Project setup instructions.
        - Steps for running the app locally.
        - Dependencies and required libraries.

7. Streamlit Application Deployment

Application Overview

The final improved model has been deployed as a web-based application using **Streamlit**, which enables real-time image classification through a user-friendly interface.

Features

1. **Image Uploading**: Users can upload images in `.jpg` or `.png` format to test the model.
2. **Real-Time Predictions**:
    - Displays the predicted class (e.g., building, forest, etc.).
    - Shows the confidence score for each class.
3. **Interactive User Interface**: The interface is simple, visually appealing, and intuitive, allowing non-technical users to utilize the model.
4. **Performance**: The deployed app runs the improved model, ensuring accurate predictions and fast responses.

Implementation Details

- **Backend**: The model was trained in TensorFlow/Keras and exported as an `.h5` file for inference.
- **Frontend**: Streamlit provides a Python-based interactive GUI, allowing users to upload and classify images seamlessly.
- **Hosting**: The application is hosted on Streamlit Cloud, making it accessible globally without the need for local installation.

Application Flow

1. **Upload Image**: The user uploads an image using the file uploader widget.
2. **Model Prediction**: The app processes the image and runs it through the trained model.
3. **Output Display**:
    - The predicted class is displayed alongside the uploaded image.
    - Confidence scores for all classes are shown, offering transparency.

Application Code Example

import streamlit as st

from tensorflow.keras.models import load_model

from tensorflow.keras.preprocessing.image import img_to_array, load_img

import numpy as np

```
# Load the trained model
model = load_model('improved_model.h5')


# Streamlit UI
st.title("Image Classification App")
uploaded_file = st.file_uploader("Upload an image", type=["jpg", "png"])
if uploaded_file is not None:
    # Load and preprocess image
    image = load_img(uploaded_file, target_size=(150, 150))
    image_array = img_to_array(image) / 255.0
    image_array = np.expand_dims(image_array, axis=0)

    # Make predictions
    predictions = model.predict(image_array)
    predicted_class = np.argmax(predictions)
    confidence_scores = predictions[0]

    # Display results
    st.image(image, caption="Uploaded Image", use_column_width=True)
    st.write(f"**Predicted Class:** {predicted_class}")
    st.write(f"**Confidence Scores:** {confidence_scores}")
```

Screenshots

Screenshots of the app in action:

1. The main interface allows users to upload images.
2. A predicted class (e.g., **"Buildings"**) is displayed after processing.
3. Confidence scores for each class are visualized.

Share

Image Classification App

Upload an image to classify it into one of the following categories:

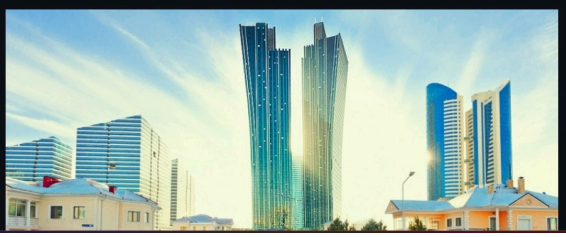
buildings, forest, glacier, mountain, sea, street

Choose an image...

Drag and drop file here
Limit 200MB per file • JPG, PNG, JPEG

Browse files

a7aae7c30f18c3a3881840264ec1b27e.jpg 125.2KB



Manage app

Share

Drag and drop file here
Limit 200MB per file • JPG, PNG, JPEG

Browse files

a7aae7c30f18c3a3881840264ec1b27e.jpg 125.2KB

Uploaded Image

Classifying...

Predicted Class: buildings

Confidence: 99.97%

Manage app

Share

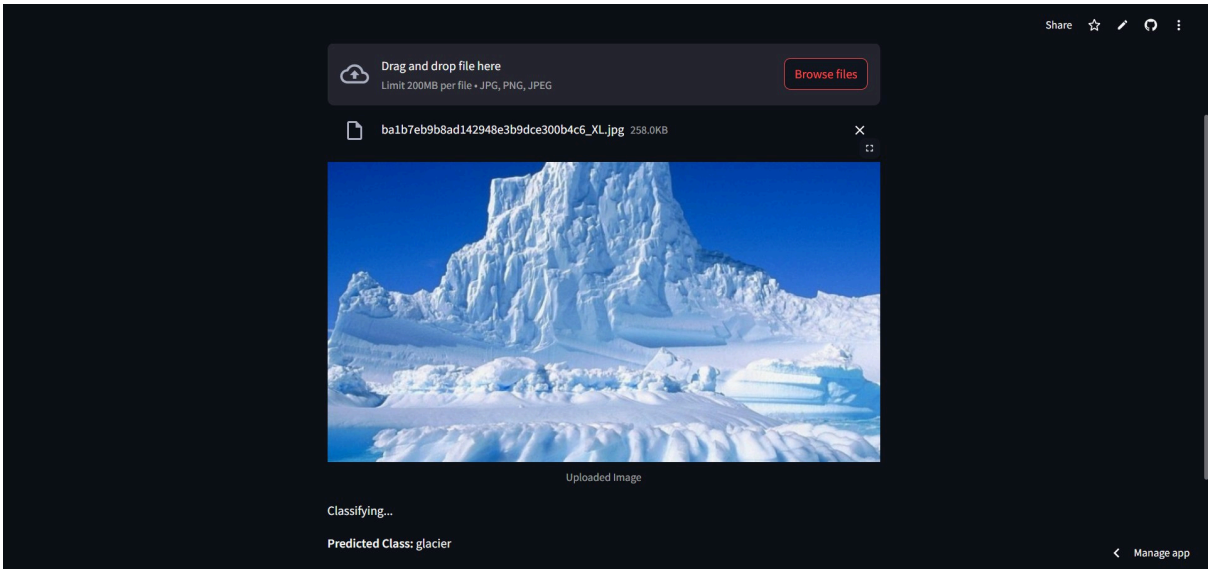
Uploaded Image

Classifying...

Predicted Class: forest

Confidence: 100.00%

Manage app



8. Conclusion

Challenges and Future Improvements

Throughout the project, several challenges were encountered during the implementation and optimization phases. Key issues and their potential solutions are outlined below:

Challenge: Imbalanced Dataset Across Subclasses
While the dataset was balanced across the main classes, some subclasses (e.g., similar images in distinct categories) introduced noise in the learning process.
Solution:Using techniques like SMOTE (Synthetic Minority Over-sampling Technique) or more robust class-specific augmentation strategies could address these issues.

Challenge: Overfitting During Training
The base model demonstrated clear overfitting, where training accuracy significantly outpaced validation accuracy after certain epochs.
Solution: Regularization techniques like Dropout, L2 weight decay, and early stopping were implemented but could be further refined. Incorporating advanced approaches such as dynamic learning rate schedulers and additional data augmentation techniques would help mitigate this.

Challenge: Inconsistent Results in Model Evaluation
The validation F1-Score fluctuated slightly due to the batch-specific evaluation metrics.
Solution: Fine-tuning the batch size, increasing the dataset size, or exploring ensemble methods might stabilize these metrics in future iterations.

Future Iterations

- **Explainable AI Integration**: Providing interpretability for the model's predictions (e.g., via Grad-CAM) could enhance its usability in practical scenarios.
- **Deployment Optimization**: Transitioning from a simple Streamlit interface to a more scalable framework like Flask or FastAPI for production environments.
- **Dataset Expansion**: Incorporating more diverse images and additional real-world test data could improve generalization.

Reflective Section: Insights and Cross-Domain Applications

The completion of this project provided numerous insights and learnings:

1. Understanding Data Pipeline Dynamics
One of the major learnings was the critical role of data preprocessing and augmentation in improving model robustness. Small changes, such as image normalization and augmentation, resulted in significant performance improvements.

2. Transfer Learning Impact
Leveraging pre-trained models demonstrated the importance of using state-of-the-art techniques to save time and resources, especially when working with complex data like image classifications.

3. Teamwork and Workflow Management
Deploying the Streamlit application and hosting it online provided practical insights into workflow coordination, including version control on GitHub and integration between local and cloud environments.

4. Cross-Domain Applications
The knowledge gained in this project can be directly applied to other domains, such as:
- Medical Imaging: Using similar techniques for classifying X-rays or CT scans.
- Autonomous Vehicles: Applying image classification for scene recognition and decision-making.
- E-commerce: Enhancing search engines with image-based product recommendations.

These reflections highlight the broader applicability of machine learning models beyond this specific project and underscore the versatility of the techniques used.