

```
In [2]: import os
import shutil
import random
import copy
from pathlib import Path

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

from PIL import Image, UnidentifiedImageError
from tqdm.auto import tqdm

import torch
from torch import nn, optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms as T, models

from sklearn.model_selection import train_test_split
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    classification_report,
    roc_curve,
    auc
)

from sklearn.preprocessing import label_binarize

import warnings
warnings.filterwarnings("ignore")
```

```
In [3]: SEED = 42

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print("PyTorch version:", torch.__version__)
print("Using device:", device)
```

PyTorch version: 2.13.0+cpu
Using device: cpu

```
In [4]: ORIGINAL_DATASET_DIR = Path(
    r"dataset"
```

```
)

RESIZED_DATASET_DIR = Path("resized_dataset")
SPLIT_DATASET_DIR = Path("new_dataset")

print("Original dataset path:", ORIGINAL_DATASET_DIR.resolve())
```

Original dataset path: C:\Users\habus\Desktop\ai master\eye_diseases\dataset

```
In [5]: if not ORIGINAL_DATASET_DIR.exists():
        raise FileNotFoundError(
            "The dataset folder was not found. "
            "Please update ORIGINAL_DATASET_DIR."
        )

class_folders = sorted(
    folder.name
    for folder in ORIGINAL_DATASET_DIR.iterdir()
    if folder.is_dir()
)

print("Detected classes:", class_folders)
print("Number of classes:", len(class_folders))
```

Detected classes: ['cataract', 'diabetic_retinopathy', 'glaucoma', 'normal']
Number of classes: 4

```
In [6]: IMAGE_EXTENSIONS = {
        ".jpg",
        ".jpeg",
        ".png",
        ".bmp",
        ".tif",
        ".tiff"
    }

def count_images(folder):
    return sum(
        1
        for file in folder.iterdir()
        if file.is_file()
        and file.suffix.lower() in IMAGE_EXTENSIONS
    )

original_counts = {
    class_name: count_images(
        ORIGINAL_DATASET_DIR / class_name
    )
    for class_name in class_folders
}

print("Original class distribution:")

for class_name, count in original_counts.items():
    print(f"{class_name:25s}: {count}")
```

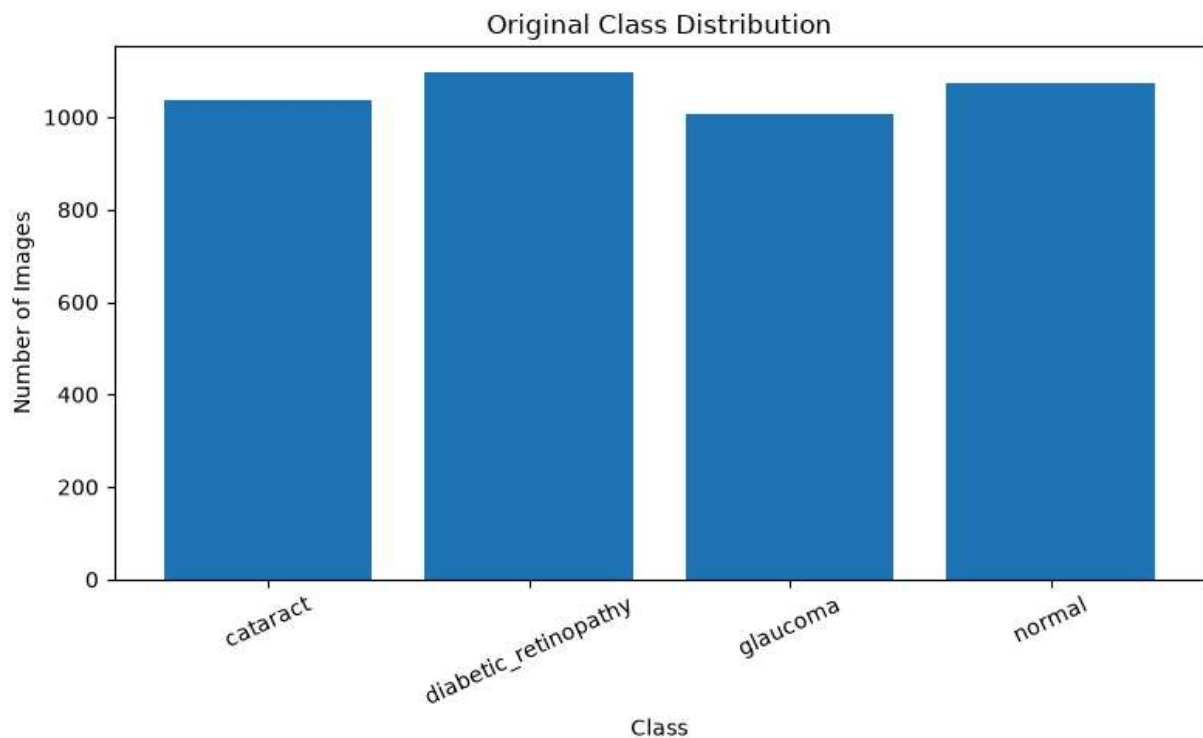
Original class distribution:

cataract	: 1038
diabetic_retinopathy	: 1098
glaucoma	: 1007
normal	: 1074

```
In [7]: plt.figure(figsize=(8, 5))

plt.bar(
    original_counts.keys(),
    original_counts.values()
)

plt.title("Original Class Distribution")
plt.xlabel("Class")
plt.ylabel("Number of Images")
plt.xticks(rotation=25)
plt.tight_layout()
plt.show()
```



```
In [9]: #Resize images
RESIZE_SIZE = (256, 256)

if RESIZED_DATASET_DIR.exists():
    shutil.rmtree(RESIZED_DATASET_DIR)

RESIZED_DATASET_DIR.mkdir(
    parents=True,
    exist_ok=True
)

processed_count = 0
failed_count = 0
```

```
for class_name in class_folders:

    input_class_dir = (
        ORIGINAL_DATASET_DIR / class_name
    )

    output_class_dir = (
        RESIZED_DATASET_DIR / class_name
    )

    output_class_dir.mkdir(
        parents=True,
        exist_ok=True
    )

    image_files = [
        file
        for file in input_class_dir.iterdir()
        if file.is_file()
        and file.suffix.lower() in IMAGE_EXTENSIONS
    ]

    for image_path in tqdm(
        image_files,
        desc=f"Resizing {class_name}"
    ):

        output_path = (
            output_class_dir /
            f"{image_path.stem}.jpg"
        )

        try:

            with Image.open(image_path) as image:

                image = image.convert("RGB")
                image = image.resize(RESIZE_SIZE)

                image.save(
                    output_path,
                    "JPEG",
                    quality=90
                )

            processed_count += 1

        except (
            UnidentifiedImageError,
            OSError
        ) as error:

            failed_count += 1
            print(
                f"Failed to process "
```

```

        f"{image_path}: {error}"
    )

print("Processed images:", processed_count)
print("Failed images:", failed_count)
print(
    "Resized dataset folder:",
    RESIZED_DATASET_DIR.resolve()
)

```

```

Resizing cataract: 100%|██████████| 1038/1038 [00:39<00:00, 26.06it/s]
Resizing diabetic_retinopathy: 100%|██████████| 1098/1098 [00:32<00:00, 34.21it/s]
Resizing glaucoma: 100%|██████████| 1007/1007 [00:40<00:00, 24.66it/s]
Resizing normal: 100%|██████████| 1074/1074 [00:31<00:00, 33.87it/s]
Processed images: 4217
Failed images: 0
Resized dataset folder: C:\Users\habus\Desktop\ai master\eye_diseases\resized_dataset

```

In [10]: *#Split dataset into train, validation, and test*

```

TRAIN_RATIO = 0.70
VALIDATION_RATIO = 0.10
TEST_RATIO = 0.20

if SPLIT_DATASET_DIR.exists():
    shutil.rmtree(SPLIT_DATASET_DIR)

train_dir = SPLIT_DATASET_DIR / "train"
validation_dir = SPLIT_DATASET_DIR / "validation"
test_dir = SPLIT_DATASET_DIR / "test"

for folder in [
    train_dir,
    validation_dir,
    test_dir
]:
    folder.mkdir(
        parents=True,
        exist_ok=True
    )

```

In [11]: *#Split and copy the images*

```

def split_and_copy_class(
    class_dir,
    destination_root
):
    all_images = [
        file
        for file in class_dir.iterdir()
        if file.is_file()
        and file.suffix.lower()
        in IMAGE_EXTENSIONS
    ]

```

```

if len(all_images) < 10:
    raise ValueError(
        f"Class {class_dir.name} "
        f"contains too few images."
    )

train_images, temporary_images = (
    train_test_split(
        all_images,
        test_size=(
            1 - TRAIN_RATIO
        ),
        random_state=SEED,
        shuffle=True
    )
)

relative_test_ratio = (
    TEST_RATIO /
    (
        VALIDATION_RATIO +
        TEST_RATIO
    )
)

validation_images, test_images = (
    train_test_split(
        temporary_images,
        test_size=relative_test_ratio,
        random_state=SEED,
        shuffle=True
    )
)

split_mapping = {
    "train": train_images,
    "validation": validation_images,
    "test": test_images
}

for split_name, image_list \
    in split_mapping.items():

    destination_class_dir = (
        destination_root /
        split_name /
        class_dir.name
    )

    destination_class_dir.mkdir(
        parents=True,
        exist_ok=True
    )

    for image_path in image_list:

```

```
shutil.copy2(  
    image_path,  
    destination_class_dir /  
    image_path.name  
)
```

```
In [12]: #Apply split to all classes  
for class_name in class_folders:  
  
    split_and_copy_class(  
        RESIZED_DATASET_DIR /  
        class_name,  
        SPLIT_DATASET_DIR  
    )  
  
print("Dataset split completed.")  
print(  
    "Split dataset folder:",  
    SPLIT_DATASET_DIR.resolve()  
)
```

Dataset split completed.

Split dataset folder: C:\Users\habus\Desktop\ai master\eye_diseases\new_dataset

```
In [13]: #Check split counts  
def get_split_counts(split_folder):  
  
    counts = {}  
  
    class_directories = sorted(  
        folder  
        for folder in split_folder.iterdir()  
        if folder.is_dir()  
    )  
  
    for class_folder in class_directories:  
  
        counts[class_folder.name] = (  
            count_images(class_folder)  
        )  
  
    return counts  
  
train_counts = get_split_counts(  
    train_dir  
)  
  
validation_counts = get_split_counts(  
    validation_dir  
)  
  
test_counts = get_split_counts(  
    test_dir  
)
```

```

print("Training counts:")
print(train_counts)

print("\nValidation counts:")
print(validation_counts)

print("\nTesting counts:")
print(test_counts)

print("\nTotal training images:")
print(sum(train_counts.values()))

print("Total validation images:")
print(sum(validation_counts.values()))

print("Total testing images:")
print(sum(test_counts.values()))

```

Training counts:

```
{'cataract': 726, 'diabetic_retinopathy': 768, 'glaucoma': 704, 'normal': 751}
```

Validation counts:

```
{'cataract': 104, 'diabetic_retinopathy': 110, 'glaucoma': 101, 'normal': 107}
```

Testing counts:

```
{'cataract': 208, 'diabetic_retinopathy': 220, 'glaucoma': 202, 'normal': 216}
```

Total training images:

2949

Total validation images:

422

Total testing images:

846

```

In [14]: #Define augmentation
IMAGE_SIZE = 224
BATCH_SIZE = 32

train_transform = T.Compose([
    T.Resize(
        (IMAGE_SIZE, IMAGE_SIZE)
    ),

    T.RandomHorizontalFlip(
        p=0.5
    ),

    T.RandomVerticalFlip(
        p=0.2
    ),

    T.RandomRotation(
        10
    ),

    T.RandomAffine(

