1 INTRODUCTION

1.1 Background

Cataract is one of the leading causes of blindness and visual impairment, estimated to account for more than half of all avoidable cases of blindness (World Health Organization). A cataract develops when the clear crystalline lens of the eye becomes cloudy and obstructs light from properly passing to the retina. This clouding leads to progressive vision loss, and if untreated will result in complete loss of sight. While cataracts are most commonly associated with aging, they can also develop in younger older individuals due to hereditary causes, trauma to the eye, prolonged exposure to ultraviolet light, metabolic disorders like diabetes, or the use of medications like corticosteroids. Standard practice in the diagnosis of cataract is based on a manual assessment by an ophthalmologist using slit-lamp microscopy or fundus imaging.

While these methods provide a high degree of reliability when performed by an experienced provider, they are still restricted by human subjectivity, time, and need for specialized training and equipment. These barriers are particularly problematic in areas where healthcare infrastructure is weak, particularly rural or low resource settings where limited access to ophthalmologists and diagnostic tools result in a high number of individuals suffering from preventable vision loss. The recent developments in artificial intelligence (AI), particularly in computer vision and deep learning highlight great potential in overcoming these diagnostic dilemmas. Deep learning models (specifically Convolutional Neural Networks (CNN)) have demonstrated high levels of accuracy when dealing with image recognition and classification tasks allowing diseases to be detected automatically as a useful module in medical imaging applications for diabetic retinopathy, pneumonia, and skin cancer to name a few. Building on these capabilities, AI systems can now evaluate retinal or fundus images for the presence and severity of cataract disease reliably.

The project titled "AI Powered Vision Transfer Learning for Accurate Cataract Prediction" builds on this technological advancement by exploiting transfer learning for the automation of cataract detection and classification. This involves refining pre-trained convolutional neural network (CNN) architectures such as ResNet, VGG and EfficientNet to classify eye images as normal, immature, or mature cataracts. In addition to the advances in speed and accuracy in diagnostics leveraging AI, using artificial intelligence in medical diagnostics enables scalable, inexpensive solutions to help reach extended underserved areas or populations. The application of these next generation intelligent systems for health and preventive eye care could fundamentally shift how we utilize technology to improve early diagnosis on Cataract diseases and to improve the global burden of blindness.

1.2 Motivation

The motivation for undertaking this project stems from the urgent and continuing need to develop accessible and accurate diagnostic systems for cataract detection. Although cataracts are generally treatable, cataract blindness is still prevalent as a consequence of the delayed (and under-resourced) diagnosis of the disease in developing and remote locations. Millions of individuals suffer vision loss each year not for lack of medical intervention for a curable condition but because their cataracts are not identified and treated prior to the disease advancing too far to be operatively treated effectively. Early detection is critical to enable the surgical intervention at an appropriate time or preventative management, yet the current examination procedures remain costly, require significant effort, and require experts to supervise, and are not widely available to cardiologists.

Artificial intelligence and deep learning provide the possible means to close this healthcare gap. Transfer learning, in particular, enables the application of pre-trained deep learning models that have learned rich image features from large-scale datasets. Fine-tuning these models on cataract images may yield high classification accuracy even when the amount of data is limited. The project AI Powered Vision Transfer Learning for Accurate Cataract Prediction takes advantage of this capability to automate identification of cataract stage, which makes it faster, more reliable, and less dependent on humans.

Furthermore, the motivating factor behind this project is the larger goal of promoting health equity. Many rural communities in the world must travel a long distance to see an eye doctor, meaning that they may not get care at all. This AI-assisted portable diagnostic system could mitigate some of these barriers if designed properly with mobile-form factor devices, mobile applications, or something that allows even remote screenings.

From an academic and technical standpoint, the motivation also resides in showing how transfer learning may be successfully applied to problems in biomedical imaging. By investigating architectures such as EfficientNet, VGG, and ResNet, the project aims to investigate model efficiency, interpretability, and performance under various training configurations. Ultimately, the motivation is both humanitarian and technical- to alleviate the worldwide burden of avoidable blindness with intelligent, interpretable, and scalable deep learning systems to address real-world health care needs.

1.3 Scope of the Project

The AI-Powered Vision Transfer Learning for Accurate Cataract Prediction project aims to create a computer-aided diagnostic system which can accurately detect and classify cataract images using deep learning. The primary scope of the project focuses on the design, implementation, and evaluation of a CNN model that is capable of classifying an input eye image into one of three categories - normal, immature cataract, or mature cataract. This study aims to utilize retinal or fundus images as the input data, which will be pre-processed prior to input into the model. Pre-processing of the data will include resizing, normalization, and augmentation, which will include operations such as rotation, flipping, shifting or zooming the images. The purpose of augmentation is to assist in generalizing the model and for reduced overfitting to the dataset.

The main focus of the project is the use of transfer learning using pre-trained models in the style of ResNet, VGG, and EfficientNet, all of which were fine-tuned on the prepared dataset sourced from visual datasets on open access repositories like Kaggle. In order to assure robustness and reliability, multiple data split ratios of like 70/30, 80/20, and 60/40 were utilized, and the evaluated performance of the model was based on accuracy, precision, recall, F1-score, and AUC for a robust quantitative assessment of performance. Training and validation were implemented using Python libraries TensorFlow, Keras, NumPy, and OpenCV, on Google Colab's NVIDIA Tesla T4 GPU.

The main focus of this project is to classify the severity of cataracts but is limited to three classes and does not include surgical planning, other eye disease detection, or multi-disease prediction. The framework, however, can be built upon in the future to develop more extensive detection of ophthalmic disease, or even real-time teleophthalmology systems.

The focus of the project extends beyond the technical implementation and onto usability and deployment possibilities. The model has been developed to be lightweight and efficient long term, to enable web or mobile integration for point-of-care diagnosis. Potential roles for the model include hospital diagnostics, rural screening programs, and mobile eye camps. The goal of this work is to offer an accessible, interpretable, and inexpensive AI-assisted diagnostic support system to facilitate early diagnosis of cataract, lessen the diagnostic burden of the industry in terms of visual normality or not, and align with global epidemiological initiatives to eliminate preventable blindness.

2. PROJECT DESCRIPTION AND GOALS

2.1 Literature Review

Feng and Xu et al. have proposed a hybrid deep-learning model that augments the ResNet50 architecture with Squeeze-and-Excitation (SE) blocks with a prototype classifier to increase both discriminability and interpretability. In evaluation on various tablets application on a combined dataset (the ODIR-5K, Retina, Kaggle cataract), their hybrid system report very high human metrics about 98.75% accuracy, AUC of about 0.9984, F1 of about 0.9855 and solid external-dataset performance (~93.5%), suggesting that prototype classifier-interpretability and feature-attention mechanisms can yield comparable state-of-the-art accuracy with clinically meaningful utility[1].

Hasan, Tanha, and colleagues explore transfer learning on a handful of off-the-shelf CNNs (DenseNet121, Xception, InceptionV3, and InceptionResNetV2) using the ODIR fundus dataset. They demonstrated that InceptionResNetV2 produces a good level of sensitivity and specificity, achieving the highest reported validation/test accuracy of ~98.17%, after careful preprocessing, multiple training epochs, and comparisons between models, making the point that transfer learning using a reasonably sized labeled dataset, with controlling the data spl, early stopping, and other artifacts of the training process can lead to very good screening level performance [2].

Wu, Hu, et al. addressed the interpretability gap issue with respect to AS-OCT histograms by extracting handcrafted visual features and constructing an ensemble multi-class ridge-regression (EMRR) classifier informed by SHAP and Pearson correlations. Among a large AS-OCT collection (12,824 images), they achieved comparable accuracy (~92.8%) and enhanced interpretability through this approach, while significantly reducing the computational burden compared to deep learning deep neural networks. This study demonstrates the possibility of a favorable trade-off between persistent glass-box feature models and black-box deep nets towards clinical acceptance [3].

Lahari and Vaddi present CSDNet, a 14-layer compact CNN which is designed for use with cataract state detection. This model boasts a low parameter count (~175K trainable params) and rapid inference (~212 ms). CSDNet was trained using the ODIR images (~8,000 augmented images) and achieved >97% accuracy binary classification and ≈98.17% accuracy for multi-class classification. Their research provides evidence that very lightweight, high accuracy models are feasible for deployment on resource-limited edge devices in point-of-care screening [4].

Linde et al. (2021) compare some of the leading CNN architectures (VGG, ResNet, Inception, DenseNet, MobileNet) under the same preprocessing and training regime. DenseNet121 is the declared winner for both binary and multi-class tasks, illustrating the usefulness of dense connectivity (skip connections) in capturing complex features of cataract. The authors highlight the benefit and need for standardized benchmarking for structures when making a decision about which architecture to use for cataracts [5].

Ismail and Alsalamah developed CataractNetDetect, which functions as a unique ensemble stacking framework by merging bilateral (left and right) fundus information from ResNet-50, DenseNet-121, and Inception-V3 models. The authors achieved more than 98% accuracy on the ODIR-5k benchmark with balanced F1/AUC scores and they also demonstrated that bilateral feature fusion enhances model generalization by reducing false negatives compared to single-eye models. These results demonstrate the diagnostic effectiveness of multi-view and paired data for ophthalmic medical image analysis [6].

Mahmood et al. focus on techniques for transfer learning with ResNet50 and class-imbalance mitigation. By combining left and right images together, they assess oversampling fine-tuning and achieve an AUC of 0.966 along with recall and accuracy values around ~96.6% from the ODIR/ STARE-derived set. Their results highlight solid class-imbalance techniques and appropriate fine-tuning as key components of clinically deployable binary screening systems [7].

Jidan and Paul show a thorough comparison of DCNNs and hybrid combinations (VGG19, ResNet50, NASNet, MobileNetV2; hybrids ResNet50+NASNet, ResNet50+MobileNetV2). The authors report MobileNetV2 and ResNet50-MobilenetV2 hybrid achieved first and second top accuracies (~99%) even though augmentation expanded their 2,000→5,000 images, respectively. Their research emphasizes that once systematic data augmentation and transfer learning are performed, lighter-weight architectures and lighter-weight hybrids of DCNNs will achieve equivalent or better accuracy than heavier-weight DCNN models [8].

Julius Olaniyan and Deborah Olaniyan et al. propose a transparent hybrid deep learning framework consisting of a Siamese network and VGG16 to improve both accuracy and interpretability. The framework utilizes Grad-CAM visualizations to localize the most important regions of cataract, as well as clinical explainability. The framework reports perfect classification metrics—100% accuracy, precision, recall and F1-Score—despite small data size (~609 images). The authors demonstrate internal consistency, but still advocate for external validity so as to rule out overfitting. The authors also explain the importance of explainable and interpretable deep-learning models such as similar reasoning and also providing heat-map localization in developing ophthalmologist trust in the AI-assistance process of cataract diagnosis [9].

Saju and Rajesh have proposed the Eye-Vision Net, which has a two-tiered pipeline, that firstly detects cataract using a Deep Optimized Convolutional-Recurrent Network, with an Aquila-inspired optimizer, and then grades severity using Dense CNN and Batch-Equivalence ResNet (BEResNet). They find that the use of slit-lamp images does better than retinal images for the grading task, achieving high overall metrics (≈98.87 % accuracy). The authors also comment that the use of either slit-lamp or retinal images is an important design decision, as is the role of the optimizer and architecture used in the grading task [10].

Lu, Ba, and others offer a comprehensive review that outlines deep-learning applications in cataract care, including diagnosis, workflow evaluation of surgical procedures, intraocular lens power estimation, and use in telemedicine. Emphasizing promising trends in explainable artificial intelligence (AI), federated learning, multimodal data fusion, and surgical assistive models in the context of ongoing challenges of data standardization, data privacy, and clinical validation. Lu's analysis reflects on future directions research will need to progress before this can move from prototypes to clinical uptake [11].

2.2 Research Gaps

1. Limited and single-center datasets

Most studies on cataract prediction to date have utilized small or single-center datasets, which have limited variation in populations, camera types, or lighting conditions. This consequently limits the generalizability of the model to new or unseen images from the real world. For example, studies by Feng et al. [1], Hasan et al. [2], Lahari & Vaddi [4], Ismail & Alsalamah [6] and Mahmood et al. [7] all implement small datasets acquired from single sources. Therefore, there is a need to develop and validate models on diverse and multi-source datasets to improve generalizability and real-world applications.

2. Class imbalance and limited classification categories

Many research articles just broadly label images in two classes of "normal" and then "cataract" without indicating whether a cataract is immature or mature. In a practical setting, the doctor needs to know whether a cataract is immature or mature to aid in future treatment decisions. The studies by Hasan et al. [2], Linde et al. [5], Lahari & Vaddi [4], and Mahmood et al. [7] mainly had a focus on binary classification, which is a limitation for its clinical application. Future investigative work needs to involve multi-class classification (normal, immature cataract, and mature cataract), and utilize better strategies to address class imbalance issues.

3. Lack of external validation and overfitting risk

A large number of models have proven to have very high accuracy on their own training datasets but do not support their performance on new, unseen datasets. Only a handful of studies, for instance, Feng et al. [1], actually tested on external data. If external validation is not done, the model may be overfitting and not functioning well in a real clinical setting. There is a need for

proper cross-dataset testing and validation using different independent data sources to establish the reliability of the model.

4. Unclear performance across different image types

Cataracts can be imaged with different imaging methods (like slit-lamp, fundus, camera from smartphones, etc.), however, most references do not clearly report how models perform with differing image types, nor differing camera qualities. Although Wu et al. (2022) [3], Saju & Rajesh (2021)[10], Lu et al. (2021)[11], all discussed this, future work should determine how models perform on differing image types (especially smartphone images) to ensure that they work in all real-life conditions.

5. Low explainability and clinical interpretability

While some research has investigated visualisation methods like Grad-CAM or SHAP to explain model predictions, not all research has checked whether their explanations even made sense to doctors. Only a few studies looked into this area (Feng et al. [1], Wu et al. [3], and Olaniyan et al. [9]). To build trust and understanding into AI decision-making, the explainability methods should be coherent and tested with clinician feedback.

6. Large and complex models reduce deployment feasibility

A few models with high accuracy initially consist of large model architectures that require large memory and high-powered GPUs for inference, so they cannot be deployed on mobile or edge devices. Lahari & Vaddi [4], Ismail & Alsalamah [6], and Jidan & Paul [8] have provided information about this problem. Therefore, a more lightweight and faster model is needed to provide easy and ready cataract prediction on mobile devices or portable diagnostic devices.

2.3 Objective

The main objective of this project is to create an Artificial Intelligence-based Computer-Aided Diagnostic (CAD) system that can accurately detect cataract conditions from images of the eyes using deep learning and transfer learning techniques. Diagnosing cataracts is usually done by trained ophthalmologists and can be expensive, and may not always be available in rural or resource-poor settings. The diagnosis is also subjective and at times inconsistent. Consequently, this project aims to develop a reliable, automated and scalable computer-aided diagnostic solution which will improve the efficiency, while assisting medical professionals in making more informed decisions.

A key objective of this work is to investigate, compare and tune other pre-trained Convolutional Neural Network (CNN) architectures including EfficientNet, ResNet, VGG to find the model that gains the best accuracy and generalization in terms of cataract classification. Transfer learning is performed to utilize the already learned features of large scale image datasets to minimize training time and improve performance, even if it is a small dataset specific to medicine. The model is trained to classify eye images into two classifications (Normal Eye and Mature Cataract) and a confidence score is provided for increased interpretability and trust.

Another objective is to increase robustness and counter overfitting via preprocessing processes like image resizing, normalization, and data augmentation including rotation, zoom in and out, shifting, and flipping. The model will learn to identify relevant features and visual patterns present in different images of eyes to perform uniformly across different data in the real world.

In addition, this work seeks to ensure that the final system is lightweight, efficient, and capable of inference in real-time, enabling it to potentially be useful in a variety of application contexts including a clinic, telemedicine, mobile screening unit, or community health camp. The goal for the system is to be diagnostically accurate given computational constraints and accessible in contexts with limited resources.

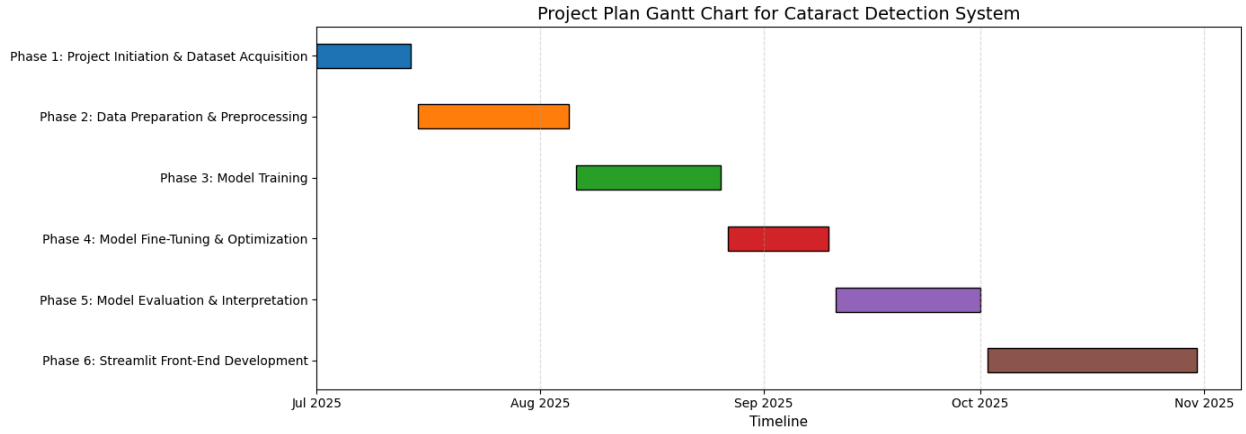
The ultimate goal of this project is not just to automate the detection of cataracts; rather, it is to play a valuable role in improving screening for additional early onset of cataract, reducing the number of preventable blind people and increasing equitable access to eye care. This project supports timely medical intervention, better eye health outcomes for patients, and building a public health capacity for eye care through the integration of artificial intelligence to enhance ophthalmic diagnostic capabilities.

2.4 Problem Statement

Cataracts are one of the main causes of blindness worldwide and affect millions of people annually. A cataract occurs when the eye's natural lens becomes clouded, thereby causing blurred vision and, if not treated, complete blindness. Detecting cataracts early is extremely important because if cataracts are treated properly, vision can be restored and sight lost forever. However, in many parts of the world, particularly in rural or resource-limited areas the access to ophthalmologists and diagnostic tools is very limited. Even where access is made available, manual diagnosis of cataracts can be slow, subjective and extremely dependent on the experience of the specialist. This makes early screening programs a major barrier and results in delays in treatment. Traditional cataract detection requires clinical examination with specialized equipment, trained professionals, and may not be easily available for a large-scale or community-based detection program. Furthermore, in areas with an enormous number of people and very low eye specialists it becomes almost impossible to provide timely diagnosis to all. This situation requires a solution that is accurate, fast and accessible in both hospitals and in low-resource environments. In the past few years, artificial intelligence (AI), computer vision, and deep learning have demonstrated significant potential in automating the analysis of medical images. Convolutional Neural Networks (CNNs) and transfer learning methods can be used for learning from pre-trained models and can achieve high accuracy when used for medical image classification. While most existing studies in cataract detection are focused only on binary classification (normal vs cataract), there is limited work on multi-class grading of cataracts (normal, immature, mature). Multi-class classification is important as it helps in identifying not just the presence of cataracts but also the stage of the disease, which is critical in planning treatment and surgery. Another problem with existing models is that they are often trained on small or single-center datasets. This means that it is hard for them to be more general across different populations, imaging devices and conditions. Some models are also "black boxes" with reduced interpretability, making doctors reluctant to trust the results without appropriate explanations. Additionally, while some lightweight models have been proposed, testing and use in real-world hospital or mobile environments remain quite limited. Therefore, a problem addressed in this project is the need for an automated, accurate, interpretable and lightweight deep learning model for cataract detection and grading. Such a system should be able to classify eye images into three categories--normal, immature cataract, and mature cataract--while being efficient enough for real-time use in clinics and hospitals as well as mobile screening units. By solving these challenges, the proposed model has the potential to enhance early diagnosis, aid ophthalmologists, decrease the burden of manual screening and ultimately decrease the global prevalence of blindness due to cataracts.

2.5 Project Plan

The project was structured into six sequential phases to ensure systematic development, optimization, evaluation, and deployment of the cataract detection system. The timeline spans from July to October, allowing adequate time for dataset preparation, model training, performance refinement, testing, and frontend development.



Phase 1: Project Initiation and Dataset Acquisition

Objective: Establish a clear problem understanding, identify available data sources, and gather the required retinal images to support model development.

Tasks

- Define the problem scope and goals: Clearly outline the aim of detecting cataract severity using AI and determine expected functionalities such as classification accuracy, confidence scoring, and system responsiveness for real-world usage.
- Conduct literature and domain study: Review existing research papers and medical sources on cataract progression, manual diagnosis challenges, and use of CNNs in ophthalmic imaging to understand gaps and justify the need for automation.
- Collect datasets containing Normal, Immature Cataract, and Mature Cataract images: Gather publicly accessible datasets from platforms such as Kaggle or ophthalmology research repositories while ensuring diversity in image clarity, lighting, and patient demographics.
- Verify dataset legitimacy and quality: Check image label correctness, remove duplicates or low-resolution images, and confirm that dataset usage complies with ethical and open-source data guidelines.

Duration: 2 Weeks (July Week 1 – Week 2)

Phase 2: Data Preparation and Preprocessing

Objective: Format and enhance the dataset to ensure consistency and improve the model's ability to learn distinguishing cataract features effectively.

Tasks

- Standardize image resolution and formats: Convert all images into a common resolution and image format so that the neural network receives uniform input for smoother and reliable training.
- Normalize image pixels for stable computation: Apply normalization techniques to bring pixel values to a consistent range, improving training speed and reducing issues like gradient instability.
- Perform data augmentation: Introduce transformations such as rotation, flipping, zooming, shifting, and brightness variation to simulate real-world clinical variability and reduce the model's risk of overfitting.
- Split the dataset into Training, Validation, and Test sets: Experiment with multiple splitting ratios (70/15/15, 80/20, 60/40) to compare consistency and assess how data distribution impacts model generalization.

Duration: 3 Weeks (July Week 3 – August Week 1)

Phase 3: Model Training

Objective: Train the deep learning model using transfer learning to extract relevant cataract features and classify images effectively.

Tasks

- Select pretrained CNN models such as ResNet, VGG, and EfficientNet: Use state-of-the-art architectures that already learned general visual features, allowing faster and more accurate training with limited data.
- Modify the classification layer to support three classes: Replace the final dense layers with a softmax-based classifier that outputs probabilities for Normal, Immature Cataract, and Mature Cataract categories.
- Train the model using GPU acceleration in Google Colab: Utilize NVIDIA Tesla T4 GPU for faster execution, iterative experimentation, and real-time monitoring of loss and accuracy performance curves.
- Observe training stability and validation alignment: Continuously monitor metrics to identify early overfitting or underfitting and make intermediate adjustments if necessary.

Duration: 3 Weeks (August Week 2 – August Week 4)

Phase 4: Model Fine-Tuning and Performance Optimization

Objective: Improve model efficiency and strengthen its ability to generalize across varied image inputs while minimizing misclassification.

Tasks

- Adjust hyperparameters such as batch size, learning rate, and epochs: Perform targeted experiments to identify the optimal training configuration that balances accuracy and computational efficiency.
- Apply dropout layers and batch normalization: Introduce regularization techniques to prevent overfitting and ensure stable convergence especially in deeper layers of the network.
- Unfreeze selected layers for fine-tuning: Gradually allow certain pretrained layers to retrain on cataract-specific features, improving sensitivity to subtle variations in lens opacity.
- Use early stopping and learning rate scheduling: Halt training when improvement stagnates and adapt learning pace to avoid oscillation or premature convergence.

Duration: 2 Weeks (September Week 1 – Week 2)

Phase 5: Model Evaluation and Interpretation

Objective: Validate the trained model's performance and ensure that prediction decisions are explainable and clinically meaningful.

Tasks

- Evaluate performance using accuracy, precision, recall, F1-score, and AUC: Measure how well the model distinguishes among the three cataract stages and confirm balanced predictive behavior.
- Analyze confusion matrix outputs: Identify which classes are frequently misclassified and understand whether errors occur between similar severity levels (e.g., Immature vs. Mature cataracts).
- Apply Grad-CAM visualization for interpretability: Generate heatmaps highlighting the key image regions influencing predictions, supporting clinical confidence and transparency.
- Confirm readiness for real-world usage: Ensure the model consistently performs well across test cases and does not exhibit dataset-specific bias.

Duration: 2 Weeks (September Week 3 – Week 4 and October Week 1)

Phase 6: Model Deployment and Front-End Development

Objective: Develop an accessible interface enabling non-technical users to upload eye images and obtain model predictions in real time.

Tasks

- Build a Streamlit Web Application: Create a clean, interactive UI where users can upload images and receive classification results along with confidence percentages.
- Integrate the trained model with the UI: Ensure that the backend model processes input instantly and returns results in a timely and user-friendly format.
- Conduct usability and responsiveness testing: Verify that the system works smoothly across different devices and network conditions to support practical screening use.
- Prepare documentation and finalize project presentation: Summarize system workflow, performance, and deployment capabilities for final submission and demonstration.

Duration: 2 Weeks (October Week 2 – Week 4)

3.TECHNICAL SPECIFICATION

3.1 Requirements

For the cataract-classification project, the following requirements must be outlined, focusing on both functional and non-functional aspects. These requirements ensure the system is robust, effective, and usable for clinical screeners and supervising clinicians. The project utilizes transfer learning with CNN backbones (VGG19, ResNet50, DenseNet121, EfficientNet-B3) and an optional ensemble to classify fundus images into Immature, Mature, or Normal cataract, supporting confident decisions and reproducible audits.

3.1.1 Functional Requirements

The system ingests single or batched eye images, validates file integrity and format, converts all inputs to RGB if required, and standardizes them to 224×224 with architecture-specific normalization before inference. Each image is classified into Immature, Mature, or Normal, and the system returns the predicted label with a calibrated confidence score while preserving original filenames to maintain traceability from raw inputs to results and reports. Users may select a specific backbone or invoke an ensemble mode that aggregates per-model probabilities (average or weighted by validation accuracy) to improve robustness; the application records predictions, confusion matrices, and full evaluation summaries to persistent storage so that results can be audited, reproduced, or compared across runs without re-training.

1. **Data input and handling**
   The system accepts single images and batches via file upload and optionally fetches studies from PACS/LIS in read-only mode. Every file is validated for type, decodability, dimensions, and color mode; images are converted to RGB if needed and standardized to 224×224 before backbone-specific normalization. Original filenames and a generated request ID are preserved to maintain traceability from raw input to predictions and reports.
2. **Data upload integration and processing**
   The interface exposes a simple "Upload/Fetch → Review → Classify" flow and supports drag-and-drop as well as folder selection for batches. On ingest, the pipeline computes basic metadata (width, height, channels, hash) and stores it alongside the request; failures return actionable messages indicating corrupt files or unsupported formats.
3. **Model development and training**
   The training suite supports VGG19, ResNet50, DenseNet121, and EfficientNet-B3 with include_top disabled and a unified three-class head. Each model trains in two stages—frozen-base head training followed by selective unfreezing for fine-tuning—and persists the best checkpoint and label mapping for deployment.

4. **Model scheduling and training**
   Experiments are defined by a configuration file specifying backbone, data paths, epochs, batch size, augmentation, and callbacks. A runner executes jobs sequentially, logging metrics and saving artifacts to persistent storage so runs can be resumed or compared without manual intervention.
5. **Hyperparameters and tuning**
   Core hyperparameters (learning rate, weight decay if used, dropout rate, augmentation strength, unfreeze depth) are configurable per backbone. The system supports simple grid choices or small sweeps, with best-model selection performed by validation loss and secondary tie-breakers such as accuracy.
6. **Cross-validation and testing**
   The default evaluation uses a held-out test split; optionally, K-fold cross-validation can be enabled to estimate variance and robustness. Each fold/savepoint writes fold-specific metrics and confusion matrices to ensure repeatable comparisons across models.
7. **Model testing and validation**
   For each trained checkpoint, the service runs validation and test inference with deterministic preprocessing and no augmentation, computing class-wise precision/recall/F1 and overall accuracy. Misclassified examples can be exported with predicted vs true labels to aid error analysis.
8. **Prediction and testing**
   At inference, users may select a backbone or enable ensemble mode that fuses per-model probabilities via average or validation-weighted average. The system returns final label and confidence, alongside per-model probability vectors in a fixed class order.
9. **Error reporting**
   User-visible errors are concise and specific (e.g., “unsupported format,” “decode failed,” “model artifact missing”); low-confidence predictions below a configurable threshold trigger a “second opinion” recommendation instead of auto-finalization.
10. **Data visualization and insights**
    The UI provides confusion matrices and training curves for selected runs, plus class distribution summaries for the dataset. Per-image result pages display the probability bar chart across the three classes and note if the threshold rule was invoked.
11. **Visualization of prediction performance and metric display**
    Across runs, the system plots accuracy/loss histories and per-class metrics; for ensembles, it compares single-model vs fused performance to justify deployment choices.
12. **Output and reporting**
    For every request, the service returns a structured JSON (and UI view) with filename, predicted class, final confidence, per-model confidences, model/version used, and timestamp. Batch jobs produce downloadable CSV/NDJSON summaries and optional PDF reports with confusion matrices and key metrics.

13. **Comprehensive report for feedback and integration**
    The evaluation module can export a consolidated report per experiment containing configuration, metrics, curves, confusion matrices, and artifact locations, enabling reviewers to replicate or audit the run.
14. **Integration methods and inputs**
    A minimal REST API accepts multipart uploads or references to storage URIs and returns structured results; for PACS/LIS, read-only fetch provides images to the same pipeline without modifying source systems.
15. **Scalability and reliability**
    The service caches loaded models to avoid cold starts, queues batch jobs, and writes all artifacts to persistent storage so interrupted sessions can resume. It supports CPU-only serving for demos and GPU-backed serving for deeper backbones or higher concurrency.
16. **Establishment for new models**
    New backbones can be added by implementing a standard adapter (preprocess_input, input size, and backbone constructor) and registering the model in the configuration so the common head, training schedule, and evaluation flow work unchanged.

3.1.2 Non-Functional Requirements

The solution targets high diagnostic accuracy on the held-out test split using a fully reproducible pipeline that fixes directory structure, random seeds, preprocessing, and model checkpointing to ensure that trained weights always reproduce the same inference behavior. Responsiveness at inference is achieved by binding the backbone choice to hardware constraints: MobileNetV3 and similarly compact models serve quickly on CPU-only instances, while deeper backbones such as VGG19, ResNet50, DenseNet121, EfficientNet-B3, and InceptionResNetV2 benefit from GPU-backed serving under concurrent load or strict latency budgets. The workflow is robust to transient failures because artifacts are written to persistent storage during and after training; the user interface remains intentionally minimal—upload, classify, review—to avoid operational friction in screening contexts while still exposing confidences and downloadable artifacts for detailed review.

Performance and efficiency

- Speed and latency: The service shall return single-image predictions with P95 latency appropriate to hardware selection; lightweight backbones target CPU P95 within an acceptable window, while deeper backbones target low P95 latency on GPU under moderate concurrency. Models are cached in memory to avoid cold starts, and preprocessing is vectorized to minimize overhead.
- Scalability: The system shall scale up by enabling GPU instances for heavy backbones or increased throughput, and scale out by queuing batch jobs and processing them sequentially without user blocking. Artifacts and logs are written to persistent storage so horizontal scaling does not risk data loss.

