

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("dankok/diabetic-retinopathy-image-dataset")

print("Path to dataset files:", path)
```

Downloading to /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/1.archive...
100%|██████████| 408M/408M [00:02<00:00, 156MB/s]Extracting files...

Path to dataset files: /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/versions/1

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("dankok/diabetic-retinopathy-image-dataset")

print("Path to dataset files:", path)
```

Using Colab cache for faster access to the 'diabetic-retinopathy-image-dataset' dataset.
Path to dataset files: /kaggle/input/diabetic-retinopathy-image-dataset

```
import os
```

```
print("Dataset path:", path)
print("Main folders/files:")
print(os.listdir(path))
```

Dataset path: /kaggle/input/diabetic-retinopathy-image-dataset
Main folders/files:
['Localization', 'Segmentation', 'Disease_Grading']

```
train_images_dir = None
test_images_dir = None
train_csv_path = None
test_csv_path = None

for root, dirs, files in os.walk(path):

    if "Disease_Grading" in root and "Original_Images" in root:
        if os.path.basename(root) == "Training Set":
            train_images_dir = root
        elif os.path.basename(root) == "Testing Set":
            test_images_dir = root

    for file in files:
        file_lower = file.lower()
        full_path = os.path.join(root, file)

        if "disease grading" in file_lower and "training labels" in file_lower:
            train_csv_path = full_path

        if "disease grading" in file_lower and "testing labels" in file_lower:
            test_csv_path = full_path

print("Train images dir:", train_images_dir)
print("Test images dir:", test_images_dir)
print("Train CSV:", train_csv_path)
print("Test CSV:", test_csv_path)
```

Train images dir: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Original_Images/Training Set
Test images dir: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Original_Images/Testing Set
Train CSV: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Groundtruths/IDRiD_Disease_Grading_Training_Labels.csv
Test CSV: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Groundtruths/IDRiD_Disease_Grading_Testing_Labels.csv

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

import numpy as np
import pandas as pd

from PIL import Image
from tqdm import tqdm

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

```
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 (
    confusion_matrix, accuracy_score, precision_score,
    recall_score, f1_score, classification_report
)

print("Libraries imported successfully")
```

Libraries imported successfully

```
train_images_dir = None
test_images_dir = None
train_csv_path = None
test_csv_path = None

for root, dirs, files in os.walk(path):

    if "Disease_Grading" in root and "Original_Images" in root:
        if os.path.basename(root) == "Training Set":
            train_images_dir = root
        elif os.path.basename(root) == "Testing Set":
            test_images_dir = root

    for file in files:
        file_lower = file.lower()
        full_path = os.path.join(root, file)

        if "disease grading" in file_lower and "training labels" in file_lower:
            train_csv_path = full_path

        if "disease grading" in file_lower and "testing labels" in file_lower:
            test_csv_path = full_path

print("Train images dir:", train_images_dir)
print("Test images dir:", test_images_dir)
print("Train CSV:", train_csv_path)
print("Test CSV:", test_csv_path)
```

Train images dir: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Original_Images/Training Set
 Test images dir: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Original_Images/Testing Set
 Train CSV: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Groundtruths/IDRiD_Disease_Grading_Training_Labels.csv
 Test CSV: /kaggle/input/diabetic-retinopathy-image-dataset/Disease_Grading/Groundtruths/IDRiD_Disease_Grading_Testing_Labels.csv

```
train_df = pd.read_csv(train_csv_path)
test_df = pd.read_csv(test_csv_path)

train_df.columns = train_df.columns.str.strip()
test_df.columns = test_df.columns.str.strip()

print("Train columns:")
print(train_df.columns.tolist())

print("\nTrain sample:")
display(train_df.head())

print("\nTest sample:")
display(test_df.head())
```

Train columns:

['Image name', 'Retinopathy grade', 'Risk of macular edema', 'Unnamed: 3', 'Unnamed: 4', 'Unnamed: 5', 'Unnamed: 6', 'Unnamed: 7', 'Unnamed: 8', 'Unnamed: 9', 'Unnamed: 10', 'Unnamed: 11']

Train sample:

	Image name	Retinopathy grade	Risk of macular edema	Unnamed: 3	Unnamed: 4	Unnamed: 5	Unnamed: 6	Unnamed: 7	Unnamed: 8	Unnamed: 9	Unnamed: 10	Unnamed: 11
0	IDRiD_001	3	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	IDRiD_002	3	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	IDRiD_003	2	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	IDRiD_004	3	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	IDRiD_005	4	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Test sample:

	Image name	Retinopathy grade	Risk of macular edema
0	IDRiD_001	4	0
1	IDRiD_002	4	1
2	IDRiD_003	4	0
3	IDRiD_004	4	0

```
image_col = "Image name"
label_col = "Retinopathy grade"
```

```
# نخلي فقط الأعمدة المهمة
```

```
train_df = train_df[[image_col, label_col]].copy()
test_df = test_df[[image_col, label_col]].copy()
```

```
print("Training label distribution:")
print(train_df[label_col].value_counts().sort_index())
```

```
print("\nTesting label distribution:")
print(test_df[label_col].value_counts().sort_index())
```

Training label distribution:

Retinopathy grade

```
0    134
1     20
2    136
3     74
4     49
```

Name: count, dtype: int64

Testing label distribution:

Retinopathy grade

```
0     34
1      5
2     32
3     19
4     13
```

Name: count, dtype: int64

```
class_names = {
    0: "grade_0_no_DR",
    1: "grade_1_mild",
    2: "grade_2_moderate",
    3: "grade_3_severe",
    4: "grade_4_proliferative_DR"
}
```

```
organized_dir = "/content/dataset_organized"
train_full_dir = os.path.join(organized_dir, "train_full")
test_raw_dir = os.path.join(organized_dir, "test")
```

```
if os.path.exists(organized_dir):
    shutil.rmtree(organized_dir)
```

```
os.makedirs(train_full_dir, exist_ok=True)
os.makedirs(test_raw_dir, exist_ok=True)
```

```
def find_image_file(images_dir, image_name):
    base_name = os.path.splitext(str(image_name))[0]
```

```
    for ext in [".jpg", ".jpeg", ".png", ".bmp", ".tiff", ".tif"]:
        img_path = os.path.join(images_dir, base_name + ext)
        if os.path.exists(img_path):
            return img_path
```

```

        return None

def organize_images(df, images_dir, output_dir):
    missing = 0

    for _, row in tqdm(df.iterrows(), total=len(df)):
        img_name = row[image_col]
        label = int(row[label_col])

        class_folder = class_names[label]
        src = find_image_file(images_dir, img_name)

        if src is None:
            missing += 1
            print("Missing image:", img_name)
            continue

        dst_folder = os.path.join(output_dir, class_folder)
        os.makedirs(dst_folder, exist_ok=True)

        dst = os.path.join(dst_folder, os.path.basename(src))
        shutil.copy(src, dst)

    print("Missing images:", missing)

organize_images(train_df, train_images_dir, train_full_dir)
organize_images(test_df, test_images_dir, test_raw_dir)

print("Images organized successfully.")

```

```

100%|██████████| 413/413 [00:03<00:00, 103.76it/s]
Missing images: 0
100%|██████████| 103/103 [00:00<00:00, 103.65it/s]Missing images: 0
Images organized successfully.

```

```

for folder in [train_full_dir, test_raw_dir]:
    print("\nFolder:", folder)

    for class_name in os.listdir(folder):
        class_path = os.path.join(folder, class_name)
        if os.path.isdir(class_path):
            print(class_name, ":", len(os.listdir(class_path)))

```

```

Folder: /content/dataset_organized/train_full
grade_3_severe : 74
grade_2_moderate : 136
grade_4_proliferative_DR : 49
grade_1_mild : 20
grade_0_no_DR : 134

Folder: /content/dataset_organized/test
grade_3_severe : 19
grade_2_moderate : 32
grade_4_proliferative_DR : 13
grade_1_mild : 5
grade_0_no_DR : 34

```

```

from PIL import Image
from tqdm import tqdm
import os
import shutil

organized_dir = "/content/dataset_organized"
output_root = "/content/output"
resize_size = (256, 256)

image_extensions = (".jpg", ".jpeg", ".png", ".bmp", ".tiff", ".tif")

if os.path.exists(output_root):
    shutil.rmtree(output_root)

os.makedirs(output_root, exist_ok=True)

for split_name in ["train_full", "test"]:
    split_path = os.path.join(organized_dir, split_name)

    for class_name in os.listdir(split_path):
        class_path = os.path.join(split_path, class_name)

        if not os.path.isdir(class_path):