```

```
        degrees=0,
        translate=(0.08, 0.08),
        scale=(0.90, 1.10)
    ),

    T.ColorJitter(
        brightness=0.15,
        contrast=0.15
    ),

    T.ToTensor(),

    T.Normalize(
        mean=[
            0.485,
            0.456,
            0.406
        ],
        std=[
            0.229,
            0.224,
            0.225
        ]
    )
])
```

In [15]: *#Define validation and test preprocessing*

```
evaluation_transform = T.Compose([
    T.Resize(
        (IMAGE_SIZE, IMAGE_SIZE)
    ),

    T.ToTensor(),

    T.Normalize(
        mean=[
            0.485,
            0.456,
            0.406
        ],
        std=[
            0.229,
            0.224,
            0.225
        ]
    )
])
```

In [16]: *#Create datasets*

```
train_data = datasets.ImageFolder(
    train_dir,
    transform=train_transform
)

validation_data = datasets.ImageFolder(
```

```

        validation_dir,
        transform=evaluation_transform
    )

    test_data = datasets.ImageFolder(
        test_dir,
        transform=evaluation_transform
    )

    classes = train_data.classes
    num_classes = len(classes)

    print("Classes:", classes)
    print("Number of classes:", num_classes)

```

Classes: ['cataract', 'diabetic_retinopathy', 'glaucoma', 'normal']
 Number of classes: 4

```

In [17]: #Create DataLoaders
train_loader = DataLoader(
    train_data,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=0
)

validation_loader = DataLoader(
    validation_data,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=0
)

test_loader = DataLoader(
    test_data,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=0
)

dataloaders = {
    "train": train_loader,
    "validation": validation_loader
}

dataset_sizes = {
    "train": len(train_data),
    "validation": len(validation_data)
}

print("Training images:", len(train_data))
print("Validation images:", len(validation_data))
print("Testing images:", len(test_data))

```

Training images: 2949
Validation images: 422
Testing images: 846

```
In [18]: #Display augmented images
def denormalize(image_tensor):

    mean = torch.tensor(
        [0.485, 0.456, 0.406]
    ).view(3, 1, 1)

    std = torch.tensor(
        [0.229, 0.224, 0.225]
    ).view(3, 1, 1)

    image_tensor = (
        image_tensor.cpu() *
        std +
        mean
    )

    return image_tensor.clamp(
        0,
        1
    )

images, labels = next(
    iter(train_loader)
)

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

for index in range(
    min(8, len(images))
):

    plt.subplot(
        2,
        4,
        index + 1
    )

    image = denormalize(
        images[index]
    )

    image = (
        image
        .permute(1, 2, 0)
        .numpy()
    )

    plt.imshow(image)

    plt.title(
```

```

        classes[
            labels[index].item()
        ]
    )

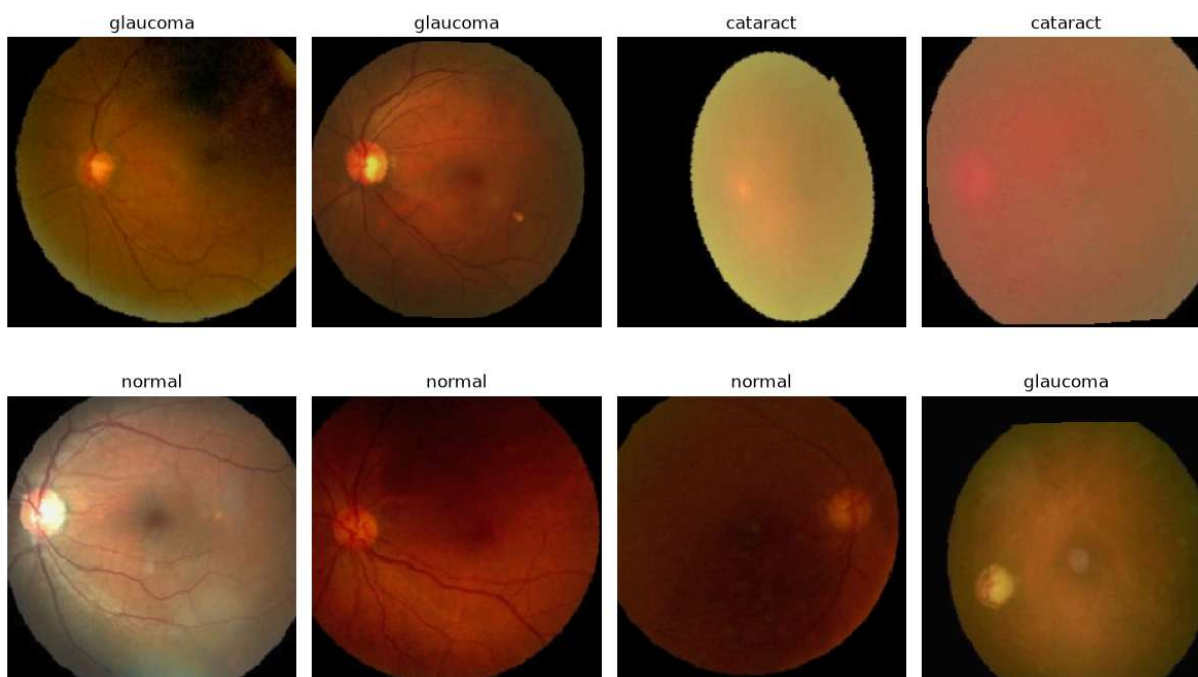
    plt.axis("off")

plt.suptitle(
    "Augmented Training Images"
)

plt.tight_layout()
plt.show()

```

Augmented Training Images



```

In [20]: #Load EfficientNet-B0
model = models.efficientnet_b0(
    weights=
        models.EfficientNet_B0_Weights.DEFAULT
)

model.classifier[1] = nn.Linear(
    model.classifier[1].in_features,
    num_classes
)

model = model.to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.AdamW(
    model.parameters(),
    lr=0.001,
    weight_decay=0.0001
)