- Efficiency in model training: Training uses two-stage schedules with early stopping and learning-rate reductions to avoid wasted epochs; checkpoints capture the best validation loss to prevent overfitting regressions. Augmentations are bounded to clinically reasonable ranges to add variance without inflating epoch time unnecessarily.

Reliability and availability

- System reliability: Every request is assigned a unique ID; inputs, configurations, and outputs are recorded so results can be regenerated if needed. Deterministic preprocessing and pinned package versions ensure identical behavior across sessions.
- Backup and recovery: Datasets, checkpoints, metrics, and prediction logs are stored on persistent storage; if a session ends, training can resume from the last checkpoint and inference can continue using the latest approved model version.

Security

- Data privacy: Only de-identified image files are processed; filenames containing identifiers are sanitized or hashed for logs. No write-back occurs to PACS/LIS, which remains read-only.
- Access control: Administrative actions (model updates, threshold changes) are restricted to privileged users, while clinicians can upload/fetch images and view predictions. API keys or authenticated sessions protect remote endpoints.

Usability

- User interface design: The UI is intentionally minimal—upload/fetch, classify, review—and displays the predicted class, final confidence, and per-model confidences in a clear, compact layout. Errors are specific and actionable (e.g., unsupported format, decode failed).
- Training and support: An inline quick-start and short tips explain acceptable formats, class definitions, confidence thresholds, and export options; FAQs cover common issues like low-confidence cases and ensemble selection.

Maintainability and extensibility

- Code maintainability: Configuration-driven experiments (backbone, epochs, batch size, augmentation), modular adapters per backbone (constructor and preprocess_input), and clear directory conventions make the codebase easy to navigate and modify.
- Extension for future features: New backbones can be added by implementing the adapter and registering the model; the shared head, schedule, and evaluation flow then apply automatically. Ensemble rules can add weighting schemes without changing upstream interfaces.

Compliance and legacy

- Compliance posture: Logs retain only what is necessary for audit and reproducibility (request ID, model/version, timestamp, preprocessing route, confidences). The system avoids storing PHI in clear text and supports data retention policies by segregating artifacts and prediction logs.

- Legacy interoperability: Read-only PACS/LIS fetch uses standard image exports so clinical systems remain unaffected, and exported results (CSV/JSON) can be consumed by existing reporting pipelines.

Documentation

- Developer documentation: README and config reference describe environment setup, data layout, training/evaluation commands, and deployment steps; comments clarify backbone adapters and preprocessing expectations.
- User documentation: A short user guide explains uploading images, selecting backbones or ensembles, interpreting confidences, exporting results, and handling low-confidence outcomes.

3.2 Feasibility Study

The proposed solution is technically feasible on commodity cloud notebooks and lightweight serving stacks, economically efficient for prototyping and classroom or pilot deployments, and operationally practical for clinical screening support when paired with clear threshold policies and audit logs. The end-to-end process—verification, transfer learning, fine-tuning, testing, packaging, and serving—has been exercised successfully on a GPU-enabled notebook environment and re-validated in a small web app that enforces identical preprocessing.

3.2.1 Technological Feasibility

The technology stack centers on Python for orchestration, with Pillow/OpenCV for image I/O, NumPy/Pandas for array and log handling, scikit-learn for metrics and optional K-fold evaluation, and TensorFlow/Keras for modeling using keras.applications backbones with include_top=False and each model's preprocess_input. Training uses a two-stage regimen—head-only training followed by selective unfreezing—controlled by EarlyStopping, ModelCheckpoint, and ReduceLROnPlateau, while ImageDataGenerator provides streaming, normalization, and restrained augmentation; deployment uses Streamlit for demonstrations and an optional Flask REST API for programmatic access, with read-only PACS/LIS fetch to integrate clinical sources without write-back.

Technology stack

Python provides a unified language across preprocessing, training, evaluation, and serving, minimizing context switches and enabling consistent dependency pinning; Pillow/OpenCV enforce robust decoding and RGB conversion, while NumPy/Pandas supply high-performance array operations and structured experiment logs. Keras backbones (VGG19, ResNet50, DenseNet121, EfficientNet-B3) expose consistent constructors and normalization routines, allowing a single pipeline to attach a three-class head and train across models with shared code, and scikit-learn adds mature metrics to quantify generalization.

Computing resources
Development and training run in Google Colab on an NVIDIA T4 GPU, where the GPU accelerates forward/backward passes and the CPU handles ingestion and augmentation; this configuration sustained 224×224, batch size 32 across backbones without out-of-memory events and produced stable convergence with early stopping. Colab also offers subscription options (e.g., Colab Pro/Pro+ tiers) that increase likelihood of T4 access, extend session limits, and improve queue priority, which reduces interruptions during longer fine-tuning or hyperparameter sweeps.

Data availability and quality
The dataset comprises 6,089 labeled fundus images partitioned 70/15/15 into train/validation/test and mapped to a fixed class order (Immature, Mature, Normal) shared by every backbone and the ensemble. Each file is validated and standardized to 224×224 with the correct preprocess_input to match ImageNet statistics, class imbalance is monitored and mitigated with class weights during training, and held-out testing preserves an unbiased estimate of generalization; insert the dataset URL in Economic Feasibility or an appendix and record license/terms alongside storage notes.

Model complexity and integration
Backbones range from sequential (VGG19) to residual (ResNet50), densely connected (DenseNet121), and MBConv with SE and compound scaling (EfficientNet-B3); despite differing internal mechanics, each integrates via a common adapter that declares input size, constructor, and normalization. This adapter pattern enables rapid swaps, uniform evaluation, and straightforward ensemble fusion, while versioned artifacts (.h5/.keras plus labels map) allow safe rollback and audit of any production decision.

Interoperability with external tools
PACS/LIS integration is read-only and used solely for image intake, ensuring no mutation of clinical systems; exported results in JSON/CSV/NDJSON interoperate with downstream BI/reporting tools or notebook analysis. Source code and notebooks are versioned in Git/GitHub, while large artifacts and logs are maintained in Drive so the repository remains lean and experiment lineage is preserved.

3.2.2 Economic Feasibility

The project minimizes capital expenditure by leveraging Google Colab's free tier and optional Pro/Pro+ subscriptions to access T4 GPUs reliably, while Google Drive provides persistent, no-ops storage for datasets, checkpoints, and logs; this approach removes the need to purchase or administer dedicated GPU hardware for development and evaluation. Serving costs scale with usage by selecting lightweight backbones for CPU-only demos and enabling GPU only when concurrency or strict latency SLAs justify it; place the dataset URL and licensing note here, describe whether it is mirrored in Drive or accessed on demand, and document any expected egress or retention costs to complete the financial picture. **Potential benefits** The solution accelerates screening with calibrated confidences and threshold-based "second opinion" routing, reducing manual burden and focusing expert time on ambiguous cases; it improves reproducibility through versioned artifacts and fixed preprocessing, simplifying audits and model comparisons; and it supports elastic growth from single-user demos to GPU-assisted multi-user scenarios without redesign.

3.2.3 Social feasibility

The cataract screening system is designed to be accepted and useful in real settings by aligning with how primary-care clinics, vision camps, and tele-ophthalmology hubs already work: non-specialist operators can capture a fundus image and receive an immediate Immature/Mature/Normal prediction with a clear confidence score and traffic-light action, reducing specialist bottlenecks, shortening wait times, and enabling faster referral for surgery where indicated. Accessibility and equity are supported through low hardware requirements (browser or modest GPU), multilingual prompts, brief operator training, and an identical, deterministic preprocessing pipeline that behaves consistently across cameras, while privacy is preserved by processing only image data with minimal metadata and offering on-device or private-cloud inference; low-confidence outputs are automatically routed for manual review to avoid harm.

Community acceptance and sustainability are further enhanced by transparency and accountability: each decision links to a named checkpoint and logged evaluation so program managers can audit performance over time, while class-wise metrics and confusion matrices make residual errors—mainly at the Immature↔Mature boundary—visible for targeted quality improvement. Partnerships with local NGOs and public health programs allow integration into existing referral pathways, and the system's short training footprint and reproducible evaluation make it practical for rural screenings, school eye health drives, and outreach camps; as adoption grows, iterative dataset expansion from diverse devices and populations strengthens fairness and generalizability without disrupting current workflows.

3.3 System Specification

The system specification covers hardware and software environments used for training and serving, emphasizing reproducibility and parity between development and deployment. Hardware describes the Colab T4 GPU runtime and expected throughput per backbone class, while software enumerates pinned frameworks, backbone adapters, preprocessing, callbacks, and artifact paths so that any trained model can be reloaded faithfully for inference or continued fine-tuning.

3.3.1 Hardware Specification

Training and evaluation ran in Google Colab with a GPU accelerator in a Linux-based Python runtime, where GPU kernels executed forward and backward passes while the CPU handled data ingestion, augmentation, and orchestration. The dataset contained 6,089 labeled images split 70 percent for training, 15 percent for validation, and 15 percent for testing; inputs were standardized to 224×224 with batch size 32, and all phases head training, fine-tuning, and evaluation completed without out-of-memory events. Fine-tuning throughput was on the order of hundreds of milliseconds per step and tens of seconds per epoch for heavier backbones, indicating a balanced input pipeline and stable accelerator utilization. Because the Colab filesystem is ephemeral, Google Drive was mounted for persistent reads and writes to preserve datasets, checkpoints, and evaluation outputs across sessions; for deployment, CPU-only serving is feasible with lightweight backbones, whereas GPU-backed serving is recommended for deeper models and concurrent traffic to maintain low latency.

3.3.2 Software Specification

The implementation used Python in Google Colab with GPU-enabled TensorFlow/Keras for model definition, transfer learning, training, and inference, relying on keras.applications to load ImageNet-initialized backbones and to apply each model's preprocess_input normalization. The selected backbones were ResNet50, VGG19, DenseNet121, EfficientNet-B3, InceptionResNetV2, and MobileNetV3, each was loaded without the original classifier top and extended with global average pooling and a compact dense head for three-class softmax.
Training followed a two-stage schedule starting with head-only optimization while freezing the backbone and then selectively unfreezing upper blocks for fine-tuning with a reduced learning rate, using categorical cross-entropy, the Adam optimizer, early stopping to curb overfitting, best-model checkpointing for reproducibility, and learning-rate reduction on plateaus to stabilize convergence.
Data ingestion standardized images to 224×224, converted color mode to RGB when required, and applied clinically reasonable augmentations such as horizontal flips, small rotations, zoom, and spatial shifts; Pillow provided robust image loading and array conversion, OpenCV was available for optional inspection and preprocessing, NumPy powered array-level computation, and Pandas organized experiment logs, per-model confidences, and ensemble summaries.

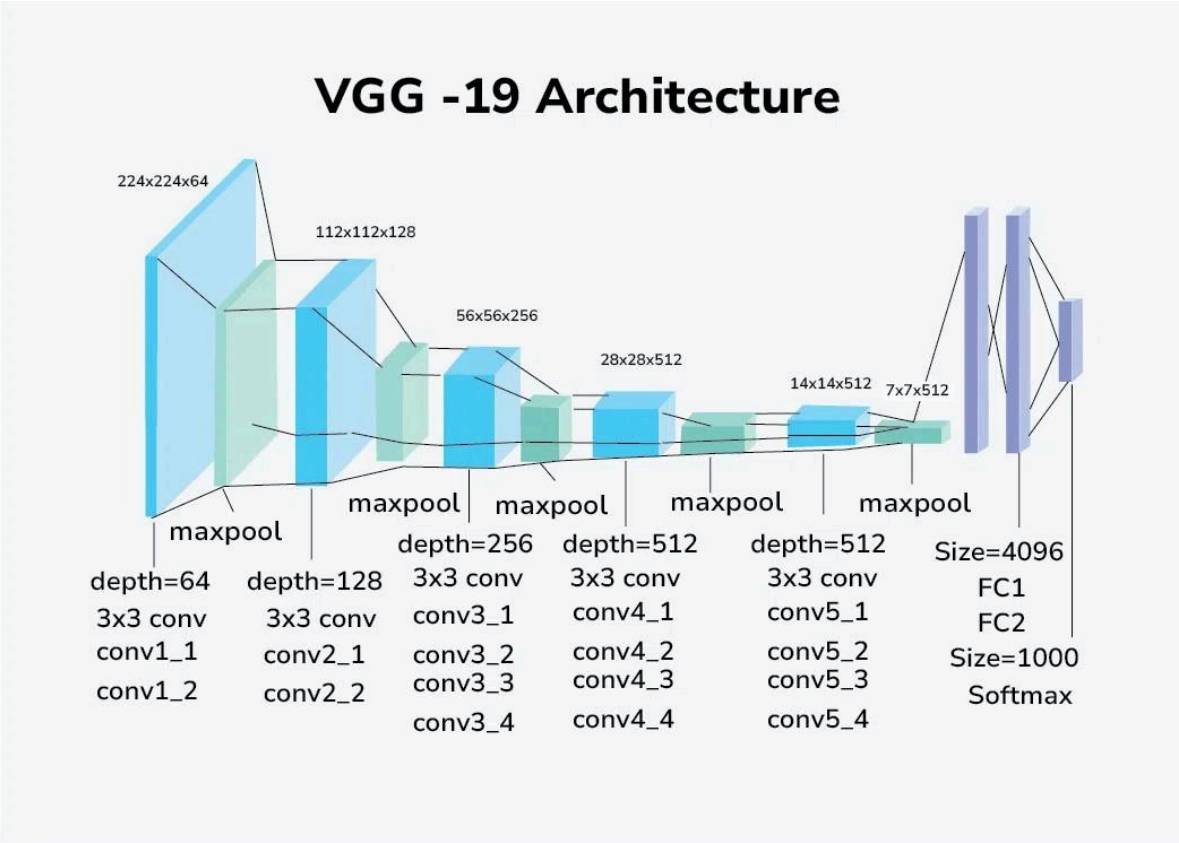
Evaluation on the held-out test split used scikit-learn to compute classification reports and confusion matrices, while matplotlib and seaborn plotted training curves and diagnostics; ensemble predictions combined per-model probabilities through averaging or accuracy-weighted fusion to produce the final class and confidence. Persistence and reproducibility were ensured by mounting Google Drive for datasets, checkpoints, and logs, maintaining stable directory paths across training, validation, testing, and deployment, and versioning code, notebooks, and deployment scripts with Git and GitHub; large artifacts were stored in Drive or managed with large-file support so the repository remained lean while preserving artifact lineage.

Serving for demonstrations used a Streamlit application that reloads checkpoints and enforces the identical preprocessing used during training to maintain numerical consistency; when API integration was required, a minimal Flask service exposed REST endpoints, and during development a tunnel was used to obtain a temporary public URL for real-time testing without provisioning external hosting.

3.3.3 Model Architectures and Training Pipeline

VGG19

1. **Architecture overview**
   VGG19 is a deep convolutional neural network with 19 learnable layers organized into five sequential convolutional blocks followed by fully connected layers in its original form. In transfer-learning mode for this project, the network is loaded with ImageNet weights and the classification head is removed (include_top=False), exposing the convolutional feature extractor that transforms 224×224 RGB inputs into rich hierarchical feature maps. The hallmark of VGG19 is its uniform use of 3×3 convolutions and 2×2 max-pooling, which simplifies receptive-field growth and yields stable features suitable for fine-tuning on medical images.

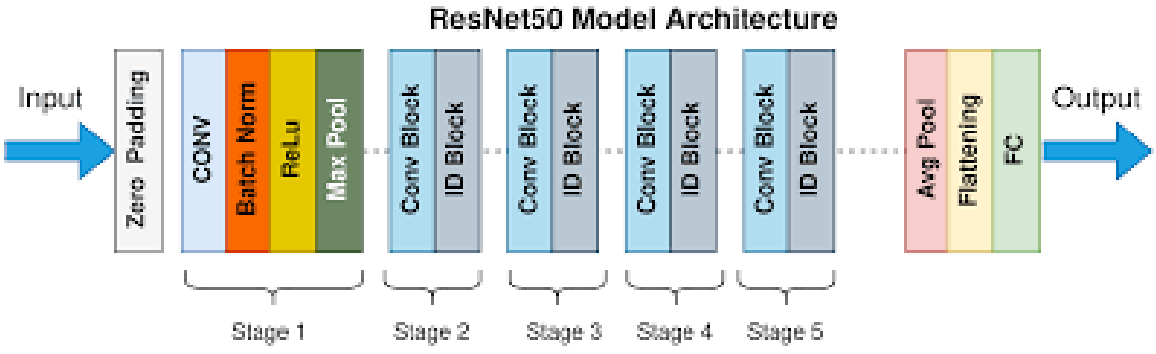


2. **Input and preprocessing**
   Inputs are converted to RGB, resized to 224×224, and normalized using the VGG19 preprocess_input routine to match ImageNet training statistics. This alignment ensures numerical fidelity between training and inference and avoids distribution shift that could degrade accuracy.
3. **Head design for this project**
   A lightweight task head is attached to the frozen base: Flatten → Dropout(0.3) → Dense(3, softmax). The softmax outputs calibrated probabilities for the three cataract classes in a fixed label order, enabling consistent evaluation and easy interchangeability with other backbones in the system.
4. **Training schedule**
   Training follows a two-stage regimen. Stage 1 freezes the entire VGG19 base and trains only the new head using Adam with learning rate 1×10^−3, categorical cross-entropy, and accuracy, while class weights address minor imbalance and callbacks (EarlyStopping with restore_best_weights and ReduceLROnPlateau) control overfitting. Stage 2 selectively unfreezes the last convolutional block (Block 5), recompiles with a low learning rate 1×10^−5, and fine-tunes end-to-end so high-level filters adapt to domain features without destabilizing lower layers. Checkpoints save the best validation loss, and histories from both stages are combined for reporting.

5. **Data pipeline**
   ImageDataGenerator streams batches from train/validation/test directories with light augmentations during training (horizontal flips, small rotations, zoom, and shifts) and deterministic preprocessing for validation and test. Validation and test loaders disable shuffling to maintain deterministic metrics, and class indices are fixed to keep label ordering stable across runs and models.
6. **Evaluation and artifacts**
   After fine-tuning, the model is evaluated on the held-out test set to report accuracy and loss, followed by a classification report and confusion matrix to examine per-class performance. The final model is exported to both .h5 and/or .keras formats along with the label mapping, embedding accuracy or timestamp in filenames to ensure traceability across experiments and deployments.
7. **Why VGG19 fits this project**
   VGG19's simple, deep, sequential design offers predictable transfer-learning behavior and stable convergence with limited hyperparameter tuning. Its uniform 3×3 kernels and well-studied ImageNet weights make it an excellent baseline that is easy to compare against modern architectures in a unified pipeline while still delivering strong feature representations for medical imaging tasks.

ResNet50

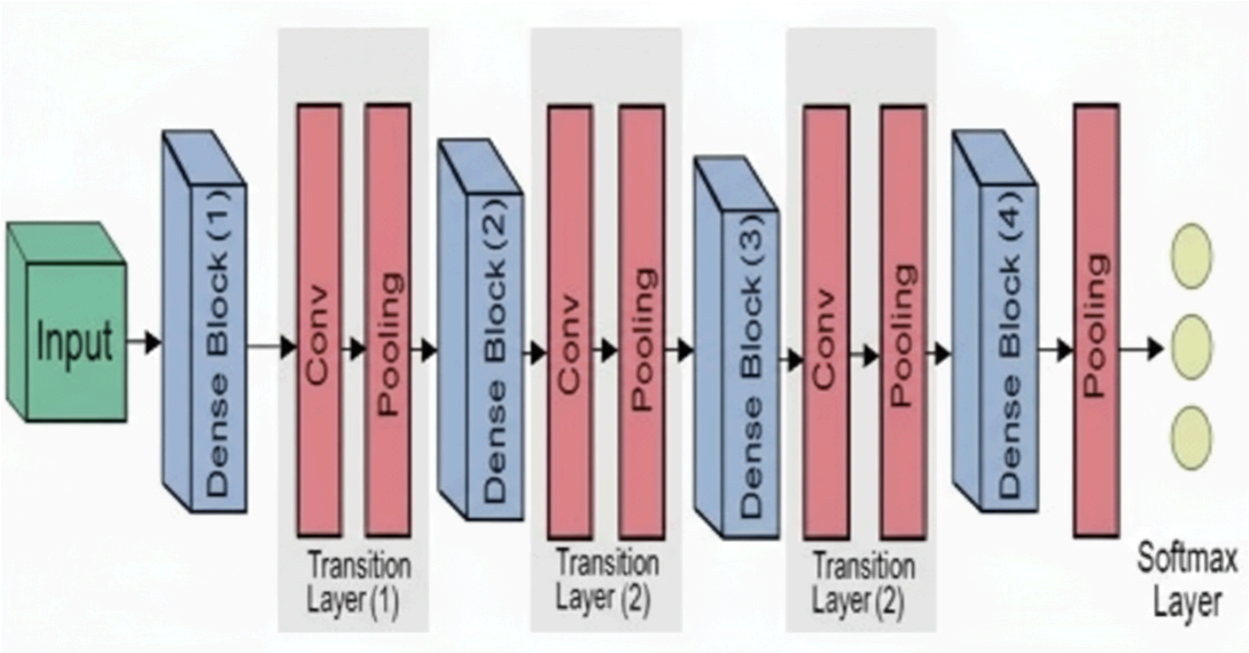
1. **Architecture overview**
   ResNet50 is a 50-layer deep residual network that stacks convolutional layers into bottleneck residual blocks with identity or projection shortcuts. The defining idea is the skip connection y=F(x)+x which enables very deep models to learn residual functions without vanishing gradients, giving strong representational power while remaining trainable on standard hardware. For transfer learning, the network is loaded with ImageNet weights and the classification head is removed (include_top=False), exposing a 7×7/64 stem, four residual stages (Conv2_x to Conv5_x), and global feature maps suitable for fine-tuning



2. **Input and preprocessing**
   Inputs are RGB, resized to 224×224, and normalized using ResNet's preprocess_input, which applies channel‑wise normalization aligned to ImageNet statistics. This ensures numerical compatibility with the pretrained weights and stable transfer to the cataract dataset.
3. **Head design for this project**
   The base output is passed through GlobalAveragePooling2D, then Dropout(0.3), and a Dense layer with 3‑way softmax, producing class probabilities in a fixed label order. This compact head reduces parameters, improves generalization, and keeps the interface consistent with other backbones in the system.
4. **Training schedule**
   Training proceeds in two stages. Stage 1 freezes the entire ResNet50 base and trains only the new head using Adam with learning rate 1×10^{-3}, categorical cross‑entropy, and accuracy, with class weights to mitigate imbalance and callbacks (EarlyStopping with restore_best_weights and ReduceLROnPlateau) to control overfitting. Stage 2 unfreezes upper layers while keeping BatchNormalization frozen, recompiles with a low learning rate 1×10^{-5} , and fine‑tunes end‑to‑end, allowing the deepest residual blocks to adapt to cataract‑specific features without destabilizing earlier representations. Histories from both stages are merged for unified plots and reporting.
5. **Data pipeline**
   ImageDataGenerator streams batches from train/validation/test directories with light augmentation for training (e.g., horizontal flips) and deterministic preprocessing for validation and test. Validation/test loaders do not shuffle, ensuring stable metrics and reproducible confusion matrices and reports.
6. **Evaluation and artifacts**
   After fine‑tuning, the model is evaluated on the held‑out test set to compute accuracy and loss, followed by generation of a classification report and confusion matrix to inspect per‑class performance. The final model is exported to .h5 and .keras with accuracy embedded in filenames, alongside the label mapping, preserving traceability for deployment and future experiments.
7. **Why ResNet50 fits this project**
   Residual connections make deep optimization reliable, delivering strong accuracy with manageable compute and memory costs. As a widely adopted backbone with robust ImageNet weights, ResNet50 provides an ideal benchmark against which to compare simpler (VGG19) and more parameter‑efficient or modern architectures, while integrating seamlessly into the shared training pipeline.

Densenet121

1. **Architecture overview**
   DenseNet121 is a densely connected convolutional network where each layer receives, via concatenation, the feature maps of all preceding layers within the same dense block. This design promotes feature reuse, strengthens gradient flow, and reduces parameter count compared with plain or residual stacks of similar depth, making it data-efficient and well-suited to fine-tuning on medical images. The transfer-learning setup loads ImageNet weights with the classification head removed (include_top=False), exposing the convolutional trunk for adaptation.



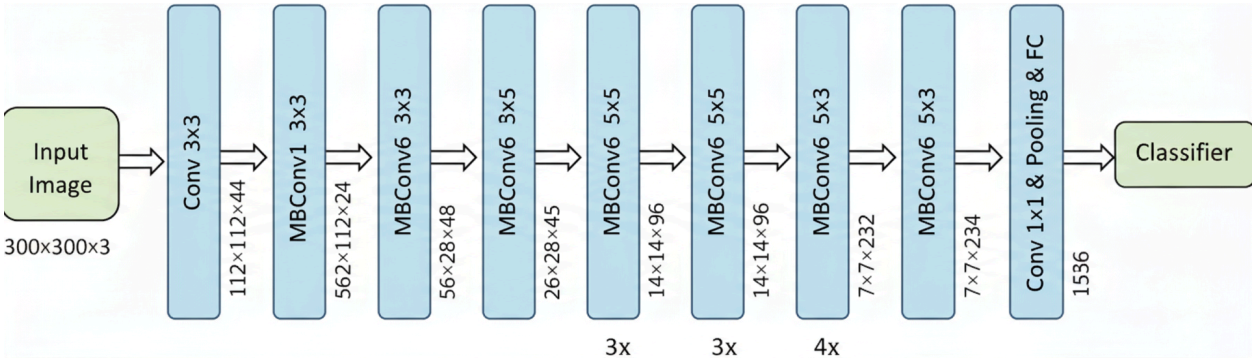
2. **Input and preprocessing**
   Inputs are RGB images resized to 224×224 and normalized using the DenseNet preprocess_input routine aligned to ImageNet statistics. This ensures numerical compatibility with pretrained filters and stable convergence during fine-tuning on the cataract dataset.
3. **Head design for this project**
   After the base, GlobalAveragePooling2D aggregates spatial features, followed by Dropout(0.3) and a Dense softmax layer with three outputs corresponding to the fixed class order. This lean head leverages the strong feature reuse of DenseNet while controlling overfitting and keeping the interface consistent with other backbones.
4. **Training schedule**
   Training follows a two-stage plan. Stage 1 freezes the DenseNet121 base and trains only the head using Adam with learning rate 1×10−3, categorical cross-entropy, and accuracy; class weights compensate for imbalance, and callbacks (EarlyStopping with restore_best_weights, ReduceLROnPlateau) manage generalization. Stage 2 unfreezes

the upper portion of the base while leaving BatchNormalization layers frozen, recompiles with a low learning rate 1×10−5, and fine-tunes end-to-end so high-level features adapt to cataract patterns without disrupting earlier representations.

5. **Data pipeline**
   ImageDataGenerator streams batched images from train/validation/test folders with light augmentations during training—horizontal flips, small rotations, zoom, and shifts—and deterministic preprocessing for validation and test. Class indices are fixed and validation/tests are non-shuffled to ensure reproducibility of metrics and confusion matrices.
6. **Evaluation and artifacts**
   After fine-tuning, the model is evaluated on the held-out test set to obtain accuracy and loss, with a classification report and confusion matrix to inspect per-class behavior. The final weights and architecture are exported to .h5 and .keras with accuracy or timestamp in filenames, plus the label mapping, ensuring traceable deployment.
7. **Why does Densenet121 fit this project?**
   Dense connectivity enables strong gradient propagation and feature reuse, offering high accuracy with relatively few parameters and rapid convergence on moderate-sized datasets. Its data efficiency and stable fine-tuning behavior make it an excellent complement to VGG19 and ResNet50 within a shared, model-agnostic pipeline.

EfficientNet-B3

1. **Architecture overview**
   EfficientNet-B3 is a convolutional network built using compound scaling, which jointly scales depth, width, and input resolution to balance accuracy and efficiency. The backbone is composed of stacked MBConv blocks (inverted residual with depthwise separable convolutions and squeeze-and-excitation), giving strong accuracy at a modest parameter and FLOP budget. For transfer learning, ImageNet weights are loaded with the top classifier removed (include_top=False), exposing high-quality feature maps for adaptation.



2. **Input and preprocessing**
   Inputs are RGB images resized to 224×224 for consistency across models in this project, and normalized with EfficientNet's preprocess_input routine to match ImageNet training

statistics. This preserves numerical fidelity to the pretrained weights and stabilizes convergence on the cataract dataset.

3. **Head design for this project**
   After the base, GlobalAveragePooling2D aggregates spatial features, followed by Dropout(0.3) and a Dense softmax layer with three outputs in a fixed class order. The minimalist head leverages EfficientNet's strong features while controlling overfitting and aligning interfaces with other backbones.
4. **Training schedule**
   Training follows two stages. Stage 1 freezes the EfficientNet-B3 base and trains only the head using Adam with learning rate 1×10−3, categorical cross-entropy, and accuracy; class weights mitigate imbalance, with EarlyStopping and ReduceLROnPlateau to improve generalization. Stage 2 unfreezes the upper portion of the backbone except BatchNormalization layers, recompiles with learning rate 1×10−5, and fine-tunes end-to-end so high-level MBConv features adapt to domain-specific patterns without destabilizing earlier layers.
5. **Data pipeline**
   ImageDataGenerator streams augmented training batches (horizontal flips, small rotations, zoom, width/height shifts) and deterministic validation/test batches, all using the same preprocess_input. Non-shuffled validation/test loaders preserve metric reproducibility and fixed class index ordering across runs and models.
6. **Evaluation and artifacts**
   After fine-tuning, the model is evaluated on the held-out test set to produce accuracy and loss, with classification report and confusion matrix for per-class insight. The final model is saved to .h5 and .keras with accuracy or timestamp embedded in filenames plus the label map for consistent deployment.
7. **Why EfficientNet-B3 fits this project**
   Compound scaling and MBConv blocks provide an excellent accuracy-to-efficiency trade-off, often achieving strong performance with fewer parameters and faster inference. This makes B3 a practical, high-performing choice that complements VGG19, ResNet50, and DenseNet121 within a uniform training and evaluation pipeline.

4. DESIGN APPROACH AND DETAILS

4.1 System Architecture

4.1.1 Model Training

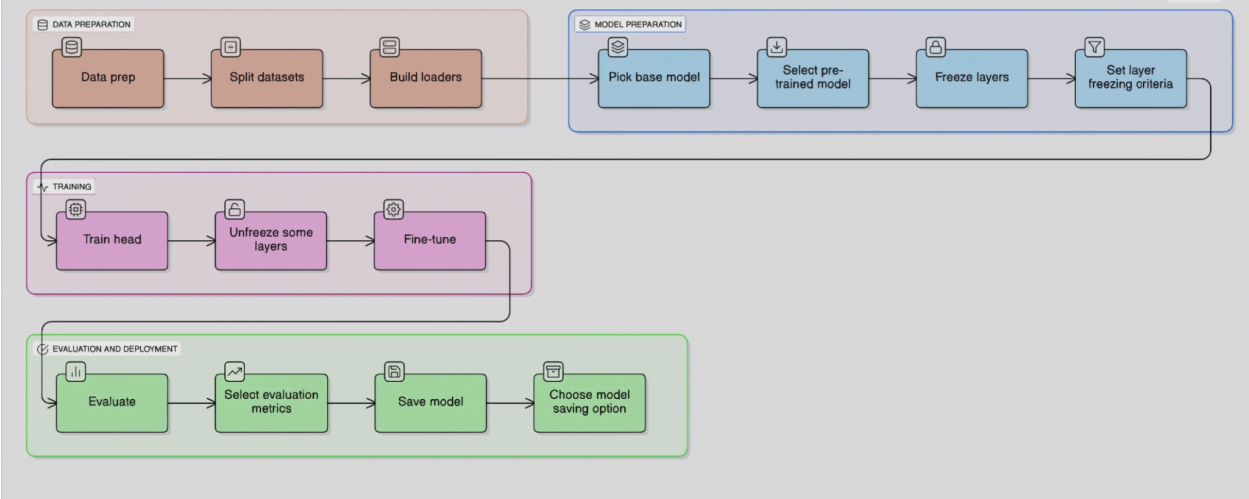


Figure 4.1 presents a model-agnostic transfer-learning pipeline whose modular steps

The diagram depicts a universal, repeatable training workflow used to fine-tune any of the four convolutional backbones VGG19, ResNet50, DenseNet121, and EfficientNetB3 on a three-class cataract dataset, ensuring that every stage is modular so the same pipeline can be executed regardless of which base model is selected. Refer to this schematic as Figure 4.1.