```

```

continue

output_class_path = os.path.join(output_root, split_name, class_name)
os.makedirs(output_class_path, exist_ok=True)

for image_name in tqdm(os.listdir(class_path), desc=f"{split_name}/{class_name}"):
    if image_name.lower().endswith(image_extensions):
        input_image_path = os.path.join(class_path, image_name)
        output_image_name = os.path.splitext(image_name)[0] + ".jpg"
        output_image_path = os.path.join(output_class_path, output_image_name)

        try:
            with Image.open(input_image_path) as img:
                img = img.convert("RGB")
                img = img.resize(resize_size)
                img.save(output_image_path, "JPEG")
        except Exception as e:
            print(f"Failed to process {input_image_path}: {e}")

print("Image resizing completed.")

train_full/grade_3_severe: 100%|██████████| 74/74 [00:20<00:00, 3.69it/s]
train_full/grade_2_moderate: 100%|██████████| 136/136 [00:35<00:00, 3.78it/s]
train_full/grade_4_proliferative_DR: 100%|██████████| 49/49 [00:12<00:00, 3.97it/s]
train_full/grade_1_mild: 100%|██████████| 20/20 [00:05<00:00, 3.39it/s]
train_full/grade_0_no_DR: 100%|██████████| 134/134 [00:33<00:00, 3.97it/s]
test/grade_3_severe: 100%|██████████| 19/19 [00:05<00:00, 3.24it/s]
test/grade_2_moderate: 100%|██████████| 32/32 [00:07<00:00, 4.32it/s]
test/grade_4_proliferative_DR: 100%|██████████| 13/13 [00:04<00:00, 2.93it/s]
test/grade_1_mild: 100%|██████████| 5/5 [00:01<00:00, 4.30it/s]
test/grade_0_no_DR: 100%|██████████| 34/34 [00:08<00:00, 4.17it/s]Image resizing completed.

```

```

for split_name in ["train_full", "test"]:
    print(f"\n{split_name.upper()}:")
    split_path = os.path.join("/content/output", split_name)

    for class_name in os.listdir(split_path):
        class_path = os.path.join(split_path, class_name)
        print(class_name, ":", len(os.listdir(class_path)))

```

```

TRAIN_FULL:
grade_3_severe : 74
grade_2_moderate : 136
grade_4_proliferative_DR : 49
grade_1_mild : 20
grade_0_no_DR : 134

TEST:
grade_3_severe : 19
grade_2_moderate : 32
grade_4_proliferative_DR : 13
grade_1_mild : 5
grade_0_no_DR : 34

```

```

from sklearn.model_selection import train_test_split
import shutil
import os

original_train_dir = "/content/output/train_full"
original_test_dir = "/content/output/test"

base_dir = "/content/new_dataset"

if os.path.exists(base_dir):
    shutil.rmtree(base_dir)

train_dir = os.path.join(base_dir, "train")
validation_dir = os.path.join(base_dir, "validation")
test_dir = os.path.join(base_dir, "test")

for d in [train_dir, validation_dir, test_dir]:
    os.makedirs(d, exist_ok=True)

validation_ratio = 0.2

def split_train_validation(class_dir, dest_base_dir):
    all_images = [
        os.path.join(class_dir, f)
        for f in os.listdir(class_dir)
        if os.path.isfile(os.path.join(class_dir, f))
    ]

```

```

train_images, val_images = train_test_split(
    all_images,
    test_size=validation_ratio,
    random_state=42
)

class_name = os.path.basename(class_dir)

for images, folder_name in zip([train_images, val_images], ["train", "validation"]):
    dest_dir = os.path.join(dest_base_dir, folder_name, class_name)
    os.makedirs(dest_dir, exist_ok=True)

    for file in images:
        shutil.copy(file, dest_dir)

for class_name in os.listdir(original_train_dir):
    class_dir = os.path.join(original_train_dir, class_name)
    if os.path.isdir(class_dir):
        split_train_validation(class_dir, base_dir)

for class_name in os.listdir(original_test_dir):
    src_class_dir = os.path.join(original_test_dir, class_name)
    dst_class_dir = os.path.join(test_dir, class_name)

    if os.path.isdir(src_class_dir):
        os.makedirs(dst_class_dir, exist_ok=True)
        for file in os.listdir(src_class_dir):
            shutil.copy(os.path.join(src_class_dir, file), dst_class_dir)

print("Dataset split completed.")

```

Dataset split completed.

```

from torchvision import datasets
import os

def get_classes(data_dir):
    all_data = datasets.ImageFolder(data_dir)
    return all_data.classes

classes = get_classes("/content/new_dataset/train")
num_classes = len(classes)

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

for split in ["train", "validation", "test"]:
    print(f"\n{split.upper()}:")
    split_path = os.path.join("/content/new_dataset", split)

    for class_name in os.listdir(split_path):
        class_path = os.path.join(split_path, class_name)
        print(class_name, ":", len(os.listdir(class_path)))

```

Classes: ['grade_0_no_DR', 'grade_1_mild', 'grade_2_moderate', 'grade_3_severe', 'grade_4_proliferative_DR']
 Number of classes: 5

TRAIN:
 grade_3_severe : 59
 grade_2_moderate : 108
 grade_4_proliferative_DR : 39
 grade_1_mild : 16
 grade_0_no_DR : 107

VALIDATION:
 grade_3_severe : 15
 grade_2_moderate : 28
 grade_4_proliferative_DR : 10
 grade_1_mild : 4
 grade_0_no_DR : 27

TEST:
 grade_3_severe : 19
 grade_2_moderate : 32
 grade_4_proliferative_DR : 13
 grade_1_mild : 5
 grade_0_no_DR : 34

```

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

```

```

def get_data_loaders(data_dir, batch_size, train=False):

    if train:
        transform = T.Compose([
            T.RandomHorizontalFlip(),
            T.RandomVerticalFlip(),
            T.RandomRotation(10),
            T.Resize((224, 224)),
            T.ToTensor(),
            T.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

        train_data = datasets.ImageFolder(
            os.path.join(data_dir, "train"),
            transform=transform
        )

        train_loader = DataLoader(
            train_data,
            batch_size=batch_size,
            shuffle=True,
            num_workers=2
        )

        return train_loader, len(train_data)

    else:
        transform = T.Compose([
            T.Resize((224, 224)),
            T.ToTensor(),
            T.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

        val_data = datasets.ImageFolder(
            os.path.join(data_dir, "validation"),
            transform=transform
        )

        test_data = datasets.ImageFolder(
            os.path.join(data_dir, "test"),
            transform=transform
        )

        val_loader = DataLoader(
            val_data,
            batch_size=batch_size,
            shuffle=False,
            num_workers=2
        )

        test_loader = DataLoader(
            test_data,
            batch_size=batch_size,
            shuffle=False,
            num_workers=2
        )

        return val_loader, test_loader, len(val_data), len(test_data)

```

```

dataset_path = "/content/new_dataset"
batch_size = 16

train_loader, train_data_len = get_data_loaders(dataset_path, batch_size, train=True)
val_loader, test_loader, valid_data_len, test_data_len = get_data_loaders(dataset_path, batch_size, train=False)

classes = get_classes("/content/new_dataset/train")
num_classes = len(classes)

dataloaders = {
    "train": train_loader,
    "val": val_loader
}

dataset_sizes = {
    "train": train_data_len,
    "val": valid_data_len
}

```

```

}

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

print("Device:", device)
print("Classes:", classes)
print("Number of classes:", num_classes)
print("Train:", train_data_len)
print("Validation:", valid_data_len)
print("Test:", test_data_len)

```

```

Device: cpu
Classes: ['grade_0_no_DR', 'grade_1_mild', 'grade_2_moderate', 'grade_3_severe', 'grade_4_proliferative_DR']
Number of classes: 5
Train: 329
Validation: 84
Test: 103

```

```

import copy
from tqdm import tqdm
from torch import nn, optim
from torchvision import models

def train_model(model, criterion, optimizer, scheduler, num_epochs=5):

    history = {
        "train_loss": [],
        "val_loss": [],
        "train_acc": [],
        "val_acc": []
    }

    best_val_acc = 0.0
    best_model_weights = copy.deepcopy(model.state_dict())

    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")
        print("-" * 10)

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

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

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in tqdm(dataloaders[phase]):
                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)
                    _, preds = torch.max(outputs, 1)

                if phase == "train":
                    loss.backward()
                    optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if phase == "train":
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print(f"{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}")

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

            if phase == "val" and epoch_acc > best_val_acc:
                best_val_acc = epoch_acc
                best_model_weights = copy.deepcopy(model.state_dict())

