Deep Learning for Automated Cataract and Glaucoma Detection:

A Comparative Study of EfficientNetB0 and DenseNet-121

Abstract

Automated detection of retinal diseases such as cataracts and glaucoma is crucial for early diagnosis and intervention. This study explores the use of deep learning (DL) models, specifically EfficientNetB0 and DenseNet-121, for classifying retinal fundus images into normal, cataract, and glaucoma categories. Eight experiments were conducted, incorporating variations in preprocessing techniques, dataset balancing, loss functions, and fine-tuning strategies. The models were trained using transfer learning, with a custom dense classification head and selective fine-tuning.

EfficientNetB0 achieved the highest accuracy (83%), demonstrating superior performance in glaucoma classification, while the best DenseNet-121 model reached 81% accuracy. Notably, a model trained without fine-tuning the base network achieved 80% accuracy, highlighting the effectiveness of pre-trained feature extraction. Dataset balancing and CLAHE filtering contributed to performance improvements, while focal loss required careful hyperparameter tuning to avoid performance degradation.

Despite these promising results, challenges such as dataset size limitations, fine-tuning instability, and class imbalance sensitivity remain. Future work should explore larger datasets, structured learning rate schedules, and explainable AI techniques for clinical implementation. These findings underscore the potential of DL models in ophthalmology while emphasizing the need for ethical considerations in AI-assisted diagnostics.

Table of Contents

List of Acronyms

Acronyms	Used for
AI	Artificial Intelligence
AUC-ROC	Area Under the Receiver Operating Characteristic Curve
CLAHE	Contrast Limited Adaptive Histogram Equalization
CNN	Convolutional Neural Network
DCNN	Deep Convolutional Neural Network
DL	Deep Learning
GAN	Generative Adversarial Network
GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
ML	Machine Learning
ReLU	Rectified Linear Unit
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative

Introduction

Background & Importance

Machine learning (ML) and deep learning (DL) have shown great potential in medical diagnostics by enabling efficient and cost-effective disease detection (Alowais et al., 2024). These technologies can assist healthcare professionals by providing data-driven insights and improving diagnostic accuracy. Several studies have focused on eye diseases (e.g., Islam et al., 2021), leveraging computer vision to analyze medical images and videos (Junayed et al., 2021). Given the abundance of patient data, ML and DL models are increasingly explored for autonomous disease detection (Syarifah et al., 2020).

Cataracts, caused by lens clouding, progressively impair vision and can lead to blindness, particularly in individuals over 60 (Hashemi et al., 2020). Early detection can reduce visual impairment and prevent costly surgeries (Pascolini et al., 2010). Cataracts account for 33% of visual impairments and 51% of global blindness cases (Yang et al., 2016).

Glaucoma, another leading cause of irreversible blindness, damages retinal ganglion cells and the optic nerve (Foster et al., 2002). Structural changes in the optic nerve head drive disease progression (Weinreb et al., 2014). Open-angle glaucoma, the most common type, increases intraocular pressure due to drainage system blockage (Junayed et al., 2021). While incurable, early detection can mitigate vision loss (Storgaard et al., 2021).

Objective & Scope

Due to their efficient architectures, DenseNet-121 and EfficientNetB0 are widely used deep convolutional neural network (DCNN) models for medical image analysis (Junayed et al., 2021; Alwakid et al., 2023). The primary objective of this study is to classify retinal images into normal, cataract, and glaucoma categories using transfer learning. By leveraging pre-trained models and fine-tuning techniques, this approach aims to achieve high accuracy and robust feature extraction, ensuring reliable disease detection.

Report Overview

This study investigates DL models for cataract and glaucoma detection, utilizing transfer learning and image preprocessing techniques to improve classification. Experiments include dataset balancing, CLAHE filtering, and fine-tuning of DenseNet-121 and EfficientNetB0. Performance is evaluated using accuracy, precision, recall, F1-score, and area under the ROC curve (AUC-ROC) metrics. The study examines feature extraction, fine-tuning strategies, and the optimization of categorical crossentropy and focal loss. It also addresses ethical considerations in AI-driven diagnostics and proposes future improvements.

Methods

Dataset & Preprocessing

The dataset comprises high-resolution retinal fundus images categorized into three classes: Normal (300 images), Cataract (100 images), and Glaucoma (101 images). Raw images vary in resolution, including 2464×1632, 2592×1728, and 1848×1224 pixels. Since both DL models used in this study require a 224×224×3 input size, images were standardized by cropping black padding, center-cropping to a square aspect ratio, and rescaling pixel values to the [0,1] range to improve numerical stability during training.

Two dataset splitting strategies were employed. The first followed a 75:15:15 ratio (350 training, 75 validation, 75 test images) and was exclusive to the first experiment. The second, used in rest of experiments, applied a 60:20:20 ratio, increasing validation and test set sizes.

```
TRAIN SET:
1_normal: 210 images
2_cataract: 70 images
3_glaucoma: 70 images

VAL SET:
1_normal: 45 images
2_cataract: 15 images
3_glaucoma: 15 images

TEST SET:
1_normal: 45 images
2_cataract: 15 images
3_glaucoma: 16 images
```

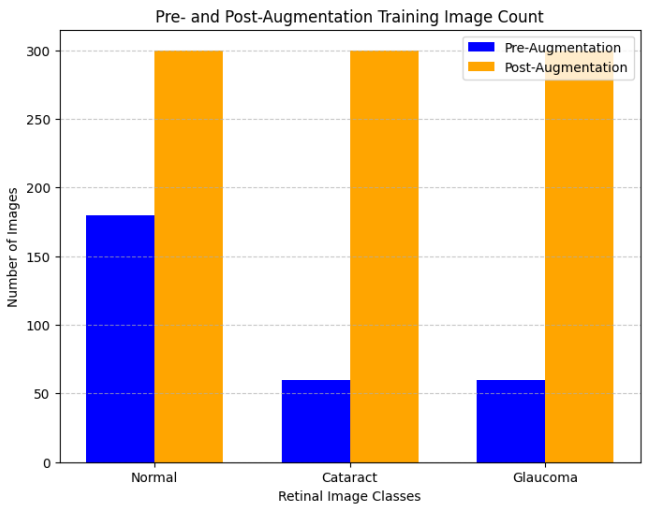
```
Class '1_normal': 180 train, 60 val, 60 test
Class '2_cataract': 60 train, 20 val, 20 test
Class '3_glaucoma': 60 train, 20 val, 21 test
```

Figure 1: Dataset Splitting Approaches.
This figure illustrates two different dataset splitting strategies used in the study. The first approach (left) employs a 70:15:15 ratio, where 70% of the images are allocated for training, 15% for validation, and 15% for testing. The second approach (right) follows a 60:20:20 ratio, distributing 60% of the images for training, 20% for validation, and 20% for testing. In both approaches, these proportions were applied separately to each class (normal, cataract, and glaucoma), ensuring that the class distributions remained consistent across the training, validation, and test sets.

To address class imbalance and improve training performance, data augmentation was implemented, expanding each class to 300 images (Figure 2). Transformations included a combination of horizontal flipping, random rotations (-180° to 180°), zooming, and width/height shifting, enhancing diversity and mitigating overfitting.

Given the variability in imaging technologies used for retinal capture, CLAHE was applied as a final preprocessing step across all datasets (training, validation, and test) to ensure consistency in image quality (Figure 3). This technique enhances contrast, refines fine details and textures, and improves low-resolution feature visibility, optimizing images for model training and reducing the risk of domain shift between different dataset splits (Reza, 2004).

Figure 2: Pre- and Post-Augmentation Training Image Count
This bar chart illustrates the impact of data augmentation in balancing the training dataset. The initial training set was imbalanced, containing 180 normal, 60 cataract, and 60 glaucoma images (blue bars). To address this imbalance and enhance model generalization, augmentation techniques were applied, generating additional images to equalize the dataset to 300 images per class (orange bars).



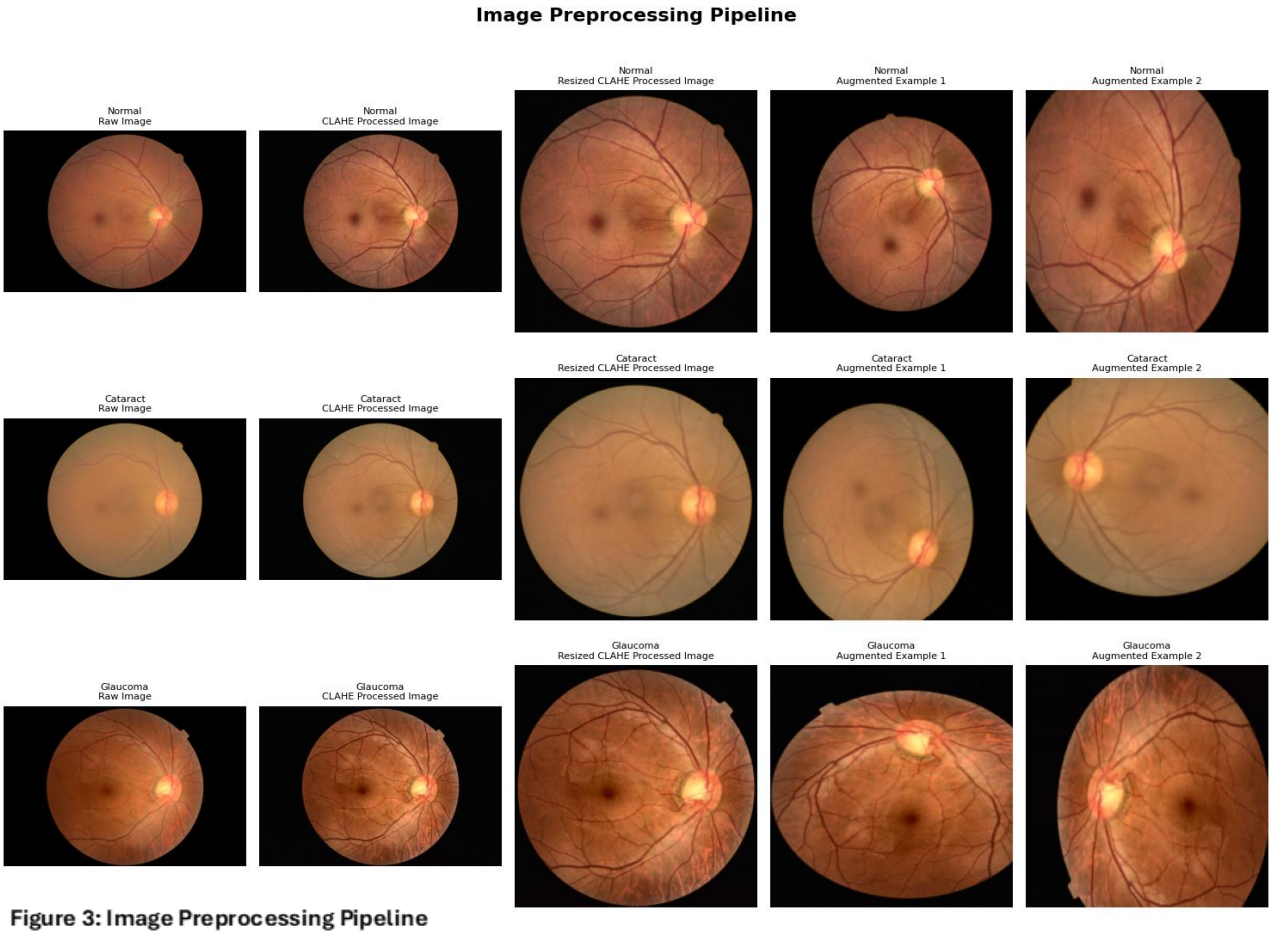


Figure 3: Image Preprocessing Pipeline

This figure illustrates the preprocessing steps applied to the retinal fundus images. Each row represents a different disease class: Normal, Cataract, and Glaucoma. The first column displays the raw images, which vary in resolution and may contain black padding. The second column shows images processed with CLAHE to enhance contrast and fine details. The third column presents resized versions of the CLAHE-processed images, cropped to a square aspect ratio and downscaled to 224×224 pixels to ensure consistency across the dataset. The final two columns showcase augmented images generated using random transformations, including horizontal flipping, rotation (-180° to 180°), zooming, and shifting.

Model & Training Details

Eight experiments were conducted to evaluate different deep learning approaches for retinal disease classification, varying in training strategy, loss function, and preprocessing techniques. These experiments examined feature extraction, dataset balancing, full versus partial fine-tuning, focal loss optimization, class weighting, learning rate scheduling, and CLAHE filtering. Detailed specifications for each experiment can be found in the Appendix and Table 1.

The models were implemented using TensorFlow Keras, leveraging its high-level API for deep learning model design and training. To optimize classification performance, a custom dense classification head was added atop the pre-trained models. The modifications included:

- A Global Average Pooling (GAP) layer: Aggregates spatial features by averaging each feature map from the convolutional layers of the pre-trained model, reducing dimensionality while retaining essential information and preventing overfitting.
- A Batch Normalization layer: Normalizes activations to stabilize training, enabling faster convergence and higher learning rates.

- A Dense Layer (512/384 units, ReLU activation): Extracts high-level abstract features from the learned representations.
- Two Dropout layers (0.3–0.6): Regularizes the model by randomly disabling neurons, reducing overfitting.
- Final Softmax Layer (3 units): Outputs class probabilities for normal, cataract, and glaucoma classification.

Training was conducted in two phases. Initially, the base model's convolutional layers were frozen, training only the custom classification head at a higher learning rate (0.00016084 or 0.00033609) for 30 iterations to stabilize learning while leveraging pre-trained features. In the second phase, selected layers or the entire base model were unfrozen for fine-tuning at a lower learning rate (1e-7 to 1e-5) for max 50 iterations.

The primary loss function was categorical crossentropy, suited for multi-class classification. It measures the difference between the predicted class probabilities and the true labels, with lower values indicating better model performance. Focal loss was used as an alternative to handle class imbalance by down-weighting well-classified samples and emphasizing hard-to-classify cases, making the model more sensitive to minority classes. The Adam optimizer was employed for its adaptive learning rate, with early stopping and learning rate reduction callbacks monitoring validation loss or F1-score respectively.

Two deep learning architectures were used as base models: DenseNet-121, a densely connected CNN that enhances feature reuse and gradient flow for improved efficiency, and EfficientNetB0, a lightweight CNN designed with compound scaling to optimize accuracy while maintaining computational efficiency. DenseNet-121 improves parameter efficiency, while EfficientNetB0 achieves high accuracy with fewer parameters (Wu et al., 2022), making it well-suited for medical imaging applications (Figures 4 & 5).

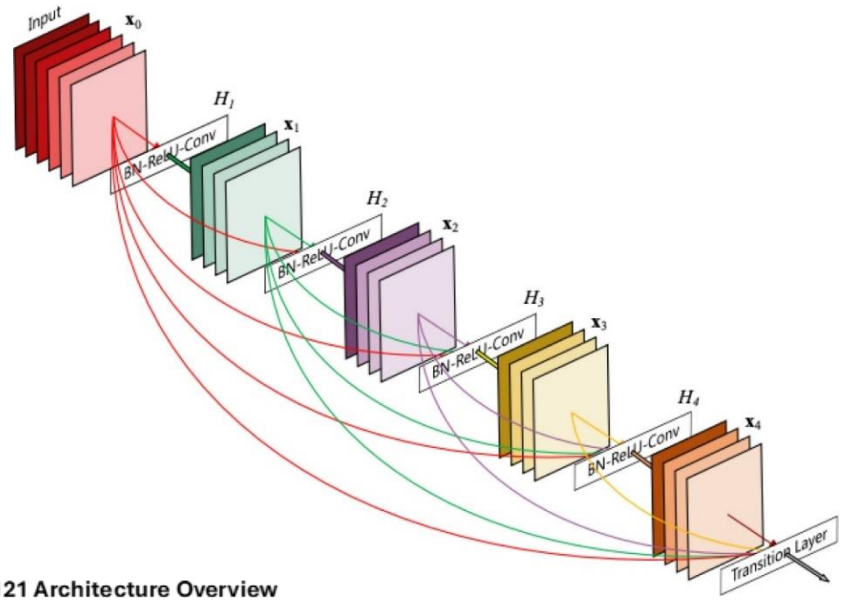


Figure 4: DenseNet-121 Architecture Overview

The figure illustrates the core structure of DenseNet-121, a densely connected CNN. Unlike traditional CNNs, where each layer passes information only to the next layer, DenseNet introduces dense connectivity, where each layer receives input from all previous layers and passes its own feature maps to all subsequent layers within the block. Each layer consists of Batch Normalization (BN), ReLU activation, and a 1x1 or 3x3 Convolution (Conv), facilitating feature reuse and improving gradient flow. Transition layers are used to downsample feature maps and control dimensionality. This structure significantly enhances parameter efficiency and enables deeper networks with fewer parameters.

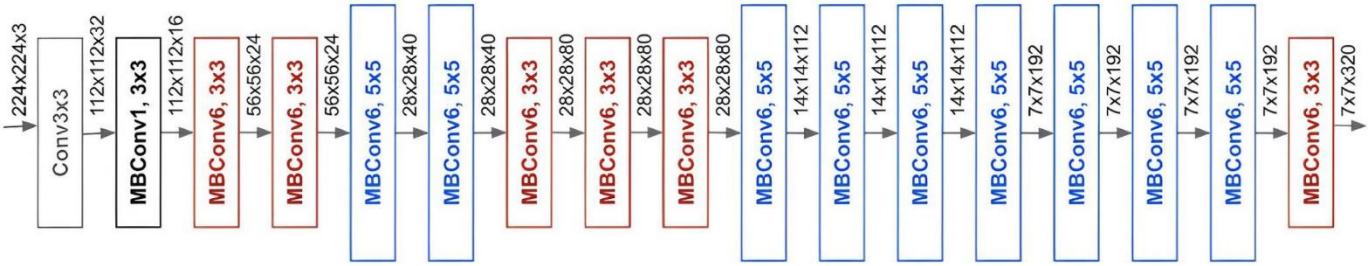


Figure 5: EfficientNetB0 Architecture Overview

The figure depicts the architecture of EfficientNetB0, a lightweight convolutional neural network optimized for efficiency and accuracy. The network starts with a 3×3 convolutional layer, followed by a series of Mobile Inverted Bottleneck Convolution (MBConv) blocks, which improve computational efficiency by using depthwise separable convolutions and squeeze-and-excitation mechanisms. The MBConv layers are shown in different sizes (3×3 and 5×5), with expansion ratios (notated as MBConv1, MBConv6) to enhance feature extraction. The network progressively reduces spatial dimensions while increasing channel depth, achieving a balance between depth, width, and resolution using compound scaling, which helps maintain high accuracy with fewer parameters compared to traditional architectures.

Evaluation Metrics

The performance of the trained models was assessed using multiple evaluation metrics to ensure a comprehensive analysis of classification effectiveness. Accuracy was used to measure the overall correctness of predictions, calculated as the ratio of correctly classified instances to the total number of instances:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP, TN, FP, and FN represent true positives, true negatives, false positives, and false negatives, respectively. Precision, given by

$$\text{Precision} = \frac{TP}{TP + FP}$$

measured the proportion of correctly predicted positive samples out of all predicted positive samples, ensuring the reliability of the model's classifications. Recall (Sensitivity), computed as

$$\text{Recall} = \frac{TP}{TP + FN}$$

evaluated the model's ability to correctly identify all actual positive cases, emphasizing its effectiveness in detecting diseased instances. F1-Score, defined as the harmonic mean of precision and recall,

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

was particularly useful in handling class imbalances by balancing precision and recall.

Additionally, the AUC-ROC was used to evaluate the model's ability to distinguish between classes based on predicted probabilities, with higher values indicating better discrimination. Since a softmax activation was used, classification was based on selecting the highest probability class rather than varying thresholds. To assess per-class performance, class-specific precision, recall, and F1-score were recorded for normal, cataract, and glaucoma cases. Additionally, macro-averaged and weighted-averaged metrics were computed, with the

former treating all classes equally and the latter accounting for class imbalance. All models were trained and evaluated using Google Colab (Google, n.d.).

Table 1: Summary of Experiment Configurations

Experiment ID	Dataset Split	Preprocessing	Model	Loss Function	Fine-tuning layers	Class Weighting	Initial Phase LR	Fine-Tuning Phase LR
1	70:15:15	Resclaling Imbalanced Dataset	DenseNet-121	Categorical Crossentropy	All	No	1.6e-4	1e-5 (Fixed)
2	60:20:20	Resclaling Balanced Dataset	DenseNet-121	Categorical Crossentropy	All	Yes	1.6e-4	1e-5 (Fixed)
3	60:20:20	Resclaling Balanced Dataset CLAHE	DenseNet-121	Categorical Crossentropy	All	Yes	1.6e-4	Warm-up (1e-7 → 5e-6, 12 epochs), then 5e-6
4	60:20:20	Resclaling Balanced Dataset CLAHE	DenseNet-121	Categorical Crossentropy	Last 82	Yes	3.3e-4	1e-5 (Fixed)
5	60:20:20	Resclaling Balanced Dataset CLAHE	DenseNet-121	Categorical Crossentropy	No fine-tuning	Yes	3.3e-4	NA
6	60:20:20	Resclaling Balanced Dataset CLAHE	DenseNet-121	Focal Loss	No fine-tuning	Yes	3.3e-4	NA
7	60:20:20	Resclaling Balanced Dataset CLAHE	EfficientNetB0	Focal Loss	Last 110	Yes	3.3e-4	Warm-up (1e-7 → 5e-6, 20 epochs), then 5e-6
8	60:20:20	Resclaling Balanced Dataset CLAHE	EfficientNetB0	Categorical Crossentropy	Last 50	Yes	3.3e-4	1e-6 (Fixed)

This table summarizes the configurations of all experiments conducted in this study, outlining dataset splitting strategies, preprocessing methods, model selection, loss functions, fine-tuning approaches, and class weighting. Additionally, the last two columns are dedicated to the learning rate schedules. The Initial Phase LR represents the learning rate used when the base model's convolutional layers were frozen, allowing only the classification head to be trained, while the Fine-Tuning Phase LR indicates the learning rate applied once selected layers or the entire base model were unfrozen. If "Fixed" is noted, a constant learning rate was used during fine-tuning, whereas "NA" denotes experiments where no fine-tuning was performed. The table captures different loss functions used per experiment as well as the effects of CLAHE preprocessing, pixel values rescaling to [0,1], and dataset balancing strategies on model performance.

Results

Model Performance

All performance metrics were computed on the test set using the *sklearn.metrics* module in Python. The key findings from each experiment are summarized in Table 2, while per-class metrics can be found in Table 3.

Among all experiments, Experiment 8 (EfficientNetB0) achieved the highest classification performance, with an accuracy of 83% and a relatively strong F1-score of 71% for the glaucoma class, which consistently posed the greatest classification challenge (Figure 6). It also had the highest discriminatory ability, as indicated by its per-class AUC values (Normal: 0.90, Cataract: 0.94, Glaucoma: 0.91; Figure 7). The best-performing DenseNet-121, Experiment 3, reached 81% accuracy and an F1-score of 67% for glaucoma (Figure 8). Experiment 7, the second-best model overall (82% accuracy), also leveraged EfficientNetB0 but employed focal loss. While it demonstrated superior recall (81%), its precision (63%) for glaucoma was lower compared to Experiment 8 (Figure 9).

Furthermore, dataset balancing and class weighting positively impacted model performance. Notably, Experiment 2, which integrated both techniques, outperformed the baseline model (Experiment 1) with a 78% accuracy and demonstrated enhanced discriminatory ability, as evidenced by its per-class AUC values (Normal: 0.90, Cataract: 0.93, Glaucoma: 0.90; Figure 10).

Although fine-tuning did not consistently yield improvements across all experiments, Experiment 7 and Experiment 3 demonstrated greater stability in accuracy and loss during the transition from feature extraction to fine-tuning (Figures 11 and 12). Experiment 8, despite achieving the highest overall accuracy, experienced a significant drop in accuracy and an increase in loss during fine-tuning, as shown in Figure 13, indicating potential instability when unfreezing additional layers without a warm-up phase (see Table 1 and Appendix).

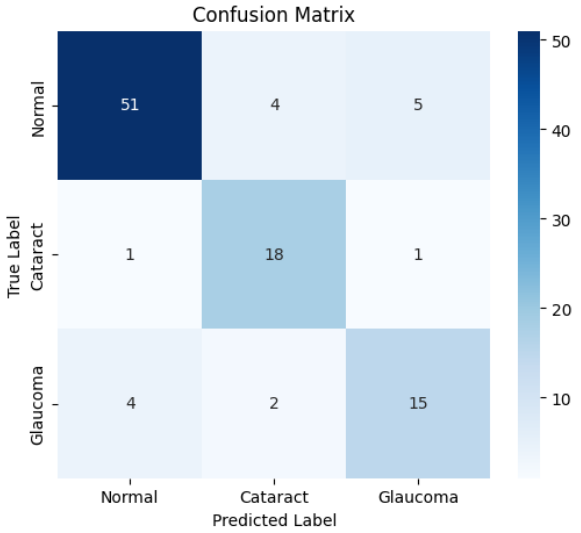
Visualization

Table 2: Summary of Overall Model Performance Across Experiments

Experiment ID	Accuracy	Macro Precision	Macro Recall	Macro F1	Weighted Precision	Weighted Recall	Weighted F1	Support
1	75%	73%	73%	73%	75%	75%	75%	76 images
2	78%	74%	79%	76%	80%	78%	79%	101 images
3	81%	79%	79%	79%	81%	81%	81%	101 images
4	78%	75%	80%	76%	79%	78%	78%	101 images
5	80%	78%	77%	77%	80%	80%	80%	101 images
6	75%	72%	75%	73%	76%	75%	76%	101 images
7	82%	78%	83%	80%	84%	82%	83%	101 images
8	83%	79%	82%	80%	84%	83%	83%	101 images

Table 3: Per-Class Classification Performance Across All Experiments

	Precision	Recall	F1-score	AUC-ROC	Total images
Experiment 1					
Normal	80%	78%	79%	0.85	45
Cataract	65%	73%	69%	0.94	15
Glaucoma	73%	69%	71%	0.93	16
Experiment 2					
Normal	88%	77%	82%	0.90	60
Cataract	67%	80%	73%	0.93	20
Glaucoma	68%	81%	74%	0.90	21
Experiment 3					
Normal	85%	85%	85%	0.87	60
Cataract	85%	85%	85%	0.98	20
Glaucoma	67%	67%	67%	0.88	21
Experiment 4					
Normal	87%	77%	81%	0.86	60
Cataract	69%	100%	82%	0.98	20
Glaucoma	68%	62%	65%	0.88	21
Experiment 5					
Normal	85%	85%	85%	0.89	60
Cataract	84%	80%	82%	0.97	20
Glaucoma	64%	67%	65%	0.87	21
Experiment 6					
Normal	84%	77%	80%	0.87	60
Cataract	77%	85%	81%	0.97	20
Glaucoma	54%	62%	58%	0.88	21
Experiment 7					
Normal	94%	82%	88%	0.90	60
Cataract	77%	85%	81%	0.96	20
Glaucoma	63%	81%	71%	0.88	21
Experiment 8					
Normal	91%	85%	88%	0.90	60
Cataract	75%	90%	82%	0.94	20
Glaucoma	71%	71%	71%	0.91	21



	precision	recall	f1-score	support
Normal	0.91	0.85	0.88	60
Cataract	0.75	0.90	0.82	20
Glaucoma	0.71	0.71	0.71	21
accuracy			0.83	101
macro avg	0.79	0.82	0.80	101
weighted avg	0.84	0.83	0.83	101

Figure 6: Confusion matrix and classification report for the best-performing model (Experiment 8, EfficientNetB0). The confusion matrix (top) visually represents the model's classification performance, showing the number of correctly and incorrectly classified instances for each class. The diagonal elements indicate correctly classified cases, while off-diagonal values represent misclassifications. The classification report (bottom) provides precision, recall, and F1-score for each class, highlighting an overall accuracy of 83% with the highest recall for the Cataract class (90%), while Glaucoma classification remained the most challenging.

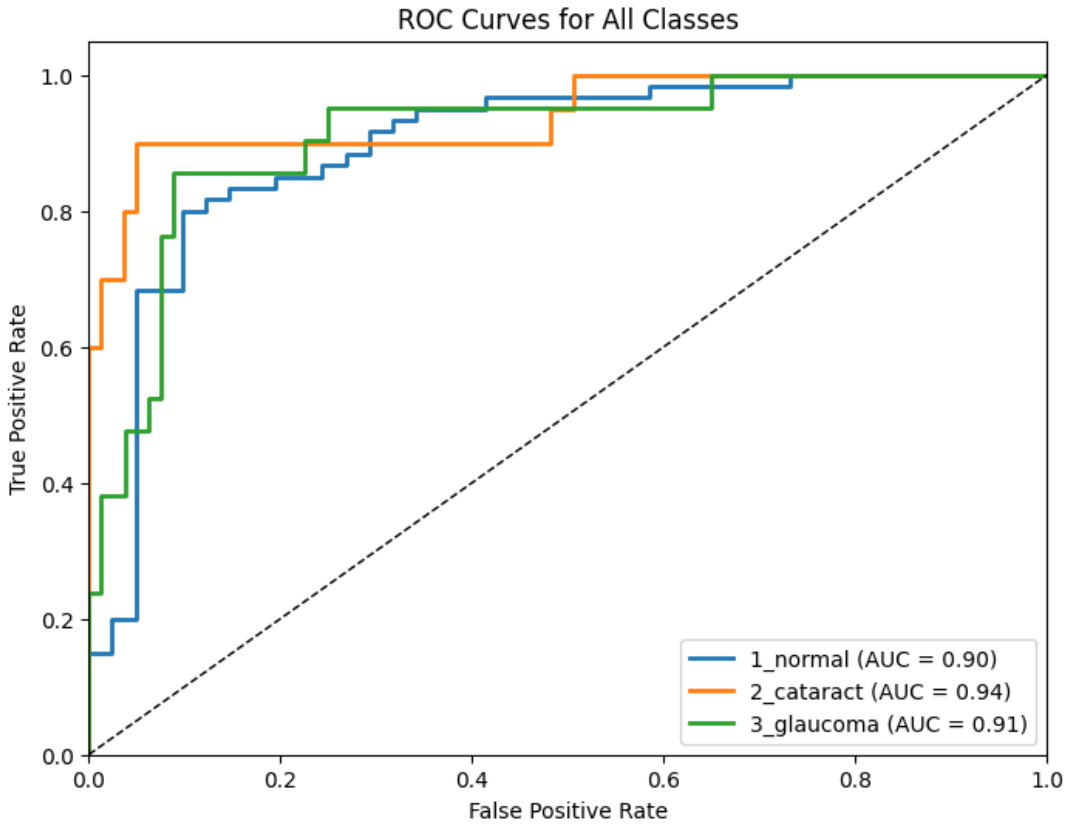
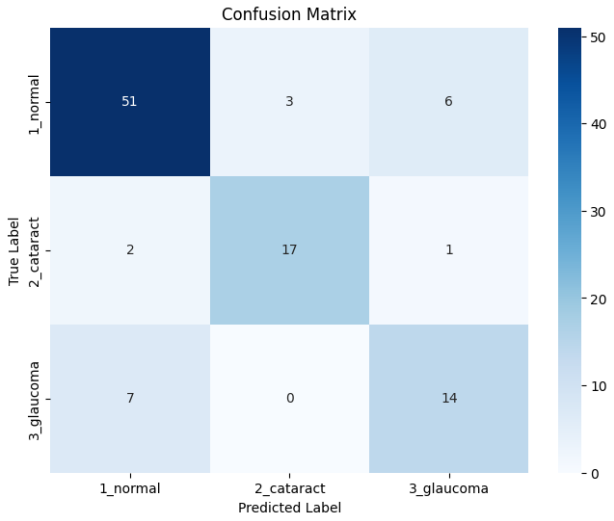


Figure 7. ROC curves for all three classes (Normal, Cataract, and Glaucoma) from the best-performing model (Experiment 8, EfficientNetB0). The ROC curve illustrates the model's ability to distinguish between classes based on predicted probabilities. The AUC values indicate the model's discriminatory power, with Cataract achieving the highest AUC (0.94), followed by Glaucoma (0.91) and Normal (0.90). The closer the curve is to the top-left corner, the better the model's classification performance. The dashed diagonal line represents random chance (AUC = 0.50).

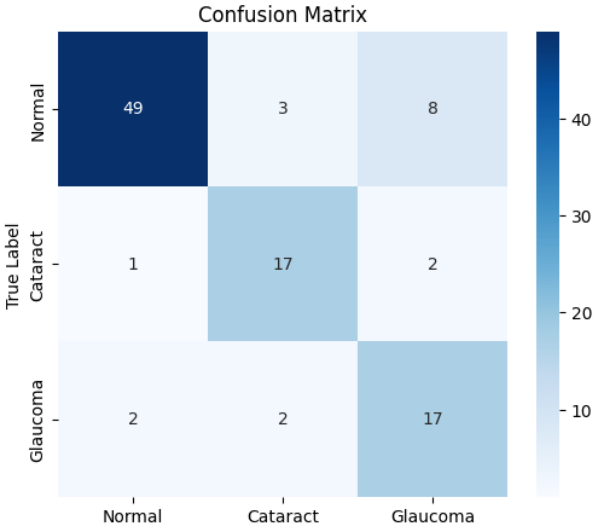


Classification Report:

	precision	recall	f1-score	support
1_normal	0.85	0.85	0.85	60
2_cataract	0.85	0.85	0.85	20
3_glaucoma	0.67	0.67	0.67	21
accuracy			0.81	101
macro avg	0.79	0.79	0.79	101
weighted avg	0.81	0.81	0.81	101

Figure 8: Confusion Matrix and Classification Report for the Best DenseNet-121 Model (Experiment 3)

The confusion matrix (top) illustrates the model's performance across the three classes: normal, cataract, and glaucoma. The majority of normal cases were correctly classified (51/60), while misclassifications primarily occurred between glaucoma and normal cases. The classification report (bottom) provides precision, recall, and F1-score values, highlighting the model's balanced performance across classes, with an overall accuracy of 81%. Despite achieving strong precision and recall for normal and cataract cases (85% each), glaucoma classification remained the most challenging, with an F1-score of 67%.



	precision	recall	f1-score	support
Normal	0.94	0.82	0.88	60
Cataract	0.77	0.85	0.81	20
Glaucoma	0.63	0.81	0.71	21
accuracy			0.82	101
macro avg	0.78	0.83	0.80	101
weighted avg	0.84	0.82	0.83	101

Figure 9: Confusion Matrix and Classification Report for Experiment 7 (EfficientNetB0 with Focal Loss)

This figure presents the confusion matrix and classification report for Experiment 7, which is the second-best performing model. The confusion matrix (top) illustrates the model's classification performance across the three retinal disease categories: Normal, Cataract, and Glaucoma. The highest misclassification occurs in the Normal class, where 8 samples were misclassified as Glaucoma. The classification report (bottom) shows an overall accuracy of 82%, with the best recall (81%) for the Glaucoma class, indicating the model's strong ability to identify glaucoma cases. However, its precision for Glaucoma (63%) is lower, suggesting a higher false positive rate. The macro-averaged F1-score is 80%, while the weighted F1-score reaches 83%, reflecting a balanced performance across the three classes.

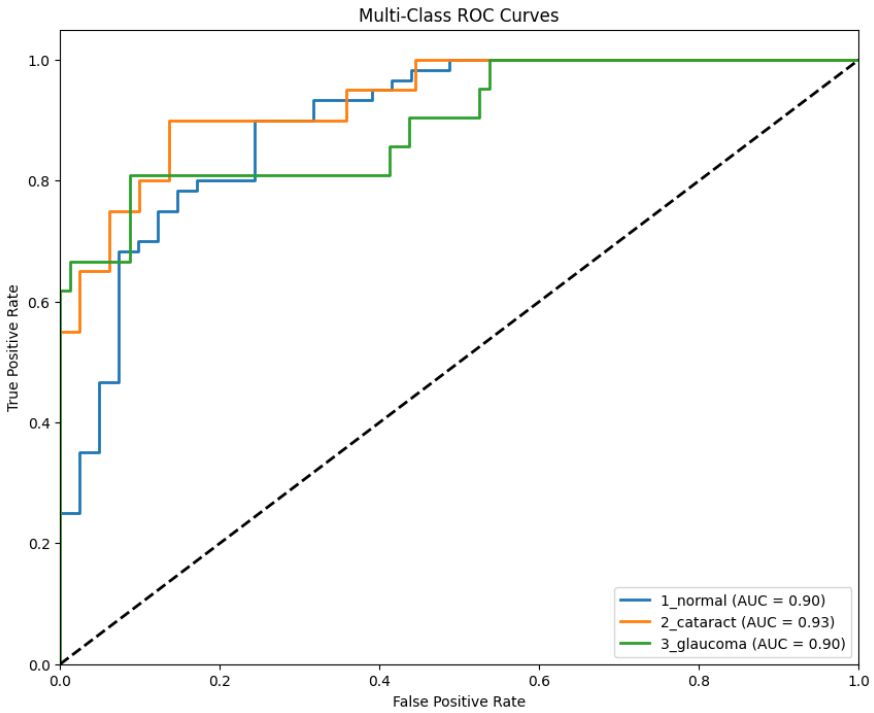


Figure 10: Multi-Class ROC Curves for Experiment 2 (DenseNet-121 with Balanced Dataset).
The ROC curve illustrates the model's ability to distinguish between classes based on predicted probabilities. The AUC values indicate the model's discriminatory power, with Cataract achieving the highest AUC (0.93), followed by Normal (0.90) and Glaucoma (0.90). The closer the curve is to the top-left corner, the better the model's classification performance. The dashed diagonal line represents random chance (AUC = 0.50), while curves further from this line indicate superior classification ability.

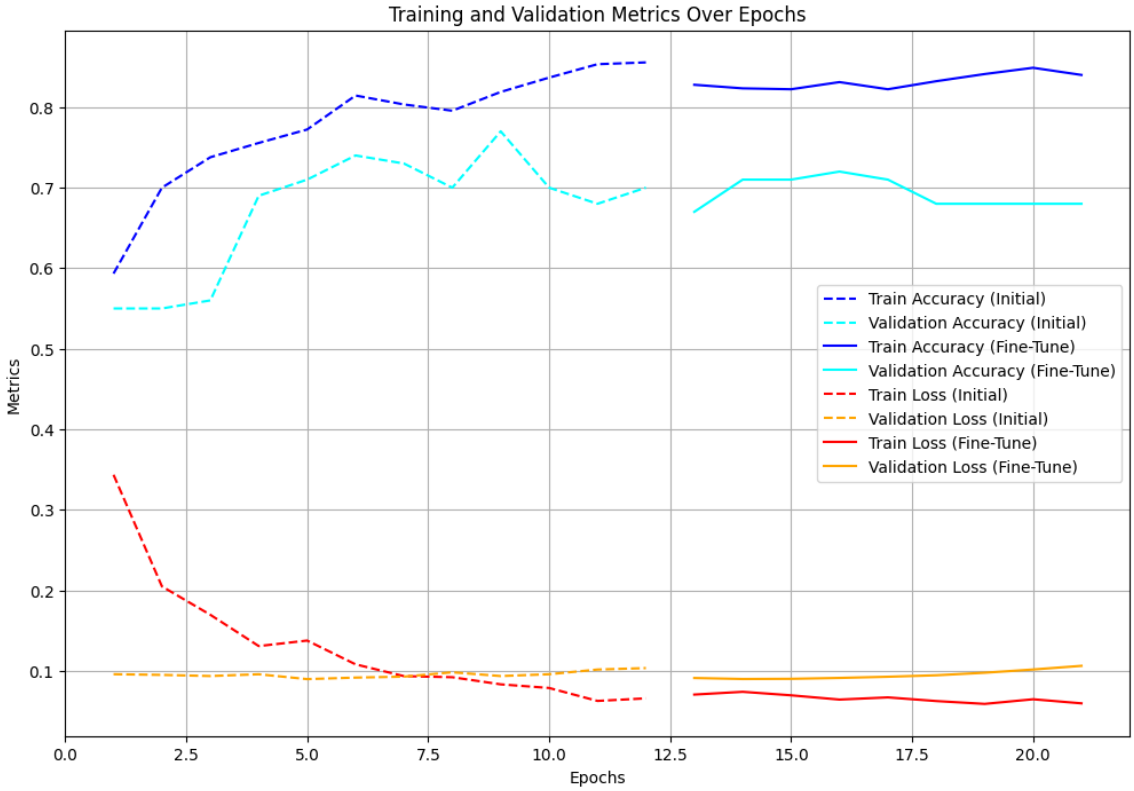


Figure 11: Training and Validation Metrics Over Epochs for Experiment 7 (EfficientNetB0, focal loss)
This figure illustrates the progression of training and validation accuracy and loss during both the initial feature extraction phase and the fine-tuning phase for Experiment 7. Dashed lines represent the initial training phase, where only the classification head was trained, while solid lines indicate the fine-tuning phase, where selected base model layers were unfrozen. Training and validation accuracy increased steadily in the initial phase, while fine-tuning maintained stability without overfitting. The loss curves show a decline in training loss, with validation loss remaining relatively stable, indicating that fine-tuning improved model generalization.

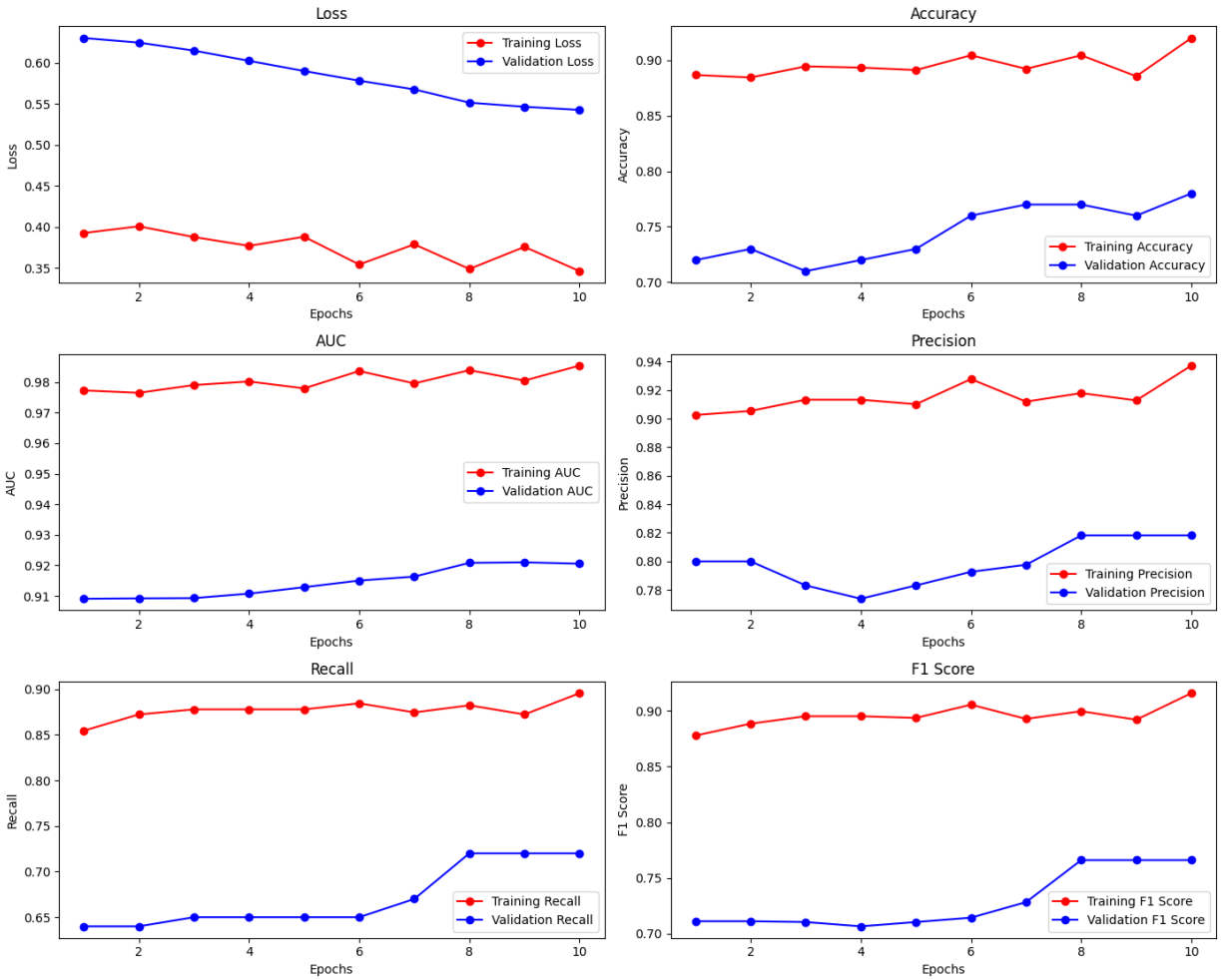


Figure 12: Training vs. Validation Metrics During Fine-Tuning (DenseNet-121, Experiment 3)

This figure presents the evolution of key training and validation metrics over 10 fine-tuning epochs. Each subplot depicts a different evaluation metric, including loss, accuracy, AUC, precision, recall, and F1-score. The red lines represent training performance, while the blue lines correspond to validation performance. Fine-tuning resulted in steady improvements across validation metrics, particularly in accuracy, recall, and F1-score, indicating better model generalization. Validation loss remained relatively stable, suggesting that fine-tuning enhanced performance without overfitting. The increasing AUC values further confirm the model's improved discriminatory capability.

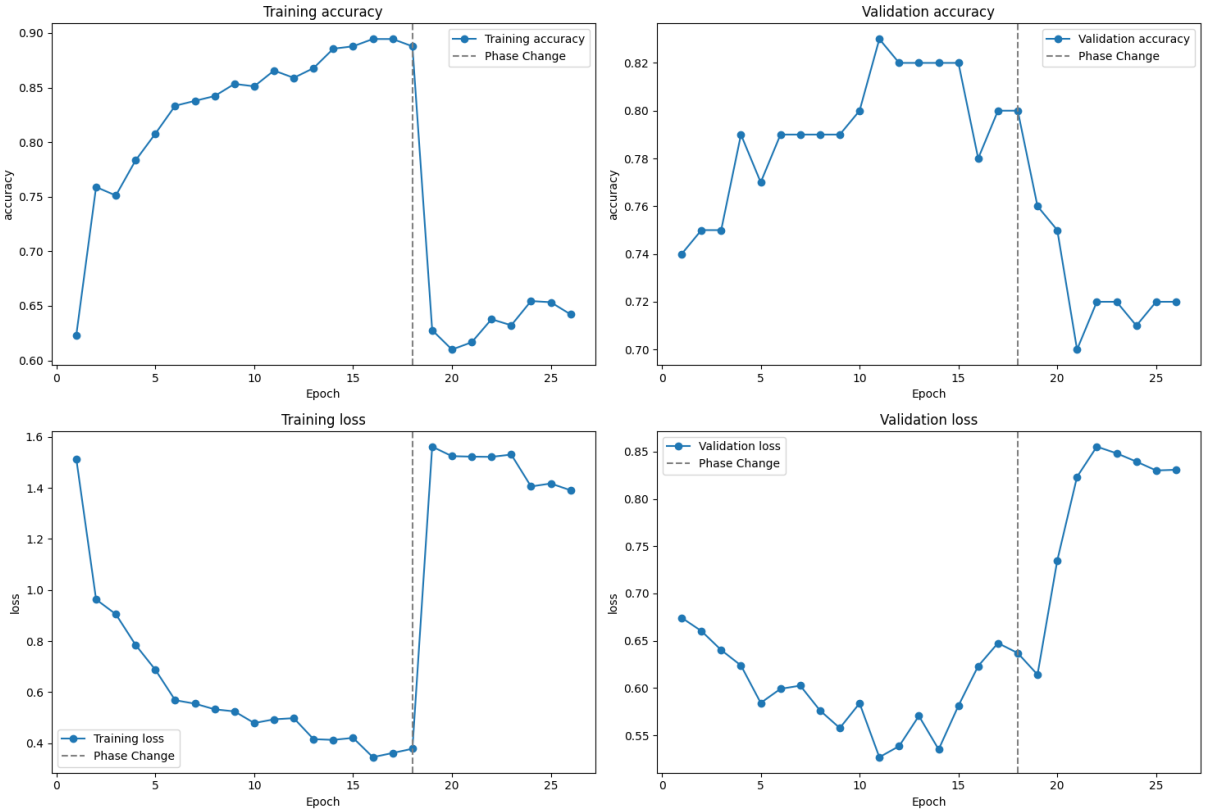


Figure 13. Training and Validation Metrics for Experiment 8 (EfficientNetB0) Over Epochs

This figure illustrates the training and validation accuracy (top) and loss (bottom) across epochs for Experiment 8. The dashed vertical line represents the transition from feature extraction to fine-tuning. Prior to fine-tuning, both training and validation accuracy steadily improve while loss decreases, indicating stable learning. However, upon unfreezing additional layers and applying a lower constant learning rate (1e-6), a significant drop in accuracy and an increase in loss are observed, suggesting instability in fine-tuning. The sharp fluctuations highlight potential issues with abrupt parameter updates, which might have been mitigated by using a more gradual learning rate adjustment similar to the warm-up strategy in Experiments 7 and 3 (see Appendix).

Discussion & Ethical Implications

Key Insights & Limitations

EfficientNetB0 and DenseNet-121 are widely used deep learning architectures in medical image analysis, each offering distinct advantages. DenseNet-121, with its densely connected layers, promotes feature reuse and efficient gradient propagation, making it particularly effective for extracting complex visual patterns in retinal images (Nandakumar et al., 2023; Alwakid et al., 2023). Conversely, EfficientNetB0 employs compound scaling, balancing depth, width, and resolution to achieve high accuracy with fewer parameters, making it well-suited for computationally constrained environments (Tan & Le, 2019). Previous studies, such as Canayaz (2022) and Islam et al. (2021), have demonstrated the effectiveness of these architectures in ophthalmology, achieving robust performance in detecting diabetic retinopathy, glaucoma, and other retinal diseases. However, many of these studies utilized significantly larger datasets than the one available for this study. Given the relatively small dataset size, EfficientNetB0 was particularly attractive due to its efficiency in learning meaningful features with fewer parameters, reducing the risk of overfitting as evidenced by Tan & Le (2019) and Wu et al. (2022). Ultimately, while both architectures yielded strong

classification results, EfficientNetB0 achieved slightly better overall performance, particularly in distinguishing glaucoma cases, across both categorical crossentropy and focal loss experiments.

A study by Alyoubi et al. (2021) used EfficientNetB0 on the APTOS 2019 dataset and achieved an accuracy of 82%. Despite working with a significantly smaller dataset, our EfficientNetB0-based model (Experiment 8) achieved an even higher accuracy of 83%, demonstrating its effectiveness even under data-constrained conditions. Conversely, dataset size remains a critical factor in deep learning performance. Alwakid et al. (2023) found that models trained on the smaller DDS dataset exhibited lower accuracy than those trained on APTOS, emphasizing the necessity of sufficient training data for deep learning models. The reduced generalization ability of models trained on small datasets can lead to biases and overfitting, highlighting the importance of data augmentation, image filtering, and proper transfer learning. In this study, small performance gains were observed with CLAHE filtering compared to non-CLAHE models, similar to findings reported by Alwakid et al. (2023). However, its overall contribution was minor, suggesting that additional contrast enhancement techniques could be explored.

This study also highlights that fine-tuning is not always necessary to achieve strong classification performance. Notably, one of the models (Experiment 5) trained without fine-tuning the base model achieved an accuracy of 80%, demonstrating that solely using the base model for feature extraction, combined with a well-designed custom classification head, can yield competitive results. Despite its simple design, the custom head effectively utilized pre-trained ImageNet features, reinforcing findings by Canayaz (2022), where feature extraction from pre-trained models, coupled with appropriate classification layers, proved sufficient for strong performance. This is particularly beneficial in low-data scenarios, where fine-tuning can introduce instability rather than improve accuracy.

As already highlighted, one of the primary limitations of this study is the dataset size, which may have constrained model generalization. While EfficientNetB0 demonstrated strong performance despite data constraints, deep learning models generally benefit from larger datasets to capture diverse feature representations and minimize overfitting. Experiment 8 showed that fine-tuning a complex base model on a small dataset without a carefully structured learning rate schedule and warm-up phase led to instability, causing a drop in accuracy and an increase in loss. This underscores the necessity of gradual fine-tuning, especially when using more complex pre-trained models on limited datasets.

Another limitation stems from the effectiveness of focal loss, which proved highly sensitive to hyperparameter selection. Experiment 6 (DenseNet-121 with focal loss) performed poorly, achieving an accuracy of only 75%, similar to the baseline model (Table 2). This aligns with findings from Li et al. (2023), emphasizing the necessity of careful tuning of the α and γ parameters alongside class weighting to avoid performance degradation. In this study, focal loss was implemented with γ=2.0 and class-specific α values of 0.20, 0.35, and 0.65 for normal, cataract, and glaucoma, respectively, alongside class weighting. While focal loss is widely used in medical imaging (Yeung et al., 2022; Mu et al., 2024), its effectiveness is highly dependent on hyperparameter tuning, as improper selection can lead to suboptimal performance.

Improvements & Future Directions

To further enhance performance and model stability, future research could incorporate larger datasets, as data availability remains a critical factor in deep learning effectiveness. Additionally, implementing a structured learning rate schedule with a proper warm-up phase, particularly when fine-tuning deeper architectures like DenseNet-121, would mitigate instability and improve convergence. Further hyperparameter optimization of focal loss, particularly α and γ values, could refine its impact on class imbalance. While data augmentation was effective in balancing the dataset, more advanced techniques such as synthetic image generation using generative adversarial networks (GANs) or domain adaptation could provide additional diversity in training samples. Lastly, exploring alternative architectures beyond EfficientNetB0 and DenseNet-121, such as transformer-based models, may offer further improvements in performance and generalization.

Ethical Considerations

The integration of DL in medical diagnostics raises ethical concerns, particularly regarding model reliability, bias, patient safety, and data privacy. Although EfficientNetB0 and DenseNet-121 achieved high classification accuracy, misclassification risks remain, potentially leading to delayed treatment or unnecessary interventions. Human oversight is essential to mitigate these risks. A key consideration in AI deployment is the trade-off between sensitivity and precision. Experiment 7 prioritized sensitivity, enhancing disease detection but increasing false positives for glaucoma. While higher sensitivity is ideal for screening, it may lead to unnecessary follow-ups and patient anxiety, highlighting the need for model selection based on clinical priorities.

Bias in AI models is a persistent challenge, often caused by imbalanced or non-representative datasets. Given this study's limited dataset size, there is a risk of reduced generalization to broader populations. Expanding datasets to include diverse demographic and clinical variations is crucial for fairness and accuracy.

Data privacy is another major concern. Retinal images contain sensitive patient information, necessitating strict adherence to data protection regulations such as the General Data Protection Regulation (European Parliament and Council, 2016) and Health Insurance Portability and Accountability Act (U.S. Congress, 1996) to ensure patient confidentiality.

Additionally, AI models require transparency and interpretability. Despite their high accuracy, deep learning models are often black boxes, making clinical decisions harder to explain. Developing explainable AI techniques would enhance trust among healthcare professionals and patients, supporting responsible AI adoption.

Lastly, human-in-the-loop approaches should be prioritized, where AI assists rather than replaces clinicians. AI should augment expert decision-making, improving diagnostic efficiency while ensuring accountability in medical practice.

Conclusion

This study evaluated EfficientNetB0 and DenseNet-121 for retinal disease classification, demonstrating that EfficientNetB0 achieved the highest accuracy (83%), particularly in detecting glaucoma cases. Notably, a model trained without fine-tuning the base network, relying solely on feature extraction from pre-trained weights and a custom classification head, still achieved 80% accuracy. This highlights the effectiveness of pre-trained representations in scenarios with limited data. Dataset balancing, class weighting, and CLAHE filtering further contributed to performance improvements.

Despite these results, limitations include the challenges of training on small datasets, the instability of fine-tuning without a structured learning rate schedule, and the sensitivity of focal loss to hyperparameters. Addressing these factors in future studies can improve model robustness.

The broader impact of this work emphasizes the potential of deep learning in retinal disease diagnosis, particularly in resource-constrained settings. Ethical concerns, including bias mitigation, data privacy, and model transparency, remain crucial for clinical adoption. Future research should focus on expanding datasets, refining learning strategies, and incorporating explainable AI techniques to enhance reliability and trust in automated medical diagnostics.

References

1. Alowais, S.A. & Huang, I.W., 2024. Revolutionizing Healthcare: The Role of Artificial Intelligence in Clinical Practice. *Angle Health Law Review*, vol. 94, pp. 95-132.
2. Alwakid, G., Gouda, W., Humayun, M., & Jhanjhi, N. Z., 2023. Deep learning-enhanced diabetic retinopathy image classification. *Digital Health*, vol. 9, 20552076231194942.
3. Alyoubi, W. L., Abulkhair, M. F., & Shalash, W. M., 2021. Diabetic retinopathy fundus image classification and lesions localization system using deep learning. *Sensors*, vol. 21(11), 3704.
4. Canayaz, M., 2022. Classification of diabetic retinopathy with feature selection over deep features using nature-inspired wrapper methods. *Applied Soft Computing*, vol. 128, 109462.
5. European Parliament and Council, 2016. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation)*. Available at: https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng. [13 February 2025].
6. Foster, P. J., Buhrmann, R., Quigley, H. A., & Johnson, G. J., 2002. The definition and classification of glaucoma in prevalence surveys. *British journal of ophthalmology*, vol. 86(2), pp. 238-242.
7. Google, n.d. *Colaboratory: Free Jupyter Notebook Environment*. Available at: https://colab.research.google.com [12 February 2025].
8. Hashemi, H., Pakzad, R., Yekta, A., Aghamirsalim, M., Pakbin, M., Ramin, S., & Khabazkhoob, M., 2020. Global and regional prevalence of age-related cataract: a comprehensive systematic review and meta-analysis. *Eye*, vol. 34(8), pp. 1357-1370.
9. Islam, M. T., Mashfu, S. T., Faisal, A., Siam, S. C., Naheen, I. T. & Khan, R., 2021. Deep learning-based glaucoma detection with cropped optic cup and disc and blood vessel segmentation. *IEEE Access*, vol. 10, pp. 2828-2841.
10. Junayed, M. S., Islam, M. B., Sadeghzadeh, A., & Rahman, S., 2021. CataractNet: An automated cataract detection system using deep learning for fundus images. *IEEE Access*, vol. 9, pp. 128799-128808.
11. Li, L., Verma, M., Wang, B., Nakashima, Y., Nagahara, H., & Kawasaki, R., 2023. Automated grading system of retinal arterio-venous crossing patterns: A deep learning approach replicating ophthalmologist's diagnostic process of arteriolosclerosis. *PLOS Digital Health*, vol. 2(1), e0000174.
12. Mu, Y., Nguyen, T., Hawickhorst, B., Wriggers, W., Sun, J. and He, J., 2024. The combined focal loss and dice loss function improves the segmentation of beta-sheets in medium-resolution cryo-electron-microscopy density maps. *Bioinformatics Advances*, vol. 4(1), p. vbae169.
13. Nandakumar, R., Saranya, P., Ponnusamy, V., Hazra, S. & Gupta, A., 2023. Detection of Diabetic Retinopathy from Retinal Images Using DenseNet Models. *Computer Systems Science & Engineering*, vol. 45(1).
14. Pascolini, D., & Mariotti, S. P., 2012. Global estimates of visual impairment: 2010. *British Journal of Ophthalmology*, vol. 96(5), pp. 614-618.
15. Reza, A. M. (2004). Realization of the contrast limited adaptive histogram equalization (CLAHE) for real-time image enhancement. *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 38, pp. 35-44.

16. Storgaard, L., Tran, T. L., Freiberg, J. C., Hauser, A. S., & Kolko, M., 2021. Glaucoma clinical research: Trends in treatment strategies and drug development. *Frontiers in Medicine*, vol. 8, 733080.

17. Syarifah, M. A., Bustamam, A. & Tampubolon, P. P., 2020. Cataract classification based on fundus image using an optimized convolution neural network with lookahead optimizer. In *AIP Conference Proceedings*, vol. 2296, No. 1. AIP Publishing.

18. Tan, M. & Le, Q., 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pp. 6105-6114. PMLR.

19. U.S. Congress, 1996. *Health Insurance Portability and Accountability Act of 1996, Public Law 104-191*. Available at: https://www.govinfo.gov/content/pkg/PLAW-104publ191/pdf/PLAW-104publ191.pdf. [13 February 2025].

20. Weinreb, R. N., Aung, T., & Medeiros, F. A., 2014. The pathophysiology and treatment of glaucoma: a review. *Jama*, vol. 311(18), pp. 1901-1911.

21. Wu, H., Ye, X., Jiang, Y., Tian, H., Yang, K., Cui, C., Shi, S., Liu, Y., Huang, S., Chen, J., Xu, J., & Dong, F., 2022. A Comparative Study of Multiple Deep Learning Models Based on Multi-Input Resolution for Breast Ultrasound Images. *Frontiers in oncology*, vol. 12, 869421.

22. Yang, J. J., Li, J., Shen, R., Zeng, Y., He, J., Bi, J., et al., 2016. Exploiting ensemble learning for automatic cataract detection and grading. *Computer methods and programs in biomedicine*, vol. 124, pp. 45-57.

23. Yeung, M., Sala, E., Schönlieb, C. B., & Rundo, L., 2022. Unified focal loss: Generalising dice and cross entropy-based losses to handle class imbalanced medical image segmentation. *Computerized Medical Imaging and Graphics*, vol. 95, 102026.