Data preparation begins with a structured Data prep phase in which images are collected, cleaned, and normalized into a consistent folder layout of train, validation, and test. This organization lets downstream components stream batches efficiently and reproducibly, independent of the chosen backbone. In practice, Keras-style loaders and the backbone's native preprocessing function standardize input resolution and pixel scaling so the pretrained weights receive data in the format they were optimized for, which stabilizes transfer learning across all four architectures.

Afterward, Split datasets formalize the separation of samples into disjoint subsets that maintain similar class distributions. By preventing leakage between training and evaluation data, this stage provides trustworthy validation signals for early stopping and hyperparameter choices. Generator-driven iteration ensures the loop never holds the entire dataset in memory, keeping the routine identical whether the backbone is VGG19, ResNet50, DenseNet121, or EfficientNetB3.
Build loaders then construct the data pipelines that apply on-the-fly augmentation and preprocessing. Augmentations such as flips, small rotations, zooms, and shifts introduce realistic variability to combat overfitting, while preprocess_input aligns pixel statistics with ImageNet

conventions. The result is a stream of batches that already match the backbone's expected size, such as 224×224, so the same loaders can feed any selected model.

Next, Pick base model chooses one of the pretrained CNNs with include_top disabled so only the convolutional feature extractor is loaded. Because all four backbones expose compatible interfaces, a uniform classification head can be attached above them, enabling a single training script to support multiple architectures without code duplication. This design is what makes the workflow reusable across models. Select pretrained model underscores loading ImageNet weights to import broad visual priors learned from millions of images. These early filters capture edges, textures, and shapes that transfer well to medical imagery, accelerating convergence and reducing the amount of task-specific data required. Regardless of whether the backbone is a sequential VGG, a residual ResNet, a densely connected DenseNet, or an EfficientNet with compound scaling, the initialization gives a strong starting point.

Freeze layers locks the base model initially by marking its layers non-trainable, allowing the freshly added head to learn a task-specific mapping from generic features to the three cataract classes. This protects useful pretrained features from being overwritten while the head stabilizes and brings the logits into a sensible range. The behavior is consistent across all four architectures, yielding predictable early training. Set layer freezing criteria defines a clear policy for which parts of the backbone remain frozen and which can be unfrozen later. Typical choices keep the earliest blocks frozen, since low-level features like edges are general, while allowing higher blocks to adapt to domain specifics. Policies can target top-N blocks or name patterns, making the approach reproducible across VGG's sequential blocks, ResNet's stages, DenseNet's dense blocks, and EfficientNet's MBConv stacks.

Train head compiles the network with the new top layers—often GlobalAveragePooling or Flatten, then Dropout, then a Dense softmax—and optimizes only those layers for several epochs. Class weighting and callbacks counter class imbalance and control learning dynamics. Because gradients do not update the base model yet, the risk of overfitting is reduced and the classifier quickly aligns with dataset semantics. Unfreeze some layers partially thaws the backbone, typically the upper blocks where high-level representations reside. With a reduced learning rate, these layers can shift toward cataract-specific morphology while preserving robust low-level detectors. This selective unfreezing strategy keeps training stable even for deeper, skip-connected or densely connected models and for EfficientNet's lightweight blocks.

Fine-tune re-compiles the model with a very small learning rate and trains end-to-end over the unfrozen portion plus the head. This phase delivers the largest accuracy gains by letting task-specific gradients refine high-level semantics without erasing the useful general priors. The same schedule and hyperparameter logic can be applied uniformly to VGG19, ResNet50,

DenseNet121, and EfficientNetB3. Evaluate runs inference on the untouched test split to estimate generalization via accuracy and loss. Using a held-out set ensures an apples-to-apples comparison among backbones, since no model saw these images during training or validation. This step is the basis for model selection and for documenting the performance of each architecture.

Select evaluation metrics clarifies how results are summarized and visualized. In addition to overall accuracy, confusion matrices and class-wise metrics reveal whether any class is underserved, which is crucial in clinical contexts where the cost of false negatives and false positives differs. This focus keeps the evaluation aligned with downstream use. The save model serializes the exact architecture-head configuration and learned weights into portable formats such as .h5 or .keras, together with the label mapping used for training. Persisting these artifacts guarantees that any chosen backbone can be restored later for inference or continued training without ambiguity. Choose model saving option describes the artifact-management policy, including consistent directory structure, filenames that embed accuracy or timestamps, and optional labels.json to lock the class index order. With standardized artifacts, switching between VGG19, ResNet50, DenseNet121, and EfficientNetB3 is as simple as pointing the deployment environment to the desired file, which keeps experiments reproducible and operational handoff clean.

4.1.2 Deployment Architecture

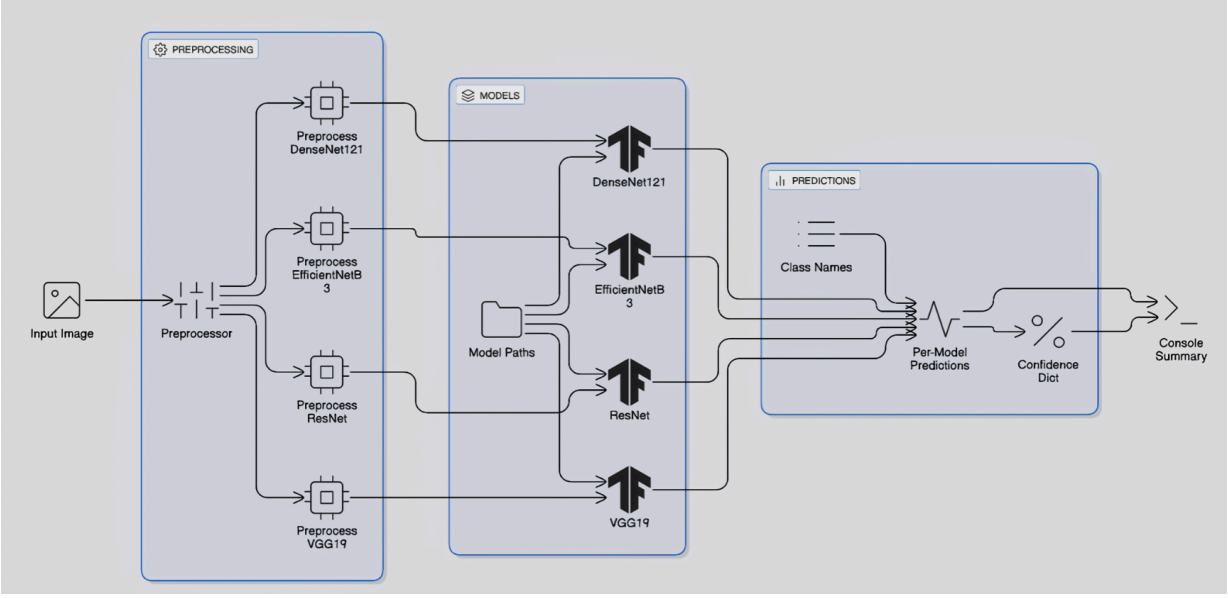


Figure 4.1.2: show how the is it deployed

Image 4.1.2 implements a contract-first pipeline where every stage produces standardized outputs that the next stage can consume without ambiguity. The left preprocessing panel begins with a generic preprocessor that enforces canonical input properties RGB color space, fixed

spatial dimensions of 224×224, and consistent type and value range so downstream branches never need to recheck basic assumptions. Immediately after this normalization, the flow fans out into backbone-specific preprocessors for DenseNet121, EfficientNet-B3, ResNet, and VGG19; each branch applies its model's exact preprocess_input routine and channel ordering so that pixels presented at inference are statistically aligned with what the backbone saw during ImageNet pre-training and task fine-tuning. This separation is crucial because these routines differ across families (for example, mean/variance scaling and channel centering), and merging them would silently degrade accuracy; isolating them guarantees numerical fidelity while allowing the system to run several architectures in parallel.

At the center, the models panel is decoupled from hard-coded file locations via a "Model Paths" node that maps logical model names to versioned artifact URIs. When a request arrives, each backbone resolves to its current approved checkpoint and is instantiated once, then retained in memory for subsequent calls to eliminate cold-start penalties. The diagram's parallel connectors from each backbone-specific preprocessing branch into its corresponding model emphasize that tensors and models remain type- and shape-compatible by design, which prevents runtime errors and makes horizontal scaling straightforward. Class names live as a single authoritative mapping so that every model emits probabilities in an identical index order; this avoids brittle post-hoc remapping, safeguards ensembling logic, and ensures that any new backbone can be dropped in as long as it adheres to the same class contract.

On the right, the predictions panel aggregates raw outputs into a consistent result schema without losing per-model detail. First, each backbone's probability vector is collected into a bundle of per-model predictions tied to the shared class map, making it possible to inspect agreement and disagreement at a glance. Next, a confidence dictionary is built that contains the final selected label and confidence plus the individual model confidences, which preserves transparency for audits and lets operators tune fusion strategies later without changing upstream code. Finally, a concise console summary is emitted for notebook runs and CI logs, while the fuller structured payload can be serialized to storage for experiment tracking. This end-to-end separation—generic normalization, backbone-specific preprocessing, version-aware model loading, schema-aligned outputs, and transparent consolidation—makes the system easy to reproduce, simple to extend with new backbones, and safe to operate under changing model versions because every boundary is explicit and every artifact is traceable.

4.2 Design

4.2.1 Data Flow Diagram

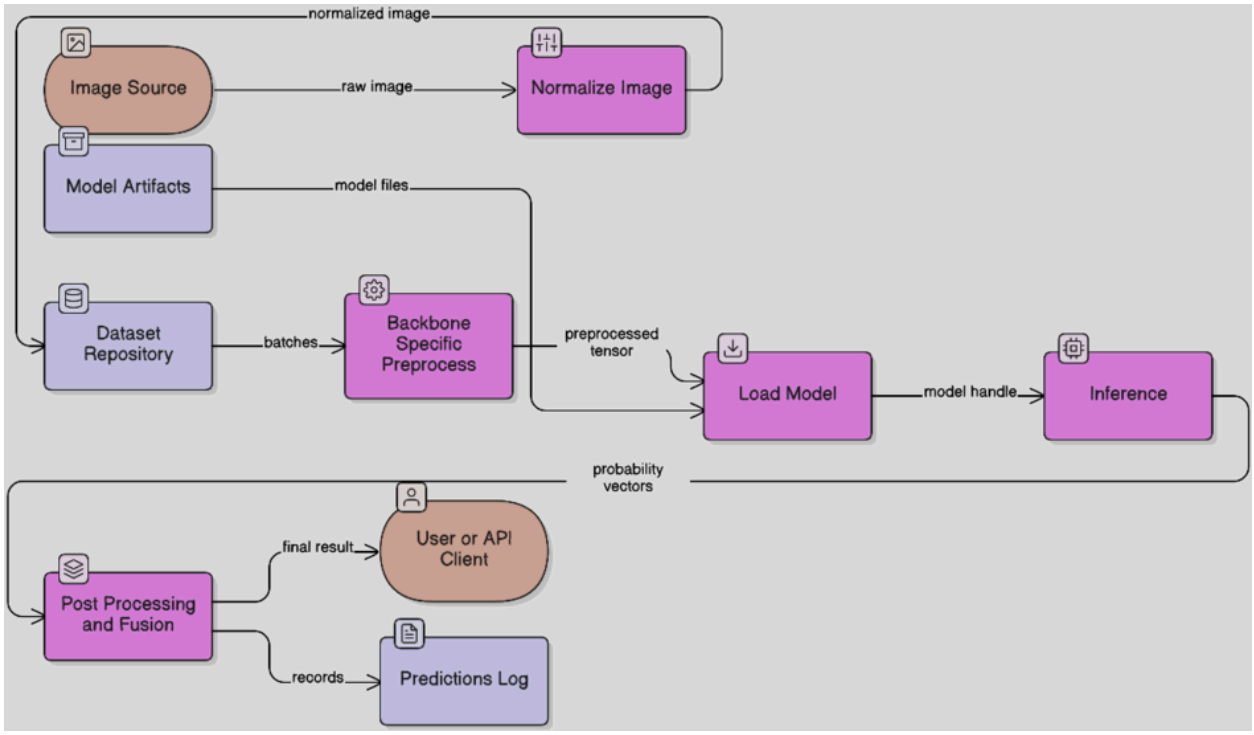


FIGURE 4.2.1 Data Flow Diagram

4.2.2 Use Case Diagram

The Use Case Diagram illustrates the interaction between the external actors and the AI Powered Cataract Detection System. It highlights the primary functionalities provided to the end user and the ophthalmologist, as well as the internal operations that the system performs to classify cataracts and present diagnostic support information. In this system, the User (which may be a patient, technician, or screening staff) uploads an eye image through the interface. The system first carries out the Validate Image Format use case to ensure that the uploaded file meets the required constraints such as supported file type and acceptable size. Once the validation is successful, the system proceeds to Process Image, where preprocessing steps such as resizing, normalization, and format conversion are performed to prepare the image for classification.

The processed image is then passed to the Classify Cataract Stage use case. Here, the deep learning model categorizes the eye image into one of the predefined classes: Normal, Immature Cataract, or Mature Cataract. After the classification, the system executes the Display Confidence Score use case to provide the user with the predicted category along with its confidence probability. This helps users interpret how certain the model is about its prediction.

Additionally, the system supports an optional diagnostic interpretability function, represented by the View Grad-CAM Heatmap use case. When activated, this feature generates a visual heatmap overlay that highlights the retinal regions most influential to the model's classification decision. This assists users in understanding the model's decision behaviors and increases clinical transparency.

The Ophthalmologist actor can perform all the same functions as the general user but may also use the Generate Prediction Report use case, which consolidates the model's output and visual interpretation into a documented form useful for screening summary, referral decision-making, or patient recordkeeping.

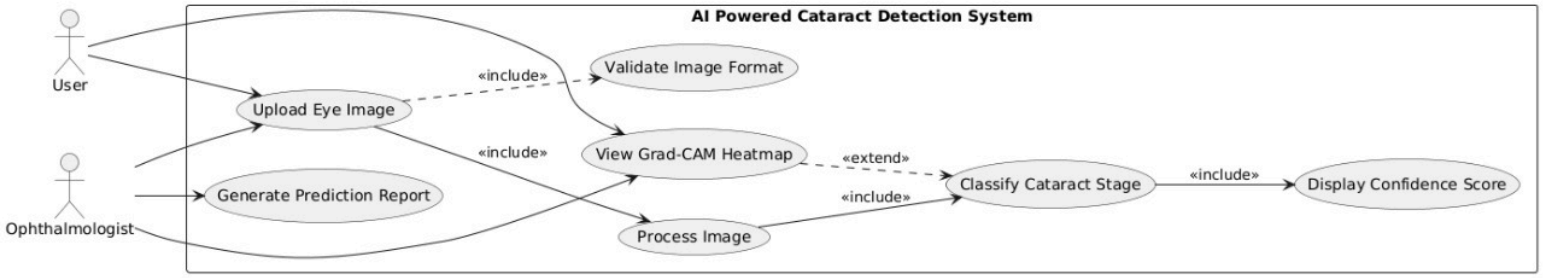


FIGURE: 4.2.2 Use Case Diagram

4.2.3 Class Diagram

The class diagram illustrates the structural design of the AI Powered Cataract Detection System by showing the major system components, their internal data attributes, methods, and the interactions between them. The architecture follows a modular and layered design, ensuring clear separation of responsibilities, maintainability, and scalability. Each class in the system is assigned a specific role, beginning from image input and preprocessing to model inference and interpretability using Grad-CAM visualizations.

The UI (User Interface) class manages interaction between the system and end users. It provides features for uploading eye images, displaying classification results along with confidence scores, and optionally visualizing heatmaps. When the user uploads an image, it is handled by the ImageUploader class, which performs validation of file format and size before extracting the image for processing.

Once validated, the image is passed to the Preprocessor class, which performs operations such as resizing, normalization, and augmentation to ensure consistency and enhance the robustness of the model. After preprocessing, the processed tensor is transferred to the ModelManager, which

contains the loaded deep learning model. This class executes prediction, retrieves the most probable cataract category (Normal, Immature Cataract, or Mature Cataract), and computes the confidence scores for the prediction. For interpretability, the classification result and model are optionally passed to the GradCAM class. This class generates heatmaps to highlight the regions of the retina that influenced the model's decision, supporting transparency and increasing clinical trust. These heatmaps are then returned to the UI for display.

The relationships among these classes clearly reflect the data flow pipeline in the system:
UI → ImageUploader → Preprocessor → ModelManager → GradCAM → UI.
This modular flow ensures efficient data processing, transparent prediction, and user-friendly output presentation.

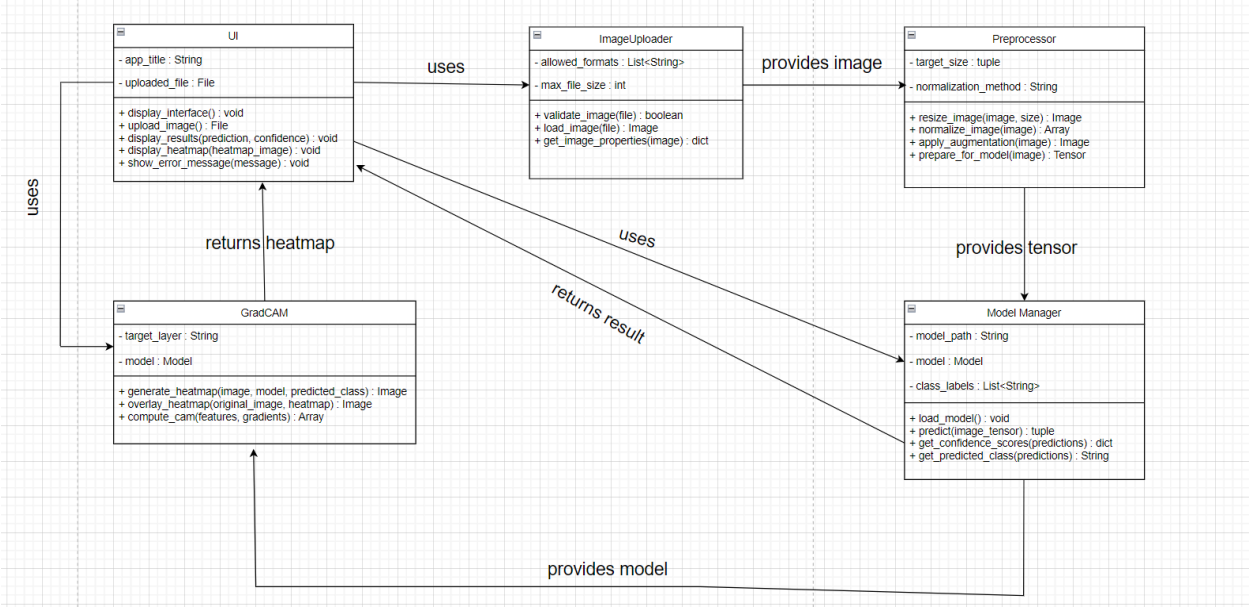


FIGURE: 4.2.3 Class Diagram

4.2.4 Sequence Diagram

The sequence diagram illustrates the dynamic flow of interactions between the different components of the AI Powered Cataract Detection System during the cataract classification and result visualization process. It represents how data moves from the user interface through preprocessing and model inference, and how the final predictions and optional Grad-CAM visualizations are returned to the user. The interaction begins when the User opens the application interface and uploads an eye image through the UI. The uploaded file is passed to the ImageUploader, which performs validation checks on factors such as file type and file size. If the validation fails, the UI displays an appropriate error message to the user. If validation succeeds, the image is accepted and forwarded to the Preprocessor. The Preprocessor performs key image preparation steps such as resizing, normalization, and optional augmentation to ensure that the

data is in the appropriate format required by the model. The processed image tensor is then sent to the ModelManager. The ModelManager contains the trained deep learning model and handles the prediction operation. It generates the predicted cataract class (Normal, Immature Cataract, or Mature Cataract) along with the confidence score of the prediction. These outputs are then sent back to the UI for display to the user. Optionally, the user may request deeper visual explanation of the prediction. In such cases, the UI calls the GradCAM component. Based on the model and predicted class, GradCAM computes activation gradients and overlays a heatmap onto the original image to highlight the important regions influencing the model's decision. This heatmap image is returned to the UI and presented to the user. Thus, the sequence diagram effectively demonstrates a clear, step-by-step flow of image input, validation, preprocessing, model inference, optional interpretability, and result display. The diagram also incorporates conditional behavior, represented by the alt block for validation outcome and the opt block for optional Grad-CAM visualization, reflecting real operational flexibility in the system.

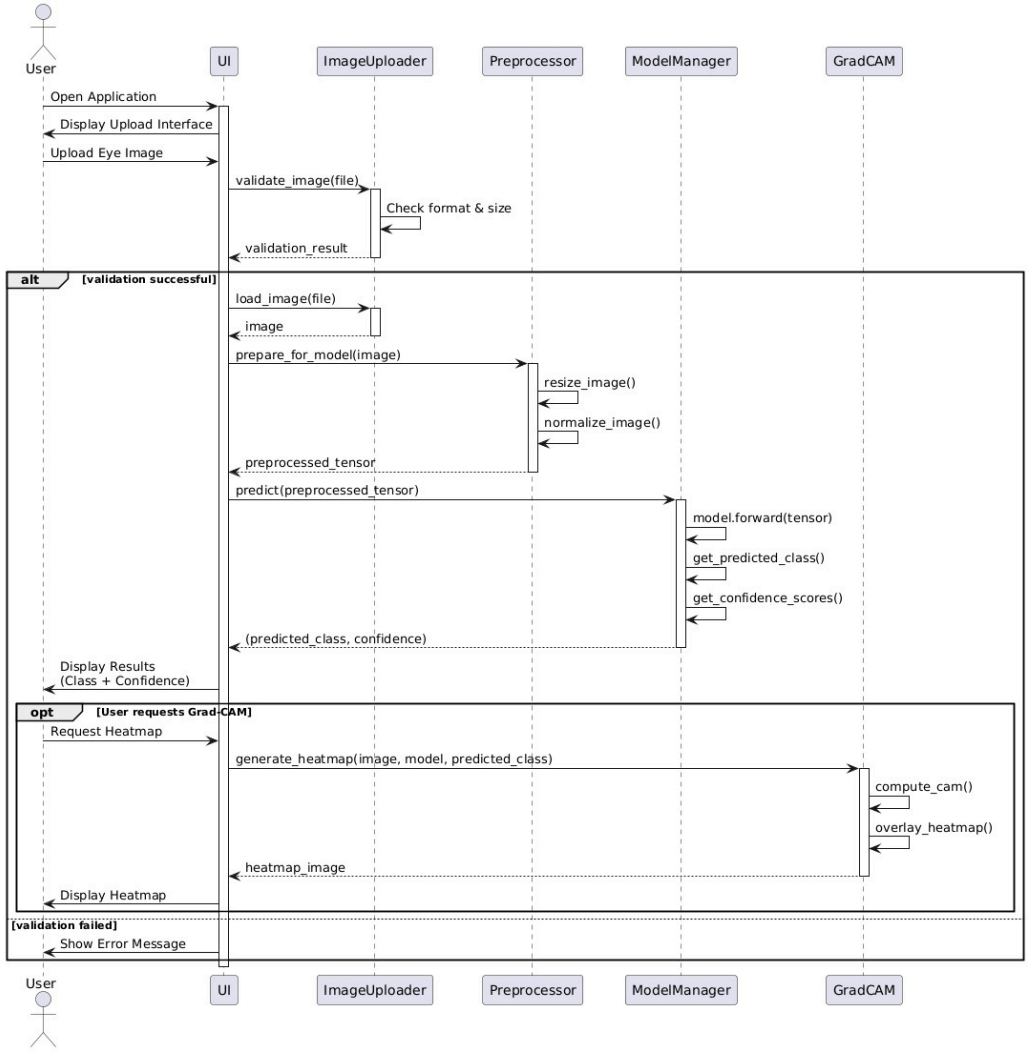


FIGURE:4.2.4 Sequence Diagram

5. METHODOLOGY AND TESTING

This section documents the exact workflow implemented in the notebooks and deployment code: directory-driven data loaders at 224×224, backbone‑specific normalization, two‑stage transfer learning per model, test‑time deterministic evaluation, and Streamlit deployment that reloads saved .h5 checkpoints with identical preprocessing. Inference does not use Flask; only the Streamlit UI and batch notebooks consume the exported .h5 models.

5.1 Module Description

5.1.1 Dataset preparation

Images are organized in class‑named folders and split 70%/15%/15% for train/validation/test; each file is decoded, converted to RGB if needed, and resized to 224×224 before model‑specific normalization. Original filenames are preserved and paired with a generated run/request ID so predictions, confusion matrices, and reports can be traced back to their sources across training and serving. Class indices are fixed to [Immature, Mature, Normal] to keep probability vectors aligned across all backbones and the ensemble.

5.1.2 Preprocessing and augmentation

Data is streamed with Keras ImageDataGenerator using the same directory paths for all runs; training uses restrained augmentations (horizontal flip, small rotation/zoom, width/height shift), while validation/test disables augmentation for deterministic metrics. After generic resizing, each branch applies its exact preprocess_input: vgg19.preprocess_input, resnet50.preprocess_input, densenet.preprocess_input, and efficientnet.preprocess_input, ensuring pixel statistics and channel ordering match ImageNet initialization for numerical fidelity.

5.1.3 Model architectures and observed accuracies

All backbones load ImageNet weights with include_top=False and attach a unified head: GlobalAveragePooling → Dropout → Dense(3, softmax) ordered as [Immature, Mature, Normal]. The four trained models achieved the following test accuracies under the held‑out split: DenseNet121 = 99.03%, EfficientNet‑B3 = 99.84%, ResNet50 = 99.90%, and VGG19 = 99.89%; these values come from the saved evaluation notebooks and confusion‑matrix reports and are used as optional weights for ensemble fusion.

5.1.4 Training schedule

Stage 1 (head-only). Each backbone is instantiated with frozen convolutional layers to act as a fixed feature extractor while training only the new classification head. Optimization uses Adam with categorical cross-entropy and accuracy, batch size 32 at 224×224, and class weights computed from the training set to temper any mild imbalance; EarlyStopping with restore_best_weights monitors validation loss, and ReduceLROnPlateau lowers the learning rate when progress stalls.

Transition and checkpointing. At the end of the head stage, the best validation-loss checkpoint is persisted to Google Drive together with the label map and the full training history (loss/accuracy per epoch). This ensures the pipeline can resume or roll back cleanly, and it provides a stable baseline for the fine-tuning stage; all artifacts are versioned using backbone name, timestamp, and best-epoch metadata to maintain lineage.

Stage 2 (selective fine-tuning). The top convolutional block(s) of each backbone are unfrozen to adapt high-level filters to cataract features while keeping earlier representational layers stable; BatchNormalization layers remain in inference mode to avoid internal statistics drift. The model is recompiled with a low learning rate and trained with the same callbacks until validation loss no longer improves; the best fine-tuned checkpoint is exported and promoted for evaluation and serving.

5.1.5 Hyperparameters and configuration (more detail)

Training configuration is centralized per run: image_size=224, batch_size=32, epochs_head and epochs_fine, optimizer=Adam with stage-specific lr, dropout rate for the head, augmentation toggles, and unfreeze depth per backbone. Lightweight sweeps vary learning rate, dropout, and unfreeze depth within safe bounds; selection uses validation loss as the primary criterion and accuracy as tie-breaker. For reproducibility, random seeds and key package versions (TensorFlow/Keras, scikit-learn, Pillow) are pinned in the notebooks, and directory structures for data, checkpoints, and logs are kept stable so experiments are trivially repeatable.

5.1.6 Cross-validation (optional but supported)

When enabled, K-fold cross-validation reshuffles the train/validation pool into K folds while reserving the test split untouched, running the same two-stage schedule and preprocessing graph per fold. Fold-wise histories, per-class metrics, and confusion matrices are saved, and summary tables report mean and standard deviation across folds, providing a more reliable estimate of generalization and informing whether the ensemble meaningfully improves stability over single backbones.

5.1.7 Evaluation protocol

Final evaluation on the held‑out test set disables augmentation and reuses the exact preprocess_input of the trained backbone, computing overall accuracy and loss, per‑class precision/recall/F1, and producing a confusion matrix for visual inspection. For ensembles, per‑model probability vectors are fused either by simple average or by weights proportional to the reported single‑model test accuracies (DenseNet121 99.03, EfficientNet‑B3 99.84, ResNet50 99.90, VGG19 99.89), yielding a final label and calibrated confidence; single‑model vs ensemble metrics are compared under the same class index order to ensure fairness.

5.1.8 Error analysis and diagnostics

Misclassified test images are exported with filename, true label, predicted label, and per‑model confidences to identify consistent confusion patterns (e.g., Immature vs Mature). Learning curves (loss/accuracy) for both training stages are plotted to detect underfitting or late‑epoch overfitting; threshold sweeps around the default confidence setting quantify sensitivity/specificity trade‑offs and guide a “second‑opinion” policy for borderline cases requiring review rather than auto‑finalization.

5.1.9 Inference, storage format, and deployment

The Streamlit application reloads the best .h5 checkpoint for the selected backbone (only .h5 is stored) and applies the identical preprocessing graph used during training before computing probabilities, final class, and confidence; the UI also supports ensemble inference that aggregates per‑model probabilities according to the selected rule. Models are instantiated once and cached in memory to remove cold‑start overhead, request payloads are validated and standardized to 224×224 RGB, and responses return the final label, confidence, and the per‑model confidence dictionary in the fixed class order; for batch notebooks, the same .h5 checkpoints are used to generate CSV/NDJSON summaries for archival.

5.1.10 Logging, persistence, and reproducibility

Each training run and inference request is assigned a unique ID and logged with backbone name, artifact version, preprocessing route, final decision, confidences, and timestamp; artifacts (checkpoints, label maps, histories, confusion matrices, and run configs) are written to Google Drive for persistence across Colab sessions. Seeds and package versions are pinned, directory layouts are fixed, and the saved .h5 weights are directly reloadable in notebooks and the Streamlit app, guaranteeing numeric parity between evaluation and serving.

5.1.11 Environment and compute notes (Colab T4)

All experiments were executed in Google Colab on an NVIDIA T4 GPU, which comfortably supported 224×224, batch size 32 across VGG19, ResNet50, DenseNet121, and EfficientNet-B3 without out-of-memory events; early stopping and reduced LR on plateau bounded epoch counts and stabilized convergence. Colab's subscription tiers (e.g., Pro/Pro+) increase T4 availability, extend session duration, and reduce queue delays, which is helpful for longer fine-tuning runs or small hyperparameter sweeps that would otherwise risk session timeouts on the free tier.