```

```
print()

model.load_state_dict(best_model_weights)
return model, history
```

```
import numpy as np

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)

# Calculate class weights
train_dataset_for_weights = datasets.ImageFolder("/content/new_dataset/train")
train_labels = [label for _, label in train_dataset_for_weights.samples]

class_counts = np.bincount(train_labels)
class_weights = len(train_labels) / (num_classes * class_counts)
class_weights = torch.tensor(class_weights, dtype=torch.float).to(device)

print("Class counts:", class_counts)
print("Class weights:", class_weights)

criterion = nn.CrossEntropyLoss(weight=class_weights).to(device)

optimizer = optim.AdamW(model.parameters(), lr=0.0005)

exp_lr_scheduler = optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.97
)

efficientnet_model, efficientnet_history = train_model(
    model,
    criterion,
    optimizer,
    exp_lr_scheduler,
    num_epochs=5
)
```

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

100%|██████████| 20.5M/20.5M [00:00<00:00, 96.5MB/s]
 Class counts: [107 16 108 59 39]
 Class weights: tensor([0.6150, 4.1125, 0.6093, 1.1153, 1.6872])
 Epoch 1/5

 100%|██████████| 21/21 [01:24<00:00, 4.01s/it]
 train Loss: 1.3857 Acc: 0.3891
 100%|██████████| 6/6 [00:05<00:00, 1.01it/s]
 val Loss: 1.1434 Acc: 0.5119

Epoch 2/5

 100%|██████████| 21/21 [01:26<00:00, 4.13s/it]
 train Loss: 0.9958 Acc: 0.5258
 100%|██████████| 6/6 [00:06<00:00, 1.03s/it]
 val Loss: 1.1165 Acc: 0.4643

Epoch 3/5

 100%|██████████| 21/21 [01:20<00:00, 3.85s/it]
 train Loss: 0.9249 Acc: 0.5866
 100%|██████████| 6/6 [00:05<00:00, 1.14it/s]
 val Loss: 1.1172 Acc: 0.4762

Epoch 4/5

 100%|██████████| 21/21 [01:26<00:00, 4.14s/it]
 train Loss: 0.7242 Acc: 0.6626
 100%|██████████| 6/6 [00:05<00:00, 1.06it/s]
 val Loss: 1.4323 Acc: 0.5238

Epoch 5/5

 100%|██████████| 21/21 [01:18<00:00, 3.75s/it]
 train Loss: 0.6861 Acc: 0.7325
 100%|██████████| 6/6 [00:05<00:00, 1.15it/s] val Loss: 1.4071 Acc: 0.5595

```

from sklearn.metrics import (
    confusion_matrix, accuracy_score, precision_score,
    recall_score, f1_score, classification_report
)

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

def evaluate_model(model_name, model, test_loader, classes):

    model.eval()

    y_true = []
    y_pred = []
    y_proba = []

    with torch.no_grad():
        for x, y in tqdm(test_loader):
            x = x.to(device)

            outputs = model(x)
            probabilities = torch.softmax(outputs, dim=1)
            preds = torch.argmax(probabilities, dim=1)

            y_true.extend(y.numpy())
            y_pred.extend(preds.cpu().numpy())
            y_proba.extend(probabilities.cpu().numpy())

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

    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"Model: {model_name}")
    print("Accuracy:", accuracy)
    print("Precision:", precision)
    print("Recall:", recall)
    print("F1-score:", f1)

    print("\nClassification Report:")
    print(classification_report(y_true, y_pred, target_names=classes, zero_division=0))

    cm = confusion_matrix(y_true, y_pred)

    plt.figure(figsize=(9, 7))
    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=classes,
        yticklabels=classes
    )
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(f"Confusion Matrix - {model_name}")
    plt.show()

    return accuracy, precision, recall, f1, y_true, y_pred, y_proba

```

```

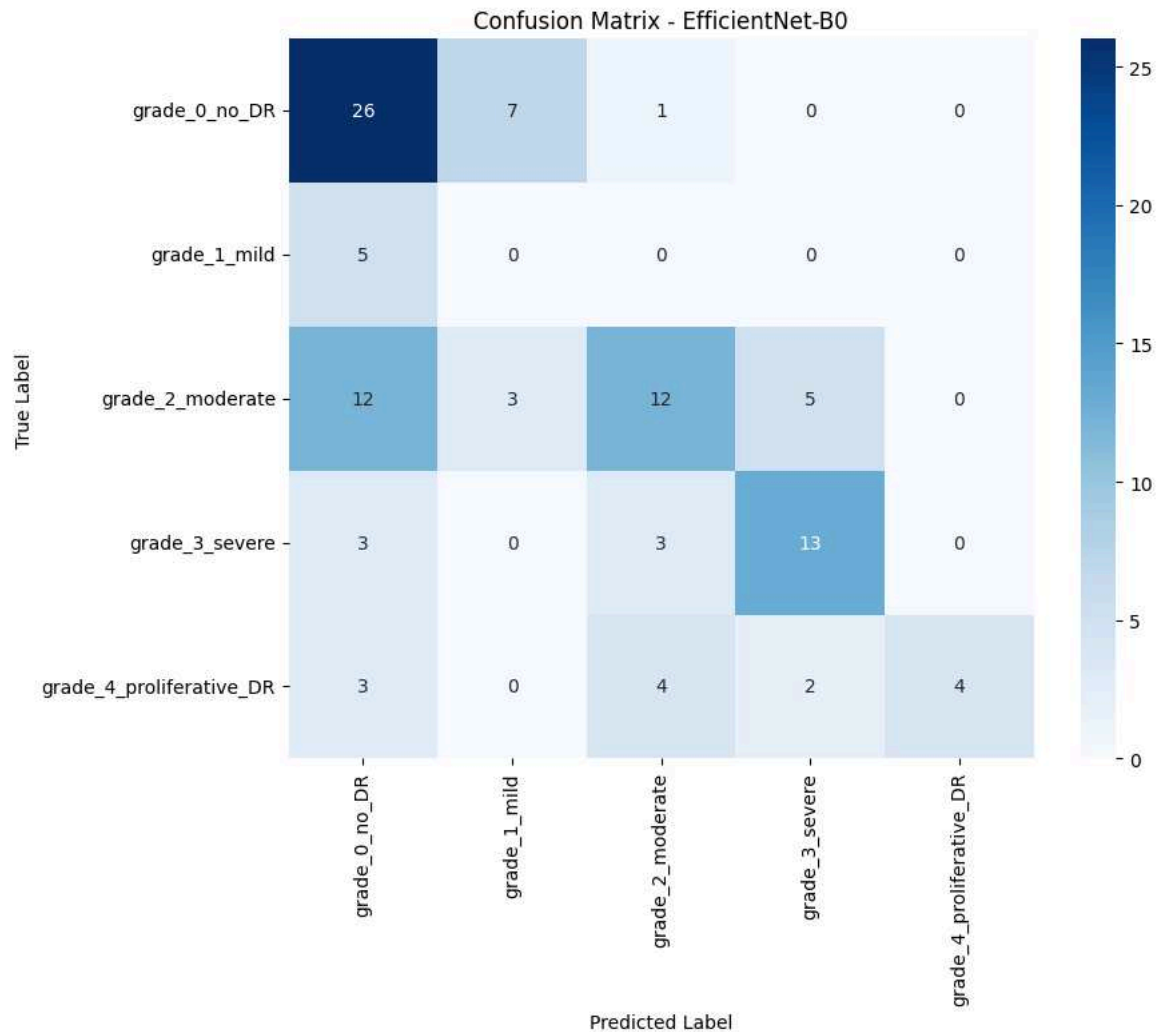
eff_results = evaluate_model(
    "EfficientNet-B0",
    efficientnet_model,
    test_loader,
    classes
)

```

100%|██████████| 7/7 [00:17<00:00, 2.56s/it]
 Model: EfficientNet-B0
 Accuracy: 0.5339805825242718
 Precision: 0.6076778284129186
 Recall: 0.5339805825242718
 F1-score: 0.5325703816893016

Classification Report:

	precision	recall	f1-score	support
grade_0_no_DR	0.53	0.76	0.63	34
grade_1_mild	0.00	0.00	0.00	5
grade_2_moderate	0.60	0.38	0.46	32
grade_3_severe	0.65	0.68	0.67	19
grade_4_proliferative_DR	1.00	0.31	0.47	13
accuracy			0.53	103
macro avg	0.56	0.43	0.45	103
weighted avg	0.61	0.53	0.53	103