```

```

)

scheduler = optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.7
)

print(model.classifier)

```

Downloading: "https://download.pytorch.org/models/efficientnet_b0_rwightman-7f5810bc.pth" to C:\Users\habus\.cache\torch\hub\checkpoints\efficientnet_b0_rwightman-7f5810bc.pth

100%|██████████| 20.5M/20.5M [00:00<00:00, 33.3MB/s]

```

Sequential(
  (0): Dropout(p=0.2, inplace=True)
  (1): Linear(in_features=1280, out_features=4, bias=True)
)

```

```

In [21]: #Define the training function
def train_model(
    model,
    criterion,
    optimizer,
    scheduler,
    num_epochs=10,
    patience=3
):

    history = {
        "train_loss": [],
        "train_accuracy": [],
        "validation_loss": [],
        "validation_accuracy": []
    }

    best_model_weights = copy.deepcopy(
        model.state_dict()
    )

    best_validation_loss = float("inf")
    epochs_without_improvement = 0

    for epoch in range(num_epochs):

        print(
            f"Epoch {epoch + 1}/"
            f"{num_epochs}"
        )

        print("-" * 60)

        for phase in [
            "train",
            "validation"
        ]:

```

```
if phase == "train":
    model.train()
else:
    model.eval()

running_loss = 0.0
running_correct = 0

for inputs, labels in tqdm(
    dataloaders[phase],
    desc=phase.capitalize()
):

    inputs = inputs.to(device)
    labels = labels.to(device)

    optimizer.zero_grad()

    with torch.set_grad_enabled(
        phase == "train"
    ):

        outputs = model(inputs)

        loss = criterion(
            outputs,
            labels
        )

        predictions = (
            outputs.argmax(dim=1)
        )

        if phase == "train":

            loss.backward()
            optimizer.step()

        running_loss += (
            loss.item() *
            inputs.size(0)
        )

        running_correct += (
            predictions ==
            labels
        ).sum().item()

    epoch_loss = (
        running_loss /
        dataset_sizes[phase]
    )

    epoch_accuracy = (
        running_correct /
```

```
        dataset_sizes[phase]
    )

    history[
        f"{phase}_loss"
    ].append(epoch_loss)

    history[
        f"{phase}_accuracy"
    ].append(epoch_accuracy)

    print(
        f"{phase.capitalize()} "
        f"Loss: {epoch_loss:.4f} "
        f"Accuracy: "
        f"{epoch_accuracy:.4f}"
    )

    if phase == "validation":

        if (
            epoch_loss <
            best_validation_loss
        ):

            best_validation_loss = (
                epoch_loss
            )

            best_model_weights = (
                copy.deepcopy(
                    model.state_dict()
                )
            )

            epochs_without_improvement = 0

        else:

            epochs_without_improvement += 1

    scheduler.step()

    if (
        epochs_without_improvement
        >= patience
    ):

        print(
            "Early stopping activated."
        )

        break

    print()
```

```

model.load_state_dict(
    best_model_weights
)

return model, history

```

```

In [24]: #Train the model
NUM_EPOCHS = 10
PATIENCE = 3

trained_model, history = train_model(
    model=model,
    criterion=criterion,
    optimizer=optimizer,
    scheduler=scheduler,
    num_epochs=NUM_EPOCHS,
    patience=PATIENCE
)

```

Epoch 1/10

Train: 100%|██████████| 93/93 [06:39<00:00, 4.30s/it]

Train Loss: 0.3250 Accuracy: 0.8806

Validation: 100%|██████████| 14/14 [00:13<00:00, 1.05it/s]

Validation Loss: 0.3519 Accuracy: 0.8507

Epoch 2/10

Train: 100%|██████████| 93/93 [06:38<00:00, 4.29s/it]

Train Loss: 0.2791 Accuracy: 0.9047

Validation: 100%|██████████| 14/14 [00:13<00:00, 1.02it/s]

Validation Loss: 0.2357 Accuracy: 0.9171

Epoch 3/10

Train: 100%|██████████| 93/93 [06:37<00:00, 4.28s/it]

Train Loss: 0.2515 Accuracy: 0.9037

Validation: 100%|██████████| 14/14 [00:13<00:00, 1.04it/s]

Validation Loss: 0.2340 Accuracy: 0.9194

Epoch 4/10

Train: 100%|██████████| 93/93 [06:38<00:00, 4.28s/it]

Train Loss: 0.2099 Accuracy: 0.9149

Validation: 100%|██████████| 14/14 [00:13<00:00, 1.03it/s]

Validation Loss: 0.2220 Accuracy: 0.9265

Epoch 5/10

Train: 100%|██████████| 93/93 [06:37<00:00, 4.27s/it]

Train Loss: 0.1793 Accuracy: 0.9366

Validation: 100%|██████████| 14/14 [00:14<00:00, 1.02s/it]

Validation Loss: 0.1643 Accuracy: 0.9408

Epoch 6/10

Train: 100%|██████████| 93/93 [06:37<00:00, 4.27s/it]

Train Loss: 0.1462 Accuracy: 0.9468

Validation: 100%|██████████| 14/14 [00:13<00:00, 1.03it/s]

Validation Loss: 0.1815 Accuracy: 0.9336

Epoch 7/10

Train: 100%|██████████| 93/93 [06:35<00:00, 4.25s/it]

Train Loss: 0.1416 Accuracy: 0.9464

Validation: 100%|██████████| 14/14 [00:13<00:00, 1.04it/s]

Validation Loss: 0.1954 Accuracy: 0.9289

Epoch 8/10

Train: 100%|██████████| 93/93 [06:35<00:00, 4.25s/it]

Train Loss: 0.1176 Accuracy: 0.9573

Validation: 100%|██████████| 14/14 [00:13<00:00, 1.03it/s]