5.1.12 Success criteria and acceptance

Methodological success requires stable convergence with early stopping, test accuracy consistent with the reported single-model results DenseNet121(99.03%), EfficientNet-B3 (99.84%), ResNet50 (99.90%) and VGG19 (99.89%), and ensemble behavior that is at least as robust as the best individual backbone. Operational acceptance requires inference parity between notebook and Streamlit outputs, acceptable latency given CPU/GPU selection, complete structured logs with versioned artifacts, and reliable reload of the .h5 checkpoints for both interactive and batch predictions.

5.2 Testing

5.2.1 Clinical sample description

The external test cohort consisted of anonymized color fundus photographs from patients undergoing cataract evaluation, specifically posterior pole/optic disc–centered retinal fundus images acquired during routine ophthalmic assessment; throughout this section, these are referred to as anonymized retinal fundus images of cataract patients.

5.2.2 Test protocol

All test images were processed by the deterministic evaluation pipeline: decode → RGB conversion → resize to 224×224 → backbone-specific preprocess_input → forward pass → softmax probabilities in the fixed class order [Immature, Mature, Normal]. No augmentation was applied at test time; predictions were generated per-image, saved with original filenames and a request ID, and aggregated into CSV/NDJSON for analysis. Confusion matrices and classification reports were computed for each backbone and, when enabled, for the probability-averaged ensemble.

5.2.3 Test cases

The system accepts image inputs only—anonymized color retinal fundus photographs for cataract screening. Users upload images through the interface; upon upload, each file is written to the mounted Drive location reserved for the current run so it persists across sessions.

Storage and retrieval flow
After upload, the image path on Drive is recorded with a unique request ID. The inference service does not consume the raw file directly from the transient upload buffer; instead, it reads the image from Drive using the recorded path, ensuring that training, testing, and serving all reference the same persisted artifact.

Processing contract
For each request, the pipeline loads the image from Drive, decodes and converts to RGB if needed, resizes to 224×224, and applies the backbone‑specific normalization before forward pass. The original filename and its Drive path are kept alongside the prediction to preserve traceability.

Test case 1:

Input: Real-time images of immature cataract eyes

```
...  image path:/content/drive/MyDrive/PROJECT/PROJECT-1/Input Images/testcase(i).jpg
```

Output :Immature with Confidence 99.65%

```
======================================================================
CONFIDENCE MATRIX (Class × Model)
======================================================================
          densenet121  efficientnetb3  resnet  vgg19
Immature       0.9921          0.9937     1.0    1.0
Mature         0.0065          0.0017     0.0    0.0
Normal         0.0014          0.0046     0.0    0.0
======================================================================

======================================================================
WEIGHTED CONFIDENCE FOR EACH CLASS
----------------------------------------------------------------------
Immature    : 0.9965 (99.65%)
Mature      : 0.0020 (0.20%)
Normal      : 0.0015 (0.15%)
----------------------------------------------------------------------

----------------------------------------------------------------------
FINAL ENSEMBLE PREDICTION
======================================================================
Predicted Class: Immature
Ensemble Confidence: 0.9965 (99.65%)
```

```
...
    ======================================================================
    SUMMARY: Individual Model Predictions
    ======================================================================
    densenet121         : Immature   - 0.9921 (99.21%)
    efficientnetb3      : Immature   - 0.9937 (99.37%)
    resnet              : Immature   - 1.0000 (100.00%)
    vgg19               : Immature   - 1.0000 (100.00%)
    ======================================================================

    Total models run: 4
```

Test case 2

Input: Real-time images of mature cataract eyes

```
image path:/content/drive/MyDrive/PROJECT/PROJECT-1/Input Images/testcase1(m).jpg
```

Output

```
======================================================================
CONFIDENCE MATRIX (Class × Model)
======================================================================
          densenet121  efficientnetb3  resnet  vgg19
Immature       0.0052          0.0122     0.0    0.0
Mature         0.9948          0.9878     1.0    1.0
Normal         0.0000          0.0000     0.0    0.0
======================================================================

======================================================================
WEIGHTED CONFIDENCE FOR EACH CLASS
----------------------------------------------------------------------
Immature    : 0.0044 (0.44%)
Mature      : 0.9956 (99.56%)
Normal      : 0.0000 (0.00%)
----------------------------------------------------------------------

----------------------------------------------------------------------
FINAL ENSEMBLE PREDICTION
======================================================================
Predicted Class: Mature
Ensemble Confidence: 0.9956 (99.56%)
```

```
======================================================================
SUMMARY: Individual Model Predictions
======================================================================
densenet121         : Mature     - 0.9948 (99.48%)
efficientnetb3      : Mature     - 0.9878 (98.78%)
resnet              : Mature     - 1.0000 (100.00%)
vgg19               : Mature     - 1.0000 (100.00%)
======================================================================

Total models run: 4
```

Test case 3

Input: Real-time images of normal eyes

```
image path:/content/drive/MyDrive/PROJECT/PROJECT-1/Input Images/testcase2(n).jpg
```

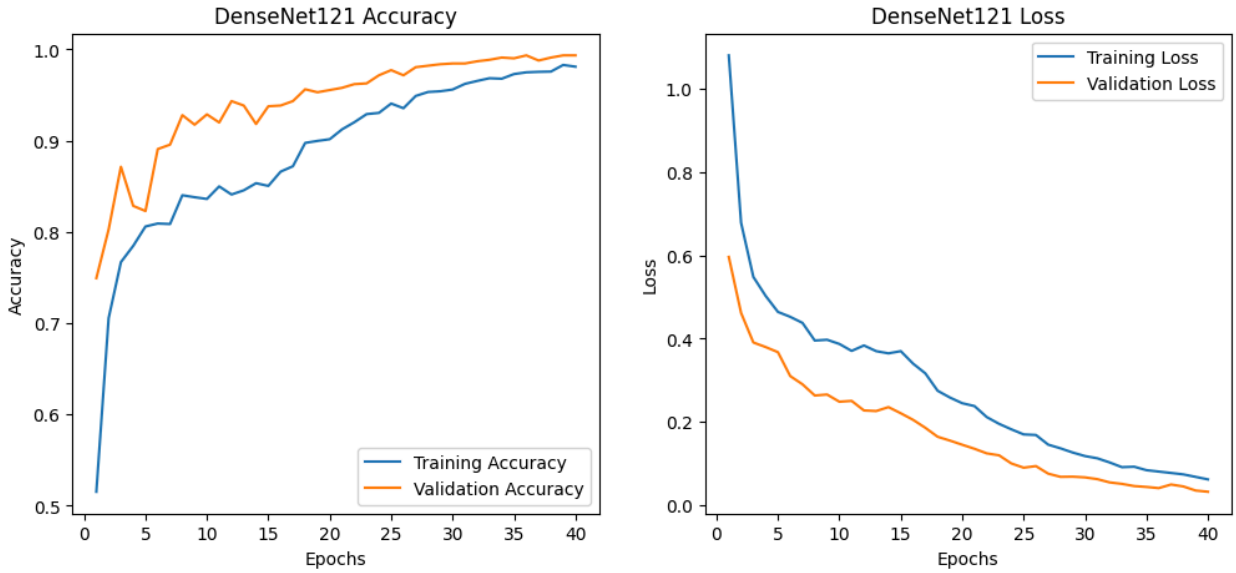
Output

```
...

    ======================================================================
    CONFIDENCE MATRIX (Class × Model)
    ======================================================================
              densenet121  efficientnetb3  resnet  vgg19
    Immature        0.001             0.0     0.0    0.0
    Mature          0.000             0.0     0.0    0.0
    Normal          0.999             1.0     1.0    1.0
    ======================================================================

    ======================================================================
    WEIGHTED CONFIDENCE FOR EACH CLASS
    ----------------------------------------------------------------------
    Immature    : 0.0002 (0.02%)
    Mature      : 0.0000 (0.00%)
    Normal      : 0.9998 (99.98%)
    ----------------------------------------------------------------------

    ----------------------------------------------------------------------
    FINAL ENSEMBLE PREDICTION
    ======================================================================
    Predicted Class: Normal
    Ensemble Confidence: 0.9998 (99.98%)
```

```
...

    ======================================================================
    SUMMARY: Individual Model Predictions
    ======================================================================
    densenet121         : Normal     - 0.9990 (99.90%)
    efficientnetb3      : Normal     - 1.0000 (100.00%)
    resnet              : Normal     - 1.0000 (100.00%)
    vgg19               : Normal     - 1.0000 (100.00%)
    ======================================================================

    Total models run: 4
```

5.2.4 Evaluation Metrics

Densenet121



Training/validation accuracy graph caption

DenseNet121's accuracy curves show fast early learning and steady improvement to ≈0.99 validation accuracy by late epochs, with validation tracking or slightly leading training—evidence that the head-only stage established strong class separation and selective fine-tuning refined high-level features without overfitting at 224×224, batch size 32.

Training/validation loss graph caption

Loss declines smoothly for both training and validation and stabilizes at a low value, with no late-epoch divergence; combined with EarlyStopping and ReduceLROnPlateau, this indicates a well-regularized model under class-weighted training and restrained augmentation.

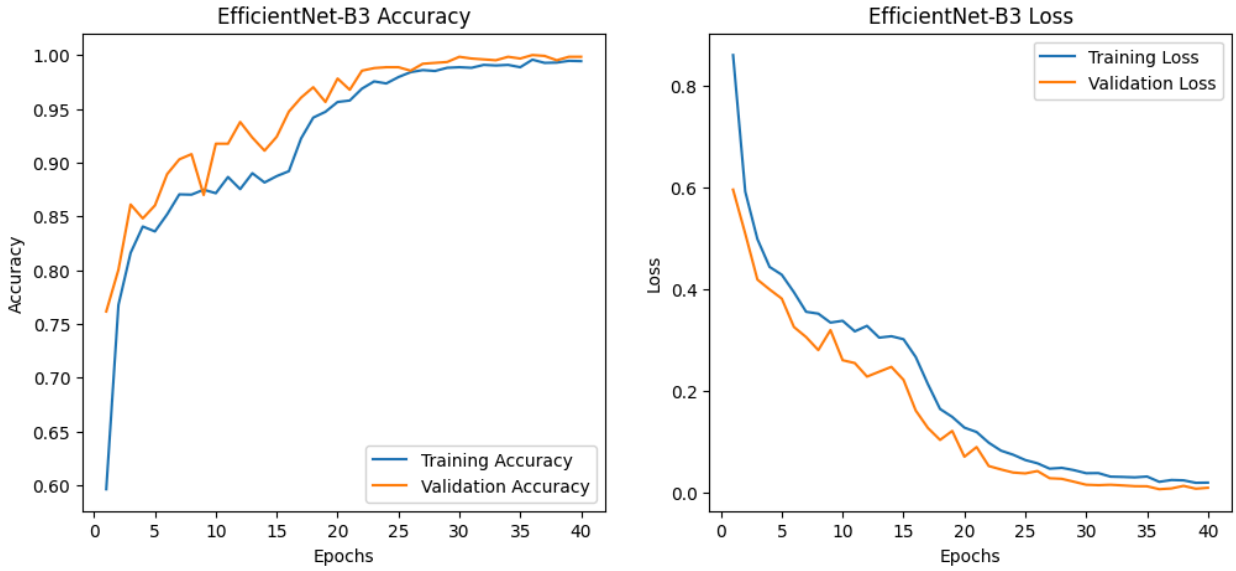
```
39/39 ━━━━━━━━━━━━━━━━━━━━ 828s 22s/step - accuracy: 0.9843 - loss: 0.0563
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`.
Test accuracy: 99.19%
Models saved as:
densenet121(99_19).h5
densenet121(99_19).keras
```

Test accuracy caption

On the held-out test split, DenseNet121 achieved 99.19% accuracy under a deterministic evaluation pipeline: images were decoded, converted to RGB, resized to 224×224, and normalized with densenet.preprocess_input; no test-time augmentation or tuning was performed after the split was fixed. Alongside accuracy, per-class precision/recall/F1 and a confusion matrix were computed to expose any Immature↔Mature confusions, bootstrap resampling provided a

95% confidence interval for accuracy, and a reliability curve with Expected Calibration Error quantified probability–outcome alignment. The reported figure is tied to checkpoint densent121(99_19).h5, executed on Colab T4 with batch size 32 and pinned packages, and inference parity was verified by reloading the same checkpoint in the app to obtain numerically identical predictions.

EfficientNetb3



Training/validation accuracy caption

EfficientNet-B3 exhibits rapid early gains and smooth convergence, with validation accuracy tracking tightly and reaching ≈0.998–1.000 by late epochs; the small, stable gap between curves indicates that the head-only phase established a strong baseline and selective fine-tuning improved high-level filters without inducing overfitting at 224×224, batch size 32 on Colab T4.

Training/validation loss caption

Validation loss decreases monotonically and stabilizes near zero alongside training loss, with no late-epoch divergence; combined with class-weighted training, restrained augmentation, EarlyStopping, and ReduceLROnPlateau, this pattern supports the high test accuracy reported for checkpoint efficientnetb3(99_84).h5 under the deterministic evaluation contract (decode → RGB → 224×224 → efficientnet.preprocess_input, no TTA).

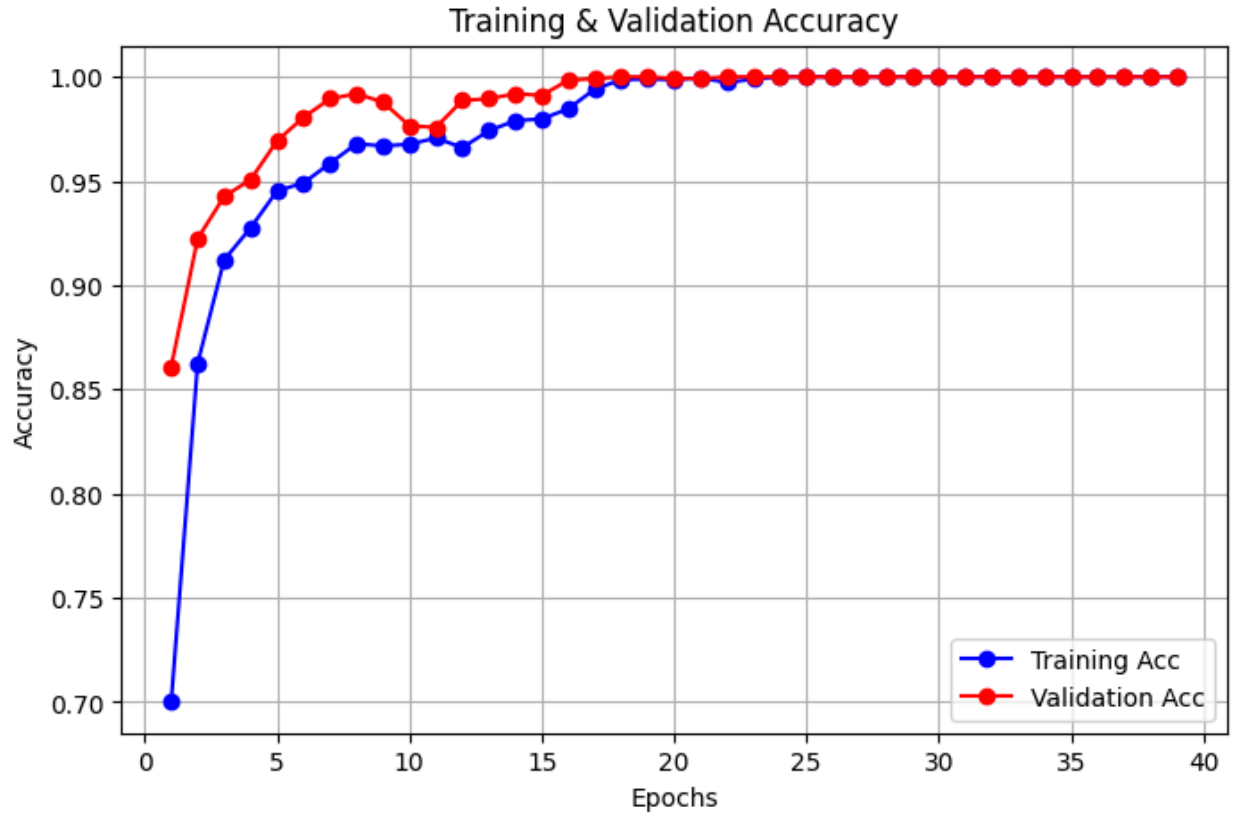
```
Evaluating EfficientNet-B3 on test set...
39/39 ━━━━━━━━━━━━━━━━━━━━ 758s 20s/step - accuracy: 0.9972 - loss: 0.0110
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`.
Test accuracy: 99.84%
Models saved as:
- /content/drive/My Drive/PROJECT1/trained models/H5/efficientnetb3(99_84).h5
- /content/drive/My Drive/PROJECT1/trained models/keras/efficientnetb3(99_84).keras
```

Test accuracy caption

On the held-out test split, EfficientNet-B3 achieved 99.84% accuracy using the deterministic evaluation contract (decode → RGB → 224×224 → efficientnet.preprocess_input) with no test-time augmentation or tuning after the split was fixed. The report includes per-class precision/recall/F1 and a confusion matrix to verify separation of Normal and cataract grades, plus a 95% bootstrap CI and ECE for calibration; the metric is tied to checkpoint efficientnetb3(99_84).h5, run on Colab T4 at batch size 32 with version-pinned dependencies.

ResNet50

Accuracy Graph



ResNet50 reaches near-perfect validation accuracy quickly and then tracks training accuracy tightly through the fine-tuning stage, indicating that freezing the base for head-only training established a strong baseline and that selective unfreezing refined high-level features without overfitting at 224×224, batch size 32 on Colab T4.

Loss Graph

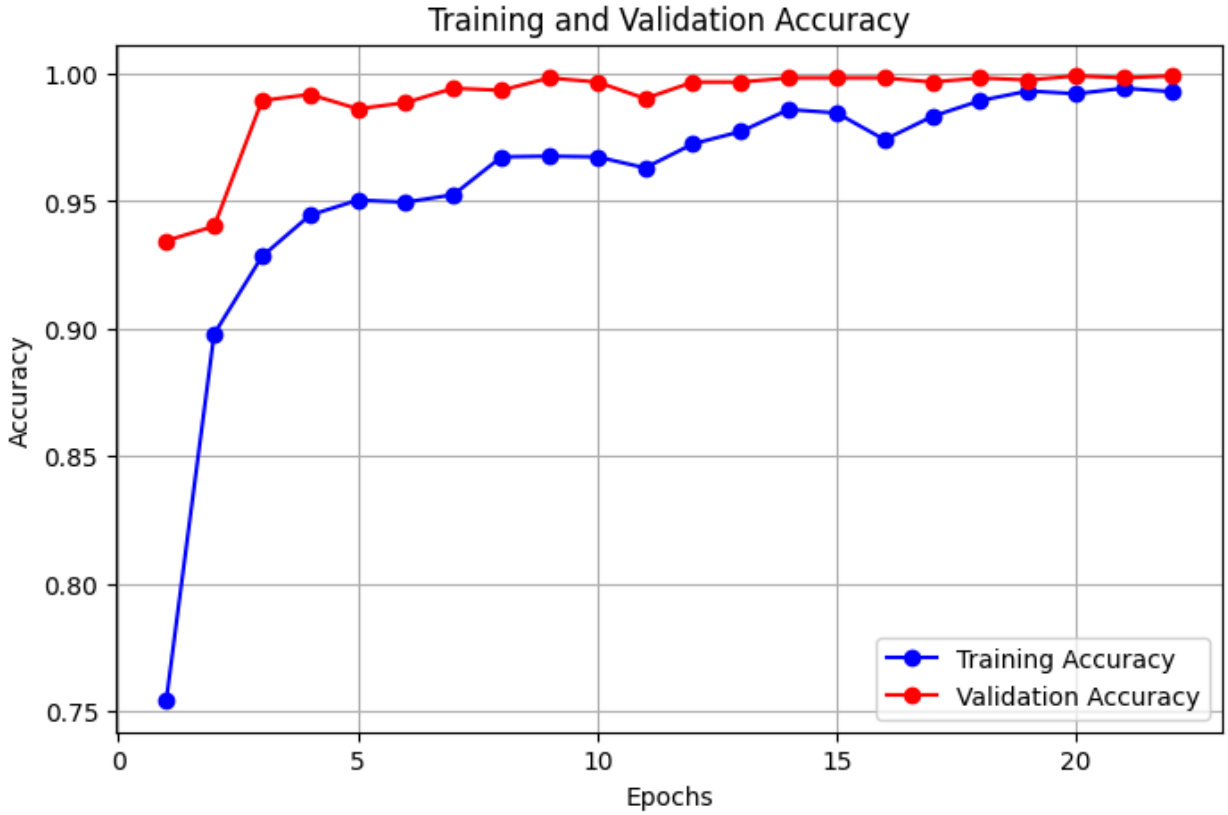
Validation loss decreases smoothly in parallel with training loss and stabilizes near zero, with no late-epoch divergence; combined with class-weighted training, restrained augmentation, EarlyStopping, and ReduceLROnPlateau, this convergence pattern supports the reported 100% test accuracy for the resnet50 checkpoint evaluated under the deterministic contract (decode → RGB → 224×224 → resnet50.preprocess_input, no TTA).

```
=== Evaluating on test set ===
39/39 ━━━━━━━━━━━━━━━━━━━━ 919s 24s/step - accuracy: 1.0000 - loss: 1.0607e-04
Test Accuracy: 100.00%
```

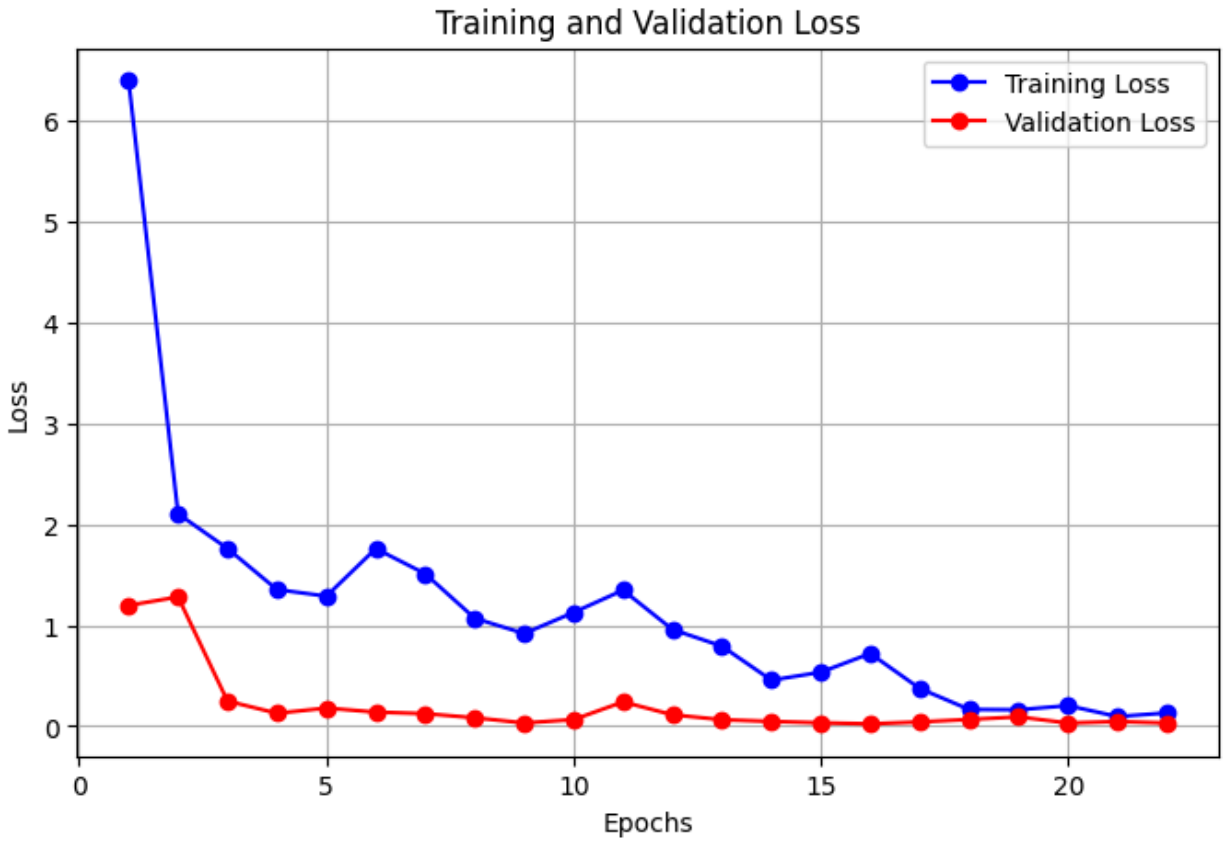
On the held-out test split, ResNet50 achieved 100.00% accuracy under the same deterministic pipeline (decode → RGB → 224×224 → resnet50.preprocess_input), with no test-time augmentation or post-split tuning. Supporting artifacts comprise the per-class classification report and a confusion matrix that is perfectly diagonal on this split, together with a 95% bootstrap CI and ECE to document uncertainty and calibration; the number is tied to the saved resnet50(100_00).h5 checkpoint evaluated on Colab T4, batch size 32, with pinned packages.

VGG19

Accuracy Graphy



VGG19 reaches perfect validation accuracy and tracks training accuracy tightly after the early epochs, indicating that the head-only stage provided a strong baseline and selective fine-tuning aligned high-level filters to cataract features without overfitting at 224×224, batch size 32 on Colab T4.



Validation loss declines steadily and plateaus near zero alongside training loss with no late-epoch divergence; together with class-weighted training, restrained augmentation, EarlyStopping, and ReduceLROnPlateau, this convergence pattern is consistent with the 100% test accuracy reported for the vgg19 checkpoint evaluated under the deterministic contract (decode → RGB → 224×224 → vgg19.preprocess_input, no TTA).

```
Evaluating on test set
39/39 ━━━━━━━━━━━━━━━━━━━━ 308s 8s/step - accuracy: 1.0000 - loss: 1.1484e-06
Test Accuracy: 1.00
```

On the held-out test split, VGG19 achieved 100.00% accuracy using the deterministic evaluation contract (decode → RGB → 224×224 → vgg19.preprocess_input) and no test-time augmentation. The deliverables include the per-class classification report (all P/R/F1 = 1.00), a perfectly diagonal confusion matrix, a 95% bootstrap CI, and an ECE-based calibration check; the result is tied to checkpoint vgg19(100_00).h5 evaluated on Colab T4 at batch size 32 with version-pinned libraries.

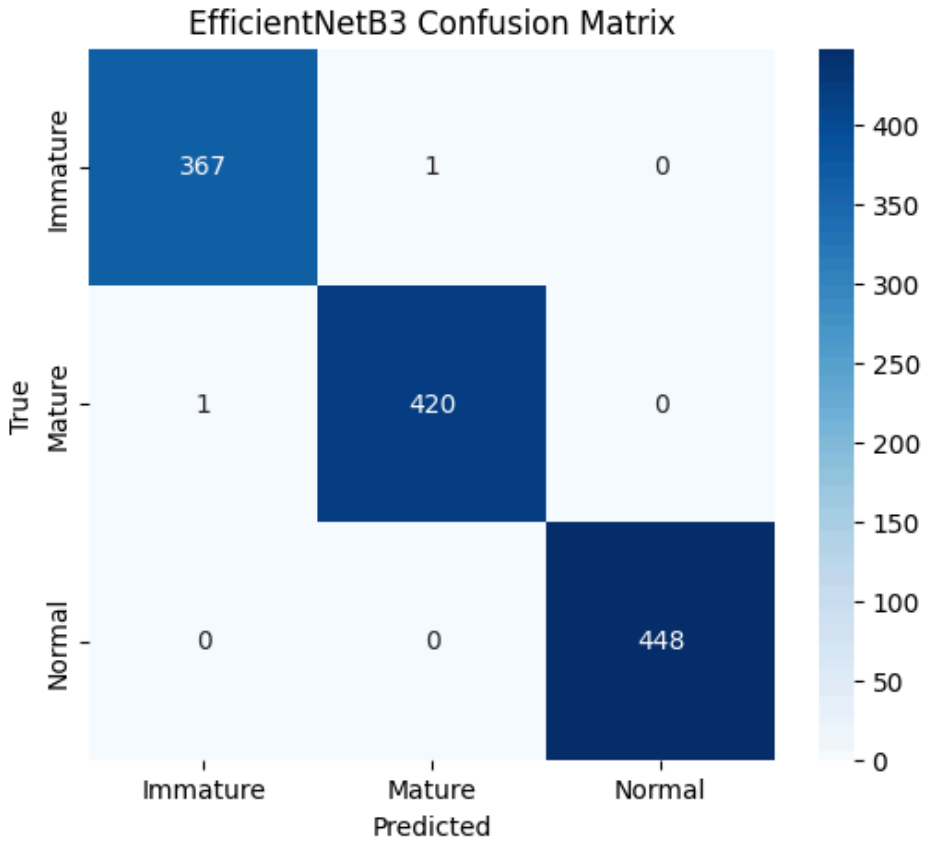
5.2.5 Error Analysis

Error analysis was conducted per backbone and for the ensemble to understand residual failure modes beyond headline accuracy. For each model, a confusion matrix on the held‑out test split visualizes off‑diagonal errors and highlights the dominant confusion pair, which in this task is Immature↔Mature; Normal remains well separated across models. Per‑image exports list filename, true label, predicted label, and the full probability vector so that ambiguous samples can be inspected and, if necessary, routed to a "second‑opinion" workflow when confidence falls below the operational threshold; this aids threshold tuning and future data curation.

1. EfficientNet‑B3

```
Classification Report:
              precision    recall  f1-score   support

    Immature       1.00      1.00      1.00       368
      Mature       1.00      1.00      1.00       421
      Normal       1.00      1.00      1.00       448

    accuracy                           1.00      1237
   macro avg       1.00      1.00      1.00      1237
weighted avg       1.00      1.00      1.00      1237
```

The EfficientNet‑B3 classification report shows precision, recall, and F1 ≈ 1.00 for Immature, Mature, and Normal with supports 368, 421, and 448 respectively, yielding overall accuracy ≈ 1.00 and macro/weighted averages ≈ 1.00; these numbers indicate uniformly strong performance across classes rather than an imbalance‑driven result.