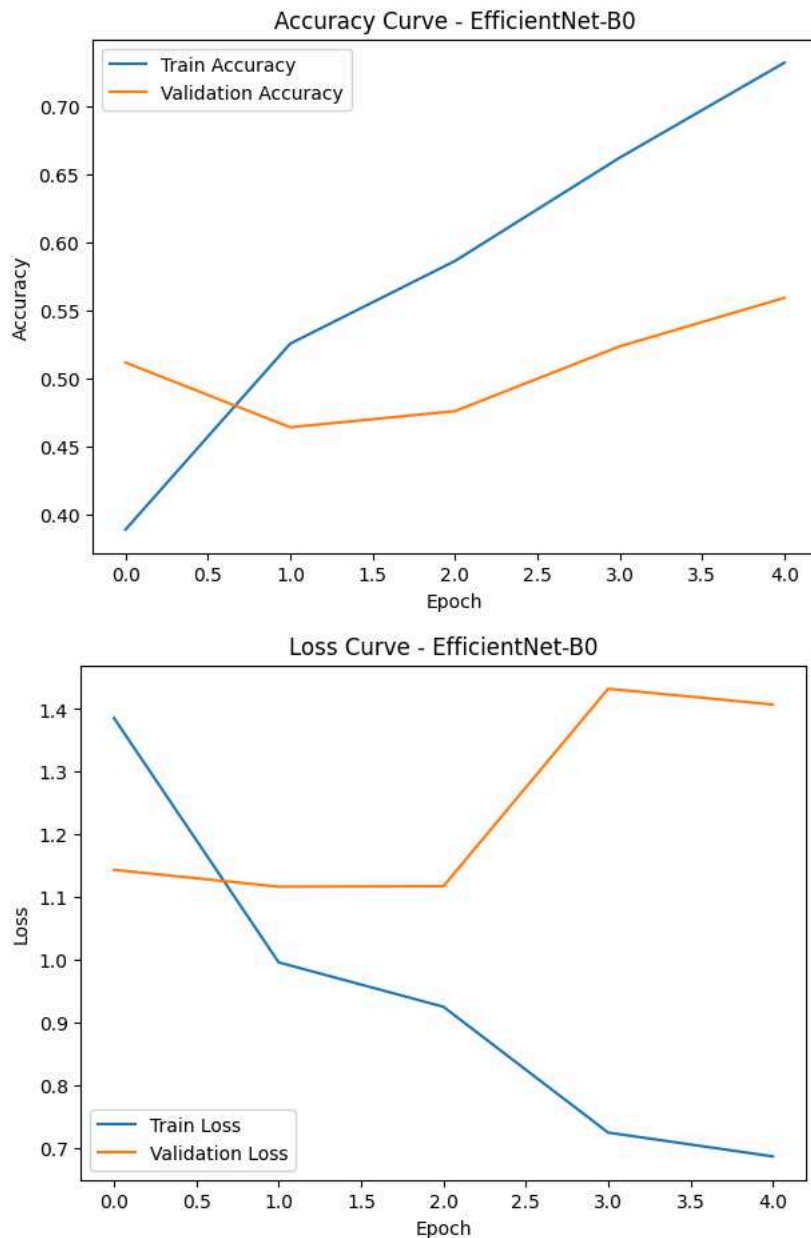


```
def plot_history(history, model_name):

    plt.figure(figsize=(7, 5))
    plt.plot(history["train_acc"], label="Train Accuracy")
    plt.plot(history["val_acc"], label="Validation Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title(f"Accuracy Curve - {model_name}")
    plt.legend()
    plt.show()

    plt.figure(figsize=(7, 5))
    plt.plot(history["train_loss"], label="Train Loss")
    plt.plot(history["val_loss"], label="Validation Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title(f"Loss Curve - {model_name}")
    plt.legend()
    plt.show()

plot_history(efficientnet_history, "EfficientNet-B0")
```



```
train_acc = efficientnet_history["train_acc"][-1]
val_acc = efficientnet_history["val_acc"][-1]

overfitting_gap = train_acc - val_acc
overfitting_percentage = overfitting_gap * 100

print("Final Train Accuracy:", train_acc)
print("Final Validation Accuracy:", val_acc)
print("Overfitting Gap:", overfitting_gap)
print("Overfitting Percentage:", overfitting_percentage, "%")
```

```
Final Train Accuracy: 0.7325227963525835
Final Validation Accuracy: 0.5595238095238095
Overfitting Gap: 0.17299898682877402
Overfitting Percentage: 17.2998986828774 %
```

```
best_epoch = np.argmax(efficientnet_history["val_acc"])

best_train_acc = efficientnet_history["train_acc"][best_epoch]
best_val_acc = efficientnet_history["val_acc"][best_epoch]

best_gap = best_train_acc - best_val_acc

print("Best Epoch:", best_epoch + 1)
print("Train Accuracy at Best Epoch:", best_train_acc)
print("Validation Accuracy at Best Epoch:", best_val_acc)
print("Overfitting Gap at Best Epoch:", best_gap)
print("Overfitting Percentage:", best_gap * 100, "%")
```

Best Epoch: 5
 Train Accuracy at Best Epoch: 0.7325227963525835
 Validation Accuracy at Best Epoch: 0.5595238095238095
 Overfitting Gap at Best Epoch: 0.17299898682877402
 Overfitting Percentage: 17.2998986828774 %

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms as T
import os

def get_data_loaders(data_dir, batch_size, train=False):

    if train:
        transform = T.Compose([
            T.RandomResizedCrop(224, scale=(0.75, 1.0)),
            T.RandomHorizontalFlip(),
            T.RandomRotation(15),
            T.ColorJitter(
                brightness=0.15,
                contrast=0.15,
                saturation=0.10
            ),
            T.ToTensor(),
            T.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

        train_data = datasets.ImageFolder(
            os.path.join(data_dir, "train"),
            transform=transform
        )

        train_loader = DataLoader(
            train_data,
            batch_size=batch_size,
            shuffle=True,
            num_workers=2
        )

        return train_loader, len(train_data)

    else:
        transform = T.Compose([
            T.Resize((224, 224)),
            T.ToTensor(),
            T.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

        val_data = datasets.ImageFolder(
            os.path.join(data_dir, "validation"),
            transform=transform
        )

        test_data = datasets.ImageFolder(
            os.path.join(data_dir, "test"),
            transform=transform
        )

        val_loader = DataLoader(
            val_data,
            batch_size=batch_size,
            shuffle=False,
            num_workers=2
        )

        test_loader = DataLoader(
            test_data,
            batch_size=batch_size,
            shuffle=False,
            num_workers=2
        )

        return val_loader, test_loader, len(val_data), len(test_data)
```

```

dataset_path = "/content/new_dataset"
batch_size = 16

train_loader, train_data_len = get_data_loaders(dataset_path, batch_size, train=True)
val_loader, test_loader, valid_data_len, test_data_len = get_data_loaders(dataset_path, batch_size, train=False)

classes = get_classes("/content/new_dataset/train")
num_classes = len(classes)

dataloaders = {
    "train": train_loader,
    "val": val_loader
}

dataset_sizes = {
    "train": train_data_len,
    "val": valid_data_len
}

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

print("Device:", device)
print("Train:", train_data_len)
print("Validation:", valid_data_len)
print("Test:", test_data_len)

```

```

Device: cpu
Train: 329
Validation: 84
Test: 103

```

```

import copy
from tqdm import tqdm

def train_model(model, criterion, optimizer, scheduler, num_epochs=15, patience=4):

    history = {
        "train_loss": [],
        "val_loss": [],
        "train_acc": [],
        "val_acc": []
    }

    best_val_acc = 0.0
    best_model_weights = copy.deepcopy(model.state_dict())
    epochs_without_improvement = 0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")
        print("-" * 10)

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

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

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in tqdm(dataloaders[phase]):
                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)
                    _, preds = torch.max(outputs, 1)

                if phase == "train":
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]

```

```

print(f"{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}")

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

if phase == "val":
    if epoch_acc > best_val_acc:
        best_val_acc = epoch_acc
        best_model_weights = copy.deepcopy(model.state_dict())
        epochs_without_improvement = 0
    else:
        epochs_without_improvement += 1

scheduler.step()
print()

if epochs_without_improvement >= patience:
    print("Early stopping triggered.")
    break

model.load_state_dict(best_model_weights)
return model, history

```

```

import numpy as np
from torch import nn, optim
from torchvision import models

model = models.efficientnet_b0(weights=models.EfficientNet_B0_Weights.DEFAULT)

# Freeze feature extractor
for param in model.features.parameters():
    param.requires_grad = False

# Replace classifier with Dropout
model.classifier = nn.Sequential(
    nn.Dropout(p=0.5),
    nn.Linear(model.classifier[1].in_features, num_classes)
)

model = model.to(device)