Validation Loss: 0.1689 Accuracy: 0.9289

Early stopping activated.

```
In [25]: #Plot training Loss
epochs = range(
    1,
    len(history["train_loss"]) + 1
)

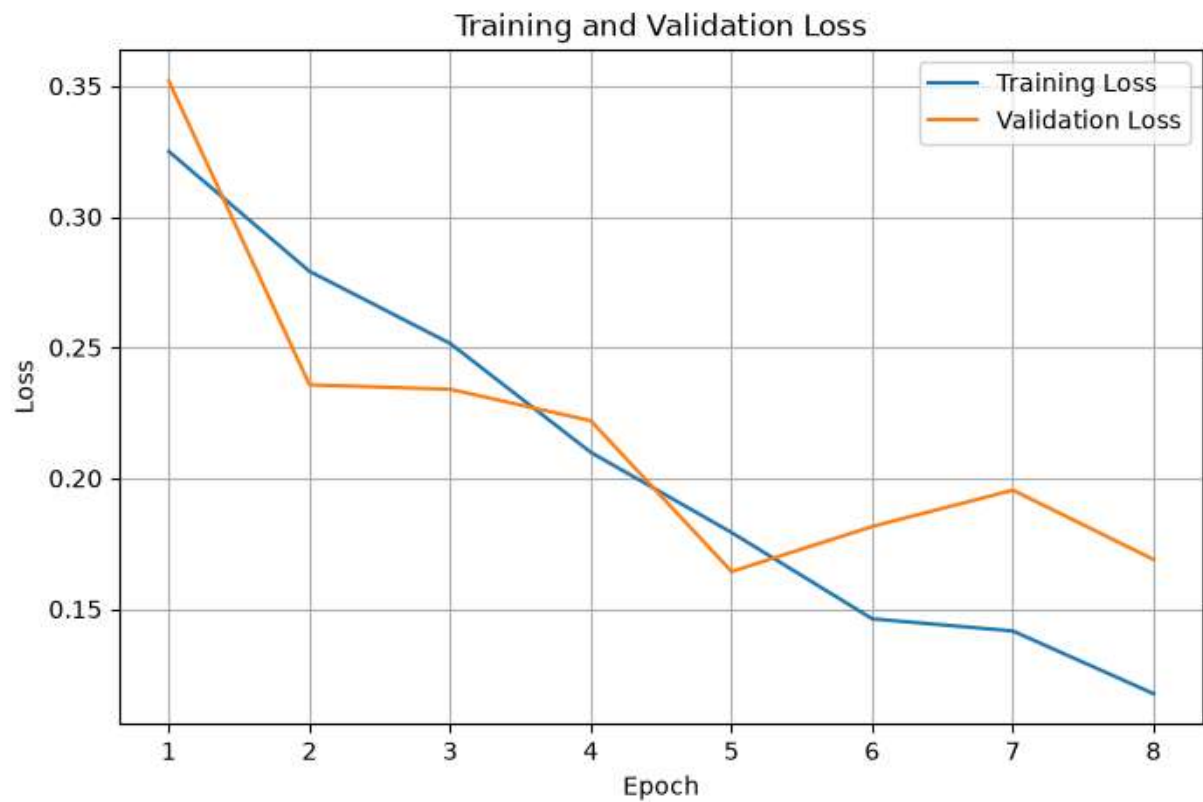
plt.figure(figsize=(8, 5))

plt.plot(
    epochs,
    history["train_loss"],
    label="Training Loss"
)

plt.plot(
    epochs,
    history["validation_loss"],
    label="Validation Loss"
)

plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title(
    "Training and Validation Loss"
)

plt.legend()
plt.grid(True)
plt.show()
```



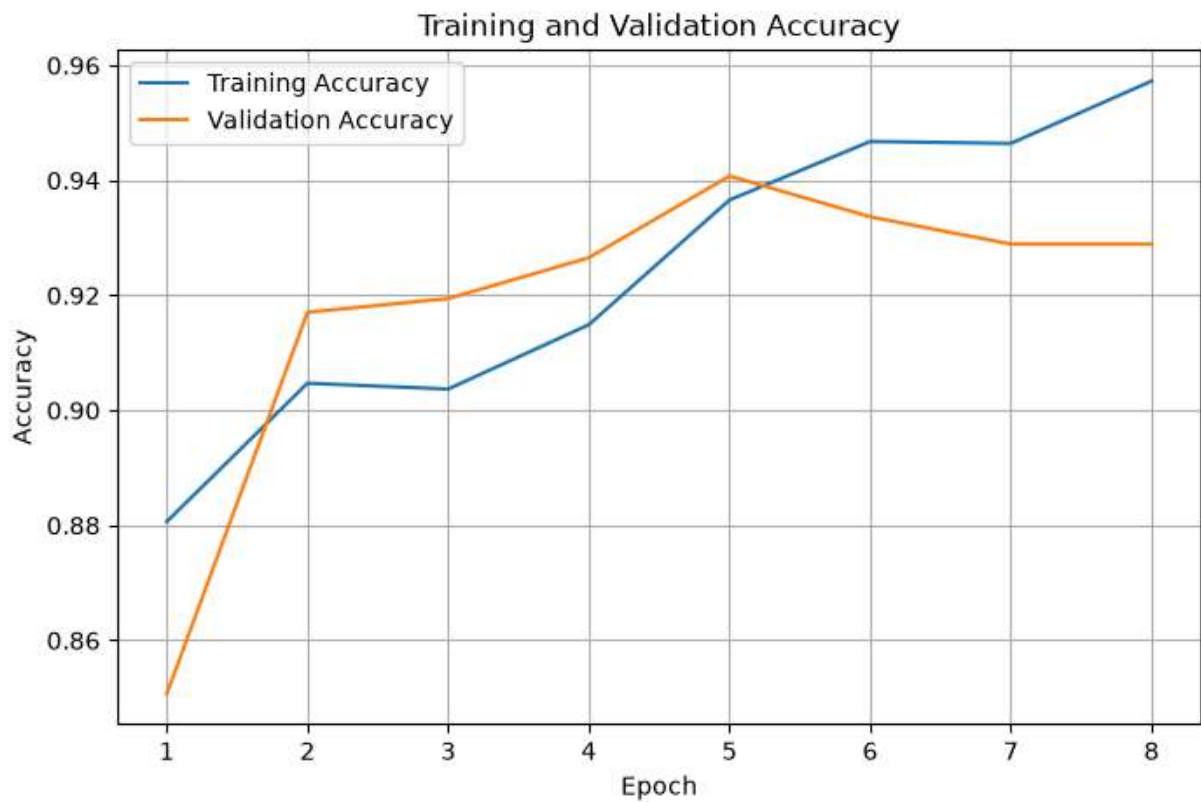
```
In [27]: #Plot training accuracy
plt.figure(figsize=(8, 5))

plt.plot(
    epochs,
    history["train_accuracy"],
    label="Training Accuracy"
)

plt.plot(
    epochs,
    history["validation_accuracy"],
    label="Validation Accuracy"
)

plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title(
    "Training and Validation Accuracy"
)

plt.legend()
plt.grid(True)
plt.show()
```



```
In [29]: #Test the Model
trained_model.eval()

all_true_labels = []
all_predicted_labels = []
all_probabilities = []

with torch.no_grad():

    for inputs, labels in tqdm(
        test_loader,
        desc="Testing"
    ):

        inputs = inputs.to(device)
        labels = labels.to(device)

        outputs = trained_model(inputs)

        probabilities = torch.softmax(
            outputs,
            dim=1
        )

        predictions = outputs.argmax(
            dim=1
        )

        all_true_labels.extend(
            labels.cpu().numpy()
        )
```

```

        all_predicted_labels.extend(
            predictions.cpu().numpy()
        )

        all_probabilities.extend(
            probabilities.cpu().numpy()
        )

y_true = np.array(
    all_true_labels
)

y_pred = np.array(
    all_predicted_labels
)

y_probability = np.array(
    all_probabilities
)

print("Testing completed.")