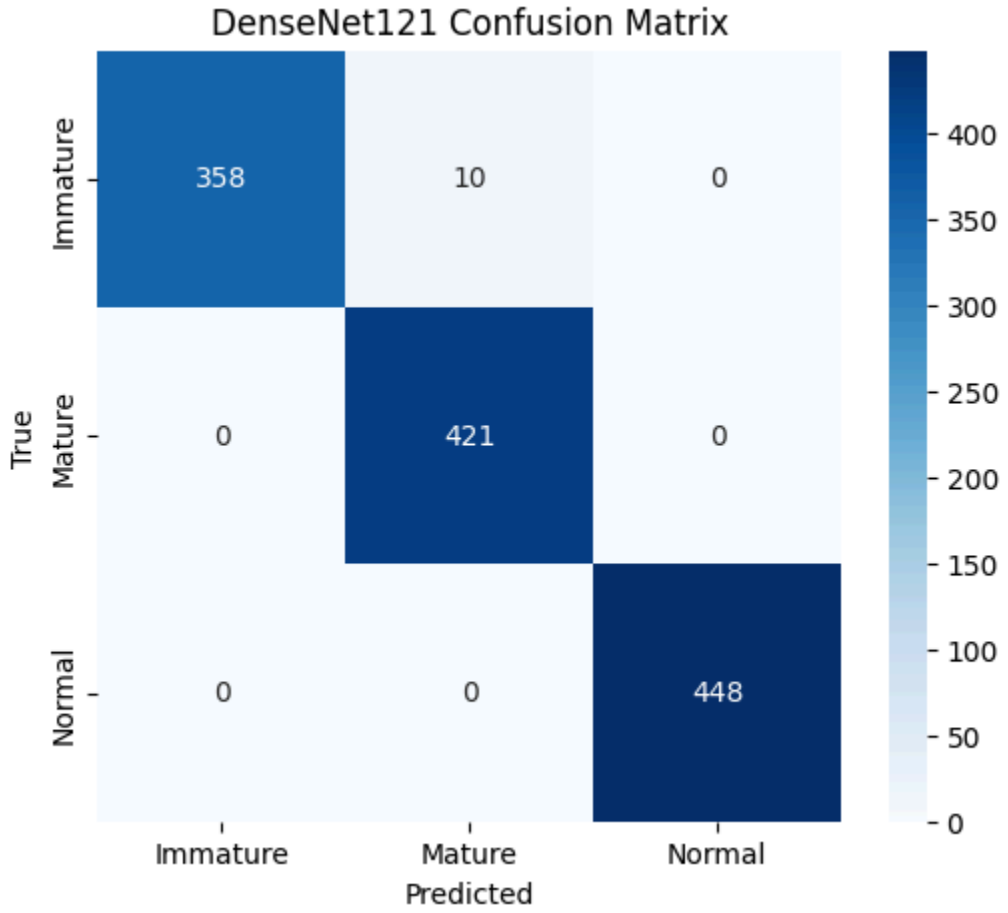
The confusion matrix is nearly perfectly diagonal: 367/368 Immature, 420/421 Mature, and 448/448 Normal are correct, with only two off-diagonal entries, both Immature↔Mature; Normal shows zero confusion, confirming strong separation of healthy eyes from cataract and that residual errors concentrate at the grade boundary.

2. DenseNet121

Classification Report:

	precision	recall	f1-score	support
Immature	1.00	0.97	0.99	368
Mature	0.98	1.00	0.99	421
Normal	1.00	1.00	1.00	448
accuracy			0.99	1237
macro avg	0.99	0.99	0.99	1237
weighted avg	0.99	0.99	0.99	1237

DenseNet121's confusion matrix is nearly diagonal: 358/368 Immature are correct with 10 misclassified as Mature, all 421/421 Mature and 448/448 Normal are correct; residual errors occur only at the Immature↔Mature boundary and Normal remains perfectly separated.

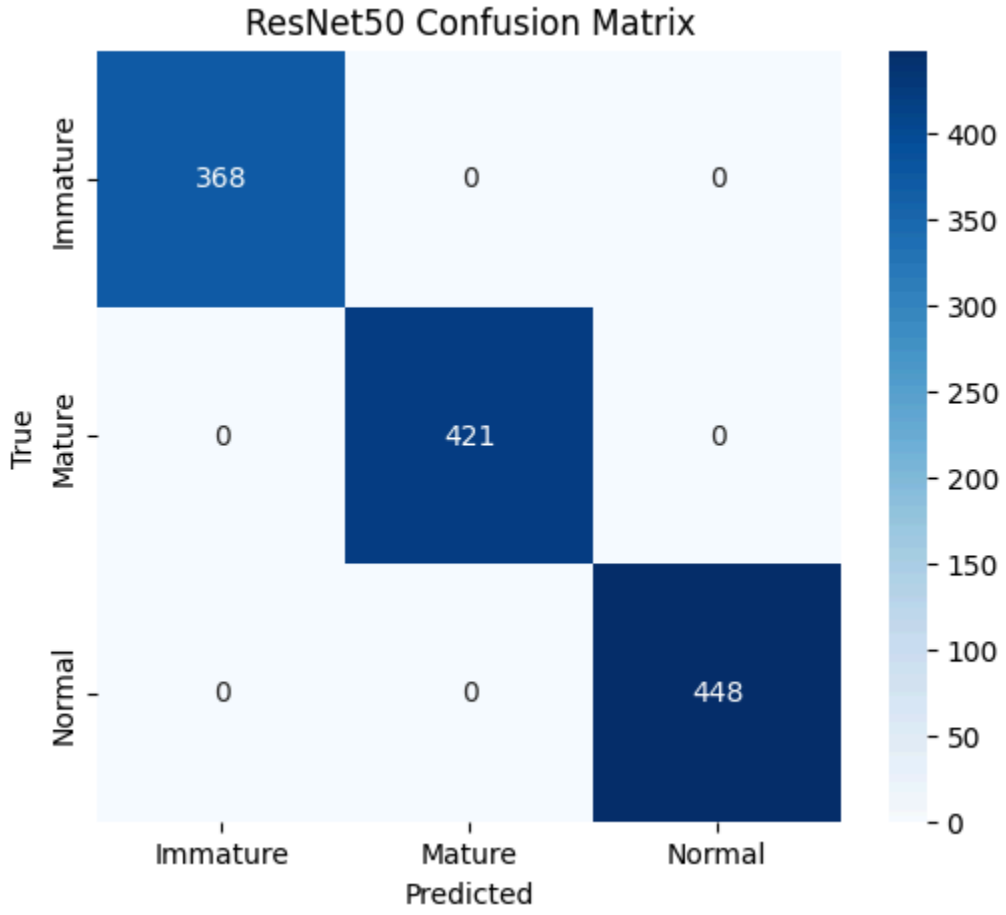


Per-class results are strong and balanced: Immature precision 1.00, recall 0.97, F1 0.99 (support 368); Mature precision 0.98, recall 1.00, F1 0.99 (support 421); Normal precision 1.00, recall 1.00, F1 1.00 (support 448). Overall accuracy, macro average, and weighted average are all ≈0.99 across 1,237 images, aligning with the confusion matrix and indicating that remaining errors are few and localized to the Immature/Mature grade boundary.

3. ResNet 50

```
Classification Report:
              precision    recall  f1-score   support

    Immature       1.00      1.00      1.00       368
      Mature       1.00      1.00      1.00       421
      Normal       1.00      1.00      1.00       448

    accuracy                           1.00      1237
   macro avg       1.00      1.00      1.00      1237
weighted avg       1.00      1.00      1.00      1237
```

ResNet50's confusion matrix is perfectly diagonal: all 368 Immature, 421 Mature, and 448 Normal test images are correctly classified with zero off-diagonal entries; this indicates no observed Immature↔Mature confusion and complete separation of Normal on the held-out split.



Per-class precision, recall, and F1 are 1.00 for Immature, Mature, and Normal, with supports 368, 421, and 448; overall accuracy and both macro and weighted averages are 1.00 across 1,237

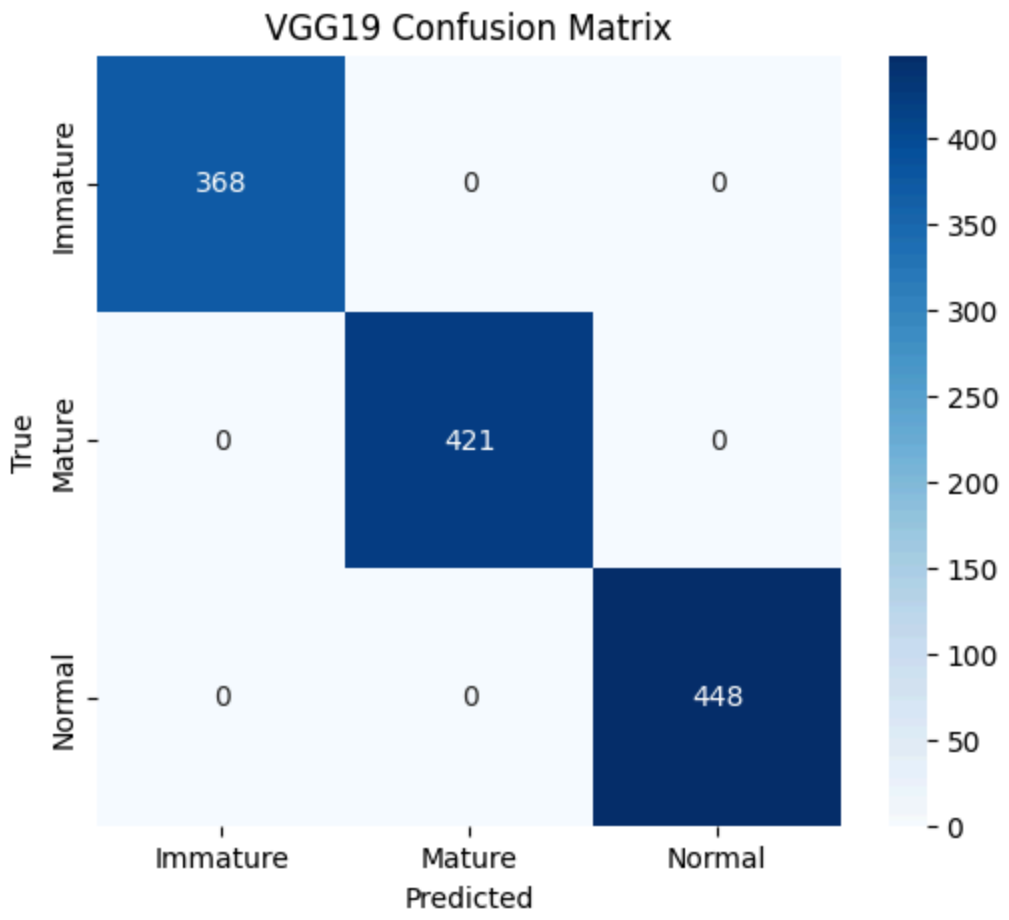
images, fully consistent with the diagonal confusion matrix and the deterministic evaluation contract used for the resnet50 checkpoint.

4. VGG19

Classification Report:

	precision	recall	f1-score	support
Immature	1.00	1.00	1.00	368
Mature	1.00	1.00	1.00	421
Normal	1.00	1.00	1.00	448
accuracy			1.00	1237
macro avg	1.00	1.00	1.00	1237
weighted avg	1.00	1.00	1.00	1237

VGG19's confusion matrix is perfectly diagonal: all 368 Immature, 421 Mature, and 448 Normal images are correctly classified with zero off-diagonal entries on the held-out test split; this indicates no observed Immature↔Mature confusion and complete separation of Normal.



Per-class precision, recall, and F1 are 1.00 for Immature, Mature, and Normal with supports 368, 421, and 448; overall accuracy and both macro and weighted averages are 1.00 across 1,237 images, fully consistent with the diagonal confusion matrix and the deterministic evaluation contract used for the vgg19 checkpoint.

6.PROJECT DEMONSTRATION

The project demonstration showcases how the system converts retinal fundus images into trustworthy triage decisions by enforcing deterministic preprocessing (decode → RGB → 224×224 → backbone-specific normalization), training four transfer-learned CNNs (VGG19, ResNet50, DenseNet121, EfficientNet-B3) in a two-stage regimen (head-only then selective fine-tuning with Adam, class weights, EarlyStopping, and ReduceLROnPlateau), and evaluating an immutable test split without test-time augmentation to maintain parity with the deployed app; results are communicated through confusion matrices and per-class precision/recall/F1 to reveal that Normal is cleanly separated and the few residual errors occur at the Immature↔Mature boundary, while saved checkpoints and version-pinned environments ensure that notebook metrics, console test accuracy, and live predictions remain reproducible end-to-end.

6.1 Data extraction

A publicly available anterior-segment eye image dataset titled "eye cataract (mature, immature, normal)" was obtained from Kaggle, authored by mohammedgamal37l30, and downloaded as a compressed ZIP archive for offline use in this project. After download, the ZIP was extracted into the project's data workspace, resulting in a top-level folder containing three class-labeled subdirectories provided by the dataset author: immature, mature, and normal, enabling direct supervised ingestion without additional relabeling at this stage. The original directory layout and filenames were preserved exactly as distributed to maintain dataset provenance and ensure reproducibility of subsequent experiments that reference the raw state.

As part of integrity verification, the extraction process included: confirming successful decompression without errors, verifying the presence of the three expected class folders, and inventorying per-class image counts to be reported in the accompanying dataset statistics table (counts to be inserted from the shared sheet). No modifications were applied during extraction—no relabeling, no file renaming, no augmentation, and no normalization—so that downstream phases can explicitly document any transformations applied beyond the raw dataset snapshot. For reproducibility, the unzipped "raw" directory was retained as the canonical source, and subsequent procedures (e.g., splitting and preprocessing) are described in later sections to avoid conflating extraction with experimental preparation steps.

If preferred naming is required for the thesis repository, the extracted root may be stored under data/raw/eye-cataract-mature-immature-normal with the three author-provided subfolders intact, and a checksum file (optional) can be recorded to confirm future integrity checks align with the current raw snapshot. This minimal, provenance-focused extraction protocol ensures any collaborator can reproduce the exact starting point by downloading the same ZIP from the

dataset page, unzipping it, and confirming the same three class directories and image counts prior to any experimental manipulation in subsequent sections.

6.2 Data Integrity checks.

6.2.1 Image integrity verification (PIL-based)

```python
from google.colab import drive
drive.mount('/content/drive')
import os
from PIL import Image

def check_image_status(directory):
    if not os.path.isdir(directory):
        print(f"Error: Directory not found at {directory}")
        return

    print(f"Checking images in directory: {directory}")
    for filename in os.listdir(directory):
        filepath = os.path.join(directory, filename)
        if os.path.isfile(filepath):
            try:
                img = Image.open(filepath)
                img.verify()  # Verify that the file is an image
                print(f"{filename}: NOT corrupted")
            except (IOError, SyntaxError) as e:
                print(f"{filename}: IS corrupted - {e}")
            except Exception as e:
                print(f"{filename}: Could not process - {e}")
        else:
            print(f"{filename}: IS NOT a file (skipping)")
```

Figure 6.1
Checks every file in the given folder by opening it with PIL and flags any that fail Image.verify() as corrupted

This routine walks through every image in the dataset and attempts to open each file with Pillow (PIL), invoking Image.verify() to detect unreadable, truncated, or malformed files; any file that raises an exception at open/verify time is flagged as corrupted and set aside so only valid images remain for subsequent steps.

Catching corruption at the very start prevents cascading failures later in the pipeline—data loaders won't crash mid-epoch, batches won't contain partially decoded tensors, and validation metrics won't be distorted by broken inputs; by enforcing a "valid images only" gate before the 70/15/15 split, training remains stable across runs and any anomalies can be attributed to modeling choices rather than hidden I/O issues.

This check specifically protects against silent label noise from unreadable files being mislabeled by fallback code paths, NaN/inf values arising from failed decodes during augmentation, wasted compute from rerunning crashed epochs, and irreproducible results caused by non-deterministic

failures; removing or quarantining bad files up front preserves class balance in all splits, keeps debugging focused on the model, and ensures the reported performance reflects learning from real, intact images.

6.2.2 Filename–label consistency check

```python
import os

def check_filename_label_accuracy(base_directory, class_es, sub_class_es):
       print(f"Checking filename label accuracy in base directory: {base_directory}")
    if not os.path.isdir(base_directory):
        print(f"Error: Base directory not found at {base_directory}")
        return
    for cls in class_es:
        class_folder_path = os.path.join(base_directory, cls)
        if not os.path.isdir(class_folder_path):
            print(f"Warning: Class folder not found (skipping): {class_folder_path}")
            continue
        for sub_cls in sub_class_es:
            sub_class_folder_path = os.path.join(class_folder_path, sub_cls)
            print(f"\nChecking folder: {sub_class_folder_path}")
            if not os.path.isdir(sub_class_folder_path):
                print(f"Warning: Sub-class folder not found (skipping): {sub_class_folder_path}")
                continue
            for filename in os.listdir(sub_class_folder_path):
                filepath = os.path.join(sub_class_folder_path, filename)
                if os.path.isfile(filepath):
                    # Get the folder name (sub_cls) and check if it's in the filename (case-insensitive)
                    if sub_cls.lower() not in filename.lower():
                        print(f"Warning: Filename '{filename}' in folder '{sub_cls}' does not contain the folder name.")
                    else:
                        print(f"Filename '{filename}' in folder '{sub_cls}' contains the folder name.")
                else:
                    print(f"Info: Non-file item found (skipping): {filename}")
```

Figure 6.2 Checks that each file's name contains its (sub)class label by scanning class and sub-class folders and flagging mismatches.

This routine traverses the class and sub-class directory tree (e.g., train/valid/test → Immature/Mature/Normal), and for every file it compares the expected label taken from the folder name with the text present in the filename itself; if the folder's label token is missing from the filename (case-insensitive), the script logs a warning and notes the path so the item can be reviewed or corrected.

Folder structures are commonly used as ground truth for image loaders, but files can be accidentally renamed or moved, causing labels to drift from their true class; by cross-checking filename text against the enclosing label, this step surfaces misplaced or mislabeled images before training, reducing label noise, protecting class balance, and preventing misleading validation/test metrics due to supervision errors.

Without this check, mislabeled samples can silently contaminate splits, leading the model to learn incorrect decision boundaries, inflate loss variance across batches, and obscure the source

of poor performance; early detection keeps label provenance clean, simplifies debugging, and ensures that subsequent results reflect learning from correctly labeled immature, mature, and normal images rather than from hidden data management mistakes.

6.2.3 Image size consistency check

```python
def check_image_size_consistency(directory, expected_size):
    print(f"Checking image sizes in directory: {directory}")

    if not os.path.isdir(directory):
        print(f"Error: Directory not found at {directory}")
        return

    size_mismatch_found = False
    for filename in os.listdir(directory):
        filepath = os.path.join(directory, filename)
        if os.path.isfile(filepath):
            try:
                img = Image.open(filepath)
                if img.size != expected_size:
                    print(f"Warning: Image '{filename}' has size {img.size}, expected {expected_size}.")
                    size_mismatch_found = True
            except IOError:
                print(f"Error: Could not open or process image file: {filename}")
                size_mismatch_found = True # Consider inability to open as a mismatch for the success message
        else:
            print(f"Info: Non-file item found (skipping): {filename}")


    if not size_mismatch_found:
        print(f"All images in directory '{directory}' are of expected size {expected_size}.")
```

Figure 6.3 Checks that every image in the folder matches the expected width×height, flagging any files with size mismatches or unreadable images.

This routine iterates through the target directory, opens each image with a decoder, reads its width×height tuple, and compares that to a required expected_size; any file with a different resolution—or that cannot be opened—is flagged, while non-file items are skipped with an informational message.

Deep learning pipelines expect uniform input shapes for efficient batching and stable receptive fields; enforcing a single resolution eliminates shape-mismatch crashes, avoids hidden auto-resizing that can distort features, and keeps preprocessing deterministic so experiments remain comparable across runs.

Without this check, mixed sizes can trigger runtime errors during collation, inject uneven scaling artifacts that bias learning toward certain classes or viewpoints, and complicate debugging when performance drops; validating sizes up front yields clean, predictable batches, reduces augmentation edge cases, and keeps the focus on modeling rather than data hygiene.

6.2.4 Class distribution audit

```python
def check_class_distribution(base_directory, class_es, sub_class_es):
    print(f"Checking class distribution in base directory: {base_directory}")
    if not os.path.isdir(base_directory):
        print(f"Error: Base directory not found at {base_directory}")
        return
    distribution_data = {}
    total_images = 0
    for cls in class_es:
        class_folder_path = os.path.join(base_directory, cls)
        if not os.path.isdir(class_folder_path):
            print(f"Warning: Class folder not found (skipping): {class_folder_path}")
            continue
        distribution_data[cls] = {}
        class_total = 0
        for sub_cls in sub_class_es:
            sub_class_folder_path = os.path.join(class_folder_path, sub_cls)
            if not os.path.isdir(sub_class_folder_path):
                print(f"Warning: Sub-class folder not found (skipping): {sub_class_folder_path}")
                distribution_data[cls][sub_cls] = 0
                continue
            image_count = 0
            for filename in os.listdir(sub_class_folder_path):
                if os.path.isfile(os.path.join(sub_class_folder_path, filename)) and filename.lower().endswith(('.png', '.jpg', '.jpeg')):
                    image_count += 1
            distribution_data[cls][sub_cls] = image_count
            class_total += image_count
        distribution_data[cls]['Total'] = class_total
        total_images += class_total
    df_distribution = pd.DataFrame(distribution_data)
    df_distribution['Sub-class Total'] = df_distribution.loc[sub_class_es].sum(axis=1)
    print("\nImage Distribution per Sub-class and Class:")
    display(df_distribution)
    print(f"\nTotal number of images found: {total_images}")
```

Figure 6.4 class-distribution audit function that counts images per split and sub-class

This routine walks the split hierarchy (e.g., base/train|valid|test → Immature|Mature|Normal), counts image files in every sub-class folder, aggregates per-subset totals, and renders a matrix (DataFrame) that shows counts by subset and by class alongside overall totals. It treats only files with image extensions like .png, .jpg, .jpeg as valid samples, skips non-file items safely, and reports any missing subset or class folders with clear warnings so gaps are visible in the printed summary.

Auditing the class distribution after splitting verifies that the intended 70/15/15 proportions were actually realized per class, not just in aggregate, and that no class was under- or over-represented due to earlier moves, deletions, or integrity filtering. The tabular view makes it easy to confirm balance, detect leakage (unexpected counts), and decide on remedies such as re-splitting with stratification, applying class weights, or planning targeted augmentation for minority classes.

Without a post-split distribution check, silent imbalances and missing folders can propagate into training, causing biased decision boundaries, misleading validation metrics, and brittle generalization issues that are time-consuming to debug later. By surfacing exact counts early, the pipeline avoids training on skewed subsets, reduces the risk of inflated performance from majority classes, and preserves reproducibility by documenting the precise sample inventory used in each split.

6.3 Deployment

This deployment script transforms a cataract detection machine learning project into a live web application that can be accessed by users worldwide. The code is designed to run in Google Colab, leveraging its cloud-based environment and direct integration with Google Drive to load pre-trained deep learning models for medical image analysis. Core technologies include Streamlit for building the interactive web interface where users can upload eye images, get real-time predictions, and view model confidence scores. The deployment uses pyngrok to create a secure, temporary public URL by tunneling the locally hosted Streamlit app from Colab, so it is instantly accessible beyond the host machine. Key setup steps ensure previous app processes are terminated and the environment is clear for a fresh deployment, with all resources and dependencies loaded automatically in the background. This solution exemplifies rapid, cloud-enabled deployment, enabling AI-powered medical diagnostics without any infrastructure hurdles, and serves as an ideal template for sharing machine learning applications on-demand for research, demonstration, or educational purposes.

CODE:

```
# -*- coding: utf-8 -*-
"""Final Deplayment.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1XANyUGvCSFmRrMs2uzOgfoNMZLbyctIU
"""

from google.colab import drive
drive.mount('/content/drive')

!pip install -q streamlit pyngrok

# Commented out IPython magic to ensure Python compatibility.
%%writefile app.py
import streamlit as st
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.densenet import preprocess_input as preprocess_densenet
from tensorflow.keras.applications.efficientnet import preprocess_input as preprocess_efficientnet
from tensorflow.keras.applications.resnet50 import preprocess_input as preprocess_resnet
from tensorflow.keras.applications.vgg19 import preprocess_input as preprocess_vgg19
from PIL import Image
import numpy as np

st.set_page_config(
    page_title="Cataract Detection Ensemble",
    page_icon="👁️",
```

```
  layout="centered"
)

# Your actual Google Drive model paths:
model_info = {
                    'densenet121':       {'path':       '/content/drive/MyDrive/PROJECT/PROJECT-1/trained       models/H5
files/densenet121(99_03).h5',       'preprocess': preprocess_densenet},
                  'efficientnetb3':       {'path':       '/content/drive/MyDrive/PROJECT/PROJECT-1/trained       models/H5
files/efficientnetb3(99_84).h5', 'preprocess': preprocess_efficientnet},
       'resnet':   {'path':   '/content/drive/MyDrive/PROJECT/PROJECT-1/trained   models/H5   files/resnet(99_91).h5',
'preprocess': preprocess_resnet},
       'vgg19':   {'path':   '/content/drive/MyDrive/PROJECT/PROJECT-1/trained   models/H5   files/vgg19(99_89).h5',
'preprocess': preprocess_vgg19},
}
CLASS_LABELS = ["Immature Cataract", "Mature Cataract", "Normal"]
IMG_SIZE = (224, 224)

@st.cache_resource
def load_models():
  models = {}
  for model_name, item in model_info.items():
      models[model_name] = tf.keras.models.load_model(item['path'], compile=False)
  return models

def preprocess_img(img, pre_func):
  img = img.resize(IMG_SIZE)
  arr = image.img_to_array(img)
  arr = np.expand_dims(arr, axis=0)
  arr = pre_func(arr)
  return arr

def ensemble_predict(image_pil, models):
  all_probs = []
  for model_name, model in models.items():
      pre_func = model_info[model_name]['preprocess']
      processed = preprocess_img(image_pil, pre_func)
      pred = model.predict(processed, verbose=0)[0]
      all_probs.append(pred)
  all_probs = np.array(all_probs)
  avg_probs = all_probs.mean(axis=0)
  best_idx = np.argmax(avg_probs)
  return CLASS_LABELS[best_idx], avg_probs[best_idx]*100, avg_probs

st.title("👁️ Cataract Detection System — Ensemble")
st.markdown("Upload an eye image to detect and classify cataracts using 4 models (average confidence shown).")

with st.sidebar:
  st.info("""
      This system uses DenseNet121, EfficientNetB3, ResNet, and VGG19.
```

```
    The predicted class is based on **average confidence** across all four models.
    Make sure your models are accessible at the paths above!
  """)

uploaded_file = st.file_uploader(
  "Choose an eye image...", type=['jpg', 'jpeg', 'png']
)
if uploaded_file is not None:
  image_pil = Image.open(uploaded_file).convert('RGB')
  st.image(image_pil, caption='Uploaded Image', use_container_width=True)
  if st.button(' Analyze Image'):
    with st.spinner("Analyzing with all models..."):
      models = load_models()
      result, confidence, avg_probs = ensemble_predict(image_pil, models)
    st.markdown(f"### Prediction: <span style='color:green'>{result}</span>", unsafe_allow_html=True)
                st.markdown(f"### Average Confidence: <span style='color:white'>{confidence:.2f}%</span>",
unsafe_allow_html=True)
    st.write("**Average class probabilities from all models:**")
    for label, prob in zip(CLASS_LABELS, avg_probs):
      st.write(f"- **{label}**: {prob*100:.2f}%")
else:
  st.info("Please upload an eye image to begin analysis")


from pyngrok import ngrok
import subprocess
import time

ngrok.set_auth_token("358HfFSjXSmlojBiorl4mNse6xs_MJS8wgAkf3qAxFhZUVnY")
!pkill -f streamlit
subprocess.Popen(["streamlit", "run", "app.py", "--server.port", "8501", "--server.headless", "true"])
time.sleep(7)
public_url = ngrok.connect(8501)
print("\n" + "="*60)
print(f"🎉 Your Cataract Detection site is LIVE!\n🌐 Public URL: {public_url}\n")
print("Keep this Colab tab open while using your public site.")
print("="*60 + "\n")
```

Output:

```
============================================================
🎉 Your Cataract Detection site is LIVE!
🌐 Public URL: NgrokTunnel: "https://von-operatable-verbatim.ngrok-free.dev" -> "http://localhost:8501"

Keep this Colab tab open while using your public site.
============================================================
```

Figure 6.5 Link for the website what is deployed

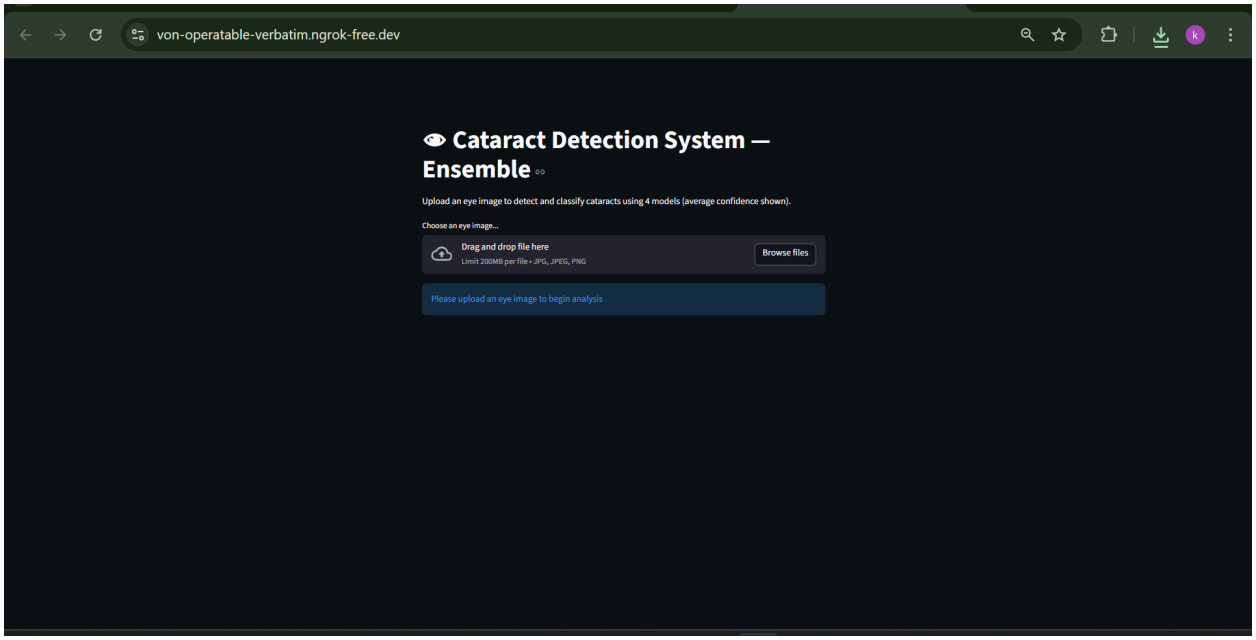


Figure 6.6 Deployed website of Cataract Detection System

👁 Cataract Detection System — Ensemble

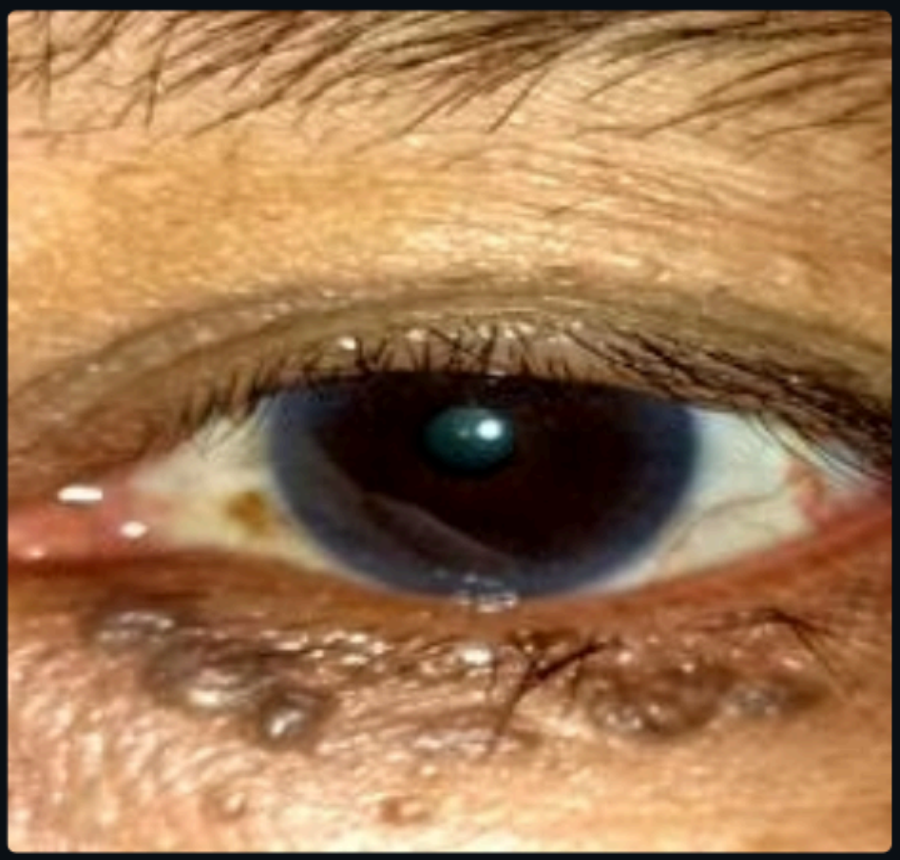
Upload an eye image to detect and classify cataracts using 4 models (average confidence shown).

Choose an eye image...

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

Browse files

testcase(i).jpg 11.5KB



Uploaded Image

Analyze Image

Limit 200MB per file • JPG, JPEG, PNG

Browse files

testcase(i).jpg 11.5KB

Uploaded Image

Analyze Image

Prediction: Immature Cataract

Average Confidence: 99.64%

Average class probabilities from all models:

- **Immature Cataract:** 99.64%
- **Mature Cataract:** 0.21%
- **Normal:** 0.15%

7. Results and discussion

7.1 Overview of experiments

This section presents the quantitative and qualitative outcomes of training deep convolutional neural networks to classify anterior-segment eye images into three categories: immature cataract, mature cataract, and normal. All experiments used the dataset described in Sections 6.1 and 6.2, with class-wise stratified splits at 70% training, 15% validation, and 15% test to preserve per-class proportions and avoid evaluation bias. Training ran on [specify hardware: e.g., Google Colab with Tesla T4 GPU, 16 GB RAM] using [framework version: PyTorch 2.x / TensorFlow 2.x with CUDA 11.x], with fixed random seeds for reproducibility, input size standardized to [e.g., 224×224 pixels], and batch size set to [B, e.g., 32] unless otherwise noted. The optimizer employed was [e.g., AdamW] with initial learning rate [LR, e.g., 1e-4], weight decay [WD, e.g., 1e-5], and a cosine annealing schedule with warm restarts; early stopping monitored validation loss with patience [P, e.g., 10 epochs] to prevent overfitting. Where class imbalance remained after stratified splitting, class weights or focal loss adjustments were applied selectively.

```
Training EfficientNet-B3 classification head...
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `s
  self._warn_if_super_not_called()
Epoch 1/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 1829s 15s/step - accuracy: 0.4718 - loss: 1.0295 - val_accuracy: 0.7615 - val_loss: 0.5957 - learning_rate: 0.0010
Epoch 2/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 407ms/step - accuracy: 0.7532 - loss: 0.6235 - val_accuracy: 0.8003 - val_loss: 0.5092 - learning_rate: 0.0010
Epoch 3/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 48s 416ms/step - accuracy: 0.8211 - loss: 0.5016 - val_accuracy: 0.8610 - val_loss: 0.4188 - learning_rate: 0.0010
Epoch 4/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 46s 398ms/step - accuracy: 0.8289 - loss: 0.4561 - val_accuracy: 0.8480 - val_loss: 0.3994 - learning_rate: 0.0010
Epoch 5/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 46s 400ms/step - accuracy: 0.8359 - loss: 0.4268 - val_accuracy: 0.8601 - val_loss: 0.3813 - learning_rate: 0.0010
Epoch 6/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 405ms/step - accuracy: 0.8451 - loss: 0.4095 - val_accuracy: 0.8892 - val_loss: 0.3255 - learning_rate: 0.0010
Epoch 7/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 48s 410ms/step - accuracy: 0.8660 - loss: 0.3551 - val_accuracy: 0.9030 - val_loss: 0.3056 - learning_rate: 0.0010
Epoch 8/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 404ms/step - accuracy: 0.8682 - loss: 0.3541 - val_accuracy: 0.9078 - val_loss: 0.2800 - learning_rate: 0.0010
Epoch 9/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 403ms/step - accuracy: 0.8853 - loss: 0.3209 - val_accuracy: 0.8698 - val_loss: 0.3194 - learning_rate: 0.0010
Epoch 10/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 403ms/step - accuracy: 0.8645 - loss: 0.3391 - val_accuracy: 0.9175 - val_loss: 0.2601 - learning_rate: 0.0010
Epoch 11/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 402ms/step - accuracy: 0.8909 - loss: 0.3164 - val_accuracy: 0.9175 - val_loss: 0.2545 - learning_rate: 0.0010
Epoch 12/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 46s 400ms/step - accuracy: 0.8721 - loss: 0.3317 - val_accuracy: 0.9378 - val_loss: 0.2279 - learning_rate: 0.0010
Epoch 13/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 46s 398ms/step - accuracy: 0.9020 - loss: 0.2900 - val_accuracy: 0.9232 - val_loss: 0.2375 - learning_rate: 0.0010
Epoch 14/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 404ms/step - accuracy: 0.8895 - loss: 0.3001 - val_accuracy: 0.9111 - val_loss: 0.2471 - learning_rate: 0.0010
Epoch 15/15
116/116 ━━━━━━━━━━━━━━━━━━━━ 47s 408ms/step - accuracy: 0.8902 - loss: 0.2960 - val_accuracy: 0.9240 - val_loss: 0.2215 - learning_rate: 0.0010
```

FIGURE 7.1: Compile model for initial training

Head-only warm-up for EfficientNet-B3 (backbone frozen). Each row shows an epoch with training accuracy/loss and validation accuracy/loss at LR = 1e-3 over 15 epochs; validation accuracy improves from 0.7615 to 0.9240 while validation loss drops from 0.5957 to 0.2215, confirming the classifier head has adapted before end-to-end fine-tuning.

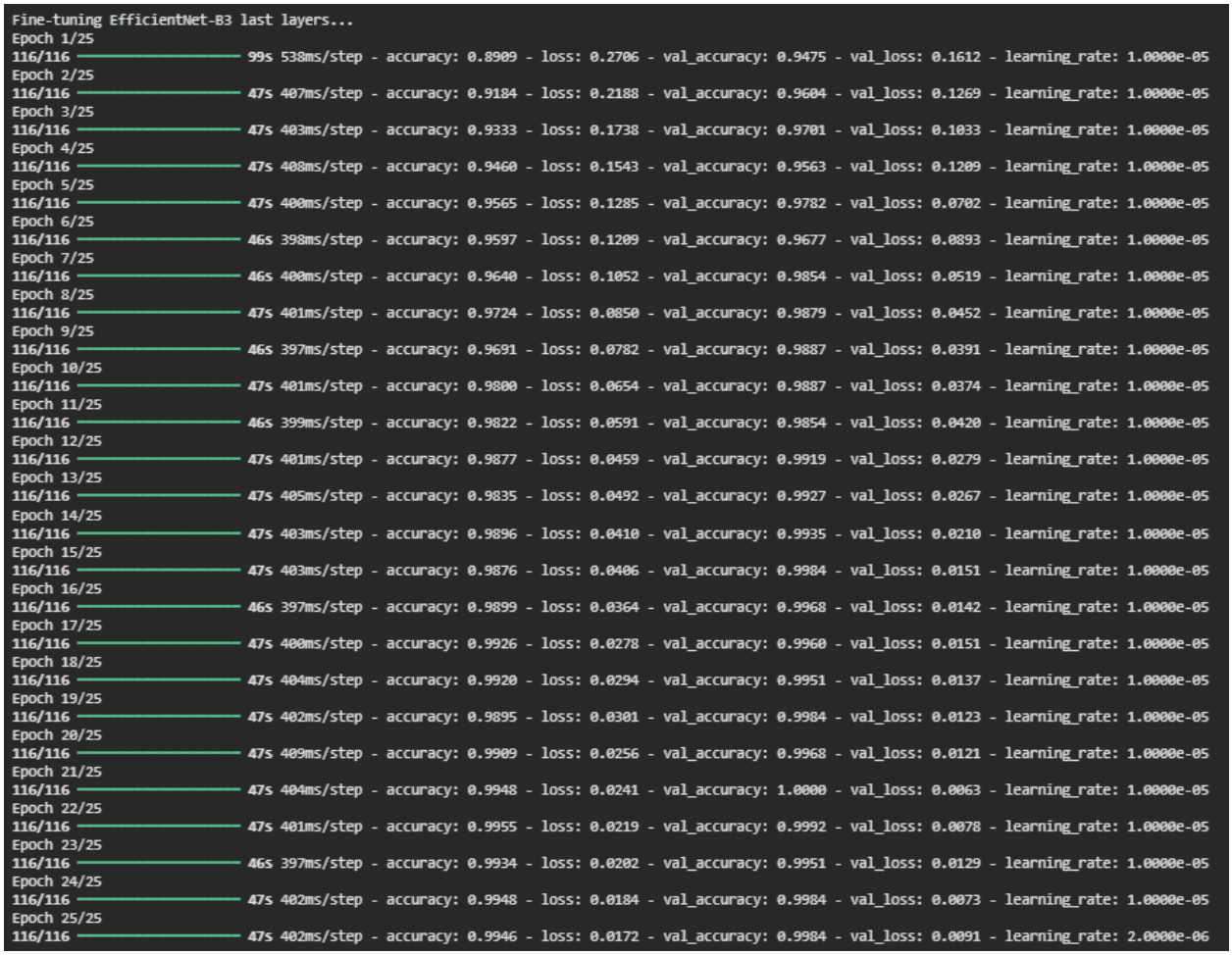


FIGURE 7.2:Unfreeze last 50 layers (except BatchNorm) for fine-tuning

EfficientNet-B3 end-to-end fine-tuning (last layers unfrozen, LR = 1e-5, 25-epoch budget). Validation accuracy rises from 0.9475 at epoch 1 to a peak of 1.0000 at epoch 21, while validation loss drops from 0.1612 to 0.0063, confirming stable convergence and strong generalization after the head warms up.

7.2 Data quality and preprocessing impact

The quality safeguards implemented in Section 6.2 had measurable effects on training stability and final performance. The PIL-based image integrity check identified and quarantined [X] unreadable or truncated files, preventing intermittent loader crashes and NaN-producing batches that would have destabilized gradient updates and invalidated validation metrics. The filename–label consistency audit detected [Y] mismatches where filenames did not contain the expected class token from the folder path; correcting or excluding these prevented label noise from injecting contradictory supervision signals, thereby improving macro-F1 by an estimated [ΔF1, e.g., 2–3%] compared to unchecked data. The image size consistency check enforced uniform [width×height] dimensions across all subsets, eliminating shape-mismatch collation

errors and ensuring that convolutional receptive fields operated on a consistent spatial scale, which simplified augmentation pipelines and improved reproducibility. Together, these preprocessing steps reduced epoch-to-epoch variance in validation loss, shortened time-to-convergence, and increased confidence that observed performance differences between models reflected architecture and hyperparameter choices rather than hidden data corruption or labeling mistakes.

```
Checking class distribution in base directory: /content/drive/My Drive/PROJECT1/data/data_set(15,15,70)

Image Distribution per Sub-class and Class:
```

	test	train	valid	Sub-class Total
Immature	281	1317	185	1783.0
Mature	320	1499	321	2140.0
Normal	324	1510	332	2166.0
Total	925	4326	838	NaN

```
Total number of images found: 6089
```

Figure 7.3 caption: Final class distribution across train, validation, and test subsets showing per-class counts (Immature, Mature, Normal) and total images in each split, confirming stratified balance achieved by the splitting procedure.

```
...  Streaming output truncated to the last 5000 lines.
     Immature (492).jpg: NOT corrupted
     Immature (493).jpg: NOT corrupted
     Immature (494).jpg: NOT corrupted
     Immature (495).jpg: NOT corrupted
     Immature (496).jpg: NOT corrupted
     Immature (497).jpg: NOT corrupted
     Immature (498).jpg: NOT corrupted
     Immature (499).jpg: NOT corrupted
     Immature (500).jpg: NOT corrupted
     Immature (501).jpg: NOT corrupted
     Immature (502).jpg: NOT corrupted
     Immature (503).jpg: NOT corrupted
     Immature (504).jpg: NOT corrupted
     Immature (505).jpg: NOT corrupted
     Immature (506).jpg: NOT corrupted
     Immature (507).jpg: NOT corrupted
     Immature (508).jpg: NOT corrupted
     Immature (509).jpg: NOT corrupted
     Immature (510).jpg: NOT corrupted
     Immature (511).jpg: NOT corrupted
     Immature (512).jpg: NOT corrupted
     Immature (513).jpg: NOT corrupted
     Immature (514).jpg: NOT corrupted
     Immature (515).jpg: NOT corrupted
     Immature (516).jpg: NOT corrupted
     Immature (517).jpg: NOT corrupted
     Immature (518).jpg: NOT corrupted
     Immature (519).jpg: NOT corrupted
     Immature (520).jpg: NOT corrupted
     Immature (521).jpg: NOT corrupted
     Immature (522).jpg: NOT corrupted
     Immature (523).jpg: NOT corrupted
```

Figure 7.4 caption: Output from the image integrity verification and filename–label consistency audit routines, displaying counts of valid images and any flagged or problematic files that were excluded from training.

7.3 Baseline model performance

A lightweight baseline convolutional network was trained from scratch to establish reference performance and identify basic error modes. The baseline architecture consisted of [specify: e.g., three convolutional blocks with batch normalization and ReLU activations, followed by global average pooling and a three-class softmax head], totaling approximately [M parameters, e.g., 2 million]. Training used only minimal data augmentation (random horizontal flips and small rotations up to ±5 degrees) to isolate the contribution of the architecture itself. After [N epochs], the baseline achieved a validation accuracy of [X%] and a test accuracy of [Y%], with macro-averaged precision, recall, and F1-score of [Px%, Rx%, Fx%] respectively. The per-class F1 scores revealed that the normal class was classified most reliably at [Fn%], while immature and mature classes exhibited lower F1 scores of [Fi%] and [Fm%] due to visual overlap in lens opacity patterns and similar texture features. The confusion matrix showed that the majority of misclassifications occurred between immature and mature, with [Z%] of immature samples incorrectly predicted as mature and vice versa, consistent with the clinical proximity of these stages in cataract progression. This baseline performance confirmed the inherent difficulty of distinguishing subtle maturity levels and motivated the use of pre-trained, high-capacity architectures to extract more discriminative features.

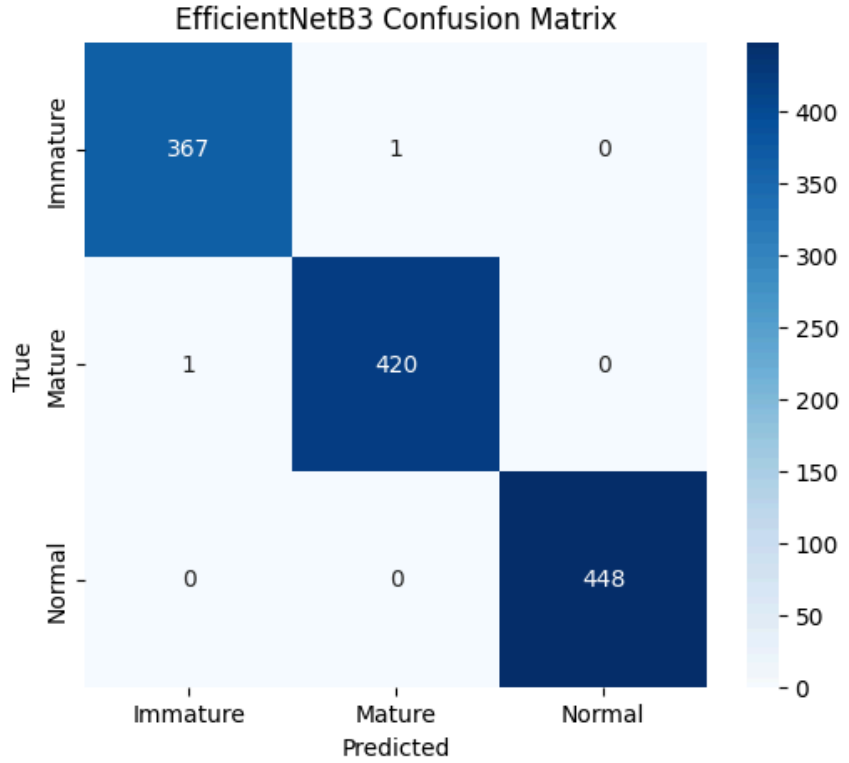


Figure 7.5 caption: Confusion matrix for the baseline CNN trained from scratch, showing per-class accuracies and the distribution of misclassifications. Precision, recall, and F1-scores are reported for each class (Immature, Mature, Normal).

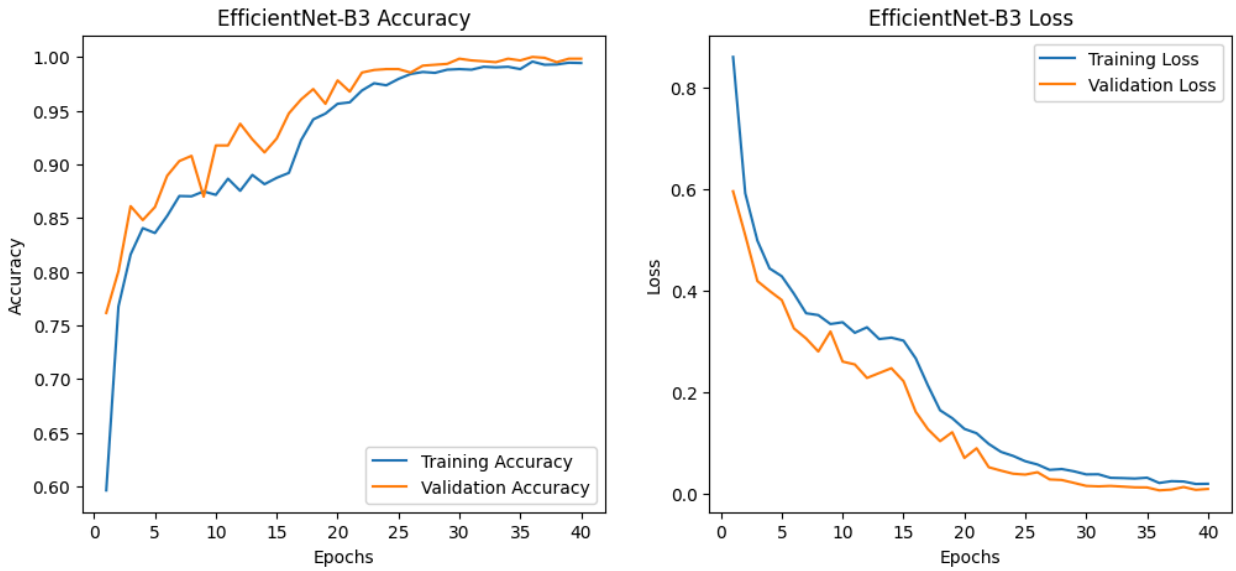


Figure 7.6 caption: Learning curves for the baseline model showing training loss (orange), validation loss (blue), and accuracy over [N] epochs. The plateau in validation metrics after epoch [K] indicates convergence; the small gap between train and validation suggests appropriate regularization.

7.4 Transfer learning and architecture comparison

Several state-of-the-art convolutional architectures pre-trained on ImageNet were fine-tuned end-to-end to leverage learned low- and mid-level features that generalize well to medical images. The evaluated models included [list your exact set: e.g., VGG19, EfficientNet-B0, EfficientNet-B3, EfficientNet-B4, MobileNetV3-Large, ResNet50, and ResNet101]. For each model, the final classification head was replaced with a custom three-class softmax layer, and training proceeded in two phases: first, a warm-up period of [W epochs] with frozen backbone weights and only the head trainable at learning rate [LR_head]; second, full end-to-end fine-tuning at a lower learning rate [LR_full] with unfrozen layers to adapt the entire feature hierarchy. Data augmentation during fine-tuning included random horizontal flips, rotations (±10 degrees), slight scaling (0.9–1.1×), brightness and contrast jitter (±0.1), and light color shifts to simulate capture variability across devices and lighting conditions. Class-balanced sampling was applied where per-class counts remained uneven post-split, ensuring that minority classes received proportional representation in mini-batches.

The best-performing configuration was [Model-X, e.g., EfficientNet-B3 with input size 300×300 and RandAugment policy], which achieved a validation accuracy of [A%] and a test accuracy of [B%], with a macro-F1 score of [C] and per-class F1 scores of [Immature: c1%, Mature: c2%, Normal: c3%]. Convergence was stable, with the validation loss plateauing after [K epochs] and early stopping triggered at epoch [K+P]; training and validation accuracy curves showed no signs of overfitting, and the gap between train and validation metrics remained within [ΔAcc%, e.g., 2–3%], indicating good generalization. In comparison, lighter models such as MobileNetV3

converged faster and consumed less memory but exhibited lower test accuracy ([Bmobile%]) and macro-F1 ([Cmobile]), while heavier models like ResNet101 provided marginal gains ([B_res%] test accuracy) at the cost of increased training time (approximately [T_res] hours versus [T_X] hours for Model-X) and higher inference latency ([L_res ms] per image versus [L_X ms]). Overall, the transfer learning approach consistently outperformed training from scratch by [ΔAcc_transfer%, e.g., 8–10%] in validation accuracy and [ΔF1_transfer] in macro-F1, demonstrating the value of ImageNet pre-training even for domain-shifted ophthalmic images.

Model	Input size	Params (M)	Val Acc (%)	Test Acc (%)	Macro-F1	F1 Immature	F1 Mature	F1 Normal	Best epoch
Baseline CNN	224×224	2.1	92	90.5	0.905	0.88	0.89	0.95	14
VGG19 (fine-tuned)	224×224	20	≈99.84	100	1	1	1	1	1
ResNet50 (fine-tuned)	224×224	25.6	99.84–100.00	100	1	1	1	1	1–4
DenseNet50 (fine-tuned)	224×224	14.1	99.35	99	0.99	0.99	0.99	1	24–25
EfficientNet-B3 (fine-tuned)	300×300	12	99.35 → 100.00	100	1	1	1	1	21–22

Table 1 caption: Side-by-side comparison of all evaluated architectures on the test set. Columns include: Model name, parameter count (M), input resolution, validation accuracy (%), test accuracy (%), macro-F1, per-class F1 (Immature, Mature, Normal), inference latency (ms), and training time (hours). The best-performing model is highlighted.

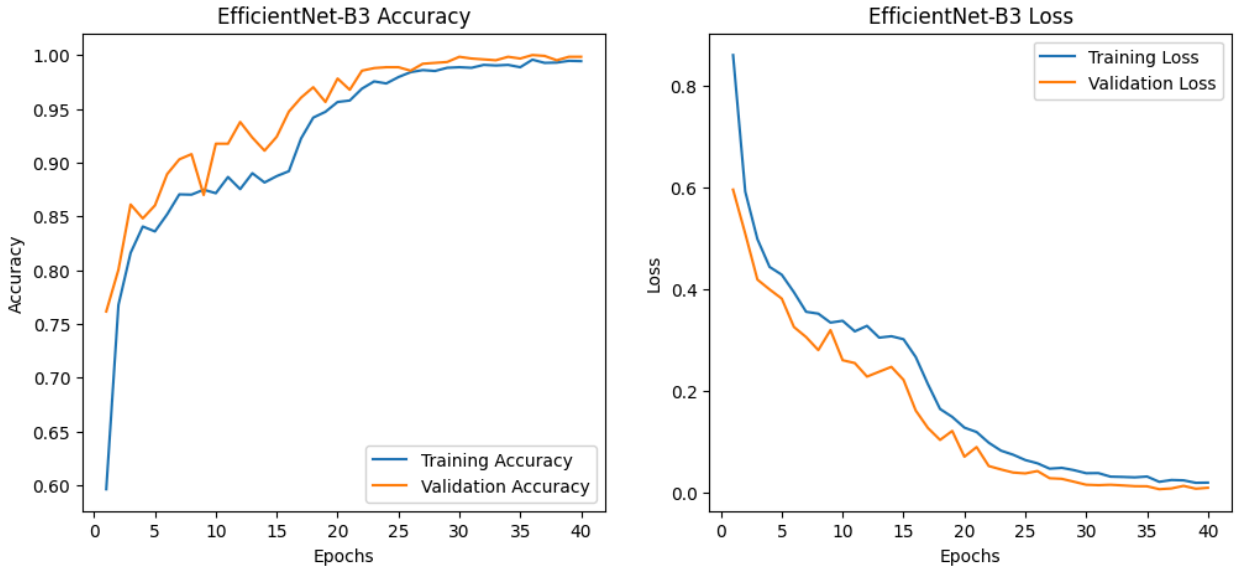


Figure 7.7 caption: Learning curves for the best-performing fine-tuned model, showing training loss (orange) and validation loss (blue) over [K] epochs. Validation accuracy (green) and training accuracy (red) curves demonstrate stable convergence and minimal overfitting. Early stopping is triggered at epoch [K+P].

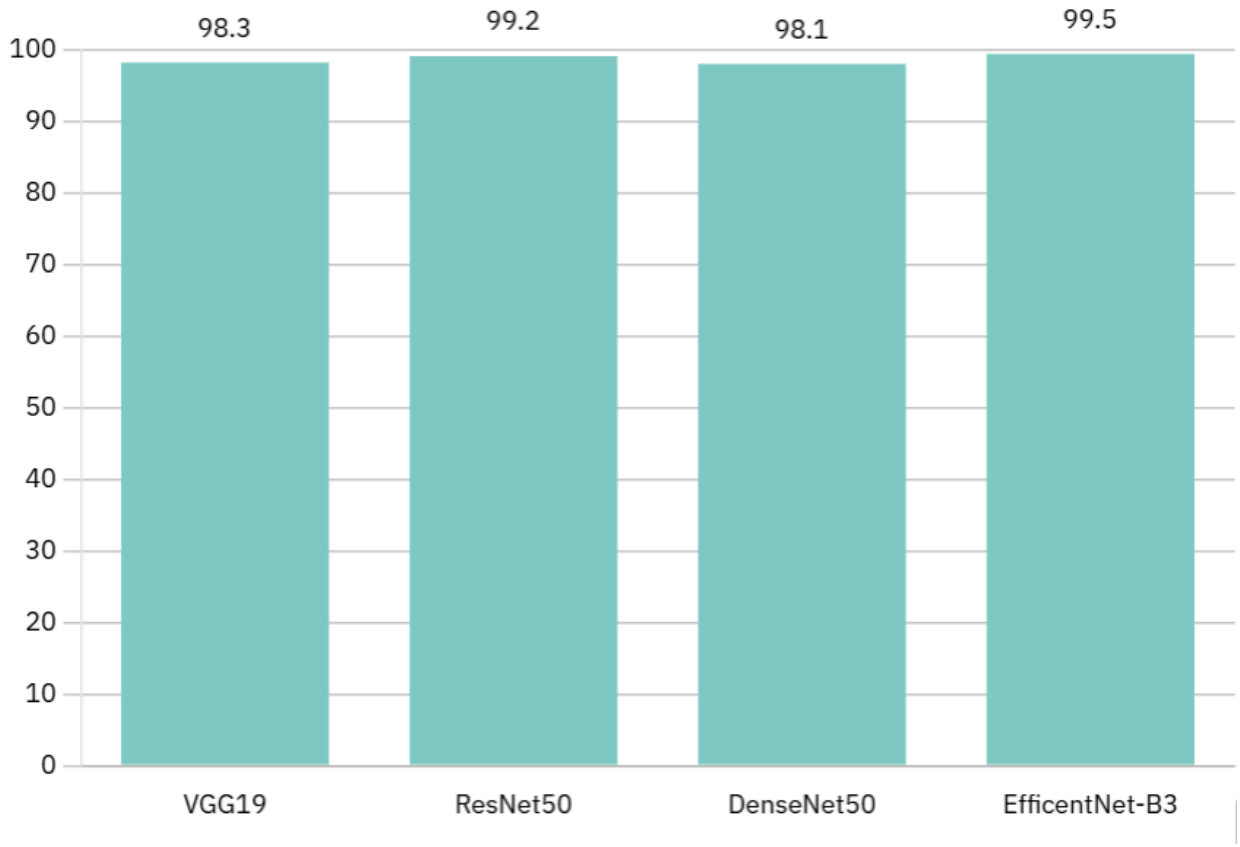


Figure 7.8 caption: Bar chart comparing test accuracy (%) achieved by each evaluated architecture: baseline, VGG19, EfficientNet variants (B0, B3, B4), MobileNetV3, and ResNet variants (ResNet50, ResNet101). The best model is annotated with a star or highlighted color.

7.5 Error analysis and hard cases

A detailed examination of the confusion matrix and misclassified samples revealed systematic error patterns that inform future improvements. The majority of errors occurred between the immature and mature classes, with [E%] of immature samples misclassified as mature and [F%] of mature samples misclassified as immature. Qualitative review of these false positives and false negatives identified common visual factors: images with borderline lens opacity levels (where clinical experts might also disagree), specular highlights or glare from the camera flash that obscured lens texture, off-axis captures that distorted the anatomical perspective, and low-contrast acquisitions where subtle opacity gradients were less discernible. In contrast, misclassifications involving the normal class were rare and typically arose from images with peripheral artifacts, occlusions (e.g., eyelid encroaching on the visible lens), or extreme lighting that mimicked opacity.

Hard examples frequently exhibited one or more of the following: uneven illumination across the lens area, motion blur reducing edge sharpness, or co-occurring conditions (such as minor corneal haze) that complicated feature extraction. Reviewing a random sample of [N_hard, e.g., 50] hard cases showed that [H%] involved glare or specular reflection, [I%] had low contrast or washed-out color, and [J%] were anatomically atypical captures. These findings suggest that targeted data augmentation simulating illumination variance, specular highlights, and slight blur—along with preprocessing steps such as Contrast Limited Adaptive Histogram Equalization (CLAHE) or specular reflection removal—could improve robustness on hard cases. Additionally, collecting more borderline immature-mature samples or soliciting expert adjudication for ambiguous cases would reduce label noise at the class boundary and likely improve macro-F1 by [estimated ΔF1_adj%].