# Class weights
بسيب عدم توازن الداتا
train_dataset_for_weights = datasets.ImageFolder("/content/new_dataset/train")
train_labels = [label for _, label in train_dataset_for_weights.samples]

class_counts = np.bincount(train_labels)
class_weights = len(train_labels) / (num_classes * class_counts)
class_weights = torch.tensor(class_weights, dtype=torch.float).to(device)

print("Class counts:", class_counts)
print("Class weights:", class_weights)

criterion = nn.CrossEntropyLoss(weight=class_weights).to(device)

optimizer = optim.AdamW(
    model.classifier.parameters(),
    lr=0.0003,
    weight_decay=0.01
)

exp_lr_scheduler = optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.85
)

efficientnet_model, efficientnet_history = train_model(
    model,
    criterion,
    optimizer,
    exp_lr_scheduler,
    num_epochs=15,
    patience=4
)

```

```

Class weights: tensor([0.6150, 4.1125, 0.6093, 1.1153, 1.6872])
Epoch 1/15
-----
100%|██████████| 21/21 [00:32<00:00, 1.54s/it]
train Loss: 1.5875 Acc: 0.2553
100%|██████████| 6/6 [00:05<00:00, 1.19it/s]

```

```
Epoch 2/15
-----
100%|██████████| 21/21 [00:29<00:00, 1.42s/it]
train Loss: 1.5466 Acc: 0.3343
100%|██████████| 6/6 [00:06<00:00, 1.02s/it]
val Loss: 1.5317 Acc: 0.3690
```

```
Epoch 3/15
-----
100%|██████████| 21/21 [00:29<00:00, 1.40s/it]
train Loss: 1.4793 Acc: 0.4316
100%|██████████| 6/6 [00:05<00:00, 1.00it/s]
val Loss: 1.5039 Acc: 0.4286
```

```
Epoch 4/15
-----
100%|██████████| 21/21 [00:29<00:00, 1.38s/it]
train Loss: 1.4665 Acc: 0.3982
100%|██████████| 6/6 [00:04<00:00, 1.22it/s]
val Loss: 1.4777 Acc: 0.4881
```

```
Epoch 5/15
-----
100%|██████████| 21/21 [00:29<00:00, 1.39s/it]
train Loss: 1.4343 Acc: 0.4498
100%|██████████| 6/6 [00:06<00:00, 1.12s/it]
val Loss: 1.4427 Acc: 0.4762
```

```
Epoch 6/15
-----
100%|██████████| 21/21 [00:29<00:00, 1.39s/it]
train Loss: 1.4137 Acc: 0.4650
100%|██████████| 6/6 [00:05<00:00, 1.18it/s]
val Loss: 1.4279 Acc: 0.4405
```

```
Epoch 7/15
-----
100%|██████████| 21/21 [00:29<00:00, 1.42s/it]
train Loss: 1.3929 Acc: 0.4924
100%|██████████| 6/6 [00:05<00:00, 1.06it/s]
val Loss: 1.4243 Acc: 0.4286
```

```
Epoch 8/15
-----
100%|██████████| 21/21 [00:29<00:00, 1.40s/it]
train Loss: 1.3497 Acc: 0.5046
100%|██████████| 6/6 [00:05<00:00, 1.04it/s]
val Loss: 1.4061 Acc: 0.4405
```

Early stopping triggered.

```
import numpy as np
from torch import nn, optim
from torchvision import models

model = models.efficientnet_b0(weights=models.EfficientNet_B0_Weights.DEFAULT)

# Freeze feature extractor
for param in model.features.parameters():
    param.requires_grad = False

# Replace classifier with Dropout
model.classifier = nn.Sequential(
    nn.Dropout(p=0.5),
    nn.Linear(model.classifier[1].in_features, num_classes)
)

model = model.to(device)

# Class weights
# بسبب عدم توازن الداتا
train_dataset_for_weights = datasets.ImageFolder("/content/new_dataset/train")
train_labels = [label for _, label in train_dataset_for_weights.samples]

class_counts = np.bincount(train_labels)
class_weights = len(train_labels) / (num_classes * class_counts)
class_weights = torch.tensor(class_weights, dtype=torch.float).to(device)

print("Class counts:", class_counts)
print("Class weights:", class_weights)

criterion = nn.CrossEntropyLoss(weight=class_weights).to(device)

optimizer = optim.AdamW(
    model.classifier.parameters(),
    lr=0.0003,
    weight_decay=0.01
```

```

exp_lr_scheduler = optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.85
)

efficientnet_model, efficientnet_history = train_model(
    model,
    criterion,
    optimizer,
    exp_lr_scheduler,
    num_epochs=15,
    patience=4
)

```

val Loss: 1.4305 Acc: 0.4048

Epoch 8/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.41s/it]
train Loss: 1.3813 Acc: 0.4286
100%|██████████| 6/6 [00:05<00:00, 1.18it/s]
val Loss: 1.4283 Acc: 0.3929

```

Epoch 9/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.41s/it]
train Loss: 1.3555 Acc: 0.4681
100%|██████████| 6/6 [00:06<00:00, 1.02s/it]
val Loss: 1.4154 Acc: 0.4167

```

Epoch 10/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.40s/it]
train Loss: 1.3319 Acc: 0.5076
100%|██████████| 6/6 [00:05<00:00, 1.19it/s]
val Loss: 1.4059 Acc: 0.4048

```

Epoch 11/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.43s/it]
train Loss: 1.3051 Acc: 0.4863
100%|██████████| 6/6 [00:05<00:00, 1.08it/s]
val Loss: 1.3786 Acc: 0.4405

```

Epoch 12/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.41s/it]
train Loss: 1.3012 Acc: 0.5076
100%|██████████| 6/6 [00:05<00:00, 1.00it/s]
val Loss: 1.3859 Acc: 0.4524

```

Epoch 13/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.41s/it]
train Loss: 1.3126 Acc: 0.4863
100%|██████████| 6/6 [00:05<00:00, 1.17it/s]
val Loss: 1.3698 Acc: 0.4405

```

Epoch 14/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.40s/it]
train Loss: 1.2577 Acc: 0.5167
100%|██████████| 6/6 [00:06<00:00, 1.00s/it]
val Loss: 1.3600 Acc: 0.4405

```

Epoch 15/15

```

-----
100%|██████████| 21/21 [00:29<00:00, 1.39s/it]
train Loss: 1.2981 Acc: 0.4833
100%|██████████| 6/6 [00:05<00:00, 1.20it/s]val Loss: 1.3503 Acc: 0.4643

```

```

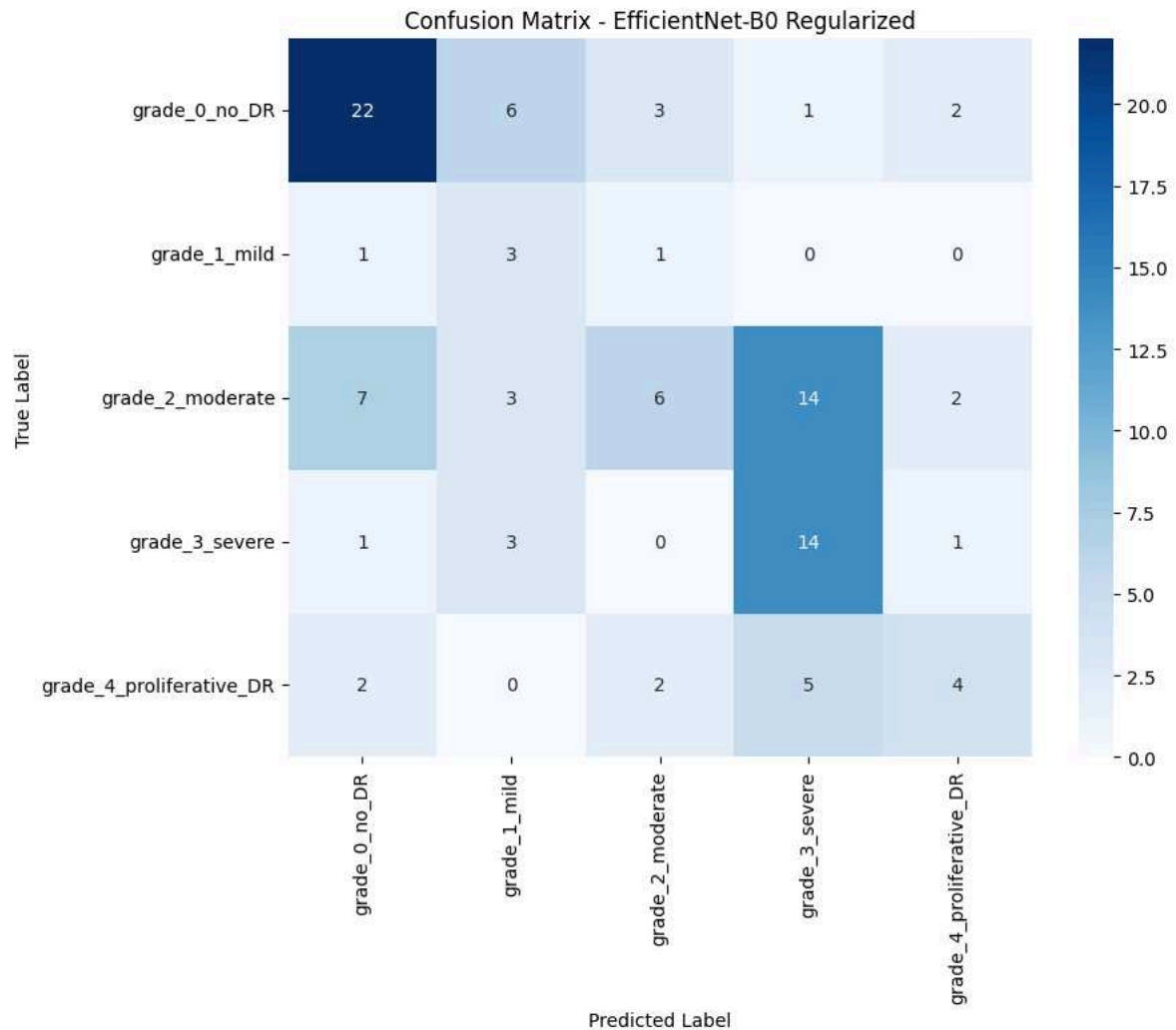
eff_results = evaluate_model(
    "EfficientNet-B0 Regularized",
    efficientnet_model,
    test_loader,
    classes
)