```

Testing: 100%|██████████| 27/27 [00:40<00:00, 1.50s/it]
 Testing completed.

```

In [31]: #Calculate evaluation metrics
accuracy = accuracy_score(
    y_true,
    y_pred
)

precision = precision_score(
    y_true,
    y_pred,
    average="weighted",
    zero_division=0
)

recall = recall_score(
    y_true,
    y_pred,
    average="weighted",
    zero_division=0
)

f1 = f1_score(
    y_true,
    y_pred,
    average="weighted",
    zero_division=0
)

print(
    f"Accuracy: {accuracy:.4f}"
)

```

```

print(
    f"Precision: {precision:.4f}"
)

print(
    f"Recall:      {recall:.4f}"
)

print(
    f"F1-score:   {f1:.4f}"
)

```

Accuracy: 0.9184
 Precision: 0.9233
 Recall: 0.9184
 F1-score: 0.9188

```

In [32]: #Classification report
print("Classification Report:")

print(
    classification_report(
        y_true,
        y_pred,
        target_names=classes,
        zero_division=0
    )
)

```

Classification Report:

	precision	recall	f1-score	support
cataract	0.92	0.96	0.94	208
diabetic_retinopathy	1.00	0.93	0.96	220
glaucoma	0.94	0.83	0.88	202
normal	0.83	0.94	0.89	216
accuracy			0.92	846
macro avg	0.92	0.92	0.92	846
weighted avg	0.92	0.92	0.92	846

```

In [33]: #Confusion matrix
matrix = confusion_matrix(
    y_true,
    y_pred
)

plt.figure(figsize=(8, 6))

sns.heatmap(
    matrix,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=classes,

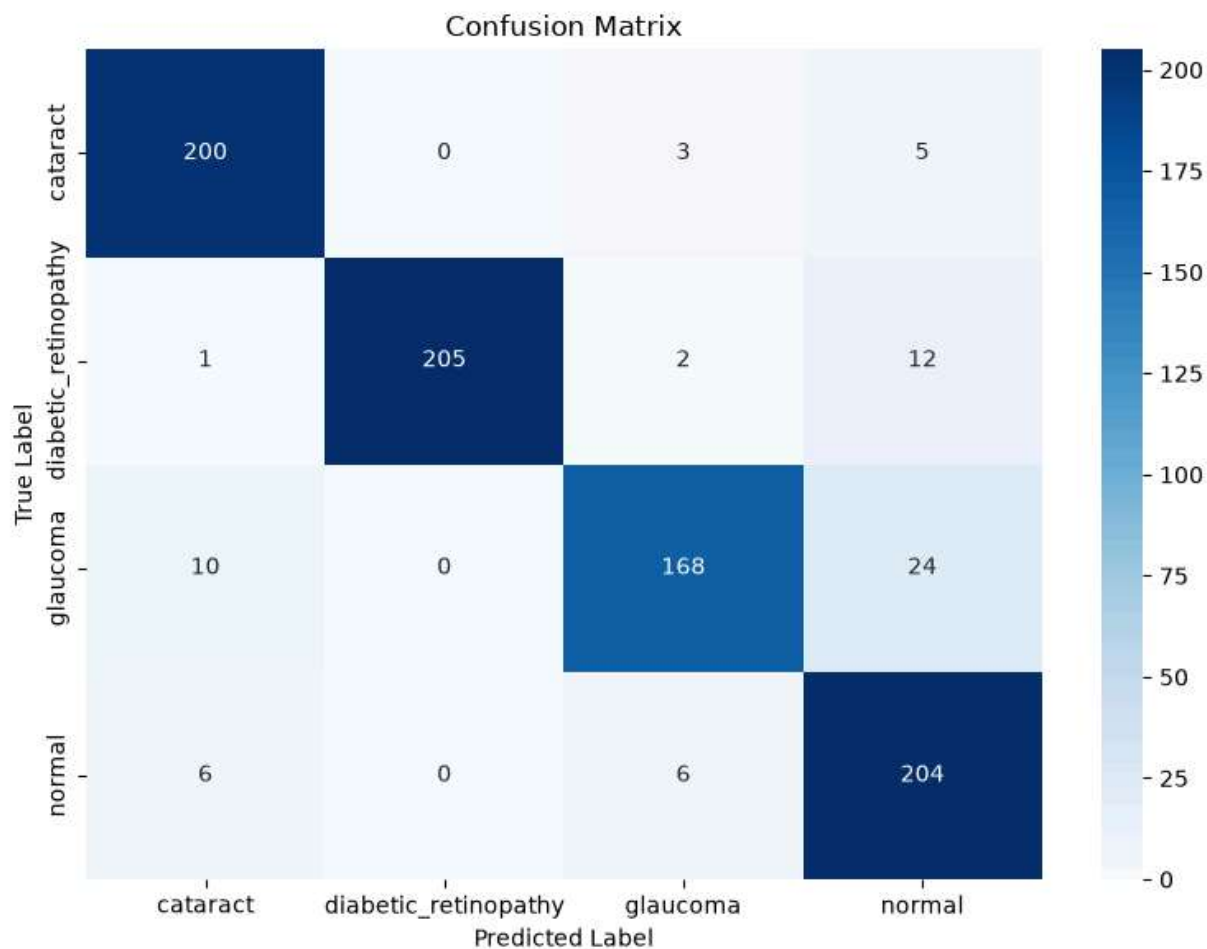
```

```

    yticklabels=classes
)

plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.tight_layout()
plt.show()