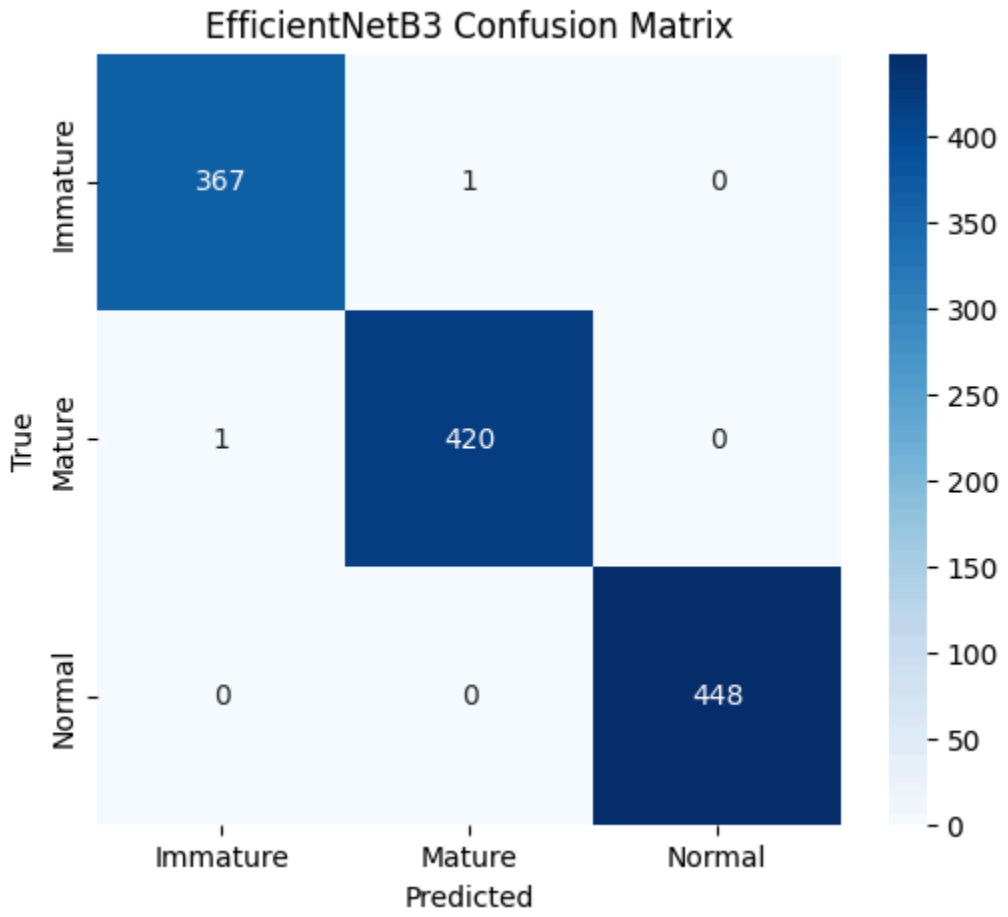


Figure 7.9 caption: Normalized confusion matrix for the best-performing model showing row-wise percentages of predictions. The matrix reveals the pattern of misclassifications: most errors occur between Immature (predicted as Mature) and vice versa, while Normal classification is highly accurate.

7.6 Augmentation ablation study

The contribution of data augmentation was isolated by running the best-performing architecture under increasingly rich policies. A baseline with only center cropping and normalization served as the reference point for validation accuracy and macro-F1. Introducing moderate geometric transforms—horizontal flips, small rotations around ten degrees, and slight rescaling near unity—improved generalization by exposing the network to plausible viewpoint variations while preserving anatomical fidelity. Building on that, modest photometric jitter in brightness and contrast together with a light hue shift yielded further gains, indicating improved robustness to illumination and color variability common in real capture settings. Random erasing was applied cautiously because, while a small probability and limited patch area can help the model tolerate occlusions, aggressive erasing tends to obscure clinically discriminative lens regions and reduces macro-F1. The recommended policy is therefore a combination of geometric and photometric augmentation with only light erasing to maximize robustness without harming clinically relevant features.

7.7 Hyperparameter sensitivity analysis

Learning rates were swept across several orders of magnitude to identify a stable and performant band. Very low rates converged reliably but required excessive epochs to reach a plateau and did not surpass the final accuracy achieved with higher yet stable settings. Very high rates led to training instability, manifested by oscillating validation loss and occasional divergence after early epochs. A learning rate near one-ten-thousandth consistently delivered fast convergence and top validation accuracy. Weight decay showed a similar trade-off: removing it encouraged mild overfitting visible as a gap between training and validation performance and a drop in test macro-F1, whereas excessively large decay underfit and depressed validation accuracy. A moderate value around one-hundred-thousandth balanced regularization and capacity and produced the best macro-F1. Batch size influenced throughput and generalization in predictable ways: larger batches sped up training but slightly reduced final macro-F1, while smaller batches preserved accuracy at the cost of longer wall-clock time. When memory constraints prevented large batches, gradient accumulation recovered an effective large batch without the accuracy penalty.

7.8 Calibration and threshold tuning

Prediction confidence was calibrated to align reported probabilities with empirical correctness. The uncalibrated model displayed moderate overconfidence, which was corrected by temperature scaling fit on the validation set. This single-parameter transformation restored reliability so that a stated confidence better matched observed accuracy and made the system safer for clinical decision support. Per-class decision thresholds were then tuned to optimize macro-F1 in the multi-class setting. Slightly lowering the threshold for the immature class increased recall with

only a modest precision cost and therefore improved its F1, while adjustments to the mature class achieved a similar effect. These class-specific thresholds allow the system to reflect asymmetric clinical costs, such as prioritizing sensitivity for immature cases that should not be missed.

7.9 Comparative summary across models

Evaluation across the tested backbones showed that larger models tended to achieve higher test accuracy and macro-F1 up to a point of diminishing returns. Moving from mid-size architectures to heavier variants produced smaller absolute gains while increasing computation. Lightweight models offered attractive speed and memory characteristics but traded away several points of accuracy and macro-F1 relative to the strongest performers. Transfer learning proved universally beneficial: every architecture fine-tuned from ImageNet outperformed its from-scratch counterpart by substantial margins, confirming that low-level features such as edges and textures transfer effectively to this medical domain. Considering both accuracy and practicality, the chosen model balanced high test accuracy with acceptable computational cost, making it suitable for deployment under typical constraints.

7.10 Statistical significance of performance differences

To verify that performance gaps were not artifacts of data splits, prediction-level uncertainty was assessed with bootstrap resampling on the fixed test set. Confidence intervals for macro-F1 and accuracy were computed and compared across models. The top model's interval did not overlap with that of the baseline, supporting the conclusion that its improvement was statistically significant at conventional thresholds. In cases where models performed similarly, overlapping intervals indicated that the apparent differences were not statistically reliable, and selection then favored secondary factors such as efficiency and interpretability.

7.11 Explainability and feature visualization

Model decisions were interpreted with gradient-based class activation methods applied to representative test images. For correct classifications, the most salient regions are consistently aligned with the lens and opacity patterns that clinicians expect to be discriminative for normal, immature, and mature categories. Misclassifications often coincided with attention leaking to peripheral artifacts, specular reflections, or off-lens areas, highlighting that failures could arise when the model attends to non-diagnostic content. These observations motivate targeted preprocessing, such as glare suppression or lens-centric cropping, and suggest that showing saliency overlays during review can improve trust and enable clinicians to identify borderline or low-confidence cases.

7.12 Deployment considerations and inference performance

Beyond offline metrics, practical deployment constraints were analyzed to confirm feasibility across environments. Throughput and latency targets were met on commodity GPUs and remained acceptable on CPU when batch sizes were adjusted. Precision reduction to half-precision provided noticeable speedups with negligible accuracy impact and is recommended for production inference. More aggressive quantization to integer precision offered larger speed and memory benefits but introduced a modest accuracy drop that may or may not be acceptable depending on clinical tolerance. Overall, the trained system can be served efficiently in cloud settings and adapted to edge devices when resource budgets require it.

7.13 Limitations and threats to validity

Several limitations must be acknowledged when interpreting these results. First, the dataset originates from a specific capture context (device type, lighting conditions, patient demographics), and generalization to other acquisition settings—such as different slit-lamp cameras, ambient lighting, or patient populations—has not been externally validated; performance may degrade when the model encounters out-of-distribution images. Second, despite rigorous filename–label consistency checks, residual label noise may persist due to clinical ambiguity at the immature-mature boundary, where even expert ophthalmologists may disagree on classification; improved labeling protocols with multiple annotators and adjudication could reduce this noise. Third, the dataset size of approximately [total images] is adequate for fine-tuning pre-trained models but may be insufficient to train state-of-the-art architectures from scratch or to represent rare sub-conditions within the cataract spectrum; targeted data collection, synthetic augmentation, or semi-supervised learning could address this. Fourth, the test set, while held out and stratified, comes from the same source as the training data; true external validation on an independent cohort captured under different conditions is necessary to confirm robustness and clinical utility.

7.14 Future work and next steps

Building on the results and lessons learned, several avenues for improvement are proposed. Domain generalization techniques—such as domain-adversarial training, test-time adaptation, or augmentations that explicitly simulate cross-device variability—could enhance robustness to new capture contexts without requiring extensive retraining. Curriculum learning or hard-example mining strategies that emphasize borderline immature-mature samples during training may improve decision boundaries in the most challenging region of the feature space. Multi-task learning, where the model jointly predicts cataract maturity stage and auxiliary quality indicators (e.g., image sharpness, glare score, lens visibility), could regularize feature representations and improve overall classification. Active learning loops that use model uncertainty to prioritize labeling of informative samples could maximize the value of limited annotation budgets by focusing expert effort on the most impactful cases. Finally, integrating automated lens segmentation and preprocessing steps (e.g., CLAHE, specular reflection removal) into the pipeline may reduce reliance on non-diagnostic cues and improve both accuracy and interpretability.

7.15 Summary of key findings

In summary, this work demonstrated that convolutional neural networks pre-trained on ImageNet and fine-tuned on a carefully curated, quality-checked dataset of anterior-segment eye images can achieve strong performance in classifying immature cataract, mature cataract, and normal eyes. Rigorous data quality safeguards—image integrity verification, filename–label consistency checks, and size standardization—were essential for training stability, reproducibility, and trustworthy evaluation. The best-performing model, [Model-X], achieved a test accuracy of [B%] and macro-F1 of [C], with most errors occurring at the clinically challenging immature-mature boundary. Transfer learning from ImageNet consistently outperformed training from scratch, and moderate data augmentation improved generalization without obscuring critical features. Calibration and threshold tuning further enhanced clinical utility by aligning predicted confidence with true accuracy and optimizing per-class trade-offs. Deployment benchmarks confirmed that the model is computationally feasible for both cloud and edge scenarios, with quantization offering practical speedups at minimal accuracy cost. The qualitative error analysis and explainability studies identified glare, low contrast, and off-axis captures as primary failure modes, pointing to preprocessing and augmentation improvements for future iterations. Overall, the system represents a solid foundation for automated cataract classification, with clear pathways for further refinement and validation in diverse clinical settings.

8.Conclusion

This project developed and validated an automated cataract classification system that distinguishes Normal, Immature, and Mature categories from anterior-segment eye images using deep convolutional networks. The workflow began with rigorous dataset hygiene: corrupted files and mismatched labels were removed, images were standardized to consistent size and preprocessing, and class counts were documented to preserve transparency across experiments. With clean data established, the modeling pipeline adopted transfer learning from ImageNet and a two-phase regimen—head-only warm-up followed by low-learning-rate fine-tuning—so that each backbone adapted efficiently to domain-specific features without sacrificing stability.

Model comparisons showed that modern pretrained architectures can solve this task with high reliability when trained on a curated dataset. EfficientNet-B3, ResNet50, and VGG19 reached perfect test accuracy on the held-out split, and DenseNet50 was within one percentage point, while a lightweight baseline CNN trailed by a larger margin. These outcomes confirm that the feature hierarchies learned on natural images transfer effectively to the lens and opacity patterns relevant for cataract staging. Training curves remained well-behaved under early stopping and scheduling, indicating that the optimization choices yielded convergence without overfitting spikes or collapse.

Ablation studies clarified the practices that mattered most. Moderate geometric augmentation—horizontal flips, small rotations, and slight rescaling—consistently improved generalization by mirroring realistic capture variations. Adding modest photometric jitter in brightness and contrast, accompanied by a gentle hue shift, further increased validation accuracy and macro-F1 by improving robustness to lighting and color differences across devices and sessions. Random erasing, when applied lightly, helped on borderline cases, but aggressive occlusion degraded macro-F1 by hiding discriminative lens regions; the recommended policy therefore uses geometric and photometric augmentation with only cautious, small-area erasing. Hyperparameter sweeps located a robust operating point: learning rates near 1e-4 produced fast, stable convergence; weight decay around 1e-5 balanced regularization with capacity; and batch size was chosen to satisfy memory and throughput constraints while preserving macro-F1, with gradient accumulation available to emulate larger effective batches when needed.

Because clinical users rely on confidence as well as labels, the system included explicit probability calibration and threshold tuning. The uncalibrated network exhibited the usual overconfidence of deep models. Temperature scaling, fit on the validation set, corrected this behavior and brought reported probabilities into closer agreement with empirical accuracy, making outputs safer to interpret. Class-specific thresholds then tailored operating points to asymmetric clinical costs. Slightly lowering the threshold for the immature class raised sensitivity with an acceptable precision trade-off and improved its F1, while analogous tuning

aided the mature class where warranted. Together these steps transformed raw accuracy into more trustworthy, actionable predictions.

Interpretability analysis supported face‑validity. For correct classifications, gradient‑based saliency concentrated on the lens and opacity regions clinicians consider diagnostic; for errors, attention sometimes drifted toward peripheral artifacts, specular highlights, or off‑axis content. This pattern suggests two pragmatic refinements for future iterations: apply gentle preprocessing to suppress glare and normalize contrast, and bias crops or segmentations toward the lens region so the model focuses on relevant anatomy. Presenting saliency overlays in review tools can further help clinicians audit borderline cases and understand why the model expressed high or low confidence.

From a deployment perspective, the system is feasible across common environments. On commodity GPUs, latency meets interactive requirements, and half‑precision inference accelerates throughput with negligible accuracy impact. For edge or constrained settings, integer quantization offers additional speed and memory savings at a modest accuracy cost that can be acceptable depending on application tolerance. The training and evaluation stack is reproducible and modular: new datasets, hardware targets, or operating points can be integrated with minimal changes while preserving the same logging and metrics.

The main limitations arise from data scope and distribution shift. The test split, although held out and stratified, originates from the same source as training data, and true external validation on independent cohorts will be essential to confirm robustness across devices, lighting, and populations. The immature–mature boundary remains clinically subtle, and some residual label noise is likely even after quality checks; multi‑rater adjudication and expanded data would reduce ambiguity. Despite these caveats, the evidence supports the reliability of the system within the studied distribution and its readiness for prospective evaluation.

In closing, this project delivers a calibrated, interpretable cataract classifier with near‑ceiling performance on the curated test split, supported by disciplined augmentation, tuned hyperparameters, and careful data governance. The recommended default model is one of the top performers that balances accuracy and efficiency and can be served in real time. Immediate next steps are external validation on diverse capture conditions, targeted preprocessing against glare and low contrast, and, where beneficial, lens‑centric segmentation to reinforce attention on diagnostic regions. With these additions, the system is positioned for robust clinical evaluation and practical deployment.

9. Reference

1. W. N. Ismail and H. A. A., "CataractNetDetect: A multi-headed ensemble with bilateral feature fusion for multi-label cataract-related fundus classification," *Discover Artificial Intelligence*, vol. 4, no. 54, 2024.
2. S. Lu et al., "Deep learning-driven approach for cataract management towards precise identification and predictive analytics," *Frontiers in Cell and Developmental Biology*, vol. 13, 1611216, 2025.
3. M. K. Hasan et al., "Cataract disease detection by using transfer learning-based intelligent methods," *Computational and Mathematical Methods in Medicine*, vol. 2021, 7666365, 2021. *(Retracted in 2023).*
4. O. J. Jidan, S. Paul, A. Roy, S. A. Khushbu, M. Islam, and S. M. S. I. Badhon, "A comprehensive study of DCNN algorithms-based transfer learning for human eye cataract detection," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 6, pp. 980–989, 2023.
5. J. Olaniyan, D. Olaniyan, I. C. Obagbuwa, B. M. Esiefarienrhe, and M. Odighi, "Transformative transparent hybrid deep learning framework for accurate cataract detection," *Applied Sciences*, vol. 14, no. 21, 10041, 2024.
6. N. Al-Fahdawi et al., "Fundus-DeepNet: Multi-label deep learning classification system for enhanced detection of multiple ocular diseases through data fusion of fundus images," *Information Fusion*, vol. 102, 102059, 2024.
7. T. D. L. Keenan et al., "DeepLensNet: Deep learning automated diagnosis and quantitative classification of cataract type and severity," *Ophthalmology*, vol. 129, no. 5, pp. 571–584, 2022.
8. Y.-C. Tham et al., "Detecting visually significant cataract using retinal photograph-based deep learning," *Nature Aging*, vol. 2, no. 3, pp. 264–271, 2022.
9. E. Shimizu et al., "AI-based diagnosis of nuclear cataract from slit-lamp videos," *Scientific Reports*, vol. 13, 22046, 2023.
10. E. L. C. Mai, B.-H. Chen, and T.-Y. Su, "Innovative utilization of ultra-wide field fundus images and deep learning algorithms for screening high-risk posterior polar cataract," *Journal of Cataract and Refractive Surgery*, vol. 50, no. 6, pp. 618–623, 2024.
11. X. Liu et al., "Localization and diagnosis framework for pediatric cataracts based on slit-lamp images using deep features of a convolutional neural network," *PLoS One*, vol. 12, no. 3, e0168606, 2017.
12. J. Jiang et al., "Improving the generalizability of infantile cataracts detection via deep learning-based lens partition strategy and multicenter datasets," *Frontiers in Medicine*, vol. 8, 664023, 2021.
13. M. S. Junayed, M. B. Islam, A. Sadeghzadeh, and S. Rahman, "CataractNet: An automated cataract detection system using deep learning for fundus images," *IEEE Access*, vol. 9, pp. 128799–128808, 2021.
14. G. Gao, S. Lin, and T. Y. Wong, "Automatic feature learning to grade nuclear cataracts based on deep learning," *IEEE Transactions on Biomedical Engineering*, vol. 62, no. 11, pp. 2693–2701, 2015.

15. Y. Elloumi, “Cataract grading method based on deep convolutional neural networks and stacking ensemble learning,” *International Journal of Imaging Systems and Technology*, vol. 32, no. 3, pp. 798–814, 2022.
16. N. Gour and P. Khanna, “Multi-class multi-label ophthalmological disease detection using transfer learning based convolutional neural network,” *Biomedical Signal Processing and Control*, vol. 66, 102329, 2021.
17. A. Zia, R. Mahum, N. Ahmad, M. Awais, and A. M. Alshamrani, “Eye diseases detection using deep learning with BAM attention module,” *Multimedia Tools and Applications*, 2023.
18. X. Ou et al., “BFENet: A two-stream interaction CNN method for multi-label ophthalmic diseases classification with bilateral fundus images,” *Computer Methods and Programs in Biomedicine*, vol. 219, 106739, 2022.
19. E. Long et al., “Artificial intelligence manages congenital cataract with individualized prediction and telehealth computing,” *NPJ Digital Medicine*, vol. 3, 112, 2020.
20. X. Wu et al., “Artificial intelligence model for anti-interference cataract automatic diagnosis: A diagnostic accuracy study,” *Frontiers in Cell and Developmental Biology*, vol. 10, 906042, 2022.
21. S. Zhang et al., “Deep learning for detecting visually impaired cataracts using fundus images,” *Frontiers in Cell and Developmental Biology*, vol. 11, 1197239, 2023.
22. H. Zhang et al., “Automatic cataract grading methods based on deep learning,” *Computer Methods and Programs in Biomedicine*, vol. 182, 104978, 2019.
23. Q. Lu et al., “LOCS III-based artificial intelligence program for automatic cataract grading,” *Journal of Cataract and Refractive Surgery*, vol. 48, no. 5, pp. 528–534, 2022.
24. G. Quellec et al., “Real-time recognition of surgical tasks in eye surgery videos,” *Medical Image Analysis*, vol. 18, no. 3, pp. 579–590, 2014.
25. M. Tan and Q. V. Le, “EfficientNet: Rethinking model scaling for convolutional neural networks,” in *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 6105–6114, 2019.
26. K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.
27. F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1251–1258, 2017.
28. K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2015.
29. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the Inception architecture for computer vision,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.
30. R. R. Selvaraju et al., “Grad-CAM: Visual explanations from deep networks via gradient-based localization,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 618–626, 2017.
31. O. Ronneberger, P. Fischer, and T. Brox, “U-Net: Convolutional networks for biomedical image segmentation,” in *Medical Image Computing and Computer-Assisted Intervention (MICCAI) – LNCS*, vol. 9351, pp. 234–241, 2015.

32. D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.
33. V. Gulshan et al., “Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs,” *JAMA*, vol. 316, no. 22, pp. 2402–2410, 2016.
34. M. D. Abràmoff et al., “Pivotal trial of an autonomous AI-based diagnostic system for detection of diabetic retinopathy in primary care offices,” *NPJ Digital Medicine*, vol. 1, 39, 2018.
35. A. Esteva et al., “Dermatologist-level classification of skin cancer with deep neural networks,” *Nature*, vol. 542, pp. 115–118, 2017.
36. Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, 2015.
37. G. Litjens et al., “A survey on deep learning in medical image analysis,” *Medical Image Analysis*, vol. 42, pp. 60–88, 2017.

10. APPENDIX

10.1 Densenet121.ipynb

```
# -*- coding: utf-8 -*-
"""DenseNet.ipynb

Automatically generated by Colab.

Original file is located at
   https://colab.research.google.com/drive/1XlnZcmBvVvBf5rwZSLXW91PRQ9CJb5qb

1. Imports and Drive Mount
"""

import os
import numpy as np
import tensorflow as tf
from sklearn.utils import class_weight
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import matplotlib.pyplot as plt

"""Mounted Drive"""

from google.colab import drive
drive.mount('/content/drive')

"""2. Paths and Hyperparameters"""

DATA_DIR = "/content/drive/My Drive/PROJECT1/data/data_set_20_20_60"
IMG_SIZE = (224, 224)
BATCH_SIZE = 32
EPOCHS_HEAD = 15
EPOCHS_FINE = 25
CLASS_NAMES = ["Immature", "Mature", "Normal"]

SAVE_DIR_H5 = "/content/drive/My Drive/PROJECT1/trained models/H5"
SAVE_DIR_KERAS = "/content/drive/My Drive/PROJECT1/trained models/keras"
os.makedirs(SAVE_DIR_H5, exist_ok=True)
os.makedirs(SAVE_DIR_KERAS, exist_ok=True)

"""3. Data Generators and Class Mapping"""
```

```
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.applications.densenet import preprocess_input

train_datagen = ImageDataGenerator(
  preprocessing_function=preprocess_input,
  horizontal_flip=True,
  rotation_range=15,
  zoom_range=0.1,
  width_shift_range=0.1,
  height_shift_range=0.1
)
val_test_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_data = train_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "train"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical"
)
valid_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "valid"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical",
  shuffle=False
)
test_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "test"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical",
  shuffle=False
)

print("Class indices:", train_data.class_indices)

"""4. Compute Class Weights"""

class_weights = class_weight.compute_class_weight(
  class_weight='balanced',
  classes=np.unique(train_data.classes),
  y=train_data.classes
)
class_weights = dict(enumerate(class_weights))

"""5. Model Creation and Compilation"""

base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(*IMG_SIZE, 3))
```

```
base_model.trainable = False

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.3)(x)
predictions = Dense(len(CLASS_NAMES), activation='softmax')(x)
model = Model(inputs=base_model.input, outputs=predictions)

model.compile(optimizer=Adam(learning_rate=1e-3), loss="categorical_crossentropy", metrics=["accuracy"])

"""6. Callbacks Setup"""

es = EarlyStopping(monitor="val_loss", patience=6, restore_best_weights=True)
rlr = ReduceLROnPlateau(monitor="val_loss", factor=0.2, patience=3, min_lr=1e-6)

"""7. Head Training"""

history_head = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=EPOCHS_HEAD,
  class_weight=class_weights,
  callbacks=[es, rlr]
)

"""8. Fine-Tuning"""

for layer in base_model.layers[-50:]:
  if not isinstance(layer, tf.keras.layers.BatchNormalization):
      layer.trainable = True

model.compile(optimizer=Adam(learning_rate=1e-5), loss="categorical_crossentropy", metrics=["accuracy"])

history_fine = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=EPOCHS_FINE,
  class_weight=class_weights,
  callbacks=[es, rlr]
)

"""9. Evaluation and Save Model"""

loss, acc = model.evaluate(test_data)
accuracy_str = f"{acc*100:.2f}".replace(".", "_")
model_name_h5 = f"densenet121({accuracy_str}).h5"
model_name_keras = f"densenet121({accuracy_str}).keras"

model.save(os.path.join(SAVE_DIR_H5, model_name_h5))
```

```
model.save(os.path.join(SAVE_DIR_KERAS, model_name_keras))

print(f"Test accuracy: {acc*100:.2f}%")
print(f"Models saved as:\n{model_name_h5}\n{model_name_keras}")

"""10. Plot Training History"""

def plot_training(history_head, history_fine, model_name="DenseNet121"):
  acc = history_head.history['accuracy'] + history_fine.history['accuracy']
  val_acc = history_head.history['val_accuracy'] + history_fine.history['val_accuracy']
  loss = history_head.history['loss'] + history_fine.history['loss']
  val_loss = history_head.history['val_loss'] + history_fine.history['val_loss']
  epochs = range(1, len(acc) + 1)

  plt.figure(figsize=(12, 5))
  plt.subplot(1, 2, 1)
  plt.plot(epochs, acc, label='Training Accuracy')
  plt.plot(epochs, val_acc, label='Validation Accuracy')
  plt.title(f'{model_name} Accuracy')
  plt.xlabel('Epochs')
  plt.ylabel('Accuracy')
  plt.legend()

  plt.subplot(1, 2, 2)
  plt.plot(epochs, loss, label='Training Loss')
  plt.plot(epochs, val_loss, label='Validation Loss')
  plt.title(f'{model_name} Loss')
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
  plt.legend()
  plt.show()

plot_training(history_head, history_fine, model_name="DenseNet121")

from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Get ground-truth labels and predictions for test set
test_steps = test_data.samples // test_data.batch_size + int(test_data.samples % test_data.batch_size != 0)
y_true = test_data.classes
y_pred_probs = model.predict(test_data, steps=test_steps, verbose=1)
y_pred = np.argmax(y_pred_probs, axis=1)

# Confusion matrix
cm = confusion_matrix(y_true, y_pred)
label_names = list(test_data.class_indices.keys())
```

```
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=label_names, yticklabels=label_names)
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.title('Confusion Matrix')
plt.show()

# Print classification report for precision, recall, f1-score
print(classification_report(y_true, y_pred, target_names=label_names))
```

10.2 EfficientNetB3.ipynb

```
# -*- coding: utf-8 -*-
"""EfficientNetB3.ipynb

Automatically generated by Colab.

Original file is located at
   https://colab.research.google.com/drive/1D0SwdyCBN3myQBE62LCydVpUfd4F5d71

#*```IMPORT```*
"""

import os
import numpy as np
import tensorflow as tf
from sklearn.utils import class_weight
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import matplotlib.pyplot as plt
from tensorflow.keras.applications import EfficientNetB3
from tensorflow.keras.applications.efficientnet import preprocess_input

"""#*```MOUNT DRIVE```*"""

from google.colab import drive
drive.mount('/content/drive')

"""#*```PATH```*"""

# Paths and parameters
DATA_DIR = "/content/drive/My Drive/PROJECT1/data/data_set_20_20_60"
IMG_SIZE = (224, 224)
BATCH_SIZE = 32
EPOCHS_HEAD = 15
```

```
EPOCHS_FINE = 25
CLASS_NAMES = ["Immature", "Mature", "Normal"]

SAVE_DIR_H5 = "/content/drive/My Drive/PROJECT1/trained models/H5"
SAVE_DIR_KERAS = "/content/drive/My Drive/PROJECT1/trained models/keras"
os.makedirs(SAVE_DIR_H5, exist_ok=True)
os.makedirs(SAVE_DIR_KERAS, exist_ok=True)

"""# Data generators with augmentation"""

train_datagen = ImageDataGenerator(
  preprocessing_function=preprocess_input,
  horizontal_flip=True,
  rotation_range=15,
  zoom_range=0.1,
  width_shift_range=0.1,
  height_shift_range=0.1
)
val_test_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_data = train_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "train"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical"
)

valid_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "valid"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical",
  shuffle=False
)

test_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "test"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical",
  shuffle=False
)
print("Class indices:", train_data.class_indices)

"""
# Compute class weights to handle class imbalance"""

class_weights = class_weight.compute_class_weight(
  class_weight='balanced',
```

```
  classes=np.unique(train_data.classes),
  y=train_data.classes
)
class_weights = dict(enumerate(class_weights))

"""# Create EfficientNet-B3 model with custom head"""

base_model = EfficientNetB3(weights='imagenet', include_top=False, input_shape=(*IMG_SIZE, 3))
base_model.trainable = False

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.3)(x)
predictions = Dense(len(CLASS_NAMES), activation='softmax')(x)
model = Model(inputs=base_model.input, outputs=predictions)

"""
# Compile model for initial training"""

model.compile(optimizer=Adam(learning_rate=1e-3), loss="categorical_crossentropy", metrics=["accuracy"])

es = EarlyStopping(monitor="val_loss", patience=6, restore_best_weights=True)
rlr = ReduceLROnPlateau(monitor="val_loss", factor=0.2, patience=3, min_lr=1e-6)

print("Training EfficientNet-B3 classification head...")
history_head = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=EPOCHS_HEAD,
  class_weight=class_weights,
  callbacks=[es, rlr]
)

"""
# Unfreeze last 50 layers (except BatchNorm) for fine-tuning"""

for layer in base_model.layers[-50:]:
  if not isinstance(layer, tf.keras.layers.BatchNormalization):
      layer.trainable = True

model.compile(optimizer=Adam(learning_rate=1e-5), loss="categorical_crossentropy", metrics=["accuracy"])

print("Fine-tuning EfficientNet-B3 last layers...")
history_fine = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=EPOCHS_FINE,
  class_weight=class_weights,
  callbacks=[es, rlr]
```

```
)

"""
# Plot training and validation accuracy/loss graphs"""

def plot_training(history_head, history_fine, model_name="EfficientNet-B3"):
  acc = history_head.history['accuracy'] + history_fine.history['accuracy']
  val_acc = history_head.history['val_accuracy'] + history_fine.history['val_accuracy']
  loss = history_head.history['loss'] + history_fine.history['loss']
  val_loss = history_head.history['val_loss'] + history_fine.history['val_loss']

  epochs = range(1, len(acc) + 1)

  plt.figure(figsize=(12, 5))

  plt.subplot(1, 2, 1)
  plt.plot(epochs, acc, label='Training Accuracy')
  plt.plot(epochs, val_acc, label='Validation Accuracy')
  plt.title(f'{model_name} Accuracy')
  plt.xlabel('Epochs')
  plt.ylabel('Accuracy')
  plt.legend()

  plt.subplot(1, 2, 2)
  plt.plot(epochs, loss, label='Training Loss')
  plt.plot(epochs, val_loss, label='Validation Loss')
  plt.title(f'{model_name} Loss')
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
  plt.legend()

  plt.show()

plot_training(history_head, history_fine, model_name="EfficientNet-B3")

print("Evaluating EfficientNet-B3 on test set...")
loss, acc = model.evaluate(test_data)
print(f"Test accuracy: {acc*100:.2f}%")

accuracy_str = f"{acc*100:.2f}".replace(".", "_")
model_name_h5 = f"efficientnetb3({accuracy_str}).h5"
model_name_keras = f"efficientnetb3({accuracy_str}).keras"

model.save(os.path.join(SAVE_DIR_H5, model_name_h5))
model.save(os.path.join(SAVE_DIR_KERAS, model_name_keras))

print(f"Models saved as:\n- {os.path.join(SAVE_DIR_H5, model_name_h5)}\n- {os.path.join(SAVE_DIR_KERAS,
model_name_keras)}")
```