```

100%|██████████| 7/7 [00:06<00:00, 1.01it/s]
 Model: EfficientNet-B0 Regularized
 Accuracy: 0.47572815533980584
 Precision: 0.5171647947204772
 Recall: 0.47572815533980584
 F1-score: 0.45942368458793476

Classification Report:

	precision	recall	f1-score	support
grade_0_no_DR	0.67	0.65	0.66	34
grade_1_mild	0.20	0.60	0.30	5
grade_2_moderate	0.50	0.19	0.27	32
grade_3_severe	0.41	0.74	0.53	19
grade_4_proliferative_DR	0.44	0.31	0.36	13
accuracy			0.48	103
macro avg	0.44	0.50	0.42	103
weighted avg	0.52	0.48	0.46	103



```
best_epoch = np.argmax(efficientnet_history["val_acc"])

best_train_acc = efficientnet_history["train_acc"][best_epoch]
best_val_acc = efficientnet_history["val_acc"][best_epoch]

overfitting_gap = best_train_acc - best_val_acc

print("Best Epoch:", best_epoch + 1)
print("Best Train Accuracy:", best_train_acc)
print("Best Validation Accuracy:", best_val_acc)
print("Overfitting Gap:", overfitting_gap)
print("Overfitting Percentage:", overfitting_gap * 100, "%")
```

```
Best Epoch: 15
Best Train Accuracy: 0.48328267477203646
Best Validation Accuracy: 0.4642857142857143
Overfitting Gap: 0.018996960486322156
Overfitting Percentage: 1.8996960486322156 %
```

```
# Unfreeze last blocks of EfficientNet only
for param in efficientnet_model.features[-2:].parameters():
    param.requires_grad = True

# Keep earlier layers frozen
for param in efficientnet_model.features[:-2].parameters():
    param.requires_grad = False

print("Last EfficientNet blocks are now trainable.")
```

Last EfficientNet blocks are now trainable.

```
train_dataset_for_weights = datasets.ImageFolder("/content/new_dataset/train")
train_labels = [label for _, label in train_dataset_for_weights.samples]

class_counts = np.bincount(train_labels)
class_weights = len(train_labels) / (num_classes * class_counts)
class_weights = torch.tensor(class_weights, dtype=torch.float).to(device)

print("Class counts:", class_counts)
print("Class weights:", class_weights)
```

Class counts: [107 16 108 59 39]
Class weights: tensor([0.6150, 4.1125, 0.6093, 1.1153, 1.6872])

```
criterion = nn.CrossEntropyLoss(
    weight=class_weights,
    label_smoothing=0.05
).to(device)

optimizer = optim.AdamW(
    [
        {"params": efficientnet_model.features[-2:].parameters(), "lr": 0.00001},
        {"params": efficientnet_model.classifier.parameters(), "lr": 0.0001}
    ],
    weight_decay=0.005
)

exp_lr_scheduler = optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.85
)

efficientnet_finetuned_model, efficientnet_finetuned_history = train_model(
    efficientnet_model,
    criterion,
    optimizer,
    exp_lr_scheduler,
    num_epochs=10,
    patience=4
)
```

Epoch 2/10

100%|██████████| 21/21 [00:33<00:00, 1.58s/it]
train Loss: 1.3715 Acc: 0.5471
100%|██████████| 6/6 [00:05<00:00, 1.14it/s]
val Loss: 1.4452 Acc: 0.4643

Epoch 3/10

100%|██████████| 21/21 [00:32<00:00, 1.56s/it]
train Loss: 1.3544 Acc: 0.5350
100%|██████████| 6/6 [00:05<00:00, 1.10it/s]
val Loss: 1.4340 Acc: 0.5000

Epoch 4/10

100%|██████████| 21/21 [00:32<00:00, 1.54s/it]
train Loss: 1.3348 Acc: 0.5258
100%|██████████| 6/6 [00:06<00:00, 1.03s/it]
val Loss: 1.4264 Acc: 0.4762

Epoch 5/10

100%|██████████| 21/21 [00:33<00:00, 1.59s/it]

```
100%|██████████| 21/21 [00:31<00:00, 1.51s/it]
train Loss: 1.3170 Acc: 0.5380
100%|██████████| 6/6 [00:05<00:00, 1.11it/s]
val Loss: 1.4154 Acc: 0.4524
```

Epoch 7/10

```
-----
100%|██████████| 21/21 [00:33<00:00, 1.57s/it]
train Loss: 1.3032 Acc: 0.5319
100%|██████████| 6/6 [00:05<00:00, 1.16it/s]
val Loss: 1.4145 Acc: 0.4286
```

Epoch 8/10

```
-----
100%|██████████| 21/21 [00:32<00:00, 1.55s/it]
train Loss: 1.2857 Acc: 0.5471
100%|██████████| 6/6 [00:05<00:00, 1.12it/s]
val Loss: 1.4103 Acc: 0.5000
```

Epoch 9/10

```
-----
100%|██████████| 21/21 [00:31<00:00, 1.51s/it]
train Loss: 1.3257 Acc: 0.5289
100%|██████████| 6/6 [00:06<00:00, 1.02s/it]val Loss: 1.4177 Acc: 0.5000
```

Early stopping triggered.

```
import numpy as np
import torch
from torch import nn, optim
from torchvision import models, datasets
import copy
from tqdm import tqdm

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
```

Device: cuda

```
import os

print("Does /content/new_dataset exist?", os.path.exists("/content/new_dataset"))
print("Does train folder exist?", os.path.exists("/content/new_dataset/train"))

if os.path.exists("/content"):
    print(os.listdir("/content"))
```

```
Does /content/new_dataset exist? False
Does train folder exist? False
['.config', 'sample_data']
```

```
# =====
# Rebuild dataset from zero
# =====

import os
import shutil
import warnings
warnings.filterwarnings("ignore")

import kagglehub
import pandas as pd
from PIL import Image
from tqdm import tqdm
from sklearn.model_selection import train_test_split

# 1) Download dataset
path = kagglehub.dataset_download("dankok/diabetic-retinopathy-image-dataset")
print("Dataset path:", path)

# 2) Find correct folders and CSV files
train_images_dir = None
test_images_dir = None
train_csv_path = None
test_csv_path = None

for root, dirs, files in os.walk(path):

    if "Disease_Grading" in root and "Original_Images" in root:
        if os.path.basename(root) == "Training Set":
            train_images_dir = root
        elif os.path.basename(root) == "Testing Set":
            test_images_dir = root
```

```

for file in files:
    file_lower = file.lower()
    full_path = os.path.join(root, file)

    if "disease grading" in file_lower and "training labels" in file_lower:
        train_csv_path = full_path

    if "disease grading" in file_lower and "testing labels" in file_lower:
        test_csv_path = full_path

print("Train images dir:", train_images_dir)
print("Test images dir:", test_images_dir)
print("Train CSV:", train_csv_path)
print("Test CSV:", test_csv_path)

# 3) Read labels
train_df = pd.read_csv(train_csv_path)
test_df = pd.read_csv(test_csv_path)

```

Downloading to /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/1.archive...
 100%|██████████| 408M/408M [00:04<00:00, 95.7MB/s]Extracting files...

Dataset path: /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/versions/1
 Train images dir: /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/versions/1/Disease_Grading/Origins
 Test images dir: /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/versions/1/Disease_Grading/Origins
 Train CSV: /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/versions/1/Disease_Grading/Groundtruths
 Test CSV: /root/.cache/kagglehub/datasets/dankok/diabetic-retinopathy-image-dataset/versions/1/Disease_Grading/Groundtruths

```

# 3) Read labels
train_df = pd.read_csv(train_csv_path)
test_df = pd.read_csv(test_csv_path)

train_df.columns = train_df.columns.str.strip()
test_df.columns = test_df.columns.str.strip()

image_col = "Image name"
label_col = "Retinopathy grade"

train_df = train_df[[image_col, label_col]].copy()
test_df = test_df[[image_col, label_col]].copy()

print("Training distribution:")
print(train_df[label_col].value_counts().sort_index())

print("\nTesting distribution:")
print(test_df[label_col].value_counts().sort_index())

# 4) Organize images into class folders
class_names = {
    0: "grade_0_no_DR",
    1: "grade_1_mild",
    2: "grade_2_moderate",
    3: "grade_3_severe",
    4: "grade_4_proliferative_DR"
}

organized_dir = "/content/dataset_organized"
train_full_dir = os.path.join(organized_dir, "train_full")
test_raw_dir = os.path.join(organized_dir, "test")

if os.path.exists(organized_dir):
    shutil.rmtree(organized_dir)

os.makedirs(train_full_dir, exist_ok=True)
os.makedirs(test_raw_dir, exist_ok=True)

def find_image_file(images_dir, image_name):
    base_name = os.path.splitext(str(image_name))[0]
    for ext in [".jpg", ".jpeg", ".png", ".bmp", ".tiff", ".tif"]:
        img_path = os.path.join(images_dir, base_name + ext)
        if os.path.exists(img_path):
            return img_path
    return None

def organize_images(df, images_dir, output_dir):
    missing = 0

    for _, row in tqdm(df.iterrows(), total=len(df)):
        img_name = row[image_col]

```

```

label = int(row[label_col])

class_folder = class_names[label]
src = find_image_file(images_dir, img_name)

if src is None:
    missing += 1
    print("Missing image:", img_name)
    continue

dst_folder = os.path.join(output_dir, class_folder)
os.makedirs(dst_folder, exist_ok=True)

dst = os.path.join(dst_folder, os.path.basename(src))
shutil.copy(src, dst)

print("Missing images:", missing)

organize_images(train_df, train_images_dir, train_full_dir)
organize_images(test_df, test_images_dir, test_raw_dir)

print("Images organized successfully.")

# 5) Resize images
output_root = "/content/output"
resize_size = (256, 256)
image_extensions = (".jpg", ".jpeg", ".png", ".bmp", ".tiff", ".tif")

if os.path.exists(output_root):
    shutil.rmtree(output_root)

os.makedirs(output_root, exist_ok=True)

for split_name in ["train_full", "test"]:
    split_path = os.path.join(organized_dir, split_name)

    for class_name in os.listdir(split_path):
        class_path = os.path.join(split_path, class_name)

        if not os.path.isdir(class_path):
            continue

        output_class_path = os.path.join(output_root, split_name, class_name)
        os.makedirs(output_class_path, exist_ok=True)

        for image_name in tqdm(os.listdir(class_path), desc=f"{split_name}/{class_name}"):
            if image_name.lower().endswith(image_extensions):
                input_image_path = os.path.join(class_path, image_name)
                output_image_name = os.path.splitext(image_name)[0] + ".jpg"
                output_image_path = os.path.join(output_class_path, output_image_name)

                try:
                    with Image.open(input_image_path) as img:
                        img = img.convert("RGB")
                        img = img.resize(resize_size)
                        img.save(output_image_path, "JPEG")
                except Exception as e:
                    print(f"Failed to process {input_image_path}: {e}")

print("Image resizing completed.")

# 6) Split train_full into train and validation, keep official test
original_train_dir = "/content/output/train_full"
original_test_dir = "/content/output/test"

base_dir = "/content/new_dataset"

if os.path.exists(base_dir):
    shutil.rmtree(base_dir)

train_dir = os.path.join(base_dir, "train")
validation_dir = os.path.join(base_dir, "validation")
test_dir = os.path.join(base_dir, "test")

for d in [train_dir, validation_dir, test_dir]:
    os.makedirs(d, exist_ok=True)

validation_ratio = 0.2

def split_train_validation(class_dir, dest_base_dir):
    all_images = [

```

```

        os.path.join(class_dir, f)
        for f in os.listdir(class_dir)
        if os.path.isfile(os.path.join(class_dir, f))
    ]

    train_images, val_images = train_test_split(
        all_images,
        test_size=validation_ratio,
        random_state=42
    )

    class_name = os.path.basename(class_dir)

    for images, folder_name in zip([train_images, val_images], ["train", "validation"]):
        dest_dir = os.path.join(dest_base_dir, folder_name, class_name)
        os.makedirs(dest_dir, exist_ok=True)

        for file in images:
            shutil.copy(file, dest_dir)

    for class_name in os.listdir(original_train_dir):
        class_dir = os.path.join(original_train_dir, class_name)
        if os.path.isdir(class_dir):
            split_train_validation(class_dir, base_dir)

    for class_name in os.listdir(original_test_dir):
        src_class_dir = os.path.join(original_test_dir, class_name)
        dst_class_dir = os.path.join(test_dir, class_name)

        if os.path.isdir(src_class_dir):
            os.makedirs(dst_class_dir, exist_ok=True)
            for file in os.listdir(src_class_dir):
                shutil.copy(os.path.join(src_class_dir, file), dst_class_dir)

    print("Dataset split completed.")

# 7) Final check
from torchvision import datasets

def get_classes(data_dir):
    all_data = datasets.ImageFolder(data_dir)
    return all_data.classes

classes = get_classes("/content/new_dataset/train")
num_classes = len(classes)

print("\nClasses:", classes)
print("Number of classes:", num_classes)

for split in ["train", "validation", "test"]:
    print(f"\n{split.upper()}:")
    split_path = os.path.join("/content/new_dataset", split)

    for class_name in os.listdir(split_path):
        class_path = os.path.join(split_path, class_name)
        print(class_name, ":", len(os.listdir(class_path)))

```

Training distribution:

Retinopathy grade

```

0    134
1     20
2    136
3     74
4     49

```

Name: count, dtype: int64

Testing distribution:

Retinopathy grade

```

0     34
1      5
2     32
3     19
4     13

```

Name: count, dtype: int64

100%|██████████| 413/413 [00:00<00:00, 646.24it/s]

Missing images: 0

100%|██████████| 103/103 [00:00<00:00, 668.60it/s]

Missing images: 0

Images organized successfully.

train_full/grade_0_no_DR: 100%|██████████| 134/134 [00:29<00:00, 4.54it/s]

train_full/grade_3_severe: 100%|██████████| 74/74 [00:13<00:00, 5.34it/s]

train_full/grade_2_moderate: 100%|██████████| 136/136 [00:25<00:00, 5.28it/s]

train_full/grade_1_mild: 100%|██████████| 20/20 [00:04<00:00, 4.63it/s]

train_full/grade_4_proliferative_DR: 100%|██████████| 49/49 [00:08<00:00, 5.46it/s]

```

test/grade_0_no_DR: 100%|██████████| 34/34 [00:00<00:00, 4.76it/s]
test/grade_3_severe: 100%|██████████| 19/19 [00:03<00:00, 5.55it/s]
test/grade_2_moderate: 100%|██████████| 32/32 [00:06<00:00, 4.73it/s]
test/grade_1_mild: 100%|██████████| 5/5 [00:00<00:00, 5.28it/s]
test/grade_4_proliferative_DR: 100%|██████████| 13/13 [00:02<00:00, 5.39it/s]
Image resizing completed.
Dataset split completed.

Classes: ['grade_0_no_DR', 'grade_1_mild', 'grade_2_moderate', 'grade_3_severe', 'grade_4_proliferative_DR']
Number of classes: 5

TRAIN:
grade_0_no_DR : 107
grade_3_severe : 59
grade_2_moderate : 108
grade_1_mild : 16
grade_4_proliferative_DR : 39

VALIDATION:
grade_0_no_DR : 27
grade_3_severe : 15
grade_2_moderate : 28
grade_1_mild : 4
grade_4_proliferative_DR : 10

TEST:
grade_0_no_DR : 34
grade_3_severe : 19
grade_2_moderate : 32
grade_1_mild : 5
grade_4_proliferative_DR : 13

```

```

import os
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms as T

dataset_path = "/content/new_dataset"
batch_size = 16

def get_data_loaders(data_dir, batch_size, train=False):

    if train:
        transform = T.Compose([
            T.RandomResizedCrop(224, scale=(0.75, 1.0)),
            T.RandomHorizontalFlip(),
            T.RandomRotation(15),
            T.ColorJitter(
                brightness=0.15,
                contrast=0.15,
                saturation=0.10
            ),
            T.ToTensor(),
            T.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

        train_data = datasets.ImageFolder(
            os.path.join(data_dir, "train"),
            transform=transform
        )

        train_loader = DataLoader(
            train_data,
            batch_size=batch_size,
            shuffle=True,
            num_workers=2
        )

        return train_loader, len(train_data)

    else:
        transform = T.Compose([
            T.Resize((224, 224)),
            T.ToTensor(),
            T.Normalize(
                mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]
            )
        ])

        val_data = datasets.ImageFolder(
            os.path.join(data_dir, "validation"),

```

```

        transform=transform
    )

    test_data = datasets.ImageFolder(
        os.path.join(data_dir, "test"),
        transform=transform
    )

    val_loader = DataLoader(
        val_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=2
    )

    test_loader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=2
    )

    return val_loader, test_loader, len(val_data), len(test_data)

```

```

train_loader, train_data_len = get_data_loaders(dataset_path, batch_size, train=True)
val_loader, test_loader, valid_data_len, test_data_len = get_data_loaders(dataset_path, batch_size, train=False)

classes = datasets.ImageFolder(os.path.join(dataset_path, "train")).classes
num_classes = len(classes)

dataloaders = {
    "train": train_loader,
    "val": val_loader
}

dataset_sizes = {
    "train": train_data_len,
    "val": valid_data_len
}

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

print("Device:", device)
print("Classes:", classes)
print("Number of classes:", num_classes)
print("Train:", train_data_len)
print("Validation:", valid_data_len)
print("Test:", test_data_len)

```

```

Device: cuda
Classes: ['grade_0_no_DR', 'grade_1_mild', 'grade_2_moderate', 'grade_3_severe', 'grade_4_proliferative_DR']
Number of classes: 5
Train: 329
Validation: 84
Test: 103

```

```

import copy
from tqdm import tqdm

def train_model_balanced(model, criterion, optimizer, scheduler, num_epochs=12, patience=4):

    history = {
        "train_loss": [],
        "val_loss": [],
        "train_acc": [],
        "val_acc": []
    }

    best_val_acc = 0.0
    best_model_weights = copy.deepcopy(model.state_dict())
    no_improve = 0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")
        print("-" * 10)

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

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

```

```

running_loss = 0.0
running_corrects = 0

for inputs, labels in tqdm(dataloaders[phase]):

    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)
        _, preds = torch.max(outputs, 1)

        if phase == "train":
            loss.backward()
            optimizer.step()

    running_loss += loss.item() * inputs.size(0)
    running_corrects += torch.sum(preds == labels.data)

epoch_loss = running_loss / dataset_sizes[phase]
epoch_acc = running_corrects.double() / dataset_sizes[phase]

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

print(f"{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}")

if phase == "val":
    if epoch_acc > best_val_acc:
        best_val_acc = epoch_acc
        best_model_weights = copy.deepcopy(model.state_dict())
        no_improve = 0
    else:
        no_improve += 1

scheduler.step()
print()

if no_improve >= patience:
    print("Early stopping triggered.")
    break

model.load_state_dict(best_model_weights)
return model, history

```

```

import numpy as np
from torch import nn, optim
from torchvision import models

model = models.efficientnet_b0(weights=models.EfficientNet_B0_Weights.DEFAULT)

# Freeze all feature layers first
for param in model.features.parameters():
    param.requires_grad = False

# Unfreeze last 3 blocks فقط
for param in model.features[-3:].parameters():
    param.requires_grad = True

# Save input features before replacing classifier
in_features = model.classifier[1].in_features

# Replace classifier
model.classifier = nn.Sequential(
    nn.Dropout(p=0.3),
    nn.Linear(in_features, num_classes)
)

model = model.to(device)

criterion = nn.CrossEntropyLoss(label_smoothing=0.05).to(device)

optimizer = optim.AdamW(
    [
        {"params": model.features[-3:].parameters(), "lr": 0.00001},
        {"params": model.classifier.parameters(), "lr": 0.0003}
    ],
    weight_decay=0.001
)

```

```

)

scheduler = optim.lr_scheduler.StepLR(
    optimizer,
    step_size=3,
    gamma=0.85
)

print("Balanced EfficientNet model is ready.")

```

Balanced EfficientNet model is ready.

```

balanced_model, balanced_history = train_model_balanced(
    model,
    criterion,
    optimizer,
    scheduler,
    num_epochs=12,
    patience=4
)

```

```

Epoch 5/12
-----
100%|██████████| 21/21 [00:01<00:00, 10.69it/s]
train Loss: 1.2623 Acc: 0.5046
100%|██████████| 6/6 [00:00<00:00, 15.61it/s]
val Loss: 1.2973 Acc: 0.5000

Epoch 6/12
-----
100%|██████████| 21/21 [00:01<00:00, 10.55it/s]
train Loss: 1.2096 Acc: 0.5380
100%|██████████| 6/6 [00:00<00:00, 16.55it/s]
val Loss: 1.2754 Acc: 0.5238

Epoch 7/12
-----
100%|██████████| 21/21 [00:02<00:00, 7.33it/s]
train Loss: 1.1976 Acc: 0.5502
100%|██████████| 6/6 [00:00<00:00, 9.54it/s]
val Loss: 1.2626 Acc: 0.5000

Epoch 8/12
-----
100%|██████████| 21/21 [00:02<00:00, 7.24it/s]
train Loss: 1.1835 Acc: 0.5380
100%|██████████| 6/6 [00:00<00:00, 16.16it/s]
val Loss: 1.2477 Acc: 0.5357

Epoch 9/12
-----
100%|██████████| 21/21 [00:02<00:00, 10.29it/s]
train Loss: 1.1791 Acc: 0.5623
100%|██████████| 6/6 [00:00<00:00, 15.19it/s]
val Loss: 1.2446 Acc: 0.4762

Epoch 10/12
-----
100%|██████████| 21/21 [00:02<00:00, 10.13it/s]
train Loss: 1.1337 Acc: 0.5866
100%|██████████| 6/6 [00:00<00:00, 15.93it/s]
val Loss: 1.2341 Acc: 0.5000

Epoch 11/12
-----
100%|██████████| 21/21 [00:02<00:00, 10.26it/s]
train Loss: 1.1278 Acc: 0.5653
100%|██████████| 6/6 [00:00<00:00, 16.09it/s]
val Loss: 1.2271 Acc: 0.5000

Epoch 12/12
-----
100%|██████████| 21/21 [00:02<00:00, 8.42it/s]
train Loss: 1.1277 Acc: 0.5653
100%|██████████| 6/6 [00:00<00:00, 9.56it/s]val Loss: 1.2190 Acc: 0.5000

Early stopping triggered.

```

```

from sklearn.metrics import (
    confusion_matrix, accuracy_score, precision_score,
    recall_score, f1_score, classification_report
)
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

```

```

def evaluate_model(model_name, model, test_loader, classes):

    model.eval()

    y_true = []
    y_pred = []
    y_proba = []

    with torch.no_grad():
        for x, y in tqdm(test_loader):

            x = x.to(device)

            outputs = model(x)
            probabilities = torch.softmax(outputs, dim=1)
            preds = torch.argmax(probabilities, dim=1)

            y_true.extend(y.numpy())
            y_pred.extend(preds.cpu().numpy())
            y_proba.extend(probabilities.cpu().numpy())

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

    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"Model: {model_name}")
    print("Accuracy:", accuracy)
    print("Precision:", precision)
    print("Recall:", recall)
    print("F1-score:", f1)

    print("\nClassification Report:")
    print(classification_report(y_true, y_pred, target_names=classes, zero_division=0))

    cm = confusion_matrix(y_true, y_pred)

    plt.figure(figsize=(9, 7))
    sns.heatmap(
        cm,
        annot=True,
        fmt="d",
        cmap="Blues",
        xticklabels=classes,
        yticklabels=classes
    )
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(f"Confusion Matrix - {model_name}")
    plt.show()

    return accuracy, precision, recall, f1, y_true, y_pred, y_