```



```

In [34]: #Multiclass ROC curves
y_true_binary = label_binarize(
    y_true,
    classes=np.arange(
        num_classes
    )
)

plt.figure(figsize=(9, 7))

for class_index, class_name \
    in enumerate(classes):

    false_positive_rate, \
    true_positive_rate, _ = (
        roc_curve(
            y_true_binary[
                :,

```

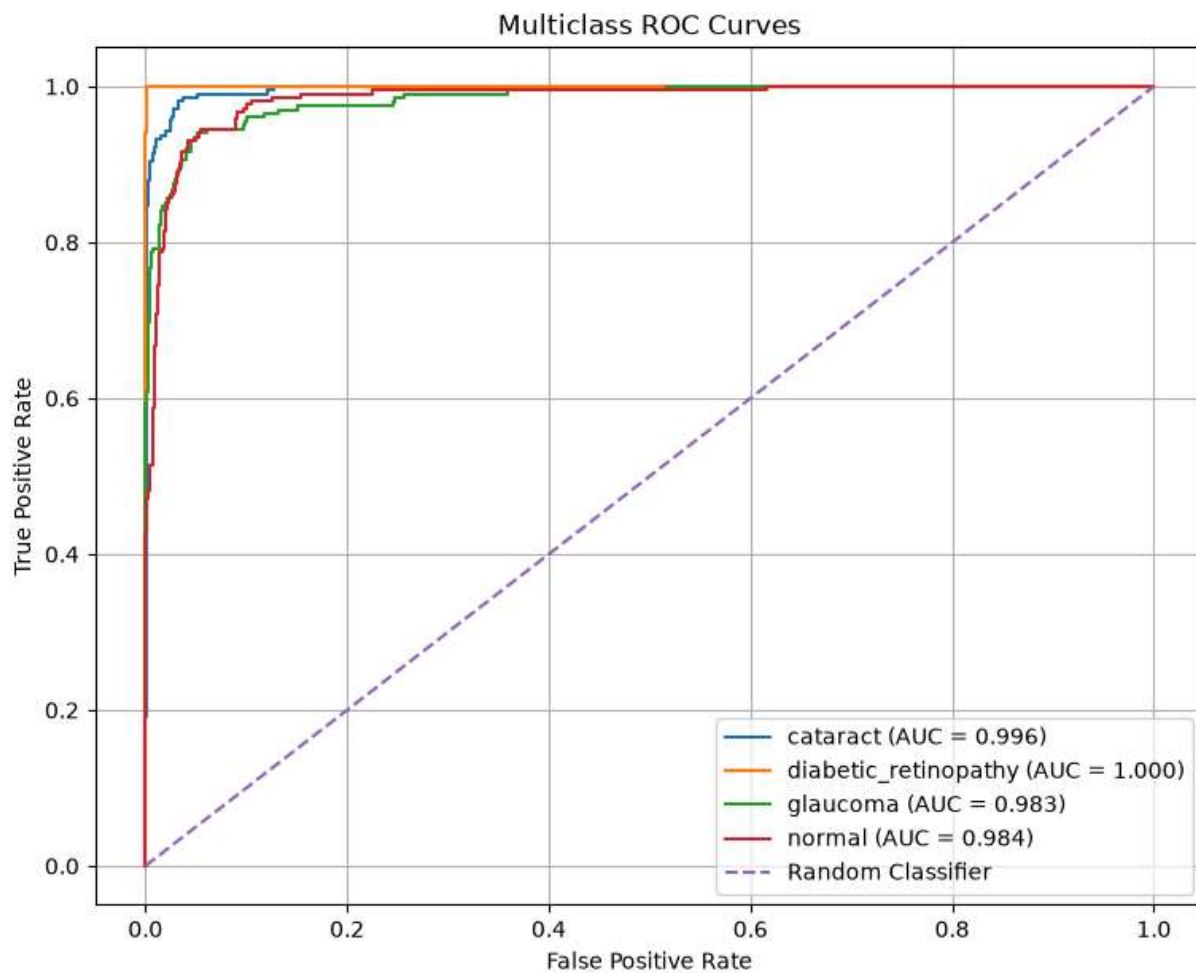
```
        class_index
    ],
    y_probability[
        :,
        class_index
    ]
)

roc_auc = auc(
    false_positive_rate,
    true_positive_rate
)

plt.plot(
    false_positive_rate,
    true_positive_rate,
    label=(
        f"{class_name} "
        f"(AUC = {roc_auc:.3f})"
    )
)

plt.plot(
    [0, 1],
    [0, 1],
    linestyle="--",
    label="Random Classifier"
)

plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("Multiclass ROC Curves")
plt.legend()
plt.grid(True)
plt.show()
```



```
In [35]: #Save the trained model
MODEL_OUTPUT_PATH = Path(
    "efficientnet_eye_disease_model.pt"
)

torch.save(
    {
        "model_name":
            "efficientnet_b0",

        "model_state_dict":
            trained_model.state_dict(),

        "classes":
            classes,

        "image_size":
            IMAGE_SIZE
    },
    MODEL_OUTPUT_PATH
)

print(
    "Model saved to:",
```

```
MODEL_OUTPUT_PATH.resolve()
)
```