Appendix

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
# Set working directory
import os
os.chdir('./drive/MyDrive/Colab Notebooks')
```

Preprocessing Phase 1

Cropping black padding and resizing raw images to 224x224

```
import cv2

# Define input and output directories
input_dir = "./raw_data"
output_dir = "./resized_images"
os.makedirs(output_dir, exist_ok=True)

# Function to crop black borders
def crop_black_padding(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)
    x, y, w, h = cv2.boundingRect(thresh)
    return image[y:y+h, x:x+w]

# Function to crop to square and resize
def crop_to_square_and_resize(image, target_size=(224, 224)):
    h, w = image.shape[:2]
    # Determine the center crop dimensions
    if h > w:  # Portrait, crop height
        diff = (h - w) // 2
        cropped = image[diff:diff+w, :]  # Center crop height
    else:  # Landscape or square, crop width
        diff = (w - h) // 2
        cropped = image[:, diff:diff+h]  # Center crop width

    # Resize to target dimensions
    resized = cv2.resize(cropped, target_size, interpolation=cv2.INTER_AREA)
    return resized

for root, subdirs, files in os.walk(input_dir):
    # Relative path from the input directory
    rel_path = os.path.relpath(root, input_dir)
    # Corresponding output subfolder path
    output_subdir = os.path.join(output_dir, rel_path)
    # Create output subfolder if not exists
    os.makedirs(output_subdir, exist_ok=True)

    for filename in files:
        if filename.lower().endswith(".png"):
            input_path = os.path.join(root, filename)
            output_path = os.path.join(output_subdir, filename)

            print(f"Processing file: {input_path} -> {output_path}")
            try:
                # Read and preprocess the image
                image = cv2.imread(input_path)
                # Remove black borders
                cropped = crop_black_padding(image)
                # Crop to square and resize
                resized = crop_to_square_and_resize(cropped)

                # Save the processed image
                cv2.imwrite(output_path, resized)
                print(f"Saved processed image to: {output_path}")
            except Exception as e:
                print(f"Error processing file {filename}: {e}")
```

Splitting dataset in a 70:15:15 ratio

```
import splitfolders

# Input directory containing your imbalanced dataset
input_dir = './resized_images'

# Output directory where splits will be saved
output_dir = './dataset_split'

# Perform the split (75:15:15) with class proportion preservation
splitfolders.ratio(
    input_dir,
    output=output_dir,
    seed=42,  # For reproducibility
    ratio=(0.70, 0.15, 0.15),  # Train, validation, test ratio
    group_prefix=None  # No grouping, treat files individually
)
```

```
Copying files: 501 files [00:09, 55.06 files/s]
```

Verify Class Proportions

```
def count_files(directory):
    for subset in ['train', 'val', 'test']:
        subset_dir = os.path.join(directory, subset)
        print(f"\n{subset.upper()} SET:")
        for class_name in os.listdir(subset_dir):
            class_path = os.path.join(subset_dir, class_name)
            print(f"{class_name}: {len(os.listdir(class_path))} images")
```

```
# Verify the split
count_files(output_dir)
```

```
TRAIN SET:
1_normal: 210 images
2_cataract: 70 images
3_glaucoma: 70 images

VAL SET:
1_normal: 45 images
2_cataract: 15 images
3_glaucoma: 15 images

TEST SET:
1_normal: 45 images
2_cataract: 15 images
3_glaucoma: 16 images
```

Experiment 1

Imbalanced Dataset Testing with DenseNet-121

Training Strategy:

- **Initial Training:** The model is first trained for a fixed number of epochs with the DenseNet121 base frozen. This stage allows the newly added layers to learn task-specific representations without perturbing the pre-trained weights.
- **Fine-Tuning:** After initial training, the base model is unfrozen, and the entire network is fine-tuned with a lower learning rate. This step refines the learned features from the base model to better fit the target dataset.

Handling Class Imbalance: Class weights are computed using scikit-learn's compute_class_weight function. These weights are passed during training to help balance the contribution of each class in the loss function, which is particularly important when some classes are underrepresented.

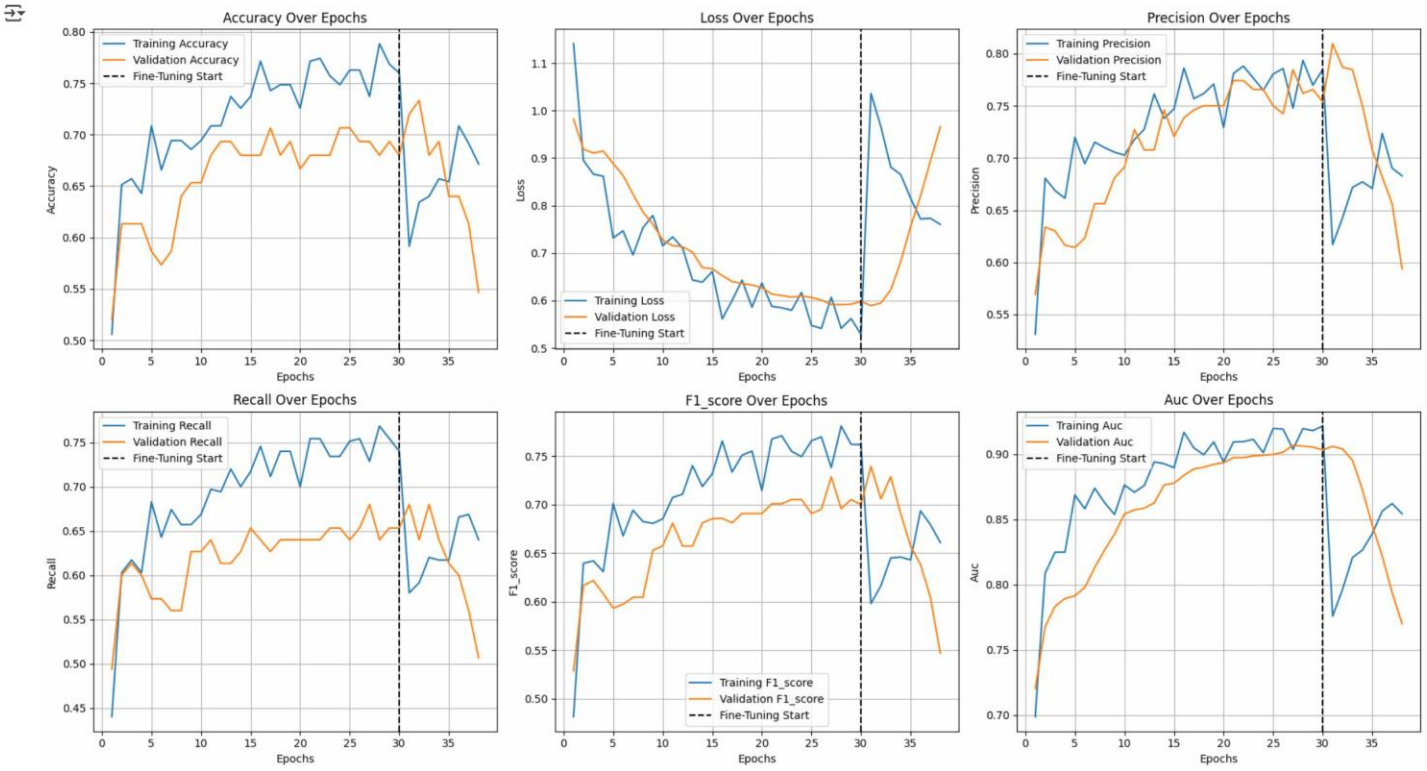
```
import tensorflow as tf
import numpy as np
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.callbacks import Callback, ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense, BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
# Keras image preprocessing (for data augmentation)
from tensorflow.keras.preprocessing.image import ImageDataGenerator


# --- Enable GPU Memory Growth ---
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError as e:
        print(e)

# --- Custom F1 Score Metric ---
class F1Score(tf.keras.metrics.Metric):
    def __init__(self, name="f1_score", **kwargs):
        super(F1Score, self).__init__(name=name, **kwargs)
        self.precision = tf.keras.metrics.Precision()
        self.recall = tf.keras.metrics.Recall()

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.precision.update_state(y_true, y_pred, sample_weight)
        self.recall.update_state(y_true, y_pred, sample_weight)

    def result(self):
        precision = self.precision.result()
        recall = self.recall.result()
        return 2 * ((precision * recall) / (precision + recall + tf.keras.backend.epsilon()))

    def reset_states(self):
        self.precision.reset_states()
        self.recall.reset_states()


# ---------------------- Paths and Parameters ----------------------
# Define the directories for training, validation, and testing datasets.
train_dir = './dataset_split/train'
val_dir = './dataset_split/val'
test_dir = './dataset_split/test'

# Define key parameters
IMG_SIZE = (224, 224)      # Target size for input images
BATCH_SIZE = 32            # Batch size for training and evaluation
EPOCHS = 30                # Number of epochs for initial training
FINE_TUNE_EPOCHS = 20      # Additional epochs for fine-tuning

# ---------------------- Data Generators ----------------------
# Create an ImageDataGenerator for training with data augmentation and rescaling.
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,         # Normalize pixel values to [0,1]
    rotation_range=180,        # Randomly rotate images up to 180 degrees
    width_shift_range=0.1,     # Shift images horizontally by up to 10%
    height_shift_range=0.1,    # Shift images vertically by up to 10%
    zoom_range=0.2,            # Randomly zoom images by up to 20%
    horizontal_flip=True,      # Randomly flip images horizontally
    fill_mode='nearest'        # Fill in missing pixels after a transformation
)

# For validation and test data, we only need rescaling.
val_test_datagen = ImageDataGenerator(rescale=1.0 / 255)

# Generate batches of augmented data for training.
```

```
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# Generate batches for validation data.
validation_generator = val_test_datagen.flow_from_directory(
    val_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# Generate batches for test data (shuffling is disabled for evaluation purposes).
test_generator = val_test_datagen.flow_from_directory(
    test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False
)


# ---------------------- Build the Model ----------------------
# Load the DenseNet121 base model pre-trained on ImageNet.
base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False  # Freeze the base model initially

# Add custom classification layers on top of the base model.
x = base_model.output
x = GlobalAveragePooling2D()(x)  # Reduce spatial dimensions to a single vector per image
x = BatchNormalization()(x)      # Normalizes activations to stabilize training
x = Dropout(0.3)(x)              # Dropout layer for regularization
x = Dense(512, activation='relu')(x)  # Fully connected layer with 512 units
x = Dropout(0.5)(x)              # Additional dropout for further regularization
output_layer = Dense(train_generator.num_classes, activation='softmax')(x)  # Output layer with softmax for multi-class classification

# Define the complete model.
model = Model(inputs=base_model.input, outputs=output_layer)

# ---------------------- Compile the Model ----------------------
# Compile the model using the Adam optimizer and categorical crossentropy loss.
model.compile(
    optimizer=Adam(learning_rate=0.00016084),
    loss='categorical_crossentropy',
    metrics=[
        'accuracy',
        tf.keras.metrics.Precision(name='precision'),
        tf.keras.metrics.Recall(name='recall'),
        tf.keras.metrics.AUC(name='auc'),
        F1Score(name='f1_score')
    ]
)

# ---------------------- Callbacks ----------------------
# Callback to reduce the learning rate when the validation loss plateaus.
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=4,
    min_lr=1e-6,
    verbose=1
)

# Early stopping callback to halt training if validation loss does not improve.
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=7,
    restore_best_weights=True
)

# ---------------------- Training (Initial Phase) ----------------------
# Calculate steps per epoch using np.ceil so that all batches are processed.
steps_per_epoch = int(np.ceil(train_generator.samples / BATCH_SIZE))
validation_steps = int(np.ceil(validation_generator.samples / BATCH_SIZE))

history = model.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    validation_data=validation_generator,
    validation_steps=validation_steps,
    epochs=EPOCHS,
    callbacks=[reduce_lr, early_stopping],
)

# ---------------------- Fine-Tuning ----------------------
# Unfreeze the base model to allow fine-tuning of the entire network.
base_model.trainable = True

# Re-compile the model with a lower learning rate suitable for fine-tuning.
model.compile(
    optimizer=Adam(learning_rate=1e-5),
    loss='categorical_crossentropy',
    metrics=[
        'accuracy',
        tf.keras.metrics.Precision(name='precision'),
        tf.keras.metrics.Recall(name='recall'),
        tf.keras.metrics.AUC(name='auc'),
        F1Score(name='f1_score')
    ]
)

history_fine = model.fit(
    train_generator,
    steps_per_epoch=steps_per_epoch,
    validation_data=validation_generator,
    validation_steps=validation_steps,
    epochs=EPOCHS + FINE_TUNE_EPOCHS,
```

```
    initial_epoch=history.epoch[-1],  # Start fine-tuning from the last epoch of initial training
    callbacks=[reduce_lr, early_stopping],
)

# ---------------------- Evaluation ----------------------
# Evaluate the final model performance on the test set.
test_loss, test_accuracy, test_precision, test_recall, test_auc, test_f1 = model.evaluate(test_generator)

# Print the test results to the console.
print(f"Test Accuracy: {test_accuracy:.2f}")
print(f"Test Precision: {test_precision:.2f}")
print(f"Test Recall: {test_recall:.2f}")
print(f"Test AUC: {test_auc:.2f}")
print(f"Test F1 Score: {test_f1:.2f}")
```

Training and Validation Metrics Over Epochs

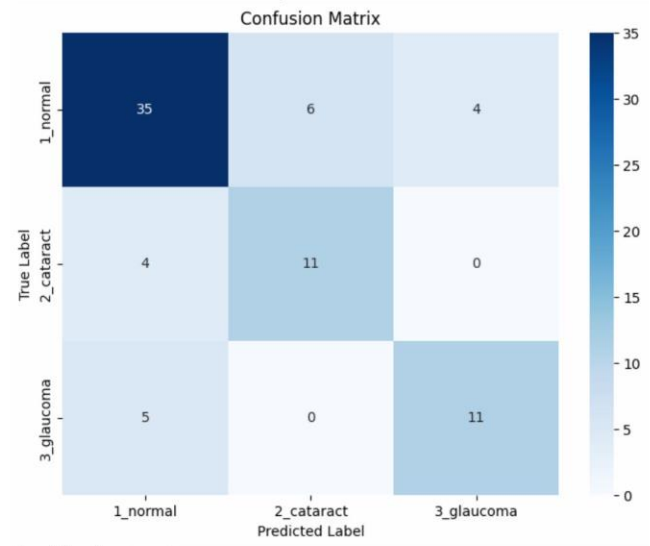
```
import matplotlib.pyplot as plt

def plot_training_metrics(history, history_fine):
    # List of metrics to plot
    metrics = ['accuracy', 'loss', 'precision', 'recall', 'f1_score', 'auc']

    # Find total epochs for separation
    initial_epochs = len(history.history['loss'])  # Number of initial training epochs

    # Create a 2x3 grid of subplots
    fig, axes = plt.subplots(2, 3, figsize=(18, 10))
    axes = axes.flatten()  # Flatten to iterate easily over the axes

    # Combine history dictionaries for seamless plotting
    full_history = {metric: history.history.get(metric, []) + history_fine.history.get(metric, []) for metric in metrics}
    full_history.update({f'val_{metric}': history.history.get(f'val_{metric}', []) + history_fine.history.get(f'val_{metric}', []) for metric in metrics})

    # Plot each metric for both training and validation data
    for i, metric in enumerate(metrics):
        epochs = range(1, len(full_history[metric]) + 1)

        axes[i].plot(epochs, full_history[metric], label=f'Training {metric.capitalize()}')
        axes[i].plot(epochs, full_history[f'val_{metric}'], label=f'Validation {metric.capitalize()}')

        # Add dashed line to separate initial training and fine-tuning phases
        axes[i].axvline(x=initial_epochs, linestyle='--', color='black', label='Fine-Tuning Start')

        axes[i].set_title(f'{metric.capitalize()} Over Epochs')
        axes[i].set_xlabel('Epochs')
        axes[i].set_ylabel(metric.capitalize())
        axes[i].legend()
        axes[i].grid(True)

    plt.tight_layout()  # Adjust layout to prevent overlap
    plt.show()

# Call the function to plot training and fine-tuning metrics
plot_training_metrics(history, history_fine)
```



Confusion Matrix & Classification Report

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

# Reset the generator to ensure predictions start from the beginning.
test_generator.reset()

# Generate predictions for the test set.
# Compute the required steps based on the batch size.
steps = int(np.ceil(test_generator.samples / test_generator.batch_size))
predictions = model.predict(test_generator, steps=steps)

# Convert predicted probabilities to class indices (highest probability class).
predicted_classes = np.argmax(predictions, axis=1)

# Retrieve the true labels from the generator.
true_classes = test_generator.classes

# Retrieve the class labels (names) based on the dataset directory structure.
class_labels = list(test_generator.class_indices.keys())

# Compute the confusion matrix.
cm = confusion_matrix(true_classes, predicted_classes)

# Plot the confusion matrix using seaborn.
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# Generate and print the classification report.
report = classification_report(true_classes, predicted_classes, target_names=class_labels, digits=2)
print("Classification Report:\n", report)
```

```
3/3 ━━━━━━━━━━━━━━━━━━━━ 14s 4s/step
```

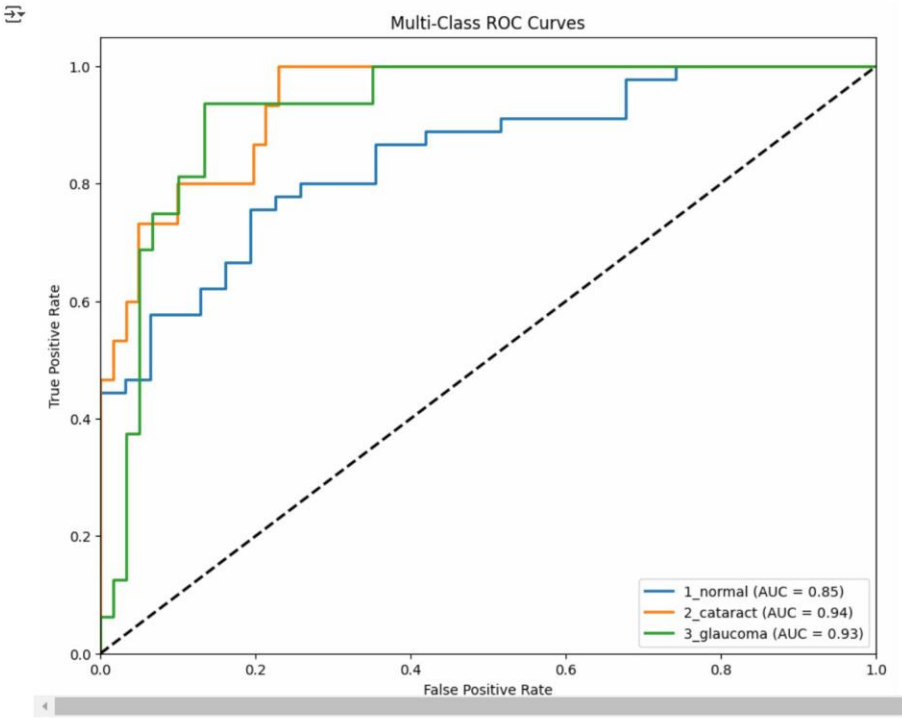



```
Classification Report:
               precision    recall  f1-score   support

    1_normal       0.80      0.78      0.79        45
  2_cataract       0.65      0.73      0.69        15
  3_glaucoma       0.73      0.69      0.71        16

    accuracy                           0.75        76
   macro avg       0.73      0.73      0.73        76
weighted avg       0.75      0.75      0.75        76
```

ROC curve

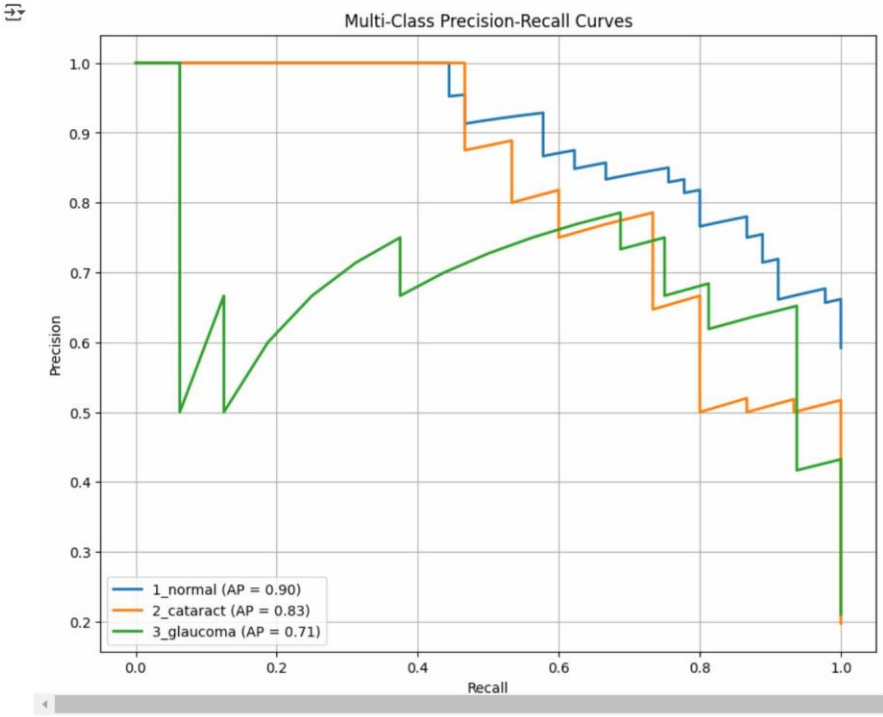
```
from sklearn.metrics import roc_curve, auc
from tensorflow.keras.utils import to_categorical

# Convert true labels to one-hot encoding.
# 'class_labels' is the list of class names retrieved earlier.
num_classes = len(class_labels)
y_true = to_categorical(test_generator.classes, num_classes=num_classes)

# 'predictions' were generated earlier using model.predict(...)
y_pred = predictions  # Shape: (n_samples, num_classes)

# Compute ROC curve and AUC for each class.
fpr = {}
tpr = {}
roc_auc = {}

for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(y_true[:, i], y_pred[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC curves for each class.
plt.figure(figsize=(10, 8))
for i, label in enumerate(class_labels):
    plt.plot(fpr[i], tpr[i], lw=2, label=f'{label} (AUC = {roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)  # Diagonal line for random classifier
plt.xlim([0.0, 1.0])
```

```
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Multi-Class ROC Curves')
plt.legend(loc="lower right")
plt.show()
```



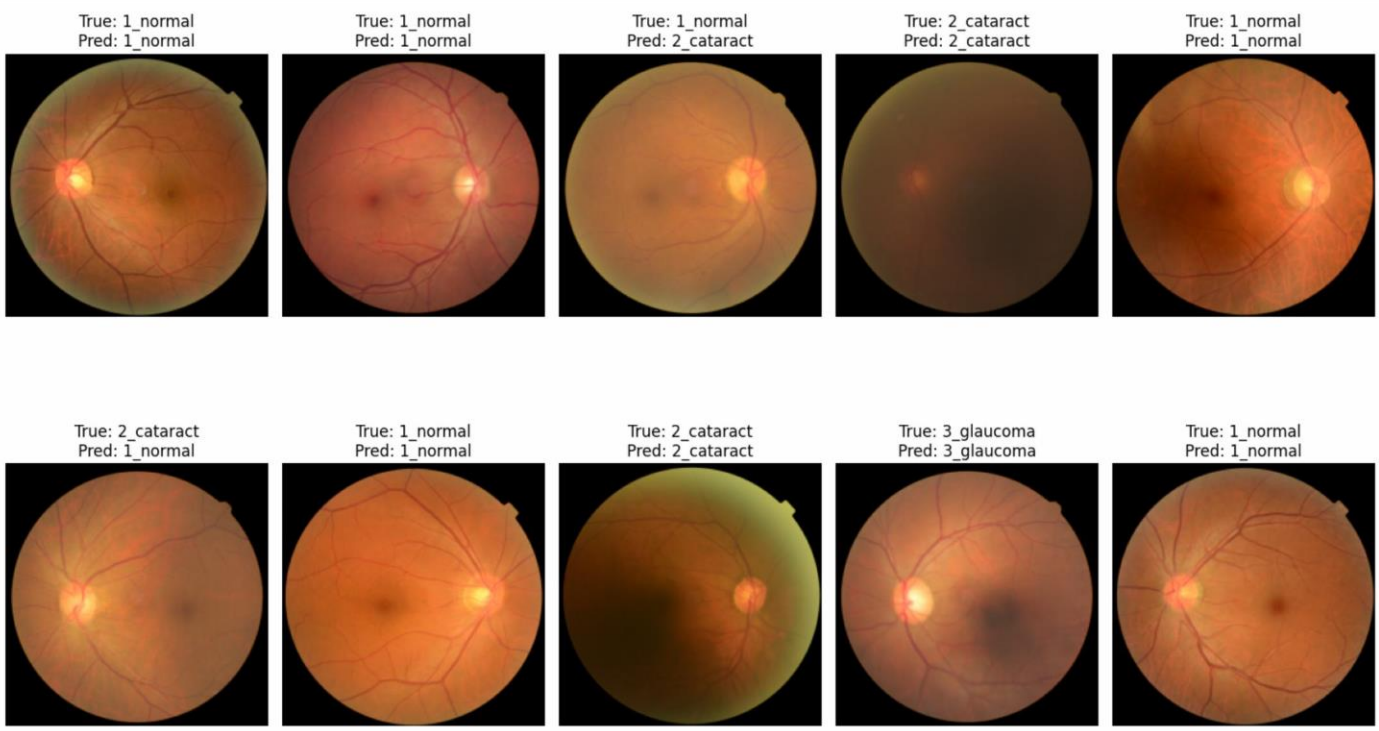
Precision-Recall curve

```
from sklearn.metrics import precision_recall_curve, average_precision_score

# Compute Precision-Recall curves and average precision (AP) for each class.
precision_dict = {}
recall_dict = {}
average_precision = {}

for i in range(num_classes):
    precision_dict[i], recall_dict[i], _ = precision_recall_curve(y_true[:, i], y_pred[:, i])
    average_precision[i] = average_precision_score(y_true[:, i], y_pred[:, i])

# Plot the Precision-Recall curves for each class.
plt.figure(figsize=(10, 8))
for i, label in enumerate(class_labels):
    plt.plot(recall_dict[i], precision_dict[i], lw=2, label=f'{label} (AP = {average_precision[i]:.2f})')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Multi-Class Precision-Recall Curves')
plt.legend(loc="lower left")
plt.grid(True)
plt.show()
```



Test Predictions Visualization

```
import random
import numpy as np
import matplotlib.pyplot as plt

# Ensure predictions is computed earlier using model.predict(...).
# Convert predictions (probabilities) to integer class indices.
predicted_labels = np.argmax(predictions, axis=1)

# Get image file paths and true labels from the test generator.
test_images, test_labels = test_generator.filepaths, test_generator.classes

# Randomly sample 10 test images.
sample_indices = random.sample(range(len(test_images)), 10)

plt.figure(figsize=(15, 10))
for i, idx in enumerate(sample_indices):
    img_path = test_images[idx]
    true_label = class_labels[test_labels[idx]]
    pred_label = class_labels[predicted_labels[idx]]

    # Load and display the image.
    img = plt.imread(img_path)
    plt.subplot(2, 5, i + 1)
    plt.imshow(img)
    plt.axis('off')
    plt.title(f'True: {true_label}\nPred: {pred_label}')

plt.tight_layout()
plt.show()
```



Preprocessing Phase 2

Splitting dataset to 60:20:20 ratio

```
def split_dataset_proportionally(input_dir, output_dir, train_ratio=0.6, val_ratio=0.2, test_ratio=0.2):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for class_name in os.listdir(input_dir):
        class_path = os.path.join(input_dir, class_name)
        files = os.listdir(class_path)
        random.shuffle(files)  # Shuffle to ensure random distribution

        # Calculate split sizes
        train_split = int(len(files) * train_ratio)
        val_split = int(len(files) * (train_ratio + val_ratio))

        train_files = files[:train_split]
        val_files = files[train_split:val_split]
        test_files = files[val_split:]

        # Create output directories for each split
        os.makedirs(os.path.join(output_dir, 'train', class_name), exist_ok=True)
        os.makedirs(os.path.join(output_dir, 'val', class_name), exist_ok=True)
        os.makedirs(os.path.join(output_dir, 'test', class_name), exist_ok=True)

        # Move files to respective directories
        for f in train_files:
            shutil.copy(os.path.join(class_path, f), os.path.join(output_dir, 'train', class_name, f))
        for f in val_files:
```

```
            shutil.copy(os.path.join(class_path, f), os.path.join(output_dir, 'val', class_name, f))
        for f in test_files:
            shutil.copy(os.path.join(class_path, f), os.path.join(output_dir, 'test', class_name, f))

        # Print status
        print(f"Class '{class_name}': {len(train_files)} train, {len(val_files)} val, {len(test_files)} test")

# Paths
input_dir = './raw_data'  # Directory containing resized images
output_dir = './raw_data_split'  # Output directory for split datasets

# Split the dataset
split_dataset_proportionally(input_dir, output_dir)
```

```
Class '1_normal': 180 train, 60 val, 60 test
Class '2_cataract': 60 train, 20 val, 20 test
Class '3_glaucoma': 60 train, 20 val, 21 test
```

Balancing dataset to 300 images per class by augmentation

```
# Paths
input_dir = './raw_data_split/train'  # Original dataset directory
output_dir = './raw_data_split/train_augmented'  # Directory to save augmented images
target_size = 300  # Desired number of images per class

# ImageDataGenerator for augmentation
datagen = ImageDataGenerator(
    rotation_range=180,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.3,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Create output directory
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Augmentation strategy
augmentation_config = {
    '1_normal': 2,  # Generate 2 augmented images per input image
    '2_cataract': 4,  # Generate 4 augmented images per input image
    '3_glaucoma': 4   # Generate 4 augmented images per input image
}

# Loop through each class directory
for class_name in os.listdir(input_dir):
    class_input_path = os.path.join(input_dir, class_name)
    class_output_path = os.path.join(output_dir, class_name)

    # Ensure the output class directory exists
    if not os.path.exists(class_output_path):
        os.makedirs(class_output_path)

    # Copy original images to the output directory
    images = os.listdir(class_input_path)
    for img_name in images:
        input_img_path = os.path.join(class_input_path, img_name)
        output_img_path = os.path.join(class_output_path, img_name)
        shutil.copy(input_img_path, output_img_path)

    # Count images now in the output directory (includes original images)
    image_count = len(os.listdir(class_output_path))

    # Determine augmentation multiplier based on class
    augmentation_multiplier = augmentation_config.get(class_name, 1)  # Default to 1 if class not in config

    # Augment images if the count is below the target size
    if image_count < target_size:
        print(f"Augmenting class: {class_name} ({image_count}/{target_size})")
        for img_name in images:
            img_path = os.path.join(class_input_path, img_name)
            img = load_img(img_path)  # Load image
            img_array = img_to_array(img)  # Convert to array
            img_array = img_array.reshape((1,) + img_array.shape)  # Reshape for augmentation

            # Generate augmented images
            save_prefix = os.path.splitext(img_name)[0]
            generated_images = 0
            for batch in datagen.flow(img_array, batch_size=1, save_to_dir=class_output_path,
                                      save_prefix=save_prefix, save_format='png'):
                generated_images += 1
                if generated_images >= augmentation_multiplier:
                    break

            # Update the image count
            image_count += augmentation_multiplier
            if image_count >= target_size:
                break

        print(f"Class {class_name} balanced to {target_size} images.")
    else:
        print(f"Class {class_name} already has {image_count} images. No augmentation needed.")

        # Validate the number of files in the output directory
    final_image_count = len(os.listdir(class_output_path))
    print(f"Final file count in '{class_name}': {final_image_count} files.")
```

```
Augmenting class: 1_normal (180/300)
Class 1_normal balanced to 300 images.
Final file count in '1_normal': 300 files.
Augmenting class: 2_cataract (60/300)
Class 2_cataract balanced to 300 images.
Final file count in '2_cataract': 300 files.
Augmenting class: 3_glaucoma (60/300)
Class 3_glaucoma balanced to 300 images.
```

```
Final file count in '3_glaucoma': 300 files.
```

Experiment 2

Balanced Dataset Testing with DenseNet-121 (no CLAHE filtering)

Fine-tuning to the whole base model

Training Strategy

The model leverages DenseNet121, which has been pre-trained on ImageNet, as a feature extractor. Initially, the base model's weights are frozen. This allows the newly added custom layers (global average pooling, batch normalization, dropout, and dense layers) to learn task-specific features without disrupting the pre-trained representations of the base model (initially trained on ImageNet dataset).

- **Initial Training Phase:** In the initial training phase, only the custom classification layers are trained for 30 epochs. During this phase, class weights are applied (see below) to help balance the contributions of each class. The training uses callbacks such as ReduceLROnPlateau (to lower the learning rate when validation loss plateaus) and EarlyStopping (to halt training if the validation loss does not improve).
- **Fine-Tuning Phase:** After the initial training, the DenseNet121 base model is unfrozen, and the entire network is recompiled with a lower learning rate (1e-5). Fine-tuning is then performed for an additional 20 epochs. This phase allows the pre-trained weights to be slightly adjusted (fine-tuned) to better capture the features specific to your dataset, often resulting in improved performance.

Class Weights: Class weights are used to address the issue of missclassification between the normal and glaucoma classes in the dataset. When some classes are underrepresented or harder to classify, the model might tend to favor the majority class. By assigning higher weights to harder to classify classes, the loss function penalizes their misclassification more heavily. This encourages the model to pay more attention to learning the characteristics of glaucoma and cataract classes.

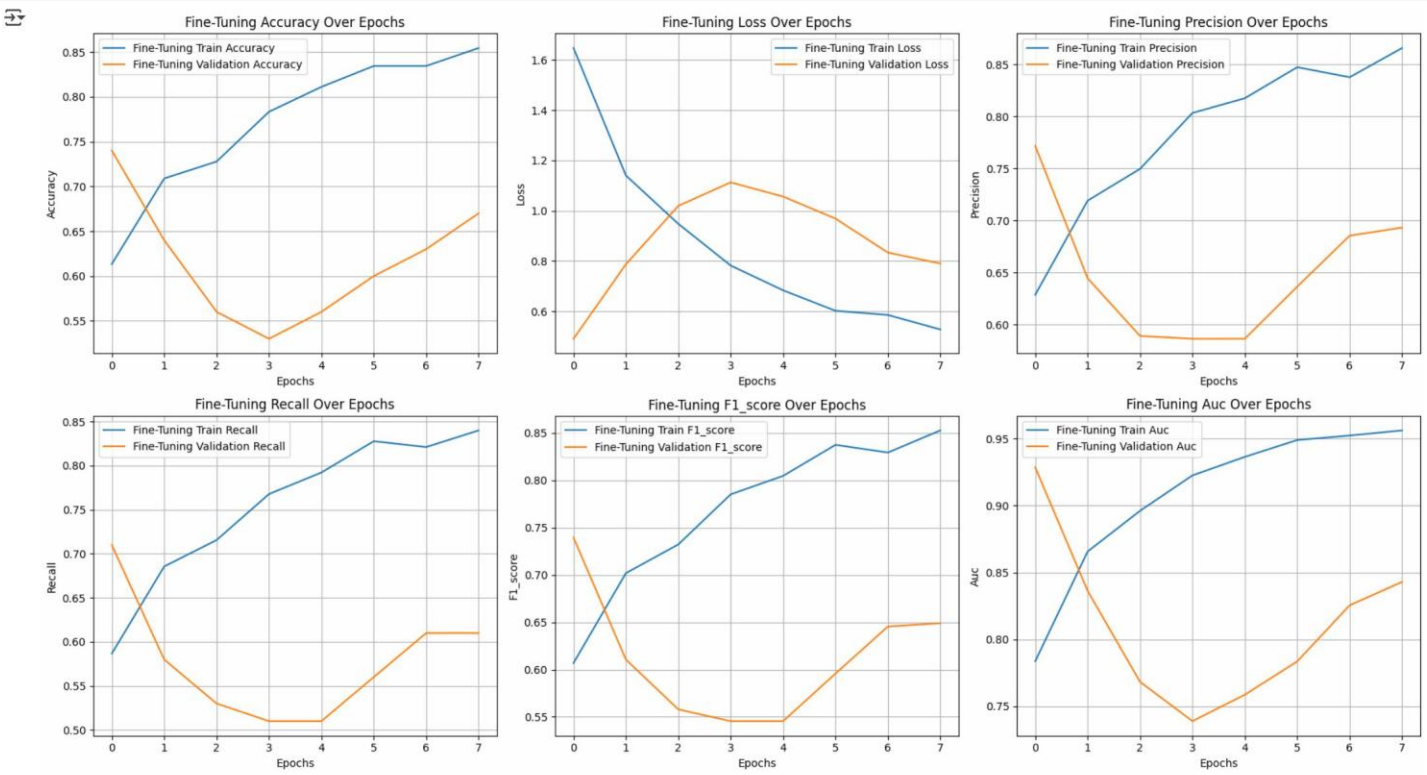
```
import tensorflow as tf
import numpy as np
from sklearn.utils.class_weight import compute_class_weight
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense, BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.metrics import Precision, Recall, AUC
from tensorflow.keras.callbacks import Callback, ReduceLROnPlateau, EarlyStopping

# --- Enable GPU Memory Growth ---
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError as e:
        print(e)

# --- Custom F1 Score Metric ---
class F1Score(tf.keras.metrics.Metric):
    def __init__(self, name="f1_score", **kwargs):
        super(F1Score, self).__init__(name=name, **kwargs)
        self.precision = tf.keras.metrics.Precision()
        self.recall = tf.keras.metrics.Recall()

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.precision.update_state(y_true, y_pred, sample_weight)
        self.recall.update_state(y_true, y_pred, sample_weight)

    def result(self):
        precision = self.precision.result()
        recall = self.recall.result()
        return 2 * ((precision * recall) / (precision + recall + tf.keras.backend.epsilon()))

    def reset_states(self):
        self.precision.reset_states()
        self.recall.reset_states()


# --- Paths ---
train_dir = './dataset_split2/train_augmented'
val_dir = './dataset_split2/val'
test_dir = './dataset_split2/test'
IMG_SIZE = (224, 224)

# --- Hyperparameters ---
BATCH_SIZE = 32
LEARNING_RATE = 0.00016084
DROPOUT_RATE_1 = 0.3
DROPOUT_RATE_2 = 0.5
DENSE_UNITS = 512
EPOCHS = 30
FINE_TUNE_EPOCHS = 20

# --- Class Weights ---
class_weights = {
    0: 1.0,
    1: 2.0,
    2: 2.0
}

# --- Build Model ---
def build_model():
    base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    base_model.trainable = False

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = BatchNormalization()(x)
    x = Dropout(DROPOUT_RATE_1)(x)
    x = Dense(DENSE_UNITS, activation='relu')(x)
    x = Dropout(DROPOUT_RATE_2)(x)
    output_layer = Dense(3, activation='softmax')(x)
```

```
    model = Model(inputs=base_model.input, outputs=output_layer)

    model.compile(
        optimizer=Adam(learning_rate=LEARNING_RATE),
        loss='categorical_crossentropy',
        metrics=[
            'accuracy',
            Precision(name='precision'),
            Recall(name='recall'),
            AUC(name='auc'),
            F1Score(name='f1_score')
        ]
    )
    model.base_model = base_model
    return model

# --- Data Generators ---
train_datagen = ImageDataGenerator(rescale=1.0 / 255)
val_test_datagen = ImageDataGenerator(rescale=1.0 / 255)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

validation_generator = val_test_datagen.flow_from_directory(
    val_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# --- Initialize the Model ---
model = build_model()

# --- Callbacks ---
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=7,
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=4,
    min_lr=1e-6,
    verbose=1
)

# --- Train the Model (Initial Phase) ---
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS,
    class_weight=class_weights,
    callbacks=[reduce_lr, early_stopping]
)

# --- Fine-Tuning Phase ---
model.base_model.trainable = True

model.compile(
    optimizer=Adam(learning_rate=1e-5),
    loss='categorical_crossentropy',
    metrics=[
        'accuracy',
        Precision(name='precision'),
        Recall(name='recall'),
        AUC(name='auc'),
        F1Score(name='f1_score')
    ]
)

history_fine = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS + FINE_TUNE_EPOCHS,
    initial_epoch=history.epoch[-1],
    class_weight=class_weights,
    callbacks=[reduce_lr, early_stopping]
)

# --- Evaluate on Test Data ---
test_generator = val_test_datagen.flow_from_directory(
    test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False
)

test_loss, test_accuracy, test_precision, test_recall, test_auc, test_f1 = model.evaluate(test_generator)
print(f"Test Results: Accuracy={test_accuracy:.2f}, Precision={test_precision:.2f}, Recall={test_recall:.2f}, Loss={test_loss:.2f}, AUC={test_auc:.2f}, F1 Score={test_f1:.2f}")
```

Training and Validation Metrics Over Epochs

```
import matplotlib.pyplot as plt

def plot_finetune_metrics(history_fine):
    # List of metrics to plot
    metrics = ['accuracy', 'loss', 'precision', 'recall', 'f1_score', 'auc']
```

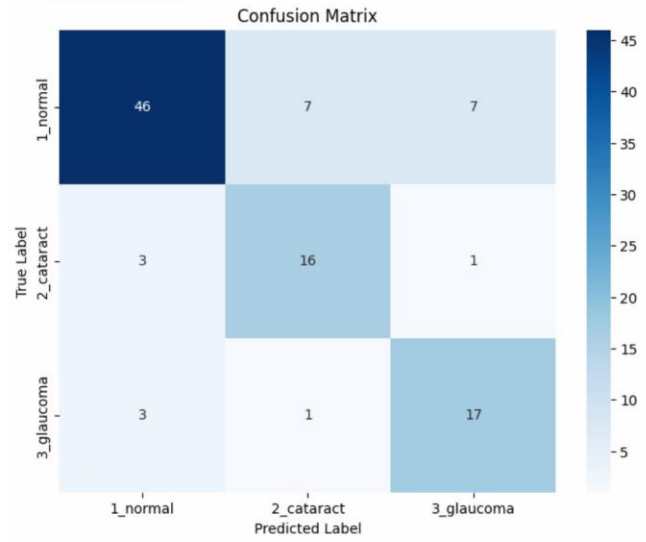
```
    # Create a 2x3 grid of subplots
    fig, axes = plt.subplots(2, 3, figsize=(18, 10))
    axes = axes.flatten()  # Flatten to iterate easily over the axes

    # Plot each metric for both training and validation data
    for i, metric in enumerate(metrics):
        axes[i].plot(history_fine.history[metric], label=f'Fine-Tuning Train {metric.capitalize()}')
        axes[i].plot(history_fine.history[f'val_{metric}'], label=f'Fine-Tuning Validation {metric.capitalize()}')
        axes[i].set_title(f'Fine-Tuning {metric.capitalize()} Over Epochs')
        axes[i].set_xlabel('Epochs')
        axes[i].set_ylabel(metric.capitalize())
        axes[i].legend()
        axes[i].grid(True)

    plt.tight_layout()  # Adjust layout to prevent overlap
    plt.show()

# Call the function to plot fine tuning metrics
plot_finetune_metrics(history_fine)
```



Confusion Matrix & Classification Report

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

# Reset the generator to ensure predictions start from the beginning.
test_generator.reset()

# Generate predictions for the test set.
# The number of steps is calculated using np.ceil to cover all test samples.
predictions = model.predict(test_generator, steps=int(np.ceil(test_generator.samples / test_generator.batch_size)))
# For multi-class classification, choose the index with the highest probability.
predicted_classes = np.argmax(predictions, axis=1)

# Get the true labels from the generator.
true_classes = test_generator.classes

# Retrieve the class labels (names) based on the directory structure.
class_labels = list(test_generator.class_indices.keys())

# Compute the confusion matrix.
cm = confusion_matrix(true_classes, predicted_classes)

# Plot the confusion matrix using seaborn.
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# Generate and print the classification report.
report = classification_report(true_classes, predicted_classes, target_names=class_labels)
print("Classification Report:\n", report)
```

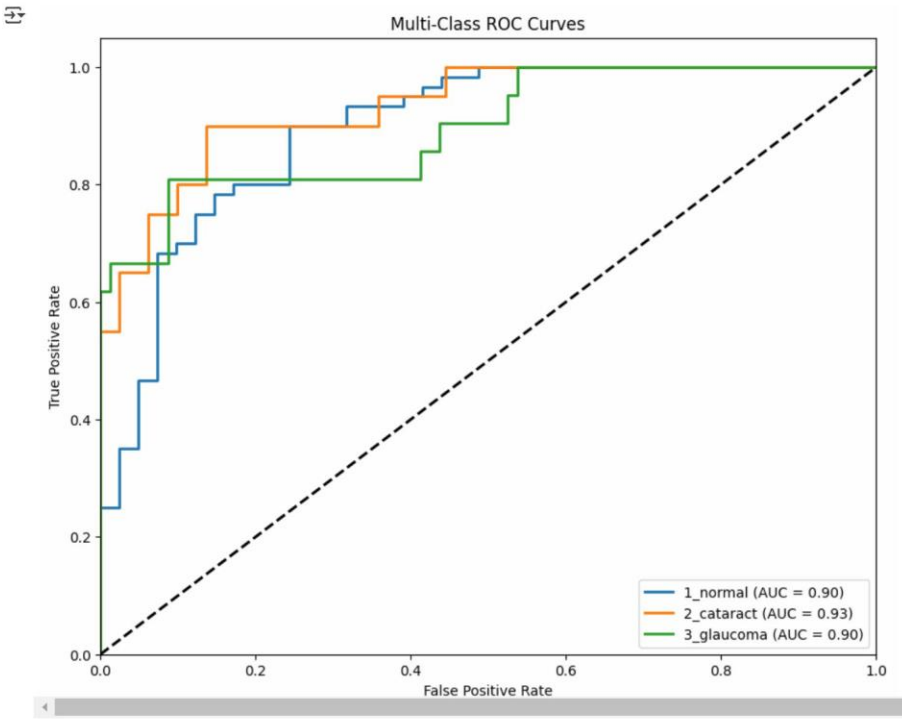
```
4/4 ━━━━━━━━━━━━━━━━━━━━ 26s 5s/step
```




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

    1_normal       0.88      0.77      0.82        60
  2_cataract       0.67      0.80      0.73        20
  3_glaucoma       0.68      0.81      0.74        21

    accuracy                           0.78       101
   macro avg       0.74      0.79      0.76       101
weighted avg       0.80      0.78      0.79       101
```

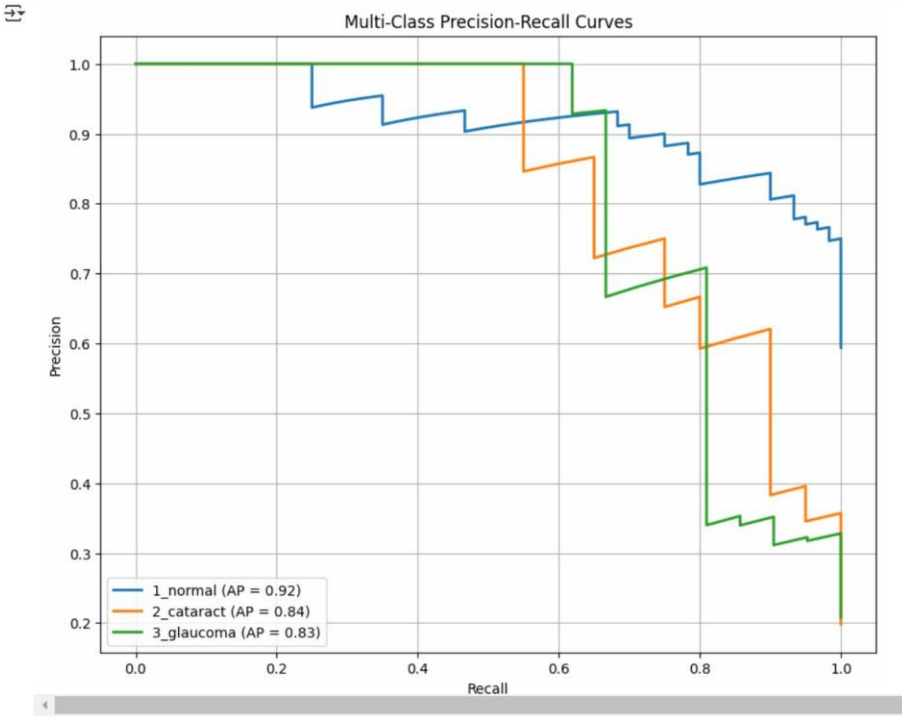
ROC curve

```
from sklearn.metrics import roc_curve, auc
from tensorflow.keras.utils import to_categorical

# Convert true labels to one-hot encoding.
# 'class_labels' is the list of class names retrieved earlier.
num_classes = len(class_labels)
y_true = to_categorical(test_generator.classes, num_classes=num_classes)

# 'predictions' were generated earlier using model.predict(...)
y_pred = predictions  # Shape: (n_samples, num_classes)

# Compute ROC curve and AUC for each class.
fpr = {}
tpr = {}
roc_auc = {}

for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(y_true[:, i], y_pred[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot ROC curves for each class.
plt.figure(figsize=(10, 8))
for i, label in enumerate(class_labels):
    plt.plot(fpr[i], tpr[i], lw=2, label=f'{label} (AUC = {roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], 'k--', lw=2)  # Diagonal line for random classifier
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Multi-Class ROC Curves')
plt.legend(loc="lower right")
plt.show()
```



Precision-Recall curve

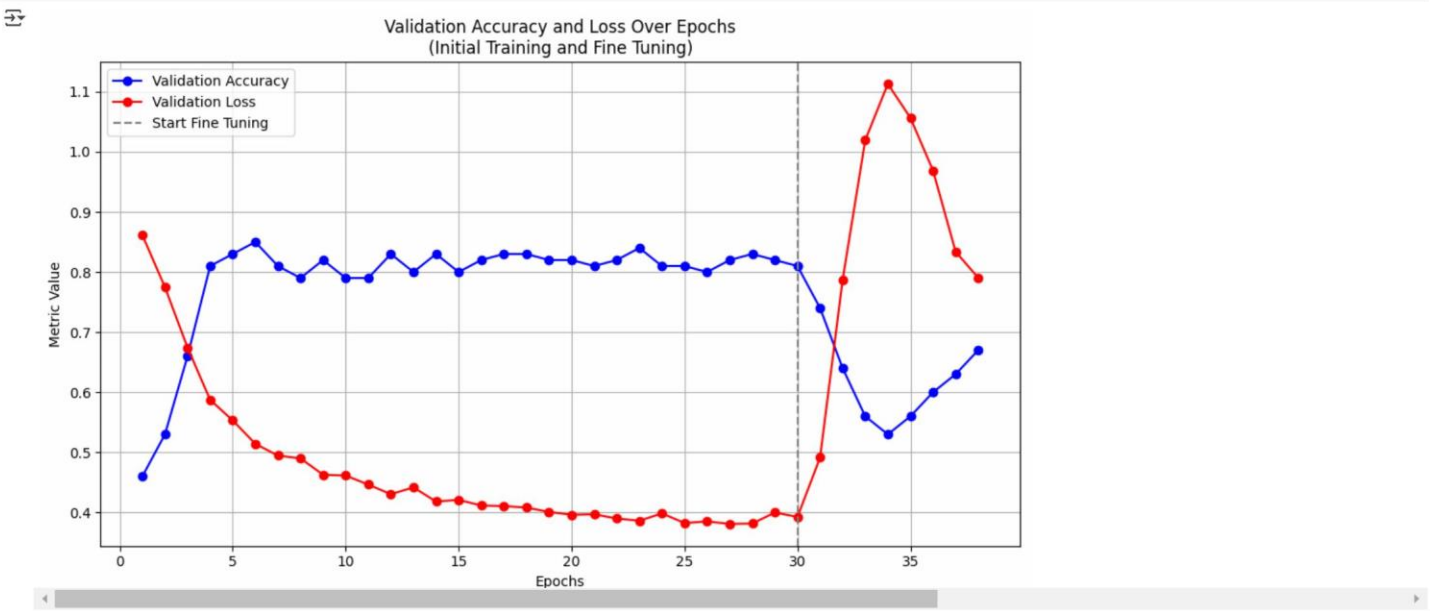
```
from sklearn.metrics import precision_recall_curve, average_precision_score

# Compute Precision-Recall curves and average precision (AP) for each class.
precision_dict = {}
recall_dict = {}
average_precision = {}

for i in range(num_classes):
    precision_dict[i], recall_dict[i], _ = precision_recall_curve(y_true[:, i], y_pred[:, i])
    average_precision[i] = average_precision_score(y_true[:, i], y_pred[:, i])

# Plot the Precision-Recall curves for each class.
plt.figure(figsize=(10, 8))
for i, label in enumerate(class_labels):
    plt.plot(recall_dict[i], precision_dict[i], lw=2, label=f'{label} (AP = {average_precision[i]:.2f})')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Multi-Class Precision-Recall Curves')
plt.legend(loc="lower left")
plt.grid(True)
plt.show()
```



Validation Accuracy and Loss over Epoch

```
import matplotlib.pyplot as plt

def plot_validation_metrics(initial_history, fine_history):
    """
```

```
    Plots combined validation accuracy and loss from the initial training phase
    and the fine-tuning phase on a single graph.

    Parameters:
      initial_history: History object from initial training.
      fine_history: History object from fine-tuning.
    """
    # Determine the number of epochs in each phase.
    initial_epochs = len(initial_history.history['val_accuracy'])
    fine_epochs = len(fine_history.history['val_accuracy'])
    total_epochs = initial_epochs + fine_epochs

    # Create a combined x-axis from 1 to total_epochs.
    epochs = range(1, total_epochs + 1)

    # Combine the validation accuracy and loss values.
    val_accuracy_total = initial_history.history['val_accuracy'] + fine_history.history['val_accuracy']
    val_loss_total = initial_history.history['val_loss'] + fine_history.history['val_loss']

    # Create the plot.
    plt.figure(figsize=(10, 6))

    # Plot validation accuracy and loss.
    plt.plot(epochs, val_accuracy_total, label='Validation Accuracy', color='blue', marker='o')
    plt.plot(epochs, val_loss_total, label='Validation Loss', color='red', marker='o')

    # Mark the transition point between initial training and fine-tuning.
    plt.axvline(x=initial_epochs, color='gray', linestyle='--', label='Start Fine Tuning')

    # Add labels and title.
    plt.xlabel('Epochs')
    plt.ylabel('Metric Value')
    plt.title('Validation Accuracy and Loss Over Epochs\n(Initial Training and Fine Tuning)')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Call the function with your history objects.
plot_validation_metrics(history, history_fine)
```

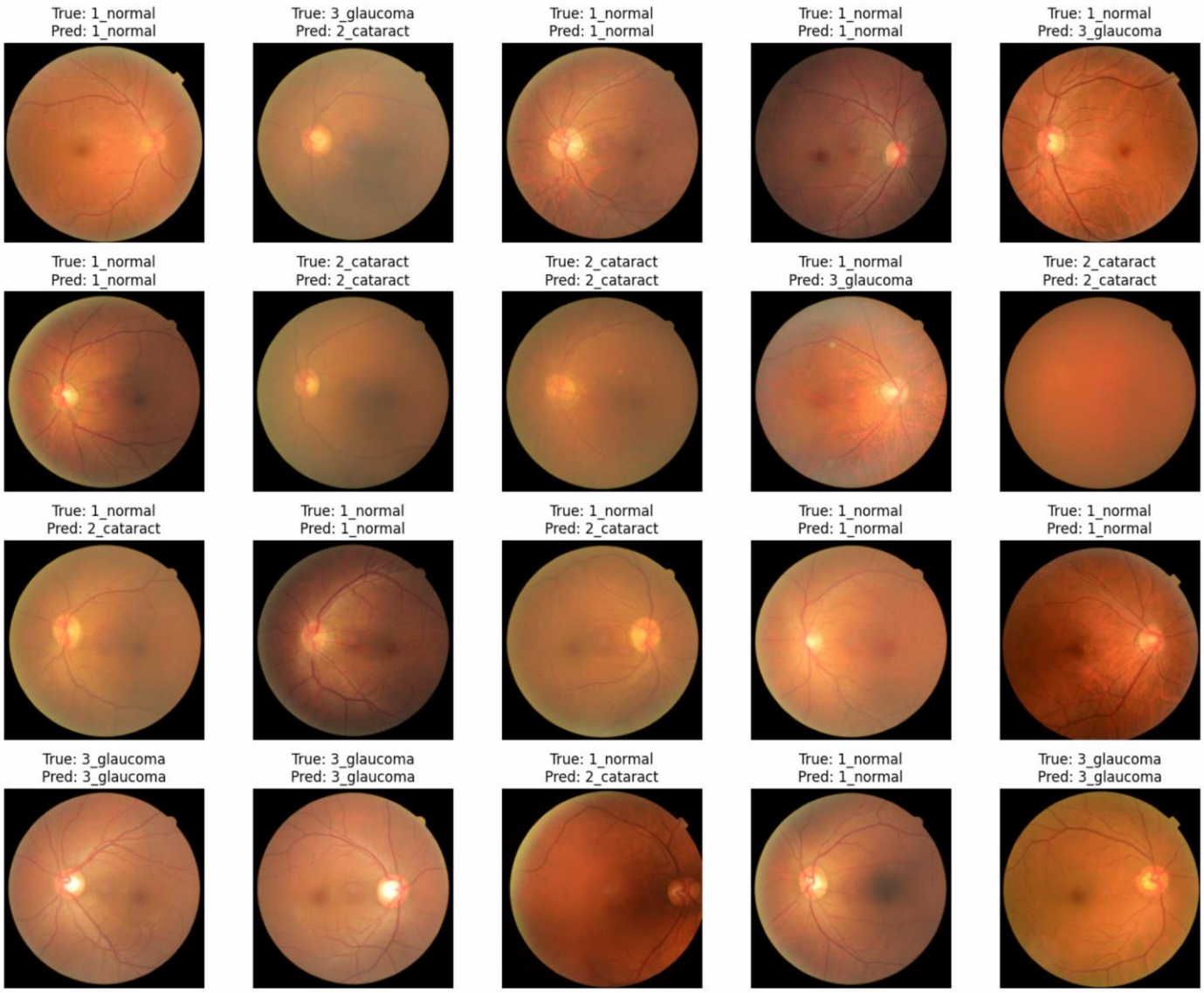


Test Predictions Visualization

```
import random
import matplotlib.pyplot as plt

# --- Compute predictions on the test set ---
predictions = model.predict(test_generator)

# Convert predictions to integer class indices.
predicted_labels = np.argmax(predictions, axis=1)

# Retrieve file paths and true labels from the test generator.
test_images = test_generator.filepaths
test_labels = test_generator.classes

# Create a mapping from class indices to class names.
# This relies on the directory names, which are stored in the generator's class_indices.
class_labels = list(test_generator.class_indices.keys())

# --- Randomly Sample 20 Test Images ---
sample_indices = random.sample(range(len(test_images)), 20)

# Set up a figure to display images in a 4 x 5 grid.
plt.figure(figsize=(15, 12))
for i, idx in enumerate(sample_indices):
    img_path = test_images[idx]
    true_label = class_labels[test_labels[idx]]
    pred_label = class_labels[predicted_labels[idx]]

    # Load the image using matplotlib.
    img = plt.imread(img_path)

    # Create subplot for each image.
    plt.subplot(4, 5, i + 1)
```

```
    plt.imshow(img)
    plt.axis('off')
    plt.title(f"True: {true_label}\nPred: {pred_label}")

plt.tight_layout()
plt.show()
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 21s 4s/step
```



Preprocessing Phase 3

Applying CLAHE filtering to Split Dataset

```
import os
import cv2
import numpy as np
from tqdm import tqdm

def apply_milder_clahe_to_directory(input_dir, output_dir, clip_limit=1.5, tile_grid_size=(8, 8)):
    """
    Applies CLAHE (Contrast Limited Adaptive Histogram Equalization) to all images in the specified directory.

    Args:
        input_dir (str): Path to the input directory containing class subdirectories.
        output_dir (str): Path to the output directory where processed images will be saved.
        clip_limit (float): Threshold for contrast limiting.
        tile_grid_size (tuple): Size of the grid for histogram equalization.
    """
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    for class_name in os.listdir(input_dir):
        class_input_path = os.path.join(input_dir, class_name)
        class_output_path = os.path.join(output_dir, class_name)

        if not os.path.exists(class_output_path):
            os.makedirs(class_output_path)

        for img_name in os.listdir(class_input_path):
            img_path = os.path.join(class_input_path, img_name)
            output_path = os.path.join(class_output_path, img_name)
```

```
            img = cv2.imread(img_path)
            if img is not None:
                # Convert to LAB color space
                lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
                l, a, b = cv2.split(lab)

                # Apply CLAHE only on the L channel (lightness)
                clahe = cv2.createCLAHE(clipLimit=clip_limit, tileGridSize=tile_grid_size)
                l_clahe = clahe.apply(l)

                # Merge the channels back and convert to BGR
                lab_clahe = cv2.merge((l_clahe, a, b))
                img_clahe = cv2.cvtColor(lab_clahe, cv2.COLOR_LAB2BGR)

                # Save the processed image
                cv2.imwrite(output_path, img_clahe)
            else:
                print(f"Warning: Could not read image {img_path}")


# Paths for Train, Validation, and Test Datasets
train_dir = './raw_data_split/train'
val_dir = './raw_data_split/val'
test_dir = './raw_data_split/test'

# Output Paths for Processed Datasets
train_clahe_dir = './raw_data_split/train_clahe'
val_clahe_dir = './raw_data_split/val_clahe'
test_clahe_dir = './raw_data_split/test_clahe'

# Apply Milder CLAHE to All Datasets
apply_milder_clahe_to_directory(train_dir, train_clahe_dir, clip_limit=1.5, tile_grid_size=(8, 8))
apply_milder_clahe_to_directory(val_dir, val_clahe_dir, clip_limit=1.5, tile_grid_size=(8, 8))
apply_milder_clahe_to_directory(test_dir, test_clahe_dir, clip_limit=1.5, tile_grid_size=(8, 8))

print("CLAHE processing complete for Train, Validation, and Test datasets.")
```

```
CLAHE processing complete for Train, Validation, and Test datasets.
```

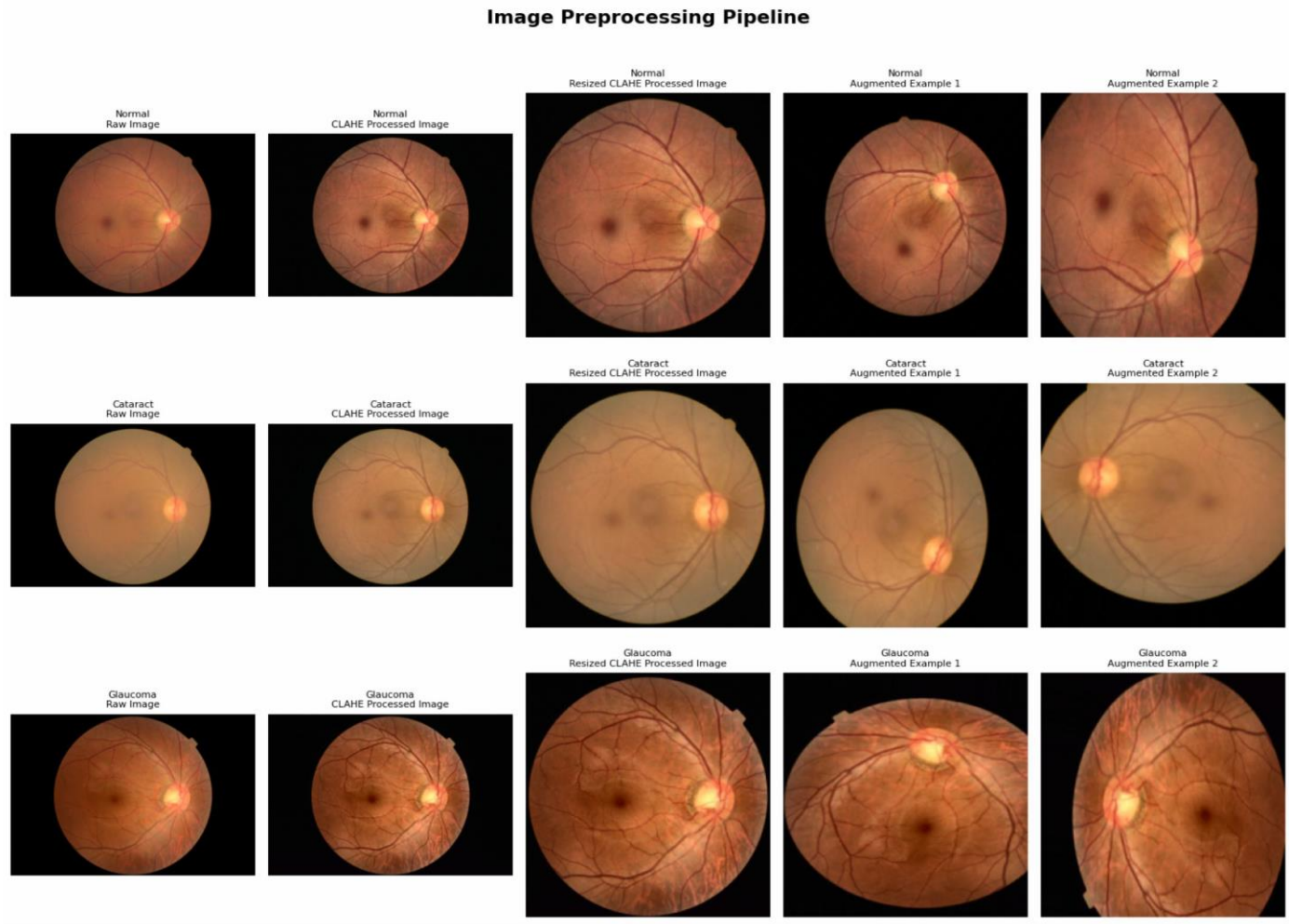
Balancing CLAHE processed training split to 300 images per class by augmentation

```
# Paths
input_dir = './raw_data_split/train_clahe'
output_dir = './raw_data_split/train_augmented_clahe'  # Directory to save augmented images
target_size = 300  # Desired number of images per class

# ImageDataGenerator for augmentation
datagen = ImageDataGenerator(
    rotation_range=180,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.3,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Create output directory
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

# Augmentation strategy
augmentation_config = {
    '1_normal': 2,  # Generate 2 augmented images per input image
    '2_cataract': 4,  # Generate 4 augmented images per input image
    '3_glaucoma': 4   # Generate 4 augmented images per input image
}

# Loop through each class directory
for class_name in os.listdir(input_dir):
    class_input_path = os.path.join(input_dir, class_name)
    class_output_path = os.path.join(output_dir, class_name)

    # Ensure the output class directory exists
    if not os.path.exists(class_output_path):
        os.makedirs(class_output_path)

    # Copy original images to the output directory
    images = os.listdir(class_input_path)
    for img_name in images:
        input_img_path = os.path.join(class_input_path, img_name)
        output_img_path = os.path.join(class_output_path, img_name)
        shutil.copy(input_img_path, output_img_path)

    # Count images now in the output directory (includes original images)
    image_count = len(os.listdir(class_output_path))

    # Determine augmentation multiplier based on class
    augmentation_multiplier = augmentation_config.get(class_name, 1)  # Default to 1 if class not in config

    # Augment images if the count is below the target size
    if image_count < target_size:
        print(f"Augmenting class: {class_name} ({image_count}/{target_size})")
        for img_name in images:
            img_path = os.path.join(class_input_path, img_name)
            img = load_img(img_path)  # Load image
            img_array = img_to_array(img)  # Convert to array
            img_array = img_array.reshape((1,) + img_array.shape)  # Reshape for augmentation

            # Generate augmented images
            save_prefix = os.path.splitext(img_name)[0]
            generated_images = 0
            for batch in datagen.flow(img_array, batch_size=1, save_to_dir=class_output_path,
                                      save_prefix=save_prefix, save_format='png'):
                generated_images += 1
                if generated_images >= augmentation_multiplier:
                    break

            # Update the image count
```

```
            image_count += augmentation_multiplier
            if image_count >= target_size:
                break

        print(f"Class {class_name} balanced to {target_size} images.")
    else:
        print(f"Class {class_name} already has {image_count} images. No augmentation needed.")

        # Validate the number of files in the output directory
    final_image_count = len(os.listdir(class_output_path))
    print(f"Final file count in '{class_name}': {final_image_count} files.")
```

```
Augmenting class: 1_normal (180/300)
Class 1_normal balanced to 300 images.
Final file count in '1_normal': 300 files.
Augmenting class: 2_cataract (60/300)
Class 2_cataract balanced to 300 images.
Final file count in '2_cataract': 300 files.
Augmenting class: 3_glaucoma (60/300)
Class 3_glaucoma balanced to 300 images.
Final file count in '3_glaucoma': 300 files.
```

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Image paths for each class and preprocessing step
image_paths = {
    "Normal": [
        "./raw_data_split/train/1_normal/NL_061.png",  # Raw image
        "./raw_data_split/train_clahe/1_normal/NL_061.png",  # CLAHE processed
        "./resized_clahe_images/train_clahe/1_normal/NL_061.png",  # Resized CLAHE processed
        "./resized_clahe_images/train_augmented_clahe/1_normal/NL_061_0_4836.png",  # Augmented example 1
        "./resized_clahe_images/train_augmented_clahe/1_normal/NL_061_0_51.png",  # Augmented example 2
    ],
    "Cataract": [
        "./raw_data_split/train/2_cataract/cataract_096.png",  # Raw image
        "./raw_data_split/train_clahe/2_cataract/cataract_096.png",  # CLAHE processed
        "./resized_clahe_images/train_clahe/2_cataract/cataract_096.png",  # Resized CLAHE processed
        "./resized_clahe_images/train_augmented_clahe/2_cataract/cataract_096_0_8519.png",  # Augmented example 1
        "./resized_clahe_images/train_augmented_clahe/2_cataract/cataract_096_0_5314.png",  # Augmented example 2
    ],
    "Glaucoma": [
        "./raw_data_split/train/3_glaucoma/Glaucoma_048.png",  # Raw image
        "./raw_data_split/train_clahe/3_glaucoma/Glaucoma_048.png",  # CLAHE processed
        "./resized_clahe_images/train_clahe/3_glaucoma/Glaucoma_048.png",  # Resized CLAHE processed
        "./resized_clahe_images/train_augmented_clahe/3_glaucoma/Glaucoma_048_0_8095.png",  # Augmented example 1
        "./resized_clahe_images/train_augmented_clahe/3_glaucoma/Glaucoma_048_0_7115.png",  # Augmented example 2
    ]
}

# Descriptions for each step
descriptions = [
    "Raw Image",
    "CLAHE Processed Image",
    "Resized CLAHE Processed Image",
    "Augmented Example 1",
    "Augmented Example 2",
]

# Plot settings
nrows = len(image_paths)  # One row per class
ncols = len(descriptions)  # Columns to match descriptions
fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=(15, 10))

# Set the main title for the plot
fig.suptitle("Image Preprocessing Pipeline", fontsize=16, weight="bold", y=1.08)

# Populate the plot with images
for row_idx, (label, paths) in enumerate(image_paths.items()):
    for col_idx, (path, desc) in enumerate(zip(paths, descriptions)):
        ax = axes[row_idx, col_idx] if nrows > 1 else axes[col_idx]
        try:
            img = mpimg.imread(path)
            ax.imshow(img)
            ax.set_title(f"{label}\n{desc}", fontsize=8)
        except FileNotFoundError:
            ax.text(0.5, 0.5, "Image Not Found", ha="center", va="center", fontsize=10)
        ax.axis("off")

plt.tight_layout()
plt.subplots_adjust(top=1)  # Adjust space for the title
plt.show()
```



Some directory paths might be different due to running various experiments that I haven't included to the final jupyter notebook.

Experiment 3

Balanced Dataset Testing with DenseNet-121 (CLAHE filtered)

Fine-tuning to the whole base model

Training Strategy

This experiment introduces CLAHE preprocessing to enhance image contrast and refines the fine-tuning process through additional stability mechanisms. The primary modifications are:

- **CLAHE filtering:** Contrast enhancement is applied to all images before training, which may improve feature extraction for retinal images.
- **More structured fine-tuning strategy:** The fine-tuning phase now includes weight transfer, where the initial classifier head's trained weights are retained before re-initializing it.
- **BatchNormalization layers remain frozen but their momentum is set to 0.9** for better adaptation during training. This allows some gradual adaptation without destabilizing the learned feature distribution.
- **Learning rate warm-up phase:** Instead of using a fixed learning rate for fine-tuning, the model **gradually increases the learning rate over 12 epochs** before stabilizing it at **5e-6**.

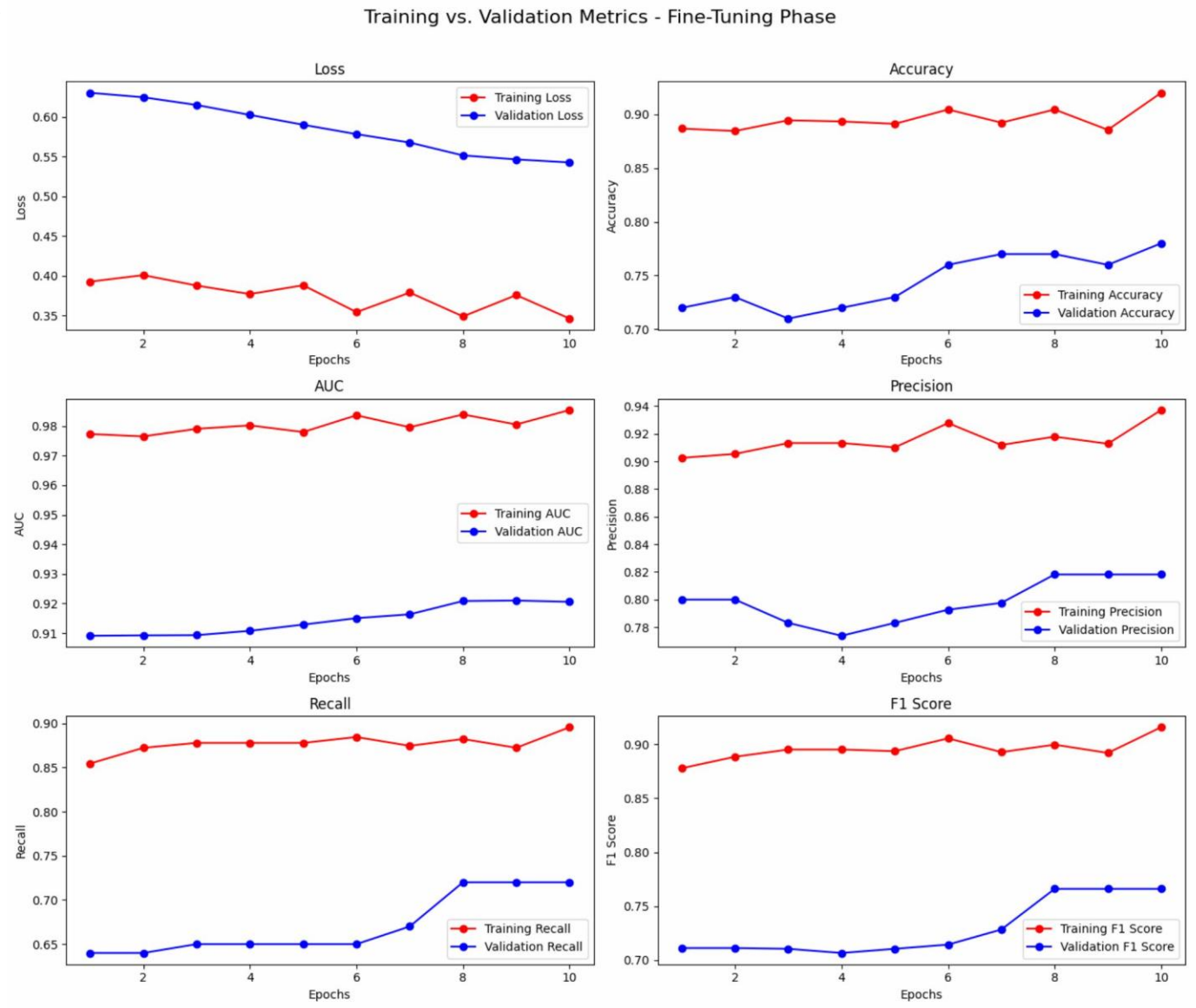
```
import tensorflow as tf
import numpy as np
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense, BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.regularizers import l2
from tensorflow.keras.metrics import Precision, Recall, AUC
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, LearningRateScheduler

# --- Enable GPU Memory Growth ---
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
    except RuntimeError as e:
        print(e)

# --- Custom F1 Score Metric ---
class F1Score(tf.keras.metrics.Metric):
```

```
    def __init__(self, name="f1_score", **kwargs):
        super(F1Score, self).__init__(name=name, **kwargs)
        self.precision = tf.keras.metrics.Precision()
        self.recall = tf.keras.metrics.Recall()

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.precision.update_state(y_true, y_pred, sample_weight)
        self.recall.update_state(y_true, y_pred, sample_weight)

    def result(self):
        precision = self.precision.result()
        recall = self.recall.result()
        return 2 * ((precision * recall) / (precision + recall + tf.keras.backend.epsilon()))

    def reset_states(self):
        self.precision.reset_states()
        self.recall.reset_states()

# --- Paths ---
train_dir = './resized_clahe_images/train_augmented_clahe'
val_dir   = './resized_clahe_images/val_clahe'
test_dir  = './resized_clahe_images/test_clahe'
IMG_SIZE  = (224, 224)

# --- Hyperparameters ---
BATCH_SIZE_TRAIN = 32
LEARNING_RATE    = 0.00016084
DROPOUT_RATE_1   = 0.3
DROPOUT_RATE_2   = 0.5
DENSE_UNITS      = 512
EPOCHS           = 30
FINE_TUNE_EPOCHS = 10

# --- Class Weights (to balance dataset) ---
class_weights = {0: 1.0, 1: 1.5, 2: 1.8}

# --- Data Generators for Image Loading ---
train_datagen = ImageDataGenerator(rescale=1.0/255)
val_datagen   = ImageDataGenerator(rescale=1.0/255)
test_datagen  = ImageDataGenerator(rescale=1.0/255)

train_generator = train_datagen.flow_from_directory(
    train_dir, target_size=IMG_SIZE, batch_size=BATCH_SIZE_TRAIN, class_mode='categorical', shuffle=True
)

validation_generator = val_datagen.flow_from_directory(
    val_dir, target_size=IMG_SIZE, batch_size=BATCH_SIZE_TRAIN, class_mode='categorical'
)

test_generator = test_datagen.flow_from_directory(
    test_dir, target_size=IMG_SIZE, batch_size=BATCH_SIZE_TRAIN, class_mode='categorical', shuffle=False
)

# --- Build Model for Initial Training ---
def build_initial_model():
    base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    base_model.trainable = False  # Freeze base model (feature extraction only)

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = BatchNormalization()(x)
    x = Dropout(DROPOUT_RATE_1)(x)
    x = Dense(DENSE_UNITS, activation='relu', name='dense')(x)
    x = Dropout(DROPOUT_RATE_2)(x)
    output = Dense(3, activation='softmax', name='dense_1')(x)

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

    model.compile(
        optimizer=Adam(learning_rate=LEARNING_RATE),
        loss='categorical_crossentropy',
        metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1_score')]
    )
    model.base_model = base_model  # Save reference for fine-tuning
    return model

model = build_initial_model()

# --- Callbacks for Initial Training ---
early_stopping = EarlyStopping(monitor='val_f1_score', mode='max', patience=10, restore_best_weights=True)
reduce_lr      = ReduceLROnPlateau(monitor='val_f1_score', mode='max', factor=0.5, patience=3, verbose=1)

# --- Initial Training (Feature Extraction) ---
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS,
    class_weight=class_weights,
    callbacks=[reduce_lr, early_stopping]
)

# --- Fine-Tuning Phase ---
# Unfreeze DenseNet121 base model for full fine-tuning
model.base_model.trainable = True

# Keep BatchNormalization layers frozen to maintain stability
for layer in model.layers:
    if isinstance(layer, BatchNormalization):
        layer.trainable = False
        layer.momentum = 0.9

# Save pre-trained weights from initial training
dense_weights = model.get_layer('dense').get_weights()
output_weights = model.get_layer('dense_1').get_weights()

# Rebuild the classification head (without dropout)
inputs = model.base_model.input
x = model.base_model.output
x = GlobalAveragePooling2D()(x)
x = BatchNormalization()(x)
```

```
x = Dense(DENSE_UNITS, activation='relu', name='dense')(x)
x = Dense(3, activation='softmax', name='dense_1')(x)
fine_tune_model = Model(inputs=inputs, outputs=x)

# Transfer trained weights to the new classification head
fine_tune_model.get_layer('dense').set_weights(dense_weights)
fine_tune_model.get_layer('dense_1').set_weights(output_weights)

# --- Learning Rate Schedule for Fine-Tuning ---
warmup_epochs = 12  # Gradually increase LR
max_lr = 5e-6       # Stable learning rate after warm-up

def lr_schedule(epoch):
    if epoch < warmup_epochs:
        return 1e-7 * (10 ** (epoch / warmup_epochs))  # Exponential warm-up
    else:
        return max_lr  # Fixate learning rate at 5e-6

lr_callback = LearningRateScheduler(lr_schedule)

# Compile model with lower LR for fine-tuning
fine_tune_model.compile(
    optimizer=Adam(learning_rate=1e-7),
    loss='categorical_crossentropy',
    metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1_score')]
)

# --- Fine-Tuning Training ---
fine_tune_history = fine_tune_model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=FINE_TUNE_EPOCHS,
    class_weight=class_weights,
    callbacks=[early_stopping, lr_callback]
)

# --- Evaluate on Test Data ---
results = fine_tune_model.evaluate(test_generator)
print("Test Results:", dict(zip(fine_tune_model.metrics_names, results)))
```

Training and Validation Metrics Over Epochs

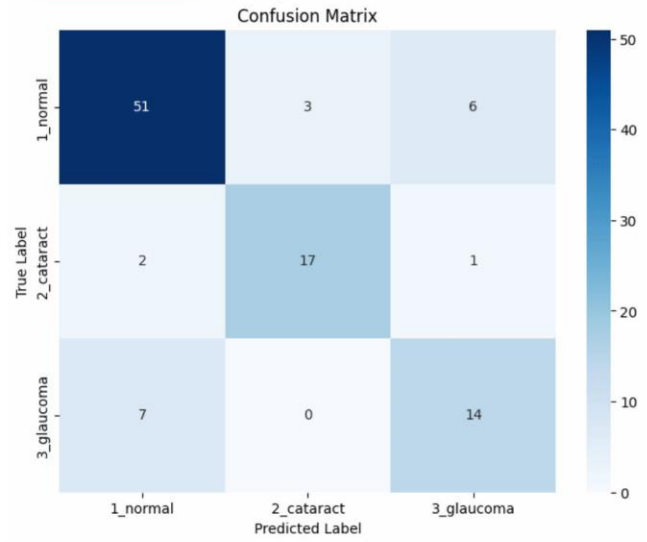
```
import matplotlib.pyplot as plt

# Extract the number of epochs from the fine-tuning history.
epochs = range(1, len(fine_tune_history.history["loss"]) + 1)

# Create subplots: 3 rows and 2 columns.
fig, axes = plt.subplots(3, 2, figsize=(14, 12))
fig.suptitle("Training vs. Validation Metrics - Fine-Tuning Phase", fontsize=16)

# Plot Loss.
axes[0, 0].plot(epochs, fine_tune_history.history["loss"], label="Training Loss", color='red', marker='o')
axes[0, 0].plot(epochs, fine_tune_history.history["val_loss"], label="Validation Loss", color='blue', marker='o')
axes[0, 0].set_title("Loss")
axes[0, 0].set_xlabel("Epochs")
axes[0, 0].set_ylabel("Loss")
axes[0, 0].legend()

# Plot Accuracy.
axes[0, 1].plot(epochs, fine_tune_history.history["accuracy"], label="Training Accuracy", color='red', marker='o')
axes[0, 1].plot(epochs, fine_tune_history.history["val_accuracy"], label="Validation Accuracy", color='blue', marker='o')
axes[0, 1].set_title("Accuracy")
axes[0, 1].set_xlabel("Epochs")
axes[0, 1].set_ylabel("Accuracy")
axes[0, 1].legend()

# Plot AUC.
axes[1, 0].plot(epochs, fine_tune_history.history["auc"], label="Training AUC", color='red', marker='o')
axes[1, 0].plot(epochs, fine_tune_history.history["val_auc"], label="Validation AUC", color='blue', marker='o')
axes[1, 0].set_title("AUC")
axes[1, 0].set_xlabel("Epochs")
axes[1, 0].set_ylabel("AUC")
axes[1, 0].legend()

# Plot Precision.
axes[1, 1].plot(epochs, fine_tune_history.history["precision"], label="Training Precision", color='red', marker='o')
axes[1, 1].plot(epochs, fine_tune_history.history["val_precision"], label="Validation Precision", color='blue', marker='o')
axes[1, 1].set_title("Precision")
axes[1, 1].set_xlabel("Epochs")
axes[1, 1].set_ylabel("Precision")
axes[1, 1].legend()

# Plot Recall.
axes[2, 0].plot(epochs, fine_tune_history.history["recall"], label="Training Recall", color='red', marker='o')
axes[2, 0].plot(epochs, fine_tune_history.history["val_recall"], label="Validation Recall", color='blue', marker='o')
axes[2, 0].set_title("Recall")
axes[2, 0].set_xlabel("Epochs")
axes[2, 0].set_ylabel("Recall")
axes[2, 0].legend()

# Plot F1 Score.
axes[2, 1].plot(epochs, fine_tune_history.history["f1_score"], label="Training F1 Score", color='red', marker='o')
axes[2, 1].plot(epochs, fine_tune_history.history["val_f1_score"], label="Validation F1 Score", color='blue', marker='o')
axes[2, 1].set_title("F1 Score")
axes[2, 1].set_xlabel("Epochs")
axes[2, 1].set_ylabel("F1 Score")
axes[2, 1].legend()

# Adjust layout to prevent overlap and keep the main title visible.
plt.tight_layout(rect=[0, 0, 1, 0.97])
plt.show()
```



Confusion Matrix & Classification Report

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

# Reset the generator to ensure predictions start from the beginning.
test_generator.reset()

# Generate predictions for the test set.
# The number of steps is calculated using np.ceil to cover all test samples.
predictions = fine_tune_model.predict(test_generator, steps=int(np.ceil(test_generator.samples / test_generator.batch_size)))
# For multi-class classification, choose the index with the highest probability.
predicted_classes = np.argmax(predictions, axis=1)

# Get the true labels from the generator.
true_classes = test_generator.classes

# Retrieve the class labels (names) based on the directory structure.
class_labels = list(test_generator.class_indices.keys())

# Compute the confusion matrix.
cm = confusion_matrix(true_classes, predicted_classes)

# Plot the confusion matrix using seaborn.
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# Generate and print the classification report.
report = classification_report(true_classes, predicted_classes, target_names=class_labels)
print("Classification Report:\n", report)
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 17s 4s/step
```

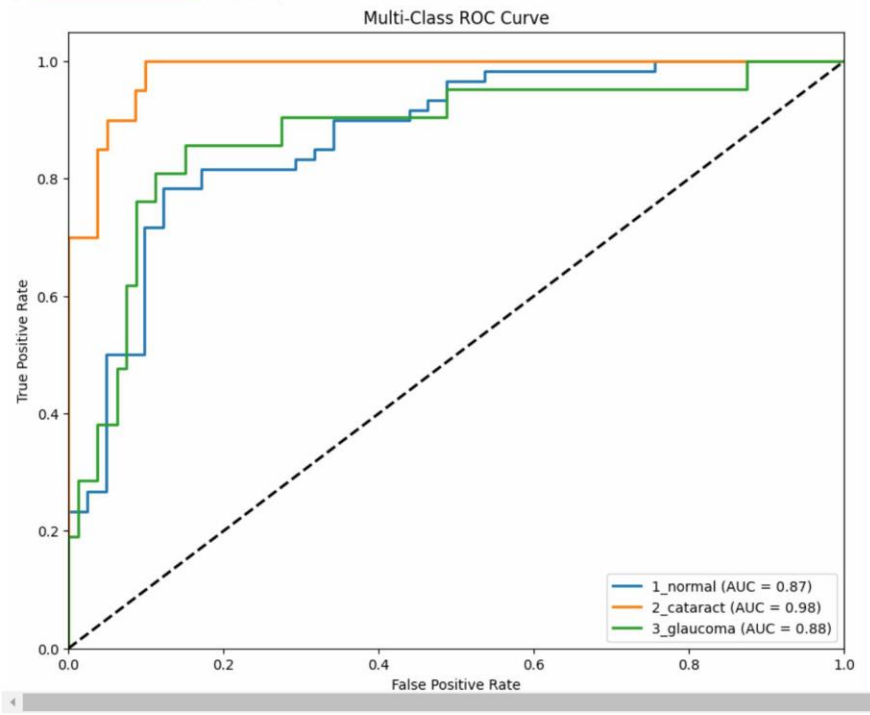



```
Classification Report:
               precision    recall  f1-score   support

    1_normal       0.85      0.85      0.85        60
  2_cataract       0.85      0.85      0.85        20
  3_glaucoma       0.67      0.67      0.67        21

    accuracy                           0.81       101
   macro avg       0.79      0.79      0.79       101
weighted avg       0.81      0.81      0.81       101
```

ROC curve

```
from sklearn.metrics import roc_curve, auc
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import numpy as np

# Generate predictions on the test set.
predictions = fine_tune_model.predict(test_generator)

# Get true class indices from the test generator.
true_classes = test_generator.classes

# Get class names from the class_indices dictionary.
class_names = list(test_generator.class_indices.keys())
num_classes = len(class_names)

# Convert true class indices to one-hot encoding.
y_true_onehot = to_categorical(true_classes, num_classes=num_classes)
y_pred = predictions  # Predicted probabilities

# Compute ROC curves and AUC for each class.
fpr = {}
tpr = {}
roc_auc = {}
for i in range(num_classes):
    fpr[i], tpr[i], _ = roc_curve(y_true_onehot[:, i], y_pred[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(10, 8))
for i, label in enumerate(class_names):
    plt.plot(fpr[i], tpr[i], lw=2, label=f'{label} (AUC = {roc_auc[i]:.2f})')
plt.plot([0, 1], [0, 1], 'k--', lw=2)  # Diagonal line for random classifier.
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Multi-Class ROC Curve')
plt.legend(loc="lower right")
plt.show()
```

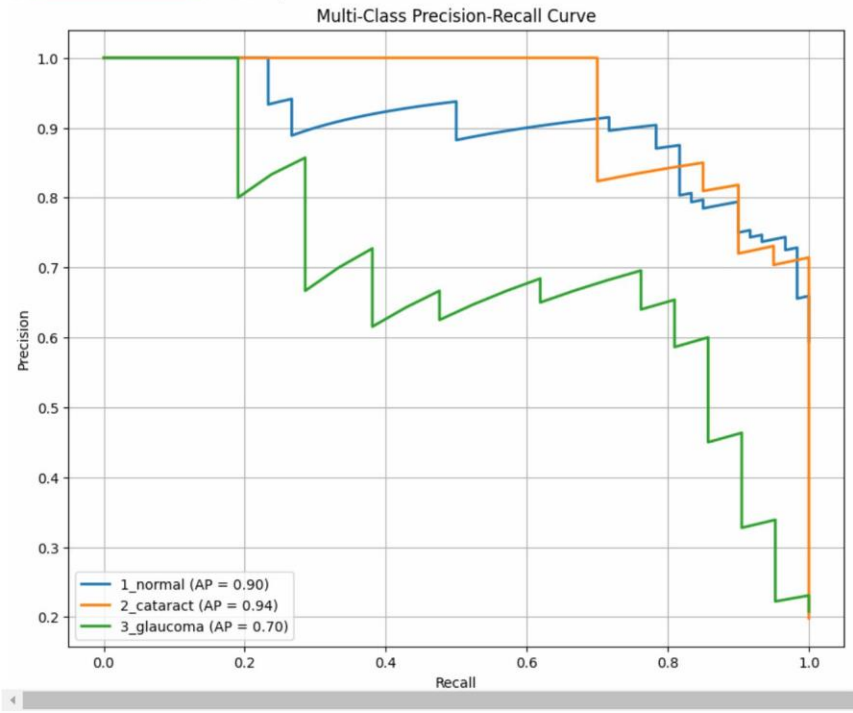
```
4/4 ━━━━━━━━━━━━━━━━━━━━ 17s 4s/step
```



Precision-Recall curve

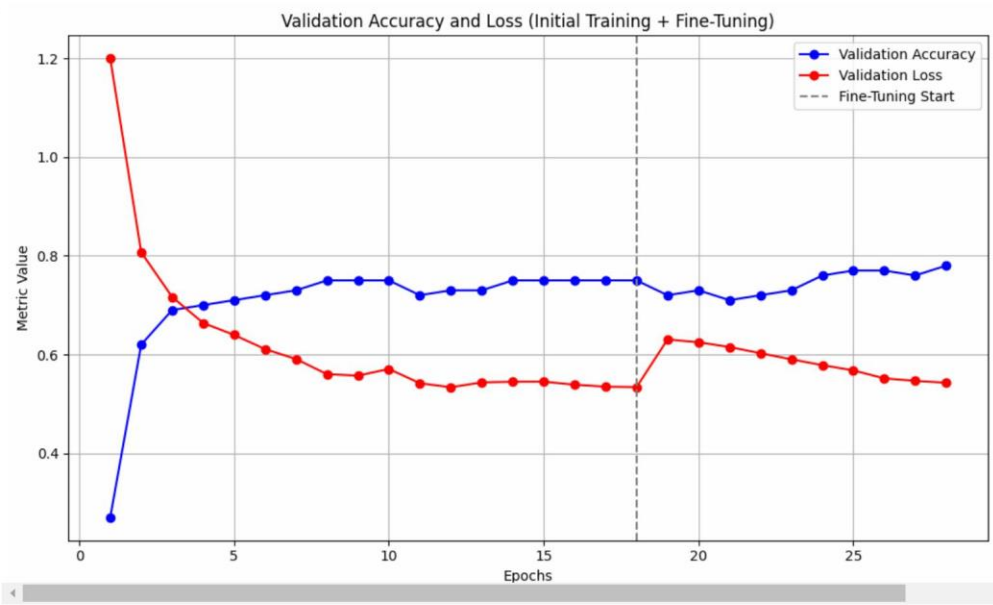
```
from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt
from tensorflow.keras.utils import to_categorical
import numpy as np

# Get class names and number of classes.
class_names = list(test_generator.class_indices.keys())
num_classes = len(class_names)

# Get true class indices from test_generator.
true_classes = test_generator.classes

# Convert true classes to one-hot encoding.
y_true_onehot = to_categorical(true_classes, num_classes=num_classes)

# Generate predictions on the test set (predicted probabilities).
y_pred = fine_tune_model.predict(test_generator)

# Compute precision-recall curves and average precision for each class.
precision_dict = {}
recall_dict = {}
average_precision = {}
for i in range(num_classes):
    precision_dict[i], recall_dict[i], _ = precision_recall_curve(y_true_onehot[:, i], y_pred[:, i])
    average_precision[i] = average_precision_score(y_true_onehot[:, i], y_pred[:, i])

# Plot the precision-recall curves.
plt.figure(figsize=(10, 8))
for i, label in enumerate(class_names):
    plt.plot(recall_dict[i], precision_dict[i], lw=2, label=f'{label} (AP = {average_precision[i]:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Multi-Class Precision-Recall Curve')
plt.legend(loc="lower left")
plt.grid(True)
plt.show()
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 17s 4s/step
```



Validation Accuracy and Loss over Epoch

```
import matplotlib.pyplot as plt

# Combine validation metrics from initial and fine-tuning phases.
initial_val_acc = history.history['val_accuracy']
initial_val_loss = history.history['val_loss']
fine_val_acc = fine_tune_history.history['val_accuracy']
fine_val_loss = fine_tune_history.history['val_loss']

total_epochs = len(initial_val_acc) + len(fine_val_acc)
epochs = range(1, total_epochs + 1)

combined_val_acc = initial_val_acc + fine_val_acc
combined_val_loss = initial_val_loss + fine_val_loss

plt.figure(figsize=(10, 6))
plt.plot(epochs, combined_val_acc, label='Validation Accuracy', marker='o', color='blue')
plt.plot(epochs, combined_val_loss, label='Validation Loss', marker='o', color='red')
plt.xlabel('Epochs')
plt.ylabel('Metric Value')
plt.title('Validation Accuracy and Loss (Initial Training + Fine-Tuning)')
plt.axvline(x=len(initial_val_acc), color='gray', linestyle='--', label='Fine-Tuning Start')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Test Predictions Visualization

```
import matplotlib.pyplot as plt
import random

# Get file paths and true labels from test_generator.
```

```
test_images = test_generator.filepaths
test_labels = test_generator.classes
class_names = list(test_generator.class_indices.keys())

# Generate predictions if not already done.
predictions = fine_tune_model.predict(test_generator)
predicted_labels = np.argmax(predictions, axis=1)

# Randomly select 20 indices.
indices = random.sample(range(len(test_images)), 20)

plt.figure(figsize=(15, 12))
for i, idx in enumerate(indices):
    img_path = test_images[idx]
    true_label = class_names[test_labels[idx]]
    pred_label = class_names[predicted_labels[idx]]
    img = plt.imread(img_path)
    plt.subplot(4, 5, i+1)
    plt.imshow(img)
    plt.axis('off')
    plt.title(f"True: {true_label}\nPred: {pred_label}")
plt.tight_layout()
plt.show()
```

Experiment 4

Balanced Dataset Testing with DenseNet-121 (CLAHE filtered)

Fine-tuning to the last 82 layers of the base model

Training Strategy

- Unlike previous experiments, which fully fine-tuned the entire DenseNet121 base model, this script **unfreezes only the last 82 layers** for fine-tuning. This reduces the risk of overfitting while still allowing the model to learn more domain-specific features.
- The fine-tuning phase starts with a **more conservative learning rate (1e-5)** for more drastic weight changes and gradual warm-up has been removed. Nonetheless, callbacks such as ReduceLROnPlateau (to lower the learning rate when validation loss plateaus) and EarlyStopping (to halt training if the validation loss does not improve) are still in place.
- **Different class weights** have been implemented

```
import tensorflow as tf
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import Precision, Recall, AUC
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

# --- Custom F1 Score Metric ---
class F1Score(tf.keras.metrics.Metric):
    def __init__(self, name="f1_score", **kwargs):
        super(F1Score, self).__init__(name=name, **kwargs)
        self.precision = tf.keras.metrics.Precision()
        self.recall = tf.keras.metrics.Recall()

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.precision.update_state(y_true, y_pred, sample_weight)
        self.recall.update_state(y_true, y_pred, sample_weight)

    def result(self):
        precision = self.precision.result()
        recall = self.recall.result()
        return 2 * ((precision * recall) / (precision + recall + tf.keras.backend.epsilon()))

    def reset_states(self):
        self.precision.reset_states()
        self.recall.reset_states()


# --- Paths ---
train_dir = './resized_clahe_images/train_augmented_clahe'
val_dir = './resized_clahe_images/val_clahe'
test_dir = './resized_clahe_images/test_clahe'

IMG_SIZE = (224, 224)

# --- Hyperparameters ---
BATCH_SIZE = 64
BASE_LR = 0.00033609 # Initial Learning Rate
FINE_TUNE_LR = 0.00001  # Lower LR for fine-tuning
DROPOUT_RATE_1 = 0.3
DROPOUT_RATE_2 = 0.5
DENSE_UNITS = 384
EPOCHS = 30
FINE_TUNE_EPOCHS = 20

# --- Compute Class Weights for Imbalanced Dataset ---
class_weights = {
    0: 1.0,  # '1_normal'
    1: 1.9,  # '2_cataract'
    2: 2.1   # '3_glaucoma'
}

# --- Build Model ---
def build_model():
    base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    base_model.trainable = False  # Initially freeze the base model

    x = base_model.output
```

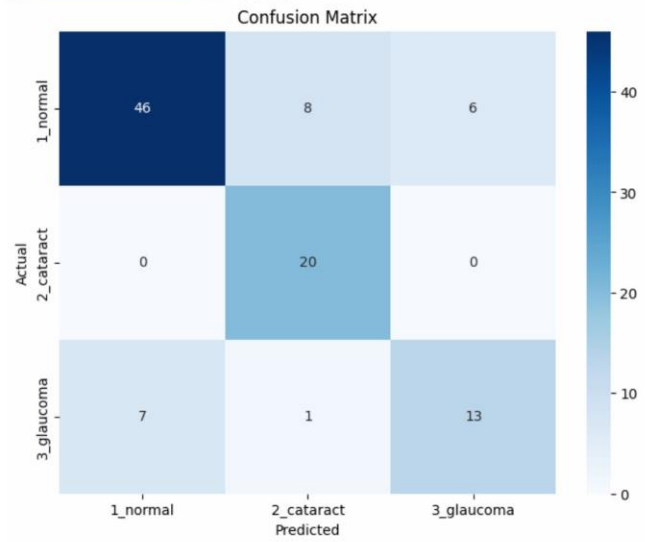
```
    x = GlobalAveragePooling2D()(x)
    x = BatchNormalization()(x)
    x = Dropout(DROPOUT_RATE_1)(x)
    x = Dense(DENSE_UNITS, activation='relu')(x)
    x = Dropout(DROPOUT_RATE_2)(x)
    output_layer = Dense(3, activation='softmax')(x)

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

    # Compile the model
    model.compile(
        optimizer=Adam(learning_rate=BASE_LR),
        loss='categorical_crossentropy',
        metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1_score')]
    )
    return model


# --- Data Generators ---
train_datagen = ImageDataGenerator(rescale=1.0 / 255)
val_test_datagen = ImageDataGenerator(rescale=1.0 / 255)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

validation_generator = val_test_datagen.flow_from_directory(
    val_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# --- Initialize the Model ---
model = build_model()

# --- Callbacks ---
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=7,
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=5,
    min_lr=1e-6,
    verbose=1
)

# --- Train the Model with Initial Frozen Base Layers ---
print("   Training the model with frozen base layers...")
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS,
    class_weight=class_weights,
    callbacks=[reduce_lr, early_stopping]
)

# --- Step 2: Unfreeze Last 82 Layers & Fine-Tune ---
print("   Unfreezing the **last 82 layers** for fine-tuning...")
for layer in model.layers[-82:]:  # Unfreeze last 82 layers
    layer.trainable = True

# Recompile with Lower Learning Rate
model.compile(
    optimizer=Adam(learning_rate=FINE_TUNE_LR),
    loss='categorical_crossentropy',
    metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1_score')]
)

# Train Again with Fine-Tuning
print("   Fine-tuning the model...")
history_fine = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS + FINE_TUNE_EPOCHS,
    initial_epoch=history.epoch[-1],
    class_weight=class_weights,
    callbacks=[reduce_lr, early_stopping]
)

# --- Evaluate on Test Data ---
test_generator = val_test_datagen.flow_from_directory(
    test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False
)

test_loss, test_accuracy, test_precision, test_recall, test_auc, test_f1 = model.evaluate(test_generator)
print(f"Test Results: Accuracy={test_accuracy:.2f}, Precision={test_precision:.2f}, Recall={test_recall:.2f}, Loss={test_loss:.2f}, AUC={test_auc:.2f}, F1 Score={test_f1:.2f}")
```

Confusion Matrix & Classification Report

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

def plot_confusion_matrix(model, test_generator):
```

```
    # Get true labels
    y_true = test_generator.classes

    # Get predicted probabilities
    y_pred_probs = model.predict(test_generator)

    # Convert probabilities to class predictions
    y_pred_classes = np.argmax(y_pred_probs, axis=1)

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred_classes)
    labels = list(test_generator.class_indices.keys())

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')
    plt.show()

    # Print classification report
    print("\n    Classification Report:\n")
    print(classification_report(y_true, y_pred_classes, target_names=labels))

# Call the function
plot_confusion_matrix(model, test_generator)
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 38s 15s/step
```



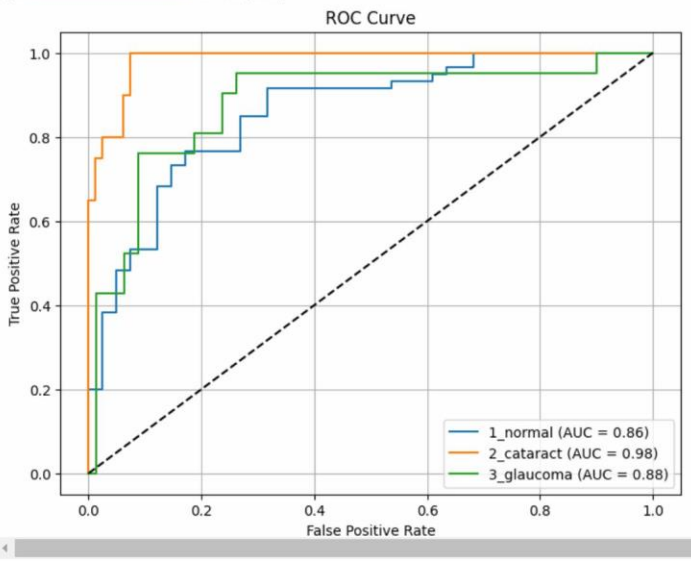

```
    Classification Report:

                  precision    recall  f1-score   support

       1_normal       0.87      0.77      0.81        60
     2_cataract       0.69      1.00      0.82        20
    3_glaucoma        0.68      0.62      0.65        21

       accuracy                           0.78       101
      macro avg       0.75      0.80      0.76       101
   weighted avg       0.79      0.78      0.78       101
```

ROC curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
import tensorflow as tf

def plot_roc_curve(model, test_generator):
    # Get true labels as one-hot encoding
    num_classes = len(test_generator.class_indices)
    y_true = tf.keras.utils.to_categorical(test_generator.classes, num_classes=num_classes)

    # Get predicted probabilities
    y_pred_probs = model.predict(test_generator, verbose=1)  # Added verbose for debugging

    # Ensure we have output to plot
    if y_pred_probs.shape[1] != num_classes:
        print("    Mismatch between predicted output shape and number of classes!")
        print(f"Expected {num_classes} classes, but got {y_pred_probs.shape[1]} outputs.")
        return

    # Plot ROC curve for each class
    plt.figure(figsize=(8, 6))
    for i, class_label in enumerate(test_generator.class_indices.keys()):
        fpr, tpr, _ = roc_curve(y_true[:, i], y_pred_probs[:, i])
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'{class_label} (AUC = {roc_auc:.2f})')

    # Diagonal reference line
    plt.plot([0, 1], [0, 1], 'k--')

    # Labels and formatting
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc='lower right')
    plt.grid()
    plt.show()
```

```
plt.show()

# Call the function
plot_roc_curve(model, test_generator)
```

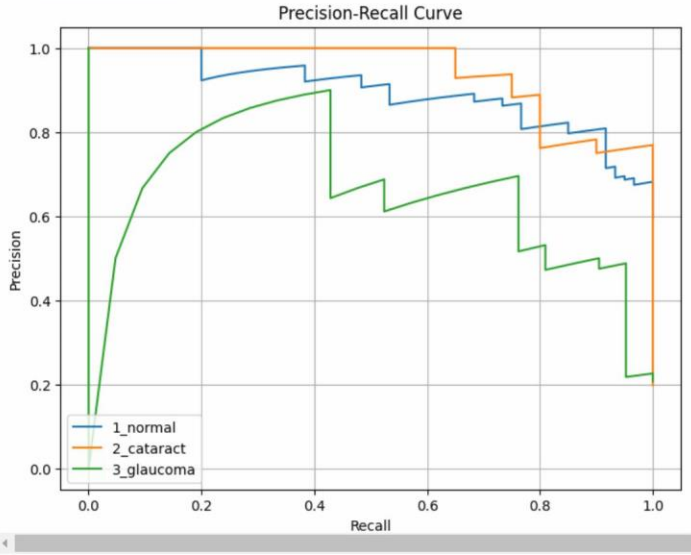
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 29s 10s/step
```



Precision-Recall curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve
import tensorflow as tf

def plot_precision_recall_curve(model, test_generator):
    # Get true labels as one-hot encoding
    num_classes = len(test_generator.class_indices)
    y_true = tf.keras.utils.to_categorical(test_generator.classes, num_classes=num_classes)

    # Get predicted probabilities
    y_pred_probs = model.predict(test_generator, verbose=1)  # Added verbose for debugging

    # Ensure we have correct number of outputs
    if y_pred_probs.shape[1] != num_classes:
        print("    Mismatch between predicted output shape and number of classes!")
        print(f"Expected {num_classes} classes, but got {y_pred_probs.shape[1]} outputs.")
        return

    # Plot Precision-Recall curve for each class
    plt.figure(figsize=(8, 6))
    for i, class_label in enumerate(test_generator.class_indices.keys()):
        precision, recall, _ = precision_recall_curve(y_true[:, i], y_pred_probs[:, i])
        plt.plot(recall, precision, label=f'{class_label}')

    # Formatting
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend(loc='lower left')
    plt.grid()
    plt.show()

# Call the function
plot_precision_recall_curve(model, test_generator)
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 22s 7s/step
```



Experiment 5

Balanced Dataset Testing with DenseNet-121 (CLAHE filtered)

No Fine-tuning to the base model

Training Strategy

Unlike previous experiments where the DenseNet121 base model was eventually unfrozen for fine-tuning, this experiment **never unfreezes the base model**. This approach only uses DenseNet121 as a fixed feature extractor while **training only the custom classification layers** (GlobalAveragePooling, BatchNorm, Dropout, and Dense layers). Class weights have also been adjusted.

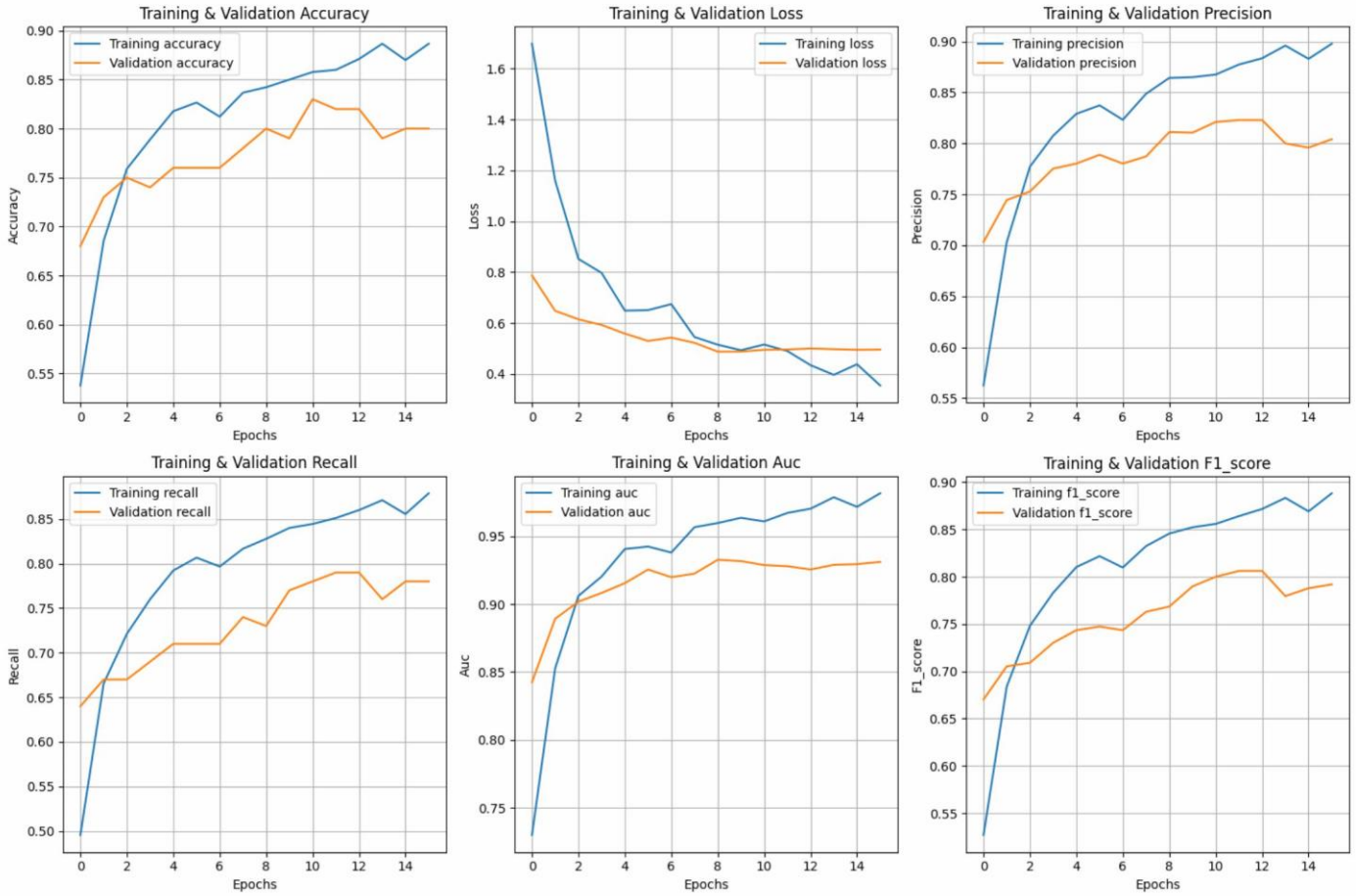
```
import tensorflow as tf
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import Precision, Recall, AUC
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np

# --- Custom F1 Score Metric ---
class F1Score(tf.keras.metrics.Metric):
    def __init__(self, name="f1_score", **kwargs):
        super(F1Score, self).__init__(name=name, **kwargs)
        self.precision = tf.keras.metrics.Precision()
        self.recall = tf.keras.metrics.Recall()

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.precision.update_state(y_true, y_pred, sample_weight)
        self.recall.update_state(y_true, y_pred, sample_weight)

    def result(self):
        precision = self.precision.result()
        recall = self.recall.result()
        return 2 * ((precision * recall) / (precision + recall + tf.keras.backend.epsilon()))

    def reset_states(self):
        self.precision.reset_states()
        self.recall.reset_states()


# --- Paths ---
train_dir = './resized_clahe_images/train_augmented_clahe'
val_dir = './resized_clahe_images/val_clahe'
test_dir = './resized_clahe_images/test_clahe'

IMG_SIZE = (224, 224)

# --- Hyperparameters ---
BATCH_SIZE = 32
BASE_LR = 0.00033609  # Initial Learning Rate
DROPOUT_RATE_1 = 0.3
DROPOUT_RATE_2 = 0.5
DENSE_UNITS = 384
EPOCHS = 30

# --- Compute Class Weights for Imbalanced Dataset ---
class_weights = {
    0: 1.0,  # '1_normal' (baseline)
    1: 1.8,  # '2_cataract' (minority)
    2: 2.0   # '3_glaucoma' (minority)
}

# --- Build Model ---
def build_model():
    base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    base_model.trainable = False  #     Never Unfreeze Base Model

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = BatchNormalization()(x)
    x = Dropout(DROPOUT_RATE_1)(x)
    x = Dense(DENSE_UNITS, activation='relu')(x)
    x = Dropout(DROPOUT_RATE_2)(x)
    output_layer = Dense(3, activation='softmax')(x)

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

    # Compile the model
    model.compile(
        optimizer=Adam(learning_rate=BASE_LR),
        loss='categorical_crossentropy',
        metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1_score')]
    )
    return model


# --- Data Generators ---
train_datagen = ImageDataGenerator(rescale=1.0 / 255)
val_test_datagen = ImageDataGenerator(rescale=1.0 / 255)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

validation_generator = val_test_datagen.flow_from_directory(
    val_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# --- Initialize the Model ---
model = build_model()
```

```
# --- Callbacks ---
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=7,  # Stops if no improvement for 7 epochs
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=4,  # Reduce LR if no improvement for 5 epochs
    min_lr=1e-6,
    verbose=1
)

# --- Train the Model (Without Fine-Tuning) ---
print("    Training the model with **frozen base layers** (No fine-tuning)...")
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS,
    class_weight=class_weights,  # Apply class weights here
    callbacks=[reduce_lr, early_stopping]
)

# --- Evaluate on Test Data ---
test_generator = val_test_datagen.flow_from_directory(
    test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False
)

test_loss, test_accuracy, test_precision, test_recall, test_auc, test_f1 = model.evaluate(test_generator)
print(f"Test Results: Accuracy={test_accuracy:.2f}, Precision={test_precision:.2f}, Recall={test_recall:.2f}, Loss={test_loss:.2f}, AUC={test_auc:.2f}, F1 Score={test_f1:.2f}")
```

Training and Validation Metrics Over Epochs

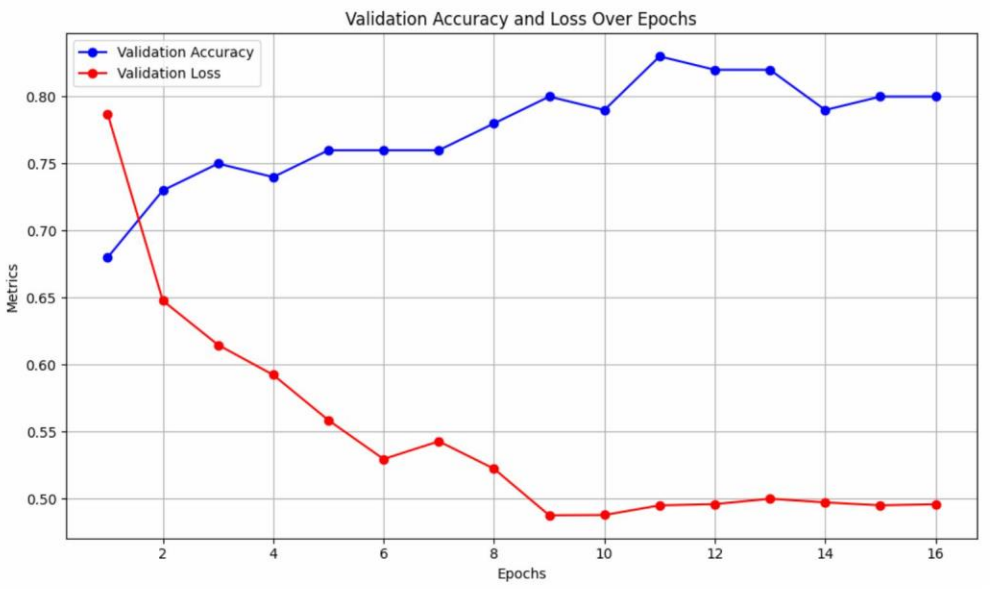
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Function to plot training history
def plot_training_history(history):
    # Define metrics to plot (excluding learning rate)
    metrics = ['accuracy', 'loss', 'precision', 'recall', 'auc', 'f1_score']

    # Check available metrics in history
    available_metrics = [m for m in metrics if m in history.history]

    # Define the figure size dynamically
    num_metrics = len(available_metrics)
    rows = (num_metrics // 3) + (num_metrics % 3 > 0)  # Dynamic row calculation

    plt.figure(figsize=(15, 5 * rows))

    for i, metric in enumerate(available_metrics):
        plt.subplot(rows, 3, i + 1)
        plt.plot(history.history[metric], label=f'Training {metric}')
        plt.plot(history.history[f'val_{metric}'], label=f'Validation {metric}')
        plt.xlabel('Epochs')
        plt.ylabel(metric.capitalize())
        plt.title(f'Training & Validation {metric.capitalize()}')
        plt.legend()
        plt.grid()

    plt.tight_layout()
    plt.show()

# Call the function
plot_training_history(history)
```

[↓]



Validation Accuracy & Loss over Epochs

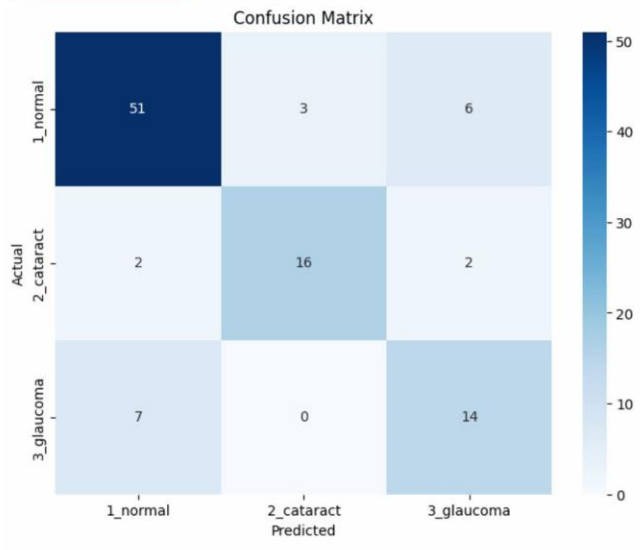
```
import matplotlib.pyplot as plt

def plot_validation_metrics(history):
    """
    Plots validation accuracy and loss on the same graph for each epoch.
    """
    epochs = range(1, len(history.history['val_accuracy']) + 1)

    # Create a figure
    plt.figure(figsize=(10, 6))

    # Plot validation accuracy
    plt.plot(epochs, history.history['val_accuracy'], label='Validation Accuracy', color='blue', marker='o')

    # Plot validation loss
    plt.plot(epochs, history.history['val_loss'], label='Validation Loss', color='red', marker='o')

    # Add labels, title, and legend
    plt.xlabel('Epochs')
    plt.ylabel('Metrics')
    plt.title('Validation Accuracy and Loss Over Epochs')
    plt.legend()

    # Add grid
    plt.grid(True)

    # Show the plot
    plt.tight_layout()
    plt.show()

# Call the function
plot_validation_metrics(history)
```



Confusion Matrix & Classification Report

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

def plot_confusion_matrix(model, test_generator):
    # Get true labels
    y_true = test_generator.classes

    # Get predicted probabilities
    y_pred_probs = model.predict(test_generator)

    # Convert probabilities to class predictions
    y_pred_classes = np.argmax(y_pred_probs, axis=1)

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred_classes)
    labels = list(test_generator.class_indices.keys())

    # Plot confusion matrix
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')
    plt.show()

    # Print classification report
    print("\n    Classification Report:\n")
    print(classification_report(y_true, y_pred_classes, target_names=labels))

# Call the function
plot_confusion_matrix(model, test_generator)
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 32s 7s/step
```

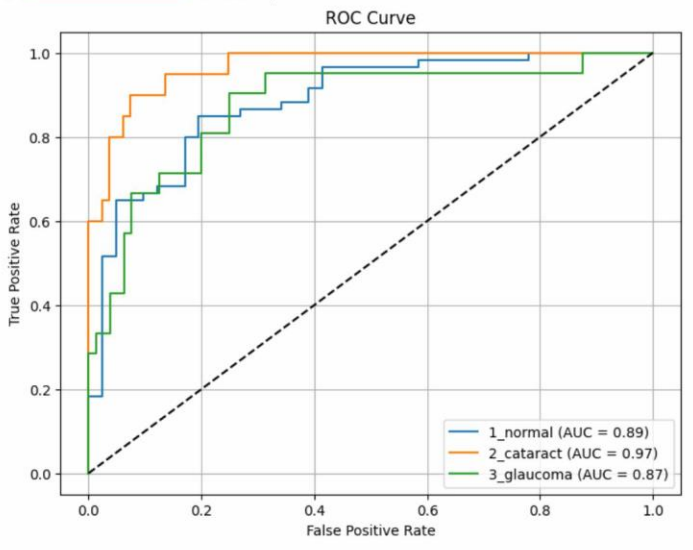



```
Classification Report:

              precision    recall  f1-score   support

    1_normal       0.85      0.85      0.85        60
  2_cataract       0.84      0.80      0.82        20
 3_glaucoma        0.64      0.67      0.65        21

    accuracy                           0.80       101
   macro avg       0.78      0.77      0.77       101
weighted avg       0.80      0.80      0.80       101
```

ROC curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
import tensorflow as tf

def plot_roc_curve(model, test_generator):
    # Get true labels as one-hot encoding
    num_classes = len(test_generator.class_indices)
    y_true = tf.keras.utils.to_categorical(test_generator.classes, num_classes=num_classes)

    # Get predicted probabilities
    y_pred_probs = model.predict(test_generator, verbose=1)  # Added verbose for debugging

    # Ensure we have output to plot
    if y_pred_probs.shape[1] != num_classes:
        print("    Mismatch between predicted output shape and number of classes!")
        print(f"Expected {num_classes} classes, but got {y_pred_probs.shape[1]} outputs.")
        return

    # Plot ROC curve for each class
    plt.figure(figsize=(8, 6))
    for i, class_label in enumerate(test_generator.class_indices.keys()):
        fpr, tpr, _ = roc_curve(y_true[:, i], y_pred_probs[:, i])
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'{class_label} (AUC = {roc_auc:.2f})')

    # Diagonal reference line
    plt.plot([0, 1], [0, 1], 'k--')

    # Labels and formatting
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc='lower right')
    plt.grid()
    plt.show()

# Call the function
plot_roc_curve(model, test_generator)
```

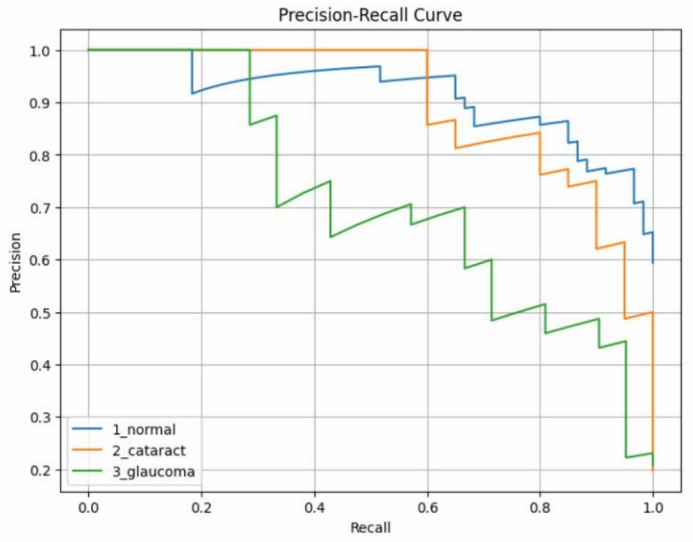
```
4/4 ━━━━━━━━━━━━━━━━━━━━ 25s 5s/step
```



Precision-Recall curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve
import tensorflow as tf

def plot_precision_recall_curve(model, test_generator):
    # Get true labels as one-hot encoding
    num_classes = len(test_generator.class_indices)
    y_true = tf.keras.utils.to_categorical(test_generator.classes, num_classes=num_classes)

    # Get predicted probabilities
    y_pred_probs = model.predict(test_generator, verbose=1)  # Added verbose for debugging

    # Ensure we have correct number of outputs
    if y_pred_probs.shape[1] != num_classes:
        print("    Mismatch between predicted output shape and number of classes!")
        print(f"Expected {num_classes} classes, but got {y_pred_probs.shape[1]} outputs.")
        return

    # Plot Precision-Recall curve for each class
    plt.figure(figsize=(8, 6))
    for i, class_label in enumerate(test_generator.class_indices.keys()):
        precision, recall, _ = precision_recall_curve(y_true[:, i], y_pred_probs[:, i])
        plt.plot(recall, precision, label=f'{class_label}')

    # Formatting
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend(loc='lower left')
    plt.grid()
    plt.show()

# Call the function
plot_precision_recall_curve(model, test_generator)
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 20s 5s/step
```



Experiment 6

Balanced Dataset Testing with DenseNet-121 (CLAHE filtered)

No Fine-tuning to the base model (focal loss)

Training Strategy

- **New Loss Function:** Instead of standard categorical crossentropy, this experiment uses **Focal Loss** to handle class imbalance effectively. Focal Loss **down-weights easy examples** and puts **more focus on hard-to-classify samples**.
- **Class weights** have been complemented with **per-class alpha values** of Normal: 0.20, Cataract: 0.35, Glaucoma: 0.65. These values ensure higher weight on Glaucoma, which is harder to classify.
- **New Learning Rate Decay Strategy:** Instead of fixed learning rates, this experiment adds a Learning Rate Scheduler.

  Strategy:

  ```
  *   For the first 10 epochs: Learning rate remains stable.
  *   Every 5 epochs after epoch 10: Learning rate is divided by 4.
  *   ReduceLROnPlateau is still used if no improvement is observed.
  ```

- Training is only enabled for the custom layers

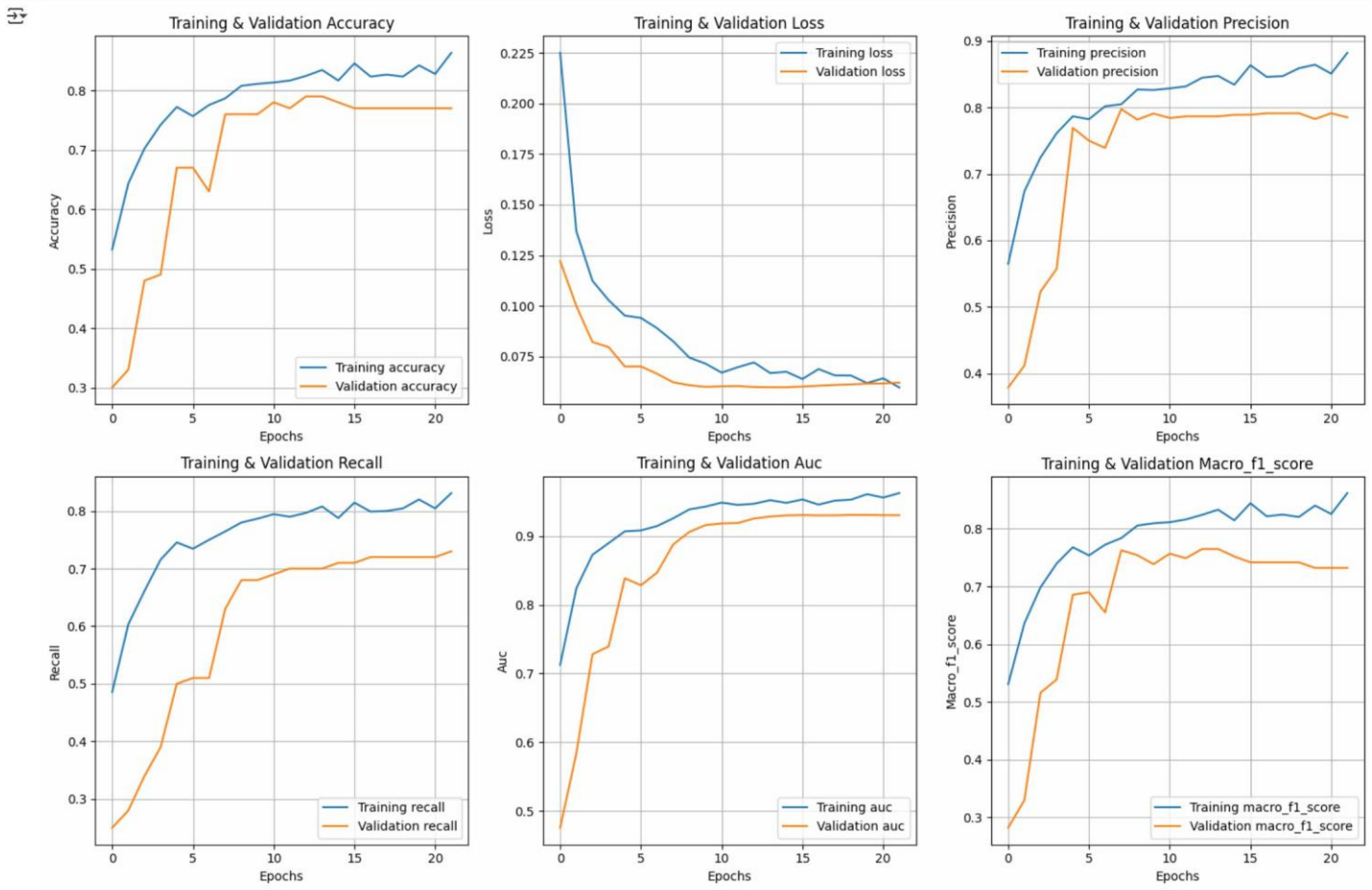
```
import tensorflow as tf
from tensorflow.keras.applications import DenseNet121
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import Precision, Recall, AUC
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping, LearningRateScheduler
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.regularizers import l2
import numpy as np

# --- Custom Macro F1 Score Metric ---
class MacroF1Score(tf.keras.metrics.Metric):
    def __init__(self, num_classes, name="macro_f1_score", **kwargs):
        super(MacroF1Score, self).__init__(name=name, **kwargs)
        self.num_classes = num_classes
        self.true_positives = self.add_weight(name="true_positives", shape=(num_classes,), initializer="zeros")
        self.false_positives = self.add_weight(name="false_positives", shape=(num_classes,), initializer="zeros")
        self.false_negatives = self.add_weight(name="false_negatives", shape=(num_classes,), initializer="zeros")

    def update_state(self, y_true, y_pred, sample_weight=None):
        y_true = tf.argmax(y_true, axis=-1)
        y_pred = tf.argmax(y_pred, axis=-1)

        for class_id in range(self.num_classes):
            true_pos = tf.reduce_sum(tf.cast((y_true == class_id) & (y_pred == class_id), tf.float32))
            false_pos = tf.reduce_sum(tf.cast((y_true != class_id) & (y_pred == class_id), tf.float32))
            false_neg = tf.reduce_sum(tf.cast((y_true == class_id) & (y_pred != class_id), tf.float32))

            # Update values: using one_hot vector to update the corresponding index
            self.true_positives.assign_add(tf.one_hot(class_id, self.num_classes) * true_pos)
            self.false_positives.assign_add(tf.one_hot(class_id, self.num_classes) * false_pos)
            self.false_negatives.assign_add(tf.one_hot(class_id, self.num_classes) * false_neg)

    def result(self):
        precision = self.true_positives / (self.true_positives + self.false_positives + tf.keras.backend.epsilon())
        recall = self.true_positives / (self.true_positives + self.false_negatives + tf.keras.backend.epsilon())
        f1_scores = 2 * (precision * recall) / (precision + recall + tf.keras.backend.epsilon())
        return tf.reduce_mean(f1_scores)

    def reset_states(self):
        self.true_positives.assign(tf.zeros_like(self.true_positives))
        self.false_positives.assign(tf.zeros_like(self.false_positives))
        self.false_negatives.assign(tf.zeros_like(self.false_negatives))

# --- Focal Loss Implementation with per-class alpha ---
def focal_loss(alpha, gamma=2.0):
    """
    Focal Loss with per-class alpha.
    Parameters:
      alpha: A list or tensor of shape (num_classes,) with alpha values per class.
      gamma: Focusing parameter.
    Returns:
      A loss function.
    """
    alpha = tf.constant(alpha, dtype=tf.float32)

    def loss_fn(y_true, y_pred):
        epsilon = tf.keras.backend.epsilon()
        y_pred = tf.clip_by_value(y_pred, epsilon, 1.0 - epsilon)  # Prevent log(0)
        cross_entropy = -y_true * tf.math.log(y_pred)
        # weight has shape (batch_size, num_classes) due to broadcasting.
        weight = alpha * tf.pow(1 - y_pred, gamma)
        loss = weight * cross_entropy
        return tf.reduce_mean(loss)

    return loss_fn

# --- Paths ---
train_dir = './resized_clahe_images/train_augmented_clahe'
val_dir = './resized_clahe_images/val_clahe'
test_dir = './resized_clahe_images/test_clahe'

IMG_SIZE = (224, 224)

# --- Hyperparameters ---
BATCH_SIZE = 32
BASE_LR = 0.00033609  # Initial Learning Rate
DROPOUT_RATE_1 = 0.3
DROPOUT_RATE_2 = 0.5
DENSE_UNITS = 384
EPOCHS = 30

# --- Class Weights (Glaucoma is harder to classify) ---
class_weights = {
    0: 1.0,  # Normal
    1: 1.5,  # Cataract
```

```
    2: 2.0   # Glaucoma
}

# --- Build Model ---
def build_model():
    base_model = DenseNet121(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
    base_model.trainable = False  # No fine-tuning yet

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = BatchNormalization()(x)
    x = Dropout(DROPOUT_RATE_1)(x)
    x = Dense(DENSE_UNITS, activation='relu', kernel_regularizer=l2(1e-4))(x)
    x = Dropout(DROPOUT_RATE_2)(x)
    output_layer = Dense(3, activation='softmax')(x)

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

    # Compile with Focal Loss and Macro F1 Score.
    # Set per-class alpha values: [0.20, 0.35, 0.65]
    macro_f1 = MacroF1Score(num_classes=3)
    model.compile(
        optimizer=Adam(learning_rate=BASE_LR),
        loss=focal_loss(alpha=[0.20, 0.35, 0.65], gamma=2.0),
        metrics=['accuracy',
                 Precision(name='precision'),
                 Recall(name='recall'),
                 AUC(name='auc'),
                 macro_f1]
    )
    return model

# --- Data Generators ---
train_datagen = ImageDataGenerator(rescale=1.0 / 255)
val_test_datagen = ImageDataGenerator(rescale=1.0 / 255)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

validation_generator = val_test_datagen.flow_from_directory(
    val_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# --- Initialize the Model ---
model = build_model()

# --- Callbacks ---
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=7,
    restore_best_weights=True
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=4,
    min_lr=1e-6,
    verbose=1,
    min_delta=1e-3
)

# Learning Rate Scheduler: Reduce LR every 5 epochs after 10 epochs
def lr_schedule(epoch, lr):
    if epoch >= 10 and epoch % 5 == 0:
        return lr / 4
    return lr

lr_scheduler = LearningRateScheduler(lr_schedule, verbose=1)

# --- Train the Model (Without Fine-Tuning) ---
print("   Training the model with **frozen base layers** (No fine-tuning)...")
history = model.fit(
    train_generator,
    validation_data=validation_generator,
    epochs=EPOCHS,
    class_weight=class_weights,  # Apply class weights
    callbacks=[reduce_lr, early_stopping, lr_scheduler]
)

# --- Evaluate on Test Data ---
test_generator = val_test_datagen.flow_from_directory(
    test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=False
)

test_loss, test_accuracy, test_precision, test_recall, test_auc, test_macro_f1 = model.evaluate(test_generator)
print(f"Test Results: Accuracy={test_accuracy:.2f}, Precision={test_precision:.2f}, Recall={test_recall:.2f}, "
      f"Loss={test_loss:.2f}, AUC={test_auc:.2f}, Macro F1 Score={test_macro_f1:.2f}")
```

Training and Validation Metrics Over Epochs

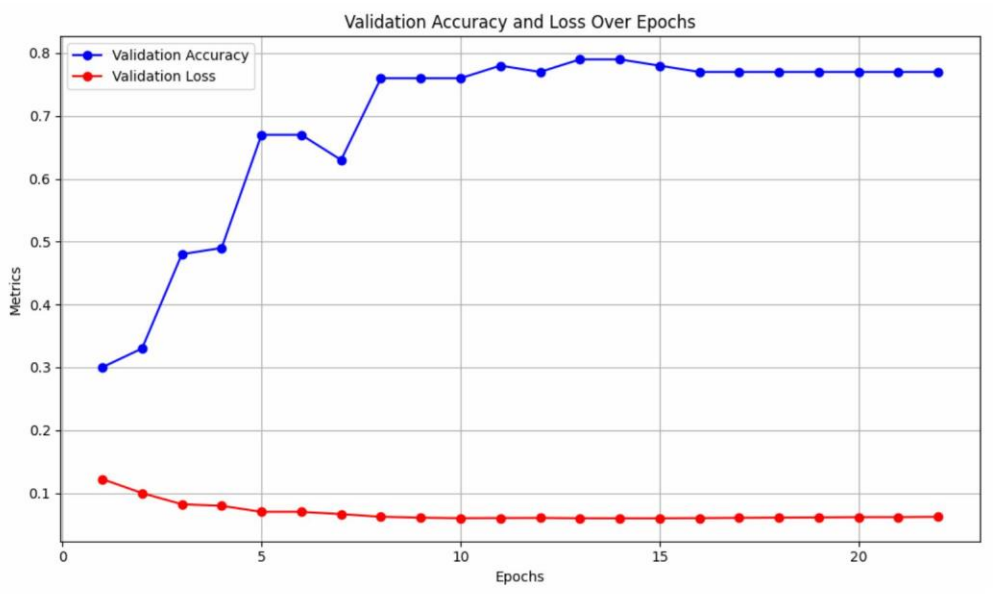
```
import numpy as np
import matplotlib.pyplot as plt

# Function to plot training history
def plot_training_history(history):
```

```
    metrics = ['accuracy', 'loss', 'precision', 'recall', 'auc', 'macro_f1_score']

    available_metrics = [m for m in metrics if m in history.history]

    num_metrics = len(available_metrics)
    rows = (num_metrics // 3) + (num_metrics % 3 > 0)

    plt.figure(figsize=(15, 5 * rows))

    for i, metric in enumerate(available_metrics):
        plt.subplot(rows, 3, i + 1)
        plt.plot(history.history[metric], label=f'Training {metric}')
        plt.plot(history.history[f'val_{metric}'], label=f'Validation {metric}')
        plt.xlabel('Epochs')
        plt.ylabel(metric.capitalize())
        plt.title(f'Training & Validation {metric.capitalize()}')
        plt.legend()
        plt.grid()

    plt.tight_layout()
    plt.show()

# Call the function
plot_training_history(history)
```



Validation Accuracy & Loss over Epochs

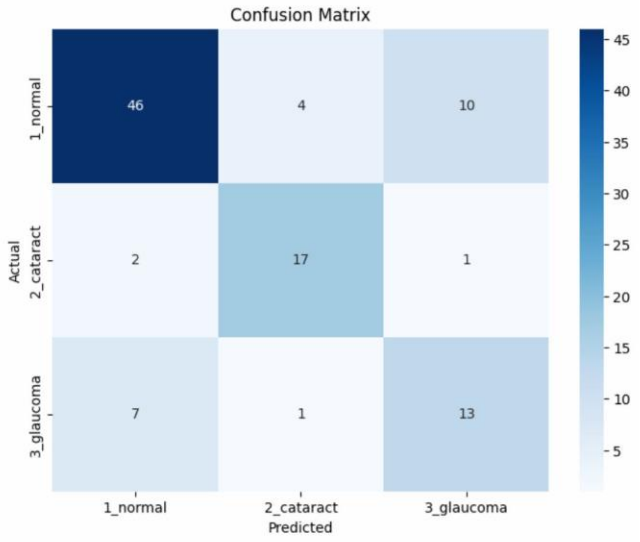
```
import matplotlib.pyplot as plt

def plot_validation_metrics(history):
    epochs = range(1, len(history.history['val_accuracy']) + 1)

    plt.figure(figsize=(10, 6))

    plt.plot(epochs, history.history['val_accuracy'], label='Validation Accuracy', color='blue', marker='o')
    plt.plot(epochs, history.history['val_loss'], label='Validation Loss', color='red', marker='o')

    plt.xlabel('Epochs')
    plt.ylabel('Metrics')
    plt.title('Validation Accuracy and Loss Over Epochs')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Call the function
plot_validation_metrics(history)
```



Confusion Matrix & Classification Report

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

def plot_confusion_matrix(model, test_generator):
    y_true = test_generator.classes
    y_pred_probs = model.predict(test_generator)
    y_pred_classes = np.argmax(y_pred_probs, axis=1)

    cm = confusion_matrix(y_true, y_pred_classes)
    labels = list(test_generator.class_indices.keys())

    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')
    plt.show()

    print("\n    Classification Report:\n")
    print(classification_report(y_true, y_pred_classes, target_names=labels))

# Call the function
plot_confusion_matrix(model, test_generator)
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 44s 6s/step
```

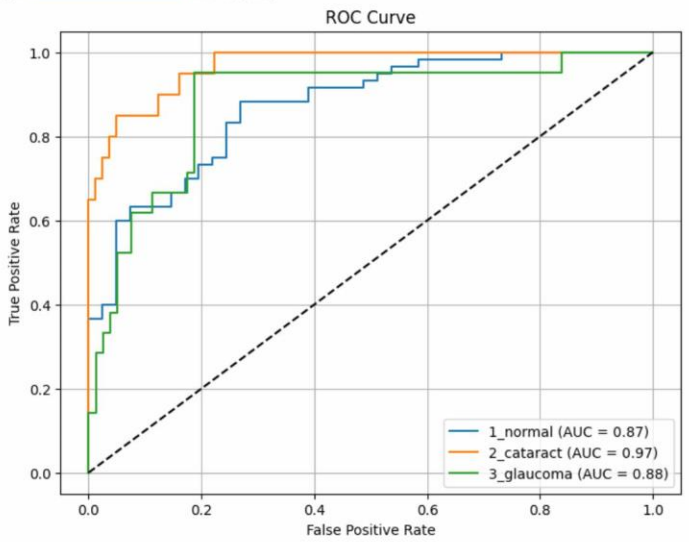



```
    Classification Report:

                  precision    recall  f1-score   support

       1_normal       0.84      0.77      0.80        60
     2_cataract       0.77      0.85      0.81        20
    3_glaucoma        0.54      0.62      0.58        21

       accuracy                           0.75       101
      macro avg       0.72      0.75      0.73       101
   weighted avg       0.76      0.75      0.76       101
```

ROC curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
import tensorflow as tf

def plot_roc_curve(model, test_generator):
    num_classes = len(test_generator.class_indices)
    y_true = tf.keras.utils.to_categorical(test_generator.classes, num_classes=num_classes)
    y_pred_probs = model.predict(test_generator, verbose=1)

    plt.figure(figsize=(8, 6))
    for i, class_label in enumerate(test_generator.class_indices.keys()):
        fpr, tpr, _ = roc_curve(y_true[:, i], y_pred_probs[:, i])
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'{class_label} (AUC = {roc_auc:.2f})')

    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc='lower right')
    plt.grid()
    plt.show()

# Call the function
plot_roc_curve(model, test_generator)
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 18s 4s/step
```



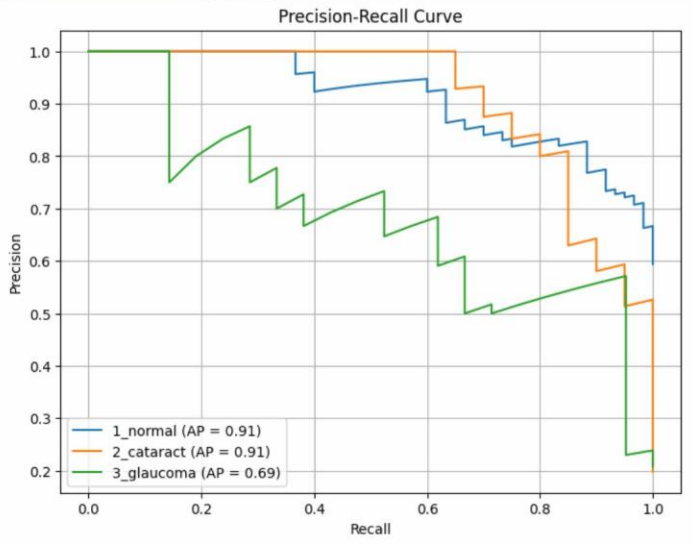
Precision-Recall curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, average_precision_score
import tensorflow as tf

def plot_precision_recall_curve(model, test_generator):
    num_classes = len(test_generator.class_indices)
    y_true = tf.keras.utils.to_categorical(test_generator.classes, num_classes=num_classes)
    y_pred_probs = model.predict(test_generator, verbose=1)

    plt.figure(figsize=(8, 6))
    for i, class_label in enumerate(test_generator.class_indices.keys()):
        precision, recall, _ = precision_recall_curve(y_true[:, i], y_pred_probs[:, i])
        ap = average_precision_score(y_true[:, i], y_pred_probs[:, i])
        plt.plot(recall, precision, label=f'{class_label} (AP = {ap:.2f})')

    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend(loc='lower left')
    plt.grid()
    plt.show()

# Call the function
plot_precision_recall_curve(model, test_generator)
```

Precision-Recall curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, average_precision_score
import tensorflow as tf

def plot_precision_recall_curve(model, test_generator):
    num_classes = len(test_generator.class_indices)
    y_true = tf.keras.utils.to_categorical(test_generator.classes, num_classes=num_classes)
    y_pred_probs = model.predict(test_generator, verbose=1)

    plt.figure(figsize=(8, 6))
    for i, class_label in enumerate(test_generator.class_indices.keys()):
        precision, recall, _ = precision_recall_curve(y_true[:, i], y_pred_probs[:, i])
        ap = average_precision_score(y_true[:, i], y_pred_probs[:, i])
        plt.plot(recall, precision, label=f'{class_label} (AP = {ap:.2f})')

    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend(loc='lower left')
    plt.grid()
    plt.show()

# Call the function
plot_precision_recall_curve(model, test_generator)
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 17s 4s/step
```



Experiment 7

Balanced Dataset Testing with EfficientNetB0 (CLAHE filtered)

Fine-tuning to the last 110 layers of the base model (focal loss)

Training Strategy

- **New Backbone Architecture:** Instead of DenseNet121, this experiment uses **EfficientNetB0**, which is lighter and more efficient.
- **Unfreezing More Layers:** Instead of unfreezing only a small part of the base model, this experiment unfreezes the last 110 layers for fine-tuning.
- **Focal Loss used Instead of Categorical Crossentropy:** custom alpha values along with class weights are used
- **New Batch Size Strategy:**
  - Phase 1 (Feature Extraction - Initial Training): Batch Size = 64. Helps stabilize training during the frozen feature extraction phase.
  - Phase 2 (Fine-Tuning): Batch Size = 32. A smaller batch size (32) improves fine-tuning stability, reducing noise in gradient updates.
- **Two-Stage Learning Rate Adjustment:**
  - **Warm-Up Phase:** First 20 epochs exponentially increase the learning rate. Prevents premature convergence of the pretrained weights.
  - **Stabilization Phase:** After warm-up, the learning rate fixes at 5e-6 for fine-tuning in a controlled manner.

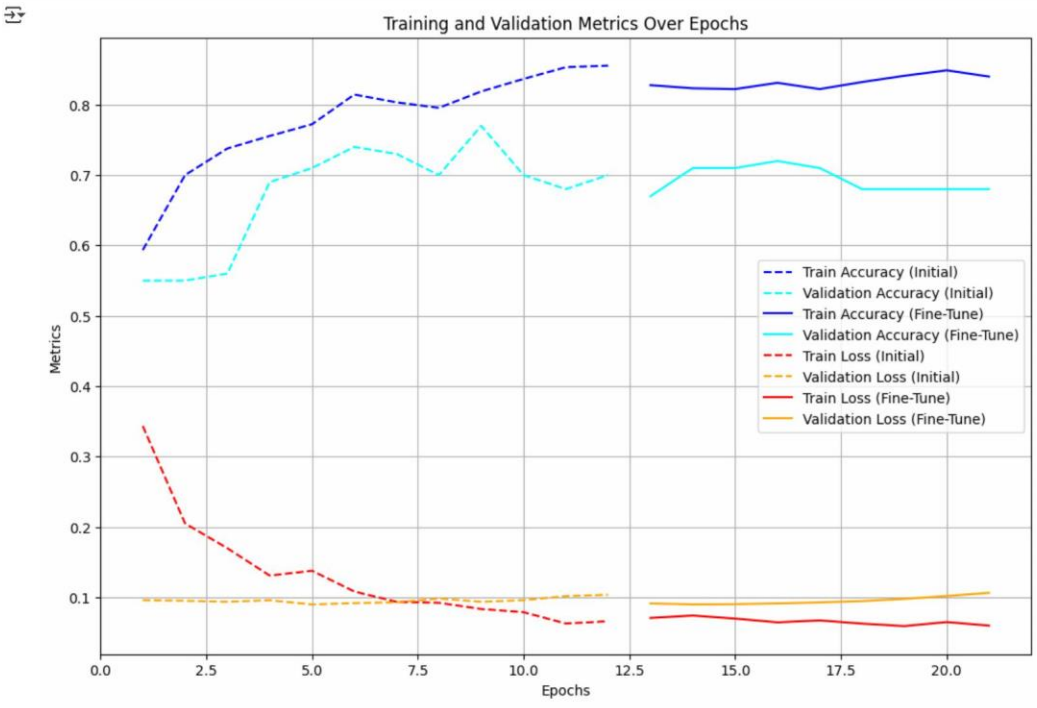
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, BatchNormalization
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping, LearningRateScheduler
from tensorflow.keras.metrics import Precision, Recall, AUC
import numpy as np

# Define dataset paths
train_dir = './resized_clahe_images/train_augmented_clahe'
val_dir = './resized_clahe_images/val_clahe'
test_dir = './resized_clahe_images/test_clahe'

# Define image size and batch sizes
IMG_SIZE = (224, 224)
```

```
BATCH_SIZE_TRAIN = 64  # Batch size for initial training
BATCH_SIZE_FINE_TUNE = 32  # Batch size for fine-tuning

# Add class weights
class_weights = {0: 1.0, 1: 1.5, 2: 2.0}  # Normal, Cataract, Glaucoma

# Define alpha values for Focal Loss (adjust class importance within the loss)
alpha = [0.20, 0.35, 0.65]  # Customize these values based on your dataset

# Define Focal Loss with custom alpha
def focal_loss(alpha, gamma=2.0):
    alpha = tf.constant(alpha, dtype=tf.float32)
    def loss_fn(y_true, y_pred):
        y_pred = tf.clip_by_value(y_pred, 1e-7, 1.0 - 1e-7)
        ce_loss = -y_true * tf.math.log(y_pred)
        focal_loss = alpha * tf.math.pow(1 - y_pred, gamma) * ce_loss
        return tf.reduce_sum(focal_loss, axis=-1)
    return loss_fn

# Load dataset (Initial Training)
AUTOTUNE = tf.data.AUTOTUNE
train_dataset = image_dataset_from_directory(train_dir, image_size=IMG_SIZE, batch_size=BATCH_SIZE_TRAIN, label_mode="categorical", shuffle=True).prefetch(buffer_size=AUTOTUNE)
val_dataset = image_dataset_from_directory(val_dir, image_size=IMG_SIZE, batch_size=BATCH_SIZE_TRAIN, label_mode="categorical").prefetch(buffer_size=AUTOTUNE)
test_dataset = image_dataset_from_directory(test_dir, image_size=IMG_SIZE, batch_size=BATCH_SIZE_TRAIN, label_mode="categorical").prefetch(buffer_size=AUTOTUNE)

# Define hyperparameters for the custom layer
DENSE_UNITS = 512
DROPOUT_RATE_1 = 0.2
DROPOUT_RATE_2 = 0.4

# Load EfficientNet model (transfer learning)
base_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False  # Freeze the base model initially

# Build model (Initial Training)
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = BatchNormalization()(x)  # Normalizing before the dense layer
x = Dropout(DROPOUT_RATE_1)(x)
x = Dense(DENSE_UNITS, activation='relu', name='dense')(x)
x = Dropout(DROPOUT_RATE_2)(x)
output = Dense(3, activation='softmax', name='dense_1')(x)

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

# Compile model with Focal Loss
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.00033609),
              loss=focal_loss(alpha=alpha),
              metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc')])
# Define callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=7, restore_best_weights=True)
reduce_lr = keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, verbose=1)

# Train model (Initial Training with Frozen Base)
history = model.fit(train_dataset, validation_data=val_dataset, epochs=30, class_weight=class_weights, callbacks=[early_stopping, reduce_lr])

# ------------------- Fine-Tuning ------------------- #
# **Unfreeze the last 110 layers**
for layer in base_model.layers[-110:]:
    layer.trainable = True

# **Freeze BatchNorm Layers**
for layer in model.layers:
    if isinstance(layer, BatchNormalization):
        layer.trainable = False

# ✅ Extract trained weights from the existing head
dense_weights = model.get_layer('dense').get_weights()
output_weights = model.get_layer('dense_1').get_weights()

# ✅ Rebuild the classification head without Dropout
inputs = base_model.input
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = BatchNormalization()(x)
x = Dense(DENSE_UNITS, activation='relu', name='dense')(x)
x = Dense(3, activation='softmax', name='dense_1')(x)

fine_tune_model = Model(inputs=inputs, outputs=x)

# ✅ Transfer the trained weights
fine_tune_model.get_layer('dense').set_weights(dense_weights)
fine_tune_model.get_layer('dense_1').set_weights(output_weights)

# **New Learning Rate Schedule: Warm-up then Stabilize at 5e-6**
warmup_epochs = 20  # Warm-up phase
total_epochs = 50  # Fine-tuning phase

def lr_schedule(epoch):
    if epoch < warmup_epochs:
        return 1e-7 * (10 ** (epoch / warmup_epochs))  # Exponential increase during warm-up
    return 5e-6  # Fixate learning rate at 5e-6 after warm-up
lr_callback = LearningRateScheduler(lr_schedule)

# **Recompile fine-tune model**
optimizer = keras.optimizers.Adam(learning_rate=1e-7, clipnorm=1.0)  # Reset optimizer state
fine_tune_model.compile(
    optimizer=optimizer,
    loss=focal_loss(alpha=alpha),
    metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc')])

# **Load Dataset for Fine-Tuning (with 32 Batch Size)**
train_dataset_finetune = image_dataset_from_directory(train_dir, image_size=IMG_SIZE,
                                                      batch_size=BATCH_SIZE_FINE_TUNE, label_mode="categorical", shuffle=True).prefetch(buffer_size=AUTOTUNE)
val_dataset_finetune = image_dataset_from_directory(val_dir, image_size=IMG_SIZE,
                                                    batch_size=BATCH_SIZE_FINE_TUNE, label_mode="categorical").prefetch(buffer_size=AUTOTUNE)
# **Fine-Tune the Model**
fine_tune_history = fine_tune_model.fit(train_dataset_finetune, validation_data=val_dataset_finetune,
                                        epochs=total_epochs, class_weight=class_weights, callbacks=[early_stopping, lr_callback])
# Evaluate the fine-tuned model
```

```
# Evaluate the fine-tuned model
results = fine_tune_model.evaluate(test_dataset)
print("Test Results:", dict(zip(fine_tune_model.metrics_names, results)))
```

Training and Validation Accuracy/Loss over Epochs

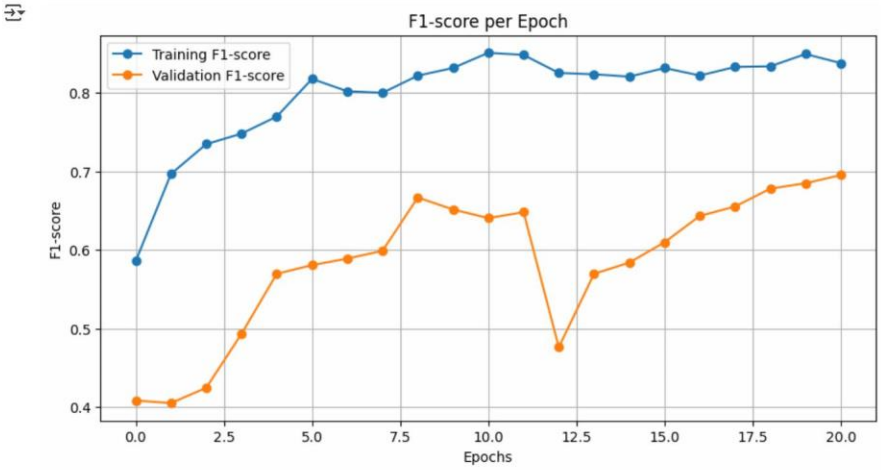
```
def plot_combined_metrics(initial_history, fine_tune_history=None):
    """
    Plots training and validation accuracy and loss in the same plot.
    """
    epochs_initial = range(1, len(initial_history.history['accuracy']) + 1)
    epochs_fine_tune = (
        range(len(epochs_initial) + 1, len(epochs_initial) + 1 + len(fine_tune_history.history['accuracy']))
        if fine_tune_history
        else []
    )

    plt.figure(figsize=(12, 8))

    # Plot accuracy
    plt.plot(epochs_initial, initial_history.history['accuracy'], label='Train Accuracy (Initial)', linestyle='--', color='blue')
    plt.plot(epochs_initial, initial_history.history['val_accuracy'], label='Validation Accuracy (Initial)', linestyle='--', color='cyan')

    if fine_tune_history:
        plt.plot(epochs_fine_tune, fine_tune_history.history['accuracy'], label='Train Accuracy (Fine-Tune)', color='blue')
        plt.plot(epochs_fine_tune, fine_tune_history.history['val_accuracy'], label='Validation Accuracy (Fine-Tune)', color='cyan')

    # Plot loss
    plt.plot(epochs_initial, initial_history.history['loss'], label='Train Loss (Initial)', linestyle='--', color='red')
    plt.plot(epochs_initial, initial_history.history['val_loss'], label='Validation Loss (Initial)', linestyle='--', color='orange')

    if fine_tune_history:
        plt.plot(epochs_fine_tune, fine_tune_history.history['loss'], label='Train Loss (Fine-Tune)', color='red')
        plt.plot(epochs_fine_tune, fine_tune_history.history['val_loss'], label='Validation Loss (Fine-Tune)', color='orange')

    # Add labels, legend, and title
    plt.title('Training and Validation Metrics Over Epochs')
    plt.xlabel('Epochs')
    plt.ylabel('Metrics')
    plt.legend()
    plt.grid(True)
    plt.show()


# Call the function with initial training history and fine-tune history
plot_combined_metrics(history, fine_tune_history)
```



Training and Validation F1-score over Epochs

```
import numpy as np
import matplotlib.pyplot as plt

# Function to compute F1-score
def compute_f1(precision, recall):
    epsilon = 1e-7  # Small value to prevent division by zero
    return (2 * precision * recall) / (precision + recall + epsilon)

# Function to plot F1-score per epoch
def plot_f1_score(history, fine_tune_history=None):
    # Extract precision and recall from history
    precision = np.array(history.history['precision'])
    recall = np.array(history.history['recall'])
    val_precision = np.array(history.history['val_precision'])
    val_recall = np.array(history.history['val_recall'])

    # Compute F1-score
    f1_train = compute_f1(precision, recall)
    f1_val = compute_f1(val_precision, val_recall)
```

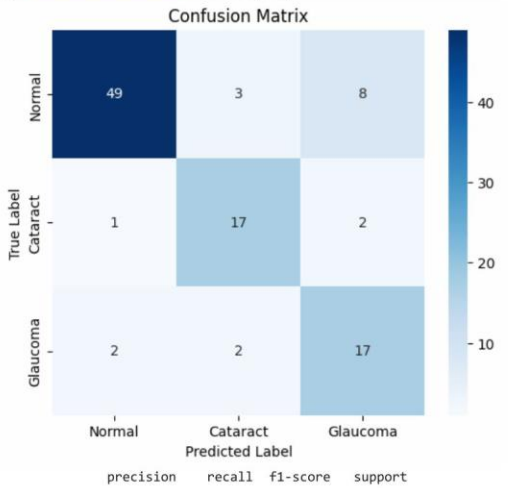
```
    # If fine-tuning history is provided, extend the plots
    if fine_tune_history:
        fine_precision = np.array(fine_tune_history.history['precision'])
        fine_recall = np.array(fine_tune_history.history['recall'])
        fine_val_precision = np.array(fine_tune_history.history['val_precision'])
        fine_val_recall = np.array(fine_tune_history.history['val_recall'])

        f1_train_fine = compute_f1(fine_precision, fine_recall)
        f1_val_fine = compute_f1(fine_val_precision, fine_val_recall)

        f1_train = np.concatenate([f1_train, f1_train_fine])
        f1_val = np.concatenate([f1_val, f1_val_fine])

    # Plot the F1-score
    plt.figure(figsize=(10, 5))
    plt.plot(f1_train, label='Training F1-score', marker='o')
    plt.plot(f1_val, label='Validation F1-score', marker='o')
    plt.xlabel('Epochs')
    plt.ylabel('F1-score')
    plt.title('F1-score per Epoch')
    plt.legend()
    plt.grid(True)
    plt.show()

# Call the function to plot F1-score
plot_f1_score(history, fine_tune_history)
```



Confusion Matrix & Classification Report

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

# Get the true labels and predictions
y_true = []
y_pred = []

# Loop through the test dataset and collect labels & predictions
for images, labels in test_dataset:
    # True labels
    y_true.extend(np.argmax(labels.numpy(), axis=1))

    # Predictions
    preds = model.predict(images)
    y_pred.extend(np.argmax(preds, axis=1))

# Convert lists to numpy arrays
y_true = np.array(y_true)
y_pred = np.array(y_pred)

# Compute the confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Class names
class_names = ["Normal", "Cataract", "Glaucoma"]

# Plot the confusion matrix
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# Print Classification Report
print(classification_report(y_true, y_pred, target_names=class_names))
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 9s 4s/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 4s 423ms/step
```

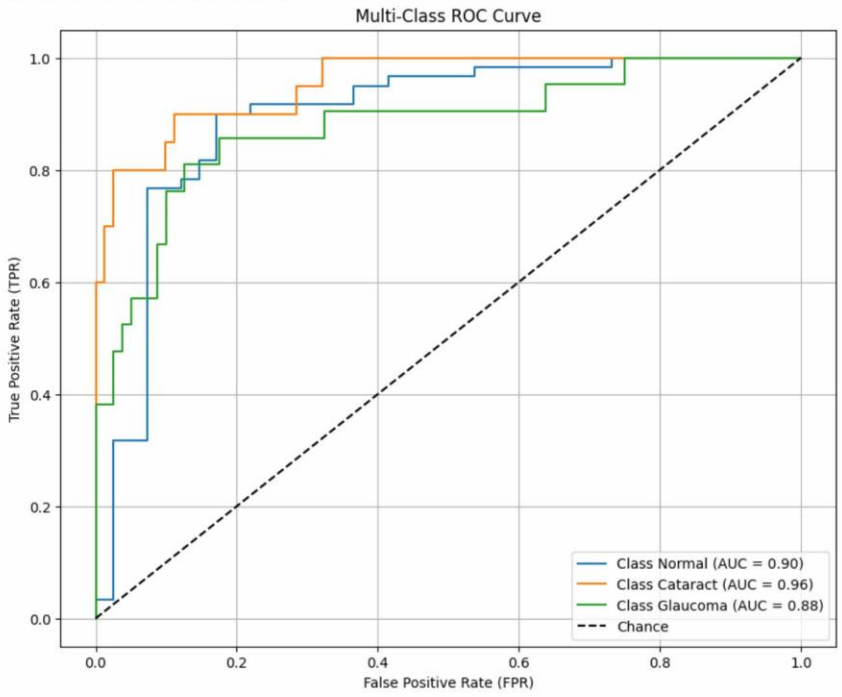


	precision	recall	f1-score	support
Normal	0.94	0.82	0.88	60
Cataract	0.77	0.85	0.81	20
Glaucoma	0.63	0.81	0.71	21
accuracy			0.82	101
macro avg	0.78	0.83	0.80	101
weighted avg	0.84	0.82	0.83	101

ROC curve

```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt
import numpy as np

def plot_roc_curve(model, test_dataset, class_names):
    # Extract true labels and predictions
    y_true = []
    y_pred = []

    for images, labels in test_dataset:
        y_true.extend(labels.numpy())
        y_pred.extend(model.predict(images))

    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    # Binarize the true labels
    y_true_binarized = label_binarize(np.argmax(y_true, axis=1), classes=range(len(class_names)))

    # Plot ROC curve for each class
    plt.figure(figsize=(10, 8))
    for i, class_name in enumerate(class_names):
        fpr, tpr, _ = roc_curve(y_true_binarized[:, i], y_pred[:, i])
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'Class {class_name} (AUC = {roc_auc:.2f})')

    # Plot the diagonal line
    plt.plot([0, 1], [0, 1], 'k--', label='Chance')

    # Add labels, title, and legend
    plt.title('Multi-Class ROC Curve')
    plt.xlabel('False Positive Rate (FPR)')
    plt.ylabel('True Positive Rate (TPR)')
    plt.legend(loc='lower right')
    plt.grid(True)
    plt.show()

# Class names in your dataset
class_names = ['Normal', 'Cataract', 'Glaucoma']

# Generate the ROC plot
plot_roc_curve(model, test_dataset, class_names)
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 9s 5s/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 5s 452ms/step
```



Test Predictions Visualization

```
import random
import matplotlib.pyplot as plt

# Get the class names
class_labels = ['Normal', 'Cataract', 'Glaucoma']  # Replace with your actual class names

# Make predictions on the test dataset
y_true = []
y_pred = []
all_images = []

for images, labels in test_dataset:
    y_true.extend(labels.numpy())
    y_pred.extend(model.predict(images))
    all_images.extend(images.numpy())  # Collect all images for visualization

y_true = np.array(y_true)
y_pred = np.array(y_pred)

# Get predicted class labels
predicted_labels = np.argmax(y_pred, axis=1)
true_labels = np.argmax(y_true, axis=1)

# Randomly sample 20 test images
sample_indices = random.sample(range(len(true_labels)), 20)

plt.figure(figsize=(20, 15))
for i, idx in enumerate(sample_indices):
    # Get the image, true label, and predicted label
    img = all_images[idx].astype('uint8')
    true_label = class_labels[true_labels[idx]]
    pred_label = class_labels[predicted_labels[idx]]

    # Display the image with title
    plt.subplot(4, 5, i + 1)  # Adjusted for a 4x5 grid to display 20 images
    plt.imshow(img)
    plt.axis('off')
    plt.title(f'True: {true_label}\nPred: {pred_label}', fontsize=10)

plt.tight_layout()
plt.show()
```

Experiment 8

Balanced Dataset Testing with EfficientNetB0 (CLAHE filtered)

Fine-tuning to the last 50 layers of the base model (loss = categorical crossentropy)

Training Strategy

- Only the **last 50 layers** of EfficientNetB0 are **unfrozen** during fine-tuning.
- **Lower Batch Size for Fine-Tuning:** Batch size is reduced from 32 → 16 during fine-tuning.
- A lower stable **learning rate of 1e-6** is used during fine-tuning.
- **Loss Function: categorical crossentropy**

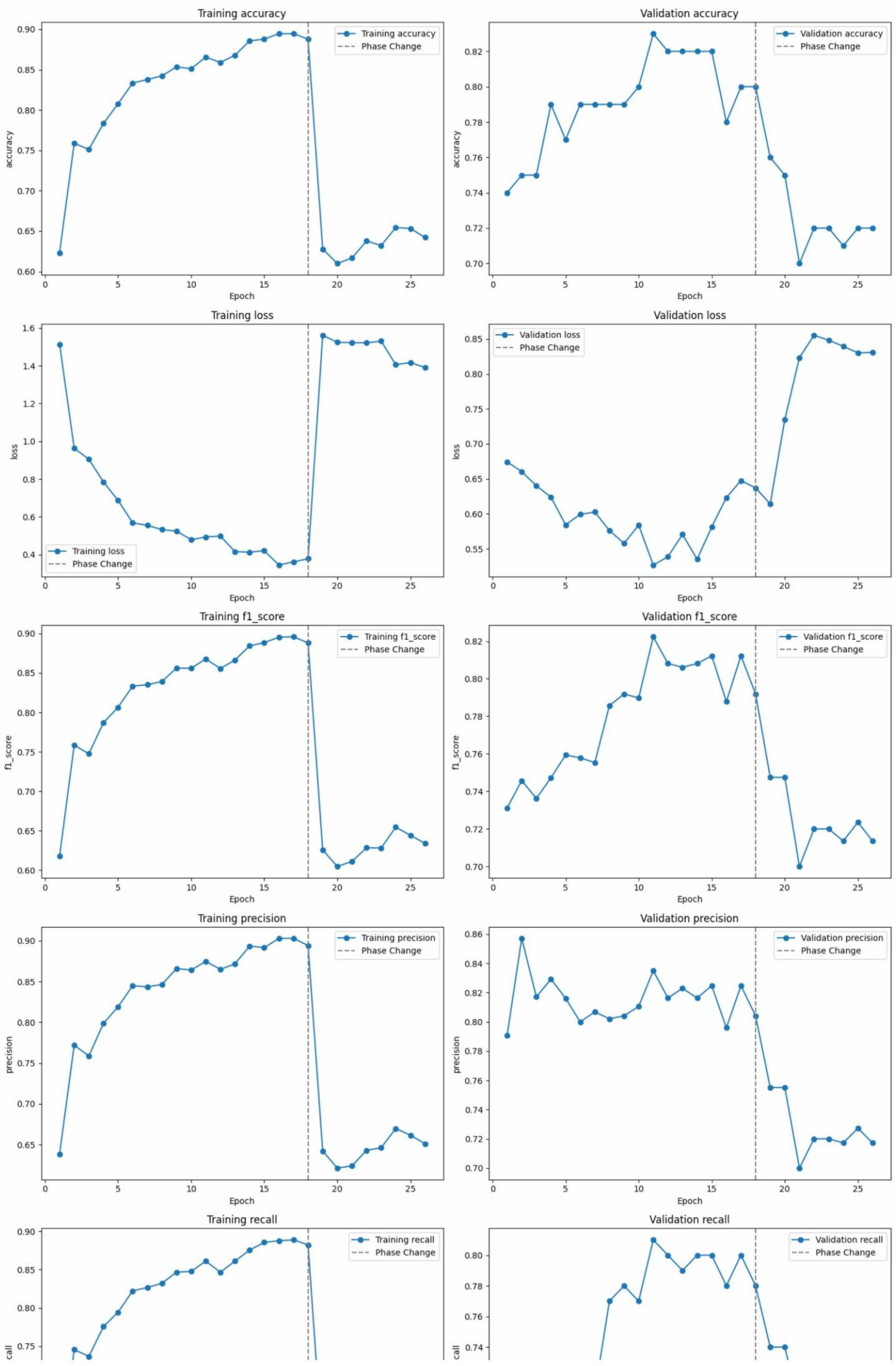
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout, BatchNormalization
```

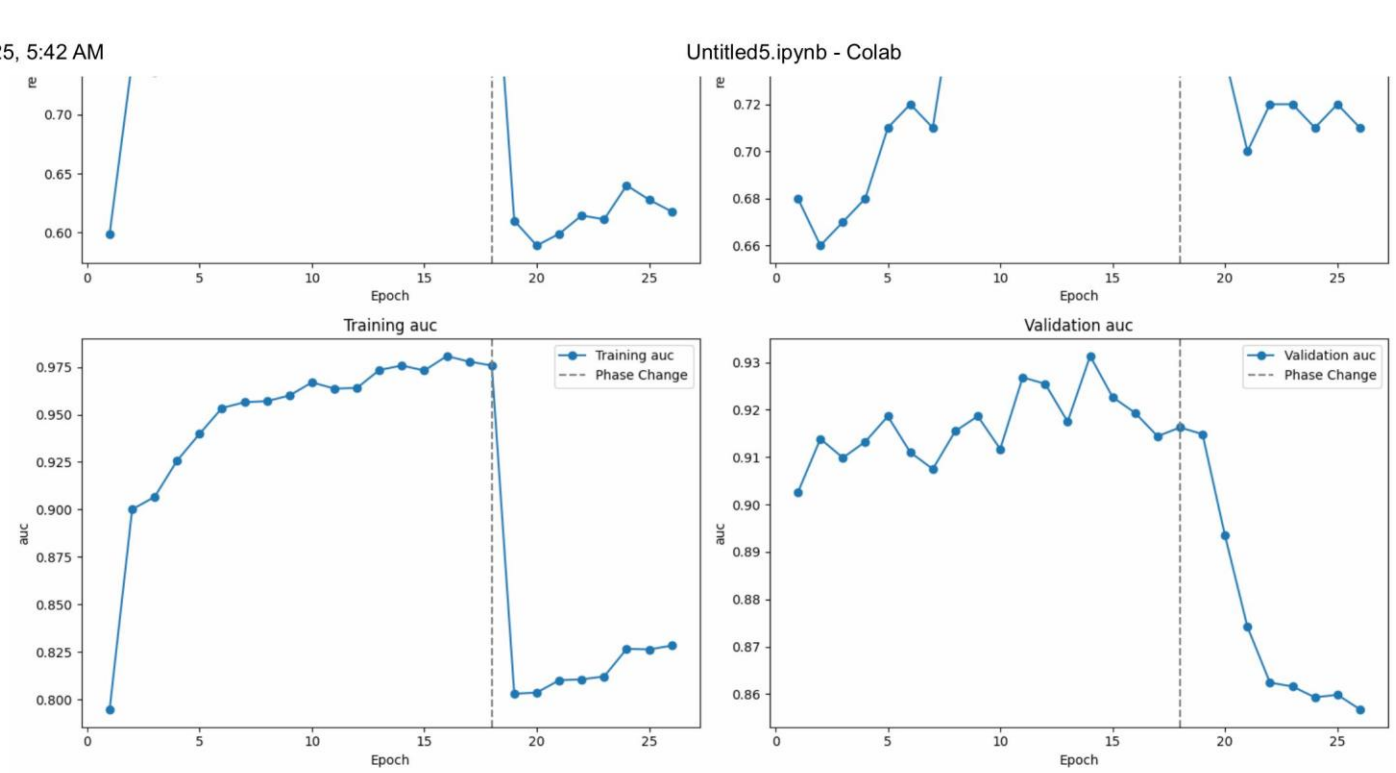
```
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.metrics import Precision, Recall, AUC

# --- Custom F1 Score Metric ---
class F1Score(tf.keras.metrics.Metric):
    def __init__(self, name="f1_score", **kwargs):
        super(F1Score, self).__init__(name=name, **kwargs)
        self.precision = tf.keras.metrics.Precision()
        self.recall = tf.keras.metrics.Recall()

    def update_state(self, y_true, y_pred, sample_weight=None):
        self.precision.update_state(y_true, y_pred, sample_weight)
        self.recall.update_state(y_true, y_pred, sample_weight)

    def result(self):
        precision = self.precision.result()
        recall = self.recall.result()
        return 2 * ((precision * recall) / (precision + recall + tf.keras.backend.epsilon()))

    def reset_states(self):
        self.precision.reset_states()
        self.recall.reset_states()

# Define dataset paths
train_dir = './resized_clahe_images/train_augmented_clahe'
val_dir   = './resized_clahe_images/val_clahe'
test_dir  = './resized_clahe_images/test_clahe'

# Define image size and batch size
IMG_SIZE = (224, 224)
BATCH_SIZE = 32  # Initial batch size

# Add class weights (Normal, Cataract, Glaucoma)
class_weights = {0: 1.0, 1: 1.5, 2: 2.0}

# Load train & validation datasets
train_dataset = image_dataset_from_directory(
    train_dir,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode="categorical"
)

val_dataset = image_dataset_from_directory(
    val_dir,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode="categorical"
)

# --- Load test dataset with shuffle set to False ---
test_dataset = image_dataset_from_directory(
    test_dir,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    label_mode="categorical",
    shuffle=False  # Ensures consistent ordering for evaluation
)

# Save test class names for later use (e.g., in confusion matrix)
test_class_names = test_dataset.class_names

# Optimize dataset loading with prefetch
AUTOTUNE = tf.data.AUTOTUNE
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
val_dataset   = val_dataset.prefetch(buffer_size=AUTOTUNE)
test_dataset  = test_dataset.prefetch(buffer_size=AUTOTUNE)

# Define hyperparameters for the custom layer
DROPOUT_RATE_1 = 0.3
DROPOUT_RATE_2 = 0.5
DENSE_UNITS    = 512

# Load EfficientNet model (transfer learning)
base_model = EfficientNetB0(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Freeze the base model initially
base_model.trainable = False

# Build model with custom top layers
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = BatchNormalization()(x)  # Normalizing before the dense layer
x = Dropout(DROPOUT_RATE_1)(x)
x = Dense(DENSE_UNITS, activation='relu')(x)
x = Dropout(DROPOUT_RATE_2)(x)
output = Dense(3, activation='softmax')(x)

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

# Compile model for initial training
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.00033609),
    loss='categorical_crossentropy',
    metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1_score')]
)

# Define callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=7, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, verbose=1)

# Train model (Initial Training with Frozen Base)
history_initial = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=30,
    class_weight=class_weights,
    callbacks=[early_stopping, reduce_lr]
)
```

```
# ------------------- Fine-Tuning ------------------- #

# Unfreeze the last 50 layers of the base model
for layer in base_model.layers[:-50]:
    layer.trainable = False
for layer in base_model.layers[-50:]:
    layer.trainable = True

# Recompile with a lower learning rate for fine-tuning
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-6),
    loss='categorical_crossentropy',
    metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1_score')]
)

# Reduce batch size for fine-tuning for more precise updates
BATCH_SIZE_FINE_TUNE = 16
train_dataset = image_dataset_from_directory(
    train_dir,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE_FINE_TUNE,
    label_mode="categorical"
)
val_dataset = image_dataset_from_directory(
    val_dir,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE_FINE_TUNE,
    label_mode="categorical"
)

# Optimize dataset loading again
train_dataset = train_dataset.prefetch(buffer_size=AUTOTUNE)
val_dataset   = val_dataset.prefetch(buffer_size=AUTOTUNE)

# Fine-tune model
history_finetune = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=30,
    class_weight=class_weights,
    callbacks=[early_stopping, reduce_lr]
)

# Evaluate the model on the test dataset
results = model.evaluate(test_dataset)
print("Test Results:", dict(zip(model.metrics_names, results)))
```

Training and Validation Metrics over Epochs

```
import matplotlib.pyplot as plt

# List of metrics you want to plot.
metrics = ['accuracy', 'loss', 'f1_score', 'precision', 'recall', 'auc']

# Create dictionaries to hold the combined history (training and validation) for each metric.
combined_train = {}
combined_val = {}

# For each metric, concatenate the initial training history with the fine-tuning history.
for metric in metrics:
    # Get training values (assumes both history objects exist)
    train_initial = history_initial.history[metric]
    train_finetune = history_finetune.history[metric]
    combined_train[metric] = train_initial + train_finetune

    # Get validation values (keys prefixed with 'val_')
    val_initial = history_initial.history['val_' + metric]
    val_finetune = history_finetune.history['val_' + metric]
    combined_val[metric] = val_initial + val_finetune

# Total number of combined epochs.
total_epochs = len(combined_train['accuracy'])
epochs_combined = list(range(1, total_epochs + 1))

# Determine the epoch number where the initial training phase ended.
phase_change_epoch = len(history_initial.history['accuracy'])

# Create a 6x2 grid of subplots:
fig, axs = plt.subplots(nrows=6, ncols=2, figsize=(15, 30))

# Loop over the metrics and plot training (left column) and validation (right column) curves.
for i, metric in enumerate(metrics):
    # Left: Training metric.
    axs[i, 0].plot(epochs_combined, combined_train[metric], marker='o', label='Training ' + metric)
    axs[i, 0].axvline(x=phase_change_epoch, color='gray', linestyle='--', label='Phase Change')
    axs[i, 0].set_xlabel('Epoch')
    axs[i, 0].set_ylabel(metric)
    axs[i, 0].set_title('Training ' + metric)
    axs[i, 0].legend()

    # Right: Validation metric.
    axs[i, 1].plot(epochs_combined, combined_val[metric], marker='o', label='Validation ' + metric)
    axs[i, 1].axvline(x=phase_change_epoch, color='gray', linestyle='--', label='Phase Change')
    axs[i, 1].set_xlabel('Epoch')
    axs[i, 1].set_ylabel(metric)
    axs[i, 1].set_title('Validation ' + metric)
    axs[i, 1].legend()

plt.tight_layout()
plt.show()
```

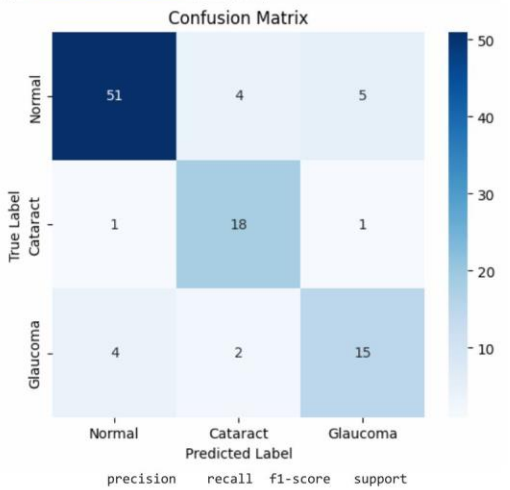

Training accuracy
Training accuracy
Phase Change
accuracy
Epoch
Validation accuracy
Validation accuracy
Phase Change
accuracy
Epoch
Training loss
Training loss
Phase Change
loss
Epoch
Validation loss
Validation loss
Phase Change
loss
Epoch
Training f1_score
Training f1_score
Phase Change
f1_score
Epoch
Validation f1_score
Validation f1_score
Phase Change
f1_score
Epoch
Training precision
Training precision
Phase Change
precision
Epoch
Validation precision
Validation precision
Phase Change
precision
Epoch
Training recall
Training recall
Phase Change
Validation recall
Validation recall
Phase Change



Confusion Matrix & Classification Report

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

# Get the true labels and predictions
y_true = []
y_pred = []

# Loop through the test dataset and collect labels & predictions
for images, labels in test_dataset:
    # True labels
    y_true.extend(np.argmax(labels.numpy(), axis=1))

    # Predictions
    preds = model.predict(images)
    y_pred.extend(np.argmax(preds, axis=1))

# Convert lists to numpy arrays
y_true = np.array(y_true)
y_pred = np.array(y_pred)

# Compute the confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Class names
class_names = ["Normal", "Cataract", "Glaucoma"]

# Plot the confusion matrix
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names, yticklabels=class_names)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# Print Classification Report
print(classification_report(y_true, y_pred, target_names=class_names))
```

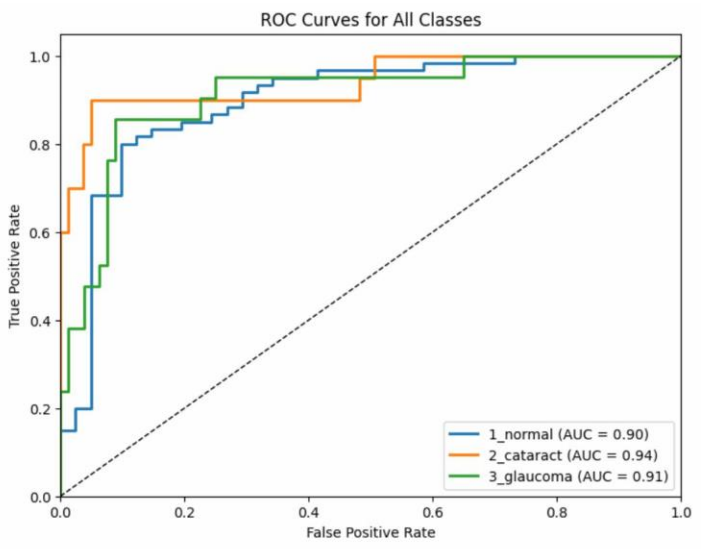
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 7s 3s/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 5s 3s/step
```



	precision	recall	f1-score	support
Normal	0.91	0.85	0.88	60
Cataract	0.75	0.90	0.82	20
Glaucoma	0.71	0.71	0.71	21
accuracy			0.83	101
macro avg	0.79	0.82	0.80	101
weighted avg	0.84	0.83	0.83	101

ROC curve

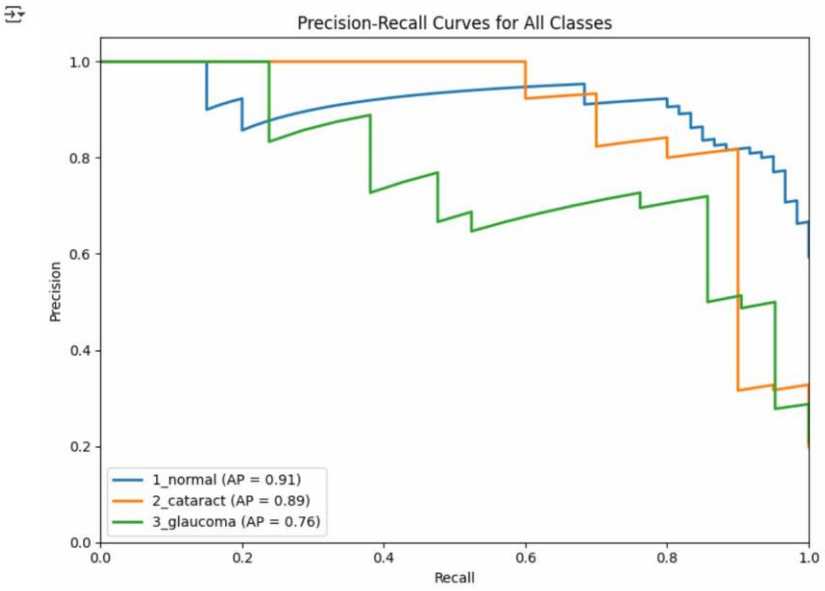
```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

plt.figure(figsize=(8, 6))
for i in range(n_classes):
    # Compute ROC curve and ROC area for class i.
    fpr, tpr, _ = roc_curve(y_true[:, i], y_pred_probs[:, i])
    roc_auc = auc(fpr, tpr)
    # Plot ROC curve for class i
    plt.plot(fpr, tpr, lw=2, label=f'{test_class_names[i]} (AUC = {roc_auc:.2f})')

# Plot the random chance line
plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for All Classes')
plt.legend(loc="lower right")
plt.show()
```



Precision-Recall curve

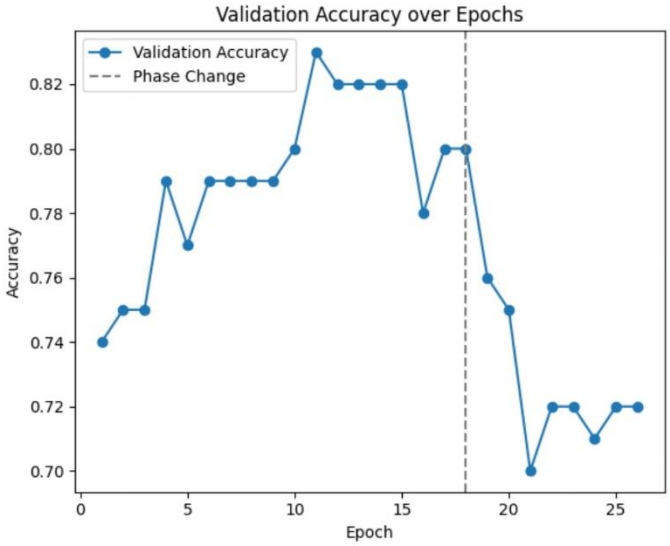
```
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, average_precision_score

plt.figure(figsize=(8, 6))
for i in range(n_classes):
    # Compute precision-recall curve and average precision for class i.
    precision_vals, recall_vals, _ = precision_recall_curve(y_true[:, i], y_pred_probs[:, i])
    ap = average_precision_score(y_true[:, i], y_pred_probs[:, i])

    # Plot all curves on the same axes
    plt.plot(recall_vals, precision_vals, lw=2, label=f'{test_class_names[i]} (AP = {ap:.2f})')

plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
```

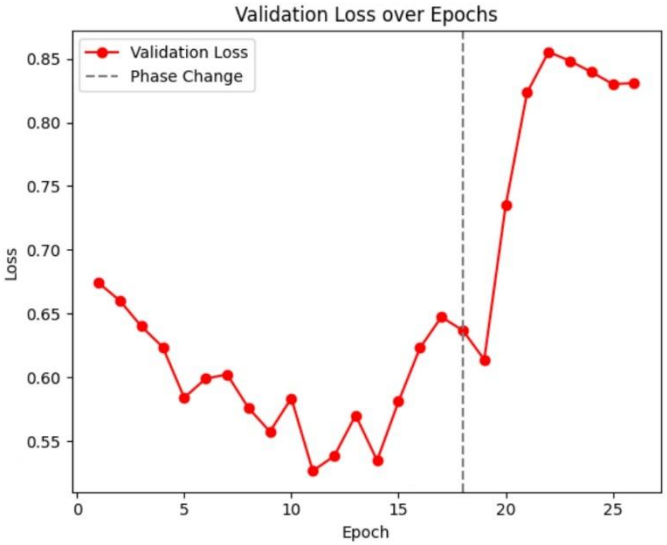
```
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves for All Classes')
plt.legend(loc="lower left")
plt.tight_layout()
plt.show()
```

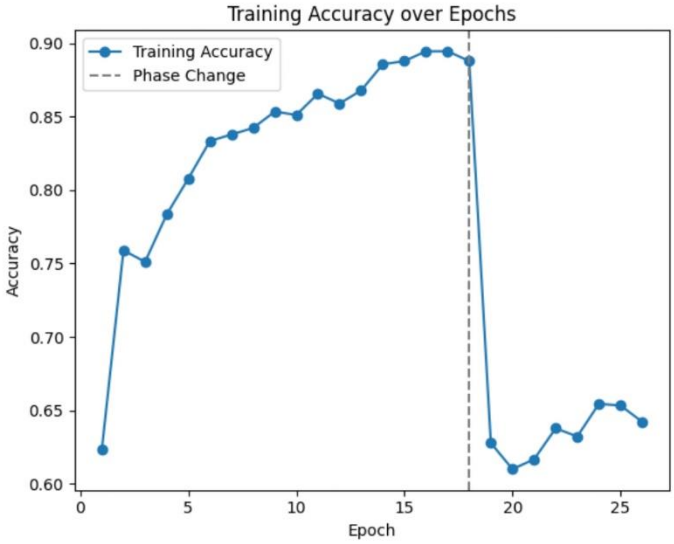


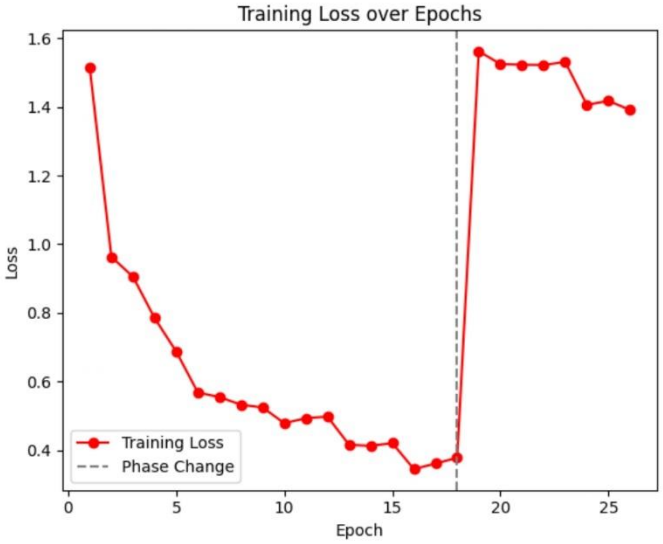
Training and Validation Accuracy/Loss over Epochs

```
import matplotlib.pyplot as plt

# --- Combine the histories from initial training and fine tuning ---
initial_epochs = len(history_initial.history['accuracy'])
finetune_epochs = len(history_finetune.history['accuracy'])
total_epochs = initial_epochs + finetune_epochs

# Combine training metrics
train_accuracy = history_initial.history['accuracy'] + history_finetune.history['accuracy']
train_loss     = history_initial.history['loss'] + history_finetune.history['loss']

# Combine validation metrics
val_accuracy = history_initial.history['val_accuracy'] + history_finetune.history['val_accuracy']
val_loss     = history_initial.history['val_loss'] + history_finetune.history['val_loss']

# Create an epoch list
epochs = list(range(1, total_epochs + 1))

# --- Graph 1: Validation Accuracy and Loss Over Epochs ---
plt.figure(figsize=(12, 5))
# Validation Accuracy
plt.subplot(1, 2, 1)
plt.plot(epochs, val_accuracy, marker='o', label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Validation Accuracy over Epochs')
plt.axvline(x=initial_epochs, color='gray', linestyle='--', label='Phase Change')
plt.legend()

# Validation Loss
plt.subplot(1, 2, 2)
plt.plot(epochs, val_loss, marker='o', color='red', label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Validation Loss over Epochs')
plt.axvline(x=initial_epochs, color='gray', linestyle='--', label='Phase Change')
plt.legend()

plt.tight_layout()
plt.show()

# --- Graph 2: Training Accuracy and Loss Over Epochs ---
plt.figure(figsize=(12, 5))
# Training Accuracy
plt.subplot(1, 2, 1)
plt.plot(epochs, train_accuracy, marker='o', label='Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.axvline(x=initial_epochs, color='gray', linestyle='--', label='Phase Change')
plt.legend()

# Training Loss
plt.subplot(1, 2, 2)
plt.plot(epochs, train_loss, marker='o', color='red', label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss over Epochs')
plt.axvline(x=initial_epochs, color='gray', linestyle='--', label='Phase Change')
plt.legend()

plt.tight_layout()
plt.show()
```









Start coding or generate with AI.