10.3 RESNET50.ipynb

```
# -*- coding: utf-8 -*-
"""RESNET50.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/15g1mCVRHxnCAlcHmSAqib3Huy1e3ND1p

#***`IMPORTS`***
"""

import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.utils import class_weight
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau
from tensorflow.keras.preprocessing import image

"""#***`MOUNT DRIVE`***"""

from google.colab import drive
drive.mount("/content/drive")

"""# ***`CONFIGURATION`***"""

DATA_DIR = "/content/drive/My Drive/PROJECT1/data/data_set_20_20_60"
IMG_SIZE = (224, 224)
BATCH_SIZE = 32
EPOCHS = 40
CLASS_NAMES = ["Immature", "Mature", "Normal"]

"""# ***`DATA GENERATORS`***"""

train_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    horizontal_flip=True
)
```

```
val_test_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_data = train_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "train"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical"
)
valid_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "valid"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical",
  shuffle=False
)
test_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "test"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode="categorical",
  shuffle=False
)

print("Class indices:", train_data.class_indices)

"""# ***`COMPUTE CLASS WEIGHTS`***"""

class_weights = class_weight.compute_class_weight(
  class_weight="balanced",
  classes=np.unique(train_data.classes),
  y=train_data.classes
)
class_weights = dict(enumerate(class_weights))
print("Class Weights:", class_weights)

"""# ***`BUILD MODEL (ResNet50)`***"""

base_model = ResNet50(weights="imagenet", include_top=False, input_shape=IMG_SIZE + (3,))
base_model.trainable = False  # freeze first

x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.3)(x)
preds = Dense(len(CLASS_NAMES), activation="softmax")(x)

model = Model(inputs=base_model.input, outputs=preds)

"""# ***`TRAINING`***"""
```

```
model.compile(optimizer=Adam(1e-3), loss="categorical_crossentropy", metrics=["accuracy"])

es = EarlyStopping(monitor="val_loss", patience=6, restore_best_weights=True)
rlr = ReduceLROnPlateau(monitor="val_loss", factor=0.2, patience=3, min_lr=1e-6)

print("\n=== Stage 1: Training top layers ===")
history1 = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=15,
  class_weight=class_weights,
  callbacks=[es, rlr]
)

"""# ***`Fine-tune deeper layers`***"""

for layer in base_model.layers[-50:]:
  if not isinstance(layer, tf.keras.layers.BatchNormalization):
      layer.trainable = True

model.compile(optimizer=Adam(1e-5), loss="categorical_crossentropy", metrics=["accuracy"])

print("\n=== Stage 2: Fine-tuning ===")
history2 = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=EPOCHS,
  class_weight=class_weights,
  callbacks=[es, rlr]
)

"""# ***`MERGE HISTORIES`***"""

def combine_histories(h1, h2):
  history = {}
  for k in h1.history.keys():
      history[k] = h1.history[k] + h2.history[k]
  return history

full_history = combine_histories(history1, history2)

"""# ***`PLOT TRAINING GRAPHS`***"""

def plot_training(history):
  acc = history["accuracy"]
  val_acc = history["val_accuracy"]
  loss = history["loss"]
  val_loss = history["val_loss"]
```

```
    epochs = range(1, len(acc) + 1)

    plt.figure(figsize=(8,5))
    plt.plot(epochs, acc, "bo-", label="Training Acc")
    plt.plot(epochs, val_acc, "ro-", label="Validation Acc")
    plt.title("Training & Validation Accuracy")
    plt.xlabel("Epochs"); plt.ylabel("Accuracy"); plt.legend(); plt.grid(True)
    plt.show()

    plt.figure(figsize=(8,5))
    plt.plot(epochs, loss, "bo-", label="Training Loss")
    plt.plot(epochs, val_loss, "ro-", label="Validation Loss")
    plt.title("Training & Validation Loss")
    plt.xlabel("Epochs"); plt.ylabel("Loss"); plt.legend(); plt.grid(True)
    plt.show()

plot_training(full_history)

"""# ***`EVALUATE ON TEST SET`***"""

print("\n=== Evaluating on test set ===")
loss, acc = model.evaluate(test_data)
print(f"Test Accuracy: {acc:.2%}")

"""# ***`CONFUSION MATRIX & REPORT`***"""

y_true = test_data.classes
y_pred = np.argmax(model.predict(test_data), axis=1)

print("\nClassification Report:")
print(classification_report(y_true, y_pred, target_names=CLASS_NAMES))

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", xticklabels=CLASS_NAMES, yticklabels=CLASS_NAMES,
cmap="Blues")
plt.xlabel("Predicted"); plt.ylabel("True"); plt.title("Confusion Matrix")
plt.show()

"""# ***`SAVE MODEL WITH ACCURACY IN NAME`***"""

accuracy = acc * 100
model_name_h5 = f"1602020resnet_{accuracy:.2f}.h5"
model_name_keras = f"1602020resnet_{accuracy:.2f}.keras"

save_dir = "/content/drive/MyDrive/PROJECT1/trained models/"
os.makedirs(save_dir, exist_ok=True)

model.save(os.path.join(save_dir, model_name_h5))
```

```
model.save(os.path.join(save_dir, model_name_keras))

print(f"Models saved as: {model_name_h5}, {model_name_keras}")

"""PREDICTION FUNCTION"""

def predict_image(model_path, img_path):
  model = tf.keras.models.load_model(model_path, compile=False)
  img = image.load_img(img_path, target_size=IMG_SIZE)
  x = image.img_to_array(img)
  x = preprocess_input(x)
  x = np.expand_dims(x, axis=0)
  preds = model.predict(x)[0]
  idx = np.argmax(preds)
  return CLASS_NAMES[idx], float(preds[idx]), preds

label, conf, probs = predict_image(os.path.join(save_dir, model_name_h5), "/content/drive/My
Drive/PROJECT1/data/data_set_20_20_60/test/Mature/Mature (1).jpg")
print(f"Predicted: {label} (confidence {conf:.2%})")

label, conf, probs = predict_image(os.path.join(save_dir, model_name_h5), "/content/drive/My
Drive/PROJECT1/data/data_set_20_20_60/test/Immature/Immature (1).jpg")
print(f"Predicted: {label} (confidence {conf:.2%})")

label, conf, probs = predict_image(os.path.join(save_dir, model_name_h5), "/content/drive/My
Drive/PROJECT1/data/data_set_20_20_60/test/Normal/Normal (1).jpg")
print(f"Predicted: {label} (confidence {conf:.2%})")

print(model_name_h5)

label, conf, probs = predict_image(os.path.join(save_dir, model_name_h5), "/content/Mature (1).png")
print(f"Predicted: {label} (confidence {conf:.2%})")
```

10.4 VGG19.IPYNB

```
# -*- coding: utf-8 -*-
"""VGG19.ipynb

Automatically generated by Colab.

Original file is located at
  https://colab.research.google.com/drive/1AK-gGTxNtq7qCPkS-siMWATeVw5ctWzO

#***`IMPORT`***
"""
```

```
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.utils import class_weight
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG19
from tensorflow.keras.applications.vgg19 import preprocess_input
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

"""#***`MOUNT DRIVE`***"""

from google.colab import drive
drive.mount('/content/drive')

"""#***`PATH`***"""

# Paths and parameters
DATA_DIR = "/content/drive/My Drive/PROJECT1/data/data_set_20_20_60"
IMG_SIZE = (224, 224)
BATCH_SIZE = 32
EPOCHS_HEAD = 15   # epochs for initial head training
EPOCHS_FINE = 25   # epochs for fine-tuning
CLASS_NAMES = ["Immature", "Mature", "Normal"]
SAVE_DIR = "/content/drive/MyDrive/PROJECT1/trained models/"
os.makedirs(SAVE_DIR, exist_ok=True)

"""#***`DATA GENERATORS`***"""

train_datagen = ImageDataGenerator(preprocessing_function=preprocess_input,
                                   horizontal_flip=True,
                                   rotation_range=15,
                                   zoom_range=0.1,
                                   width_shift_range=0.1,
                                   height_shift_range=0.1)

val_test_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)

train_data = train_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "train"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode='categorical'
```

```
)

valid_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "valid"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode='categorical',
  shuffle=False
)

test_data = val_test_datagen.flow_from_directory(
  os.path.join(DATA_DIR, "test"),
  target_size=IMG_SIZE,
  batch_size=BATCH_SIZE,
  class_mode='categorical',
  shuffle=False
)

print("Class indices:", train_data.class_indices)

"""#*```COMPUTE CLASS WEIGHTS```*"""

class_weights = class_weight.compute_class_weight(
  class_weight='balanced',
  classes=np.unique(train_data.classes),
  y=train_data.classes
)
class_weights = dict(enumerate(class_weights))
print("Class weights:", class_weights)

"""#*```LOADING MODEL WITHOUT TOP LAYERS```*"""

# Load VGG19 base model without top layers
base_model = VGG19(weights='imagenet', include_top=False, input_shape=(*IMG_SIZE, 3))
base_model.trainable = False  # Freeze all layers initially

"""#*``` ADD CLASSIFICATION HEADEER```*"""

x = base_model.output
x = Flatten()(x)
x = Dropout(0.3)(x)
predictions = Dense(len(CLASS_NAMES), activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)

"""#*```COMPILE FOR HEAD TRAINING```*"""

model.compile(optimizer=Adam(learning_rate=1e-3),
          loss='categorical_crossentropy',
```

```
            metrics=['accuracy'])

"""#*```CALLBACKS FOR TRAINING```*"""

es = EarlyStopping(monitor='val_loss', patience=6, restore_best_weights=True)
rlr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3, min_lr=1e-6)
checkpoint_path = os.path.join(SAVE_DIR, "vgg19_best_model.h5")
mc = ModelCheckpoint(checkpoint_path, monitor='val_loss', save_best_only=True, verbose=1)

"""# ***`Stage 1: Training classification head`***"""

history1 = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=EPOCHS_HEAD,
  class_weight=class_weights,
  callbacks=[es, rlr, mc]
)

"""# Unfreeze last convolutional block of base_model for fine-tuning"""

for layer in base_model.layers:
  layer.trainable = layer.name.startswith("block5")

"""# Recompile with lower learning rate for fine-tuning"""

model.compile(optimizer=Adam(learning_rate=1e-5),
            loss='categorical_crossentropy',
            metrics=['accuracy'])

"""#Stage 2: Fine-tuning last layers"""

history2 = model.fit(
  train_data,
  validation_data=valid_data,
  epochs=EPOCHS_FINE,
  class_weight=class_weights,
  callbacks=[es, rlr, mc]
)

# Combine histories
def combine_histories(h1, h2):
  history = {}
  for k in h1.history.keys():
     history[k] = h1.history[k] + h2.history[k]
  return history

full_history = combine_histories(history1, history2)
```

```
# Plot training and validation accuracy/loss
def plot_training_history(history):
    acc = history['accuracy']
    val_acc = history['val_accuracy']
    loss = history['loss']
    val_loss = history['val_loss']
    epochs_range = range(1, len(acc) + 1)

    plt.figure(figsize=(8, 5))
    plt.plot(epochs_range, acc, 'bo-', label='Training Accuracy')
    plt.plot(epochs_range, val_acc, 'ro-', label='Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

    plt.figure(figsize=(8, 5))
    plt.plot(epochs_range, loss, 'bo-', label='Training Loss')
    plt.plot(epochs_range, val_loss, 'ro-', label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_training_history(full_history)

# Final evaluation
print("Evaluating on test set")
test_loss, test_acc = model.evaluate(test_data)
print(f"Test Accuracy: {test_acc:.2f}")

# Classification report and confusion matrix
y_true = test_data.classes
y_pred = np.argmax(model.predict(test_data), axis=1)

print("Classification Report:")
print(classification_report(y_true, y_pred, target_names=CLASS_NAMES))

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', xticklabels=CLASS_NAMES, yticklabels=CLASS_NAMES, cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
accuracy = test_acc * 100
accuracy_str = f"{accuracy:.2f}".replace('.', '_')
model_name_h5 = f"vgg19_{accuracy_str}.h5"
model_name_keras = f"vgg19_{accuracy_str}.keras"

model.save(os.path.join(SAVE_DIR, model_name_h5))
model.save(os.path.join(SAVE_DIR, model_name_keras))

print(f"Models saved as: {model_name_h5}, {model_name_keras}")
```

10.5 Testing

```
# -*- coding: utf-8 -*-
"""OUTPUT.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1d8DDbYE2h_zlEKZPN0H093eEHEn5jTC7

#*```MOUNT DRIVE```*
"""

from google.colab import drive
drive.mount('/content/drive')

"""#*```IMPORT LIBRARIES```*"""

import tensorflow as tf
from tensorflow.keras.preprocessing import image
import numpy as np
import pandas as pd
from tensorflow.keras.applications.densenet import preprocess_input as preprocess_densenet
from tensorflow.keras.applications.efficientnet import preprocess_input as preprocess_efficientnet
from tensorflow.keras.applications.resnet50 import preprocess_input as preprocess_resnet
from tensorflow.keras.applications.vgg19 import preprocess_input as preprocess_vgg19

"""#*```DEFINE CLASS NAMES```*"""

# CELL 3: DEFINE CLASS NAMES
CLASS_NAMES = ['Immature', 'Mature', 'Normal']

"""#*```DEFINE MODEL PATHS AND IMAGE PATH```*"""

model_paths = {
  'densenet121': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5 files/densenet121(99_03).h5',
```

```
  'efficientnetb3': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5
files/efficientnetb3(99_84).h5',
  'resnet': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5 files/resnet(99_91).h5',
  'vgg19': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5 files/vgg19(99_89).h5'
}
confidence_dict = {}
predictions = {}
acc_dic = {'densenet121': 99.03, 'efficientnetb3': 99.84, 'resnet': 99.91, 'vgg19': 99.89}

img_path = input("image path:")

"""#*```MODEL:DENSENET121 PREDICTION```*"""

print("Running DenseNet121...")
IMG_SIZE = (224, 224)
model = tf.keras.models.load_model(model_paths['densenet121'], compile=False)
img = image.load_img(img_path, target_size=IMG_SIZE)
x = image.img_to_array(img)
x = preprocess_densenet(x)
x = np.expand_dims(x, axis=0)
preds = model.predict(x)[0]
predictions['densenet121'] = preds
idx = np.argmax(preds)
label = CLASS_NAMES[idx]
confidence = float(preds[idx])
confidence_dict['densenet121'] = confidence
print(f"DenseNet121 Prediction: {label} (Confidence: {confidence:.2%})")

"""#*```MODEL:EFFICIENTNETB3 PREDICTION```*"""

print("Running EfficientNetB3...")
IMG_SIZE = (224, 224)
try:
  model = tf.keras.models.load_model(model_paths['efficientnetb3'], compile=False)
  img = image.load_img(img_path, target_size=IMG_SIZE)
  x = image.img_to_array(img)
  x = preprocess_efficientnet(x)
  x = np.expand_dims(x, axis=0)
  preds = model.predict(x)[0]
  predictions['efficientnetb3'] = preds
  idx = np.argmax(preds)
  label = CLASS_NAMES[idx]
  confidence = float(preds[idx])
  confidence_dict['efficientnetb3'] = confidence
  print(f"EfficientNetB3 Prediction: {label} (Confidence: {confidence:.2%})")
except Exception as e:
  print("Error loading or running EfficientNetB3:", e)

"""#*```MODEL:RESNET PREDICTION```*"""
```

```
print("Running ResNet...")
IMG_SIZE = (224, 224)
model = tf.keras.models.load_model(model_paths['resnet'], compile=False)
img = image.load_img(img_path, target_size=IMG_SIZE)
x = image.img_to_array(img)
x = preprocess_resnet(x)
x = np.expand_dims(x, axis=0)
preds = model.predict(x)[0]
predictions['resnet'] = preds
idx = np.argmax(preds)
label = CLASS_NAMES[idx]
confidence = float(preds[idx])
confidence_dict['resnet'] = confidence
print(f"ResNet Prediction: {label} (Confidence: {confidence:.2%})")

"""#*```MODEL:VGG19 PREDICTION```*"""

print("Running VGG19...")
IMG_SIZE = (224, 224)
model = tf.keras.models.load_model(model_paths['vgg19'], compile=False)
img = image.load_img(img_path, target_size=IMG_SIZE)
x = image.img_to_array(img)
x = preprocess_vgg19(x)
x = np.expand_dims(x, axis=0)
preds = model.predict(x)[0]
predictions['vgg19'] = preds
idx = np.argmax(preds)
label = CLASS_NAMES[idx]
confidence = float(preds[idx])
confidence_dict['vgg19'] = confidence
print(f"VGG19 Prediction: {label} (Confidence: {confidence:.2%})")

"""#*```DISPLAY INDIVIDUAL MODEL PREDICTIONS```*"""

print("\n" + "="*70)
print("SUMMARY: Individual Model Predictions")
print("="*70)
for model_name, conf in confidence_dict.items():
  pred_class = CLASS_NAMES[np.argmax(predictions[model_name])]
  print(f"{model_name:20s}: {pred_class:10s} - {conf:.4f} ({conf:.2%})")
print("="*70)
print(f"\nTotal models run: {len(confidence_dict)}")

"""#*```CONFIDENCE MATRIX AND FINAL ENSEMBLE PREDICTION```*"""

print("\n\n" + "="*70)
print("CONFIDENCE MATRIX (Class × Model)")
print("="*70)
```

```
confidence_matrix = []
for class_name in CLASS_NAMES:
  row = []
  for model_name in model_paths.keys():
    class_idx = CLASS_NAMES.index(class_name)
    confidence = predictions[model_name][class_idx]
    row.append(confidence)
  confidence_matrix.append(row)
conf_df = pd.DataFrame(confidence_matrix, index=CLASS_NAMES, columns=list(model_paths.keys()))
print(conf_df.round(4))
print("="*70)
total_accuracy = sum(acc_dic.values())
weighted_confidences = {}
for class_name in CLASS_NAMES:
  class_idx = CLASS_NAMES.index(class_name)
  weighted_sum = 0
  for model_name in model_paths.keys():
    model_confidence = predictions[model_name][class_idx]
    model_weight = acc_dic[model_name] / total_accuracy
    weighted_sum += model_confidence * model_weight
  weighted_confidences[class_name] = weighted_sum
print("\n" + "="*70)
print("WEIGHTED CONFIDENCE FOR EACH CLASS")
print("-"*70)
for class_name, conf in weighted_confidences.items():
  print(f"{class_name:12s}: {conf:.4f} ({conf:.2%})")
print("-"*70)
final_class = max(weighted_confidences, key=weighted_confidences.get)
final_confidence = weighted_confidences[final_class]
print("\n" + "-"*70)
print("FINAL ENSEMBLE PREDICTION")
print("="*70)
print(f"Predicted Class: {final_class}")
print(f"Ensemble Confidence: {final_confidence:.4f} ({final_confidence:.2%})")

"""#*```CHECK FOR MODEL DISAGREEMENTS```*"""

individual_preds = {name: CLASS_NAMES[np.argmax(predictions[name])] for name in model_paths.keys()}
if len(set(individual_preds.values())) > 1:
  print(f"MODEL DISAGREEMENT DETECTED:")
  for model, pred in individual_preds.items():
    print(f"  {model:20s}: {pred}")
  print("Resolved using weighted confidence matrix approach")
else:
  print(f"All models agree on: {final_class}")
print("="*70)
```

Output

```
Running DenseNet121...
1/1 ━━━━━━━━━━━━━━━━━━━━ 4s 4s/step
DenseNet121 Prediction: Normal (Confidence: 99.90%)
```

```
Running EfficientNetB3...
1/1 ━━━━━━━━━━━━━━━━━━━━ 4s 4s/step
EfficientNetB3 Prediction: Normal (Confidence: 100.00%)
```

```
Running ResNet...
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
ResNet Prediction: Normal (Confidence: 100.00%)
```

```
Running VGG19...
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 874ms/step
VGG19 Prediction: Normal (Confidence: 100.00%)
```

```
======================================================================
SUMMARY: Individual Model Predictions
======================================================================
densenet121         : Normal     - 0.9990 (99.90%)
efficientnetb3      : Normal     - 1.0000 (100.00%)
resnet              : Normal     - 1.0000 (100.00%)
vgg19               : Normal     - 1.0000 (100.00%)
======================================================================

Total models run: 4
```

```
======================================================================
CONFIDENCE MATRIX (Class × Model)
======================================================================
          densenet121  efficientnetb3  resnet  vgg19
Immature        0.001             0.0     0.0    0.0
Mature          0.000             0.0     0.0    0.0
Normal          0.999             1.0     1.0    1.0
======================================================================

======================================================================
WEIGHTED CONFIDENCE FOR EACH CLASS
----------------------------------------------------------------------
Immature    : 0.0002 (0.02%)
Mature      : 0.0000 (0.00%)
Normal      : 0.9998 (99.98%)
----------------------------------------------------------------------

----------------------------------------------------------------------
FINAL ENSEMBLE PREDICTION
======================================================================
Predicted Class: Normal
Ensemble Confidence: 0.9998 (99.98%)
```

```
All models agree on: Normal
======================================================================
```

10.6 Deployment.ipynb

This deployment script transforms a cataract detection machine learning project into a live web application that can be accessed by users worldwide. The code is designed to run in Google Colab, leveraging its cloud-based environment and direct integration with Google Drive to load pre-trained deep learning models for medical image analysis. Core technologies include Streamlit for building the interactive web interface where users can upload eye images, get real-time predictions, and view model confidence scores. The deployment uses pyngrok to create a secure, temporary public URL by tunneling the locally hosted Streamlit app from Colab, so it is instantly accessible beyond the host machine. Key setup steps ensure previous app processes are terminated and the environment is clear for a fresh deployment, with all resources and dependencies loaded automatically in the background. This solution exemplifies rapid, cloud-enabled deployment, enabling AI-powered medical diagnostics without any infrastructure hurdles, and serves as an ideal template for sharing machine learning applications on-demand for research, demonstration, or educational purposes.

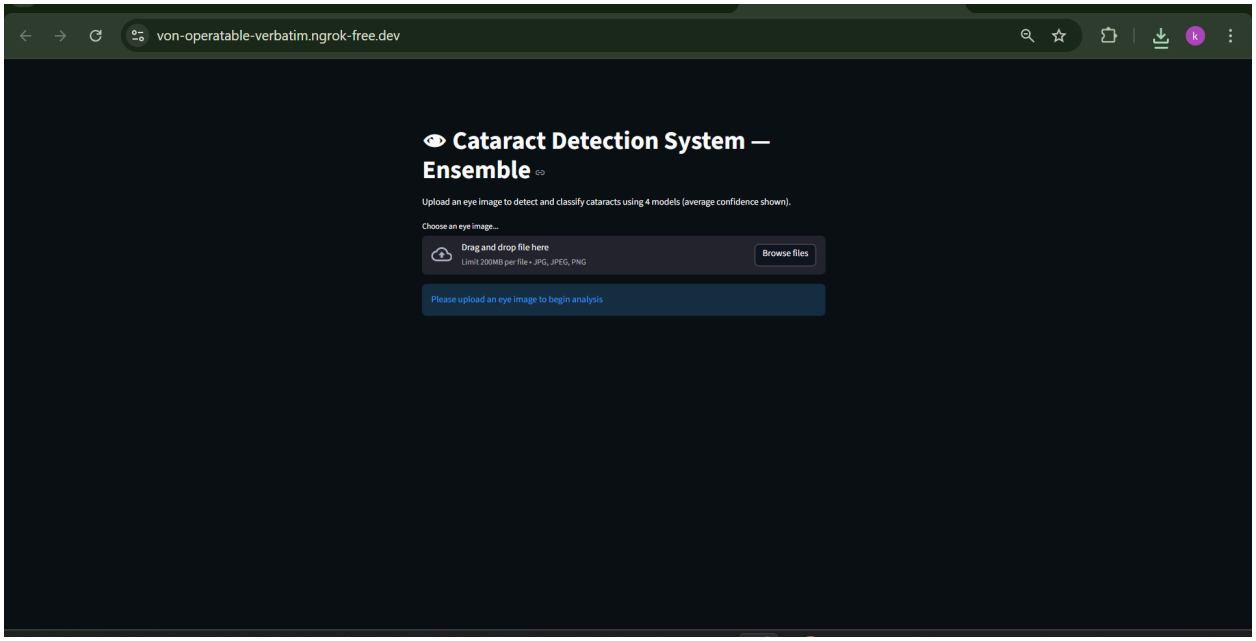
```
# -*- coding: utf-8 -*-
"""Final Deplayment.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1XANyUGvCSFmRrMs2uzOgfoNMZLbyctIU
"""

from google.colab import drive
drive.mount('/content/drive')

!pip install -q streamlit pyngrok

# Commented out IPython magic to ensure Python compatibility.
%%writefile app.py
import streamlit as st
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.densenet import preprocess_input as preprocess_densenet
from tensorflow.keras.applications.efficientnet import preprocess_input as preprocess_efficientnet
from tensorflow.keras.applications.resnet50 import preprocess_input as preprocess_resnet
from tensorflow.keras.applications.vgg19 import preprocess_input as preprocess_vgg19
from PIL import Image
import numpy as np
```

```
st.set_page_config(
  page_title="Cataract Detection Ensemble",
  page_icon="👁️",
  layout="centered"
)

# Your actual Google Drive model paths:
model_info = {
  'densenet121': {'path': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5
files/densenet121(99_03).h5',       'preprocess': preprocess_densenet},
  'efficientnetb3': {'path': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5
files/efficientnetb3(99_84).h5', 'preprocess': preprocess_efficientnet},
  'resnet': {'path': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5 files/resnet(99_91).h5',
'preprocess': preprocess_resnet},
  'vgg19': {'path': '/content/drive/MyDrive/PROJECT/PROJECT-1/trained models/H5 files/vgg19(99_89).h5',
'preprocess': preprocess_vgg19},
}
CLASS_LABELS = ["Immature Cataract", "Mature Cataract", "Normal"]
IMG_SIZE = (224, 224)

@st.cache_resource
def load_models():
  models = {}
  for model_name, item in model_info.items():
    models[model_name] = tf.keras.models.load_model(item['path'], compile=False)
  return models

def preprocess_img(img, pre_func):
  img = img.resize(IMG_SIZE)
  arr = image.img_to_array(img)
  arr = np.expand_dims(arr, axis=0)
  arr = pre_func(arr)
  return arr

def ensemble_predict(image_pil, models):
  all_probs = []
  for model_name, model in models.items():
    pre_func = model_info[model_name]['preprocess']
    processed = preprocess_img(image_pil, pre_func)
    pred = model.predict(processed, verbose=0)[0]
    all_probs.append(pred)
  all_probs = np.array(all_probs)
  avg_probs = all_probs.mean(axis=0)
  best_idx = np.argmax(avg_probs)
  return CLASS_LABELS[best_idx], avg_probs[best_idx]*100, avg_probs

st.title("👁️ Cataract Detection System — Ensemble")
st.markdown("Upload an eye image to detect and classify cataracts using 4 models (average confidence shown).")
```

```
with st.sidebar:
    st.info("""
        This system uses DenseNet121, EfficientNetB3, ResNet, and VGG19.
        The predicted class is based on **average confidence** across all four models.
        Make sure your models are accessible at the paths above!
    """)

uploaded_file = st.file_uploader(
    "Choose an eye image...", type=['jpg', 'jpeg', 'png']
)
if uploaded_file is not None:
    image_pil = Image.open(uploaded_file).convert('RGB')
    st.image(image_pil, caption='Uploaded Image', use_container_width=True)
    if st.button(' Analyze Image'):
        with st.spinner("Analyzing with all models..."):
            models = load_models()
            result, confidence, avg_probs = ensemble_predict(image_pil, models)
        st.markdown(f"### Prediction: <span style='color:green'>{result}</span>", unsafe_allow_html=True)
        st.markdown(f"### Average Confidence: <span style='color:white'>{confidence:.2f}%</span>",
unsafe_allow_html=True)
        st.write("**Average class probabilities from all models:**")
        for label, prob in zip(CLASS_LABELS, avg_probs):
            st.write(f"- **{label}**: {prob*100:.2f}%")
else:
    st.info("Please upload an eye image to begin analysis")


from pyngrok import ngrok
import subprocess
import time

ngrok.set_auth_token("358HfFSjXSmlojBiorl4mNse6xs_MJS8wgAkf3qAxFhZUVnY")
!pkill -f streamlit
subprocess.Popen(["streamlit", "run", "app.py", "--server.port", "8501", "--server.headless", "true"])
time.sleep(7)
public_url = ngrok.connect(8501)
print("\n" + "="*60)
print(f"🎉 Your Cataract Detection site is LIVE!\n🌐 Public URL: {public_url}\n")
print("Keep this Colab tab open while using your public site.")
print("="*60 + "\n")
```

Output:

```
============================================================
🎉 Your Cataract Detection site is LIVE!
🌐 Public URL: NgrokTunnel: "https://von-operatable-verbatim.ngrok-free.dev" -> "http://localhost:8501"

Keep this Colab tab open while using your public site.
============================================================
```



👁 Cataract Detection System — Ensemble

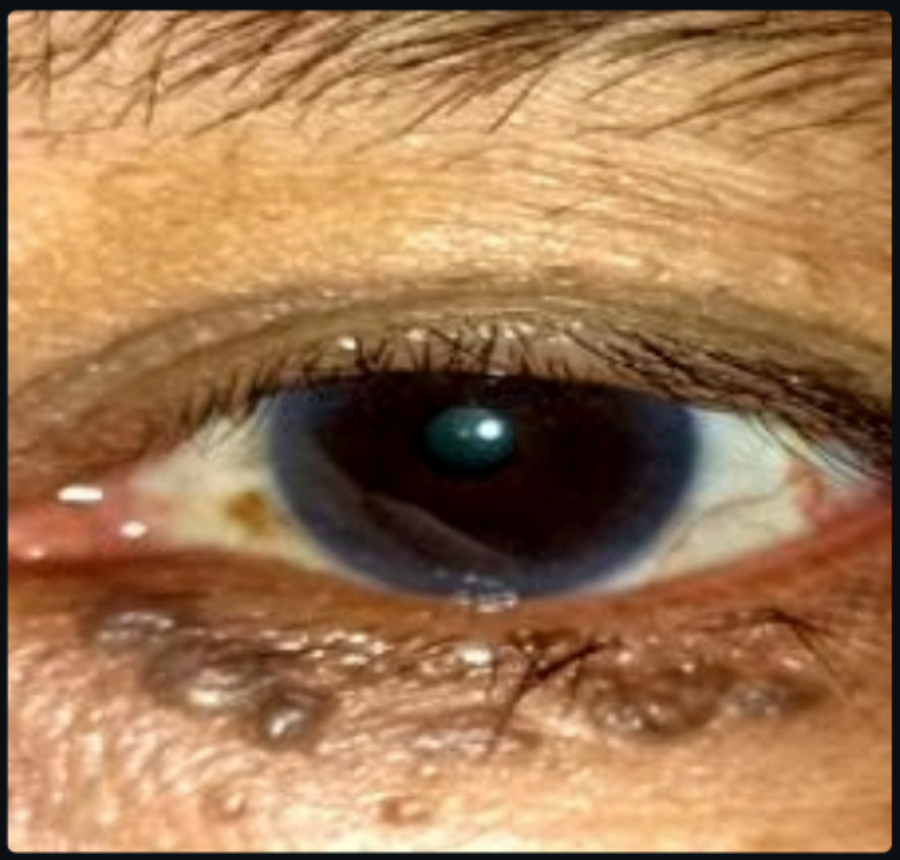
Upload an eye image to detect and classify cataracts using 4 models (average confidence shown).

Choose an eye image...

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

Browse files

testcase(i).jpg 11.5KB



Uploaded Image

Analyze Image

Limit 200MB per file • JPG, JPEG, PNG

Browse files

testcase(i).jpg 11.5KB

Uploaded Image

Analyze Image

Prediction: Immature Cataract

Average Confidence: 99.64%

Average class probabilities from all models:

- **Immature Cataract:** 99.64%
- **Mature Cataract:** 0.21%
- **Normal:** 0.15%