Model saved to: C:\Users\habus\Desktop\ai master\eye_diseases\efficientnet_eye_disease_model.pt

```
In [38]: #predict one image
#I picked up on test image with diabetic_retinopathy to test the model
IMAGE_TO_PREDICT = Path(
    r"test_eye.jpg"
)
```

```
In [39]: prediction_transform = T.Compose([
    T.Resize(
        (IMAGE_SIZE, IMAGE_SIZE)
    ),

    T.ToTensor(),

    T.Normalize(
        mean=[
            0.485,
            0.456,
            0.406
        ],
        std=[
            0.229,
            0.224,
            0.225
        ]
    )
])

def predict_single_image(
    image_path,
    model,
    class_names
):

    if not image_path.exists():

        raise FileNotFoundError(
            "Prediction image not found. "
            "Update IMAGE_TO_PREDICT."
        )

    with Image.open(
        image_path
    ) as image:

        rgb_image = image.convert(
            "RGB"
        )

        input_tensor = (
```

```
        prediction_transform(
            rgb_image
        )
        .unsqueeze(0)
        .to(device)
    )

model.eval()

with torch.no_grad():

    output = model(
        input_tensor
    )

    probabilities = torch.softmax(
        output,
        dim=1
    )[0]

    predicted_index = (
        probabilities
        .argmax()
        .item()
    )

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

plt.imshow(rgb_image)

plt.axis("off")

plt.title(
    f"Prediction: "
    f"{class_names[predicted_index]}\n"
    f"Confidence: "
    f"{probabilities[predicted_index].item():.2%}"
)

plt.show()

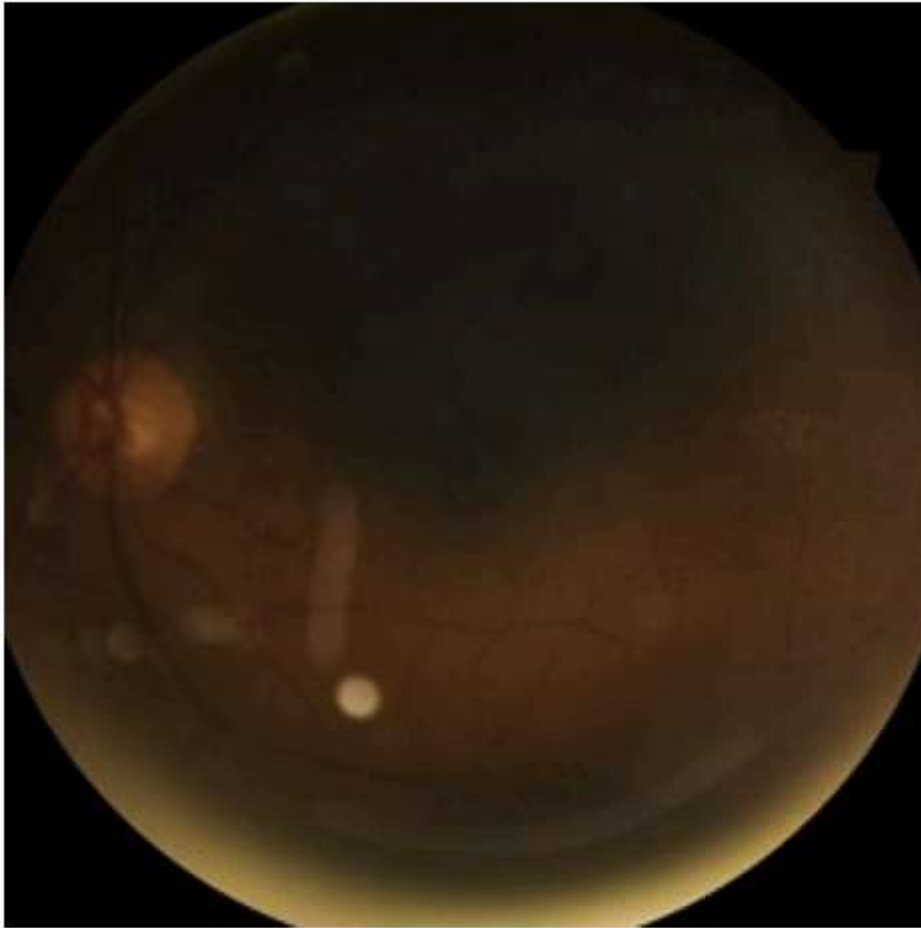
print("Class probabilities:")

for class_name, probability \
    in zip(
        class_names,
        probabilities
        .cpu()
        .numpy()
    ):

    print(
        f"{class_name:25s}: "
        f"{probability:.4f}"
    )
```

```
predict_single_image(  
    IMAGE_TO_PREDICT,  
    trained_model,  
    classes  
)
```

Prediction: diabetic_retinopathy
Confidence: 97.19%



Class probabilities:

cataract	: 0.0003
diabetic_retinopathy	: 0.9719
glaucoma	: 0.0132
normal	: 0.0146

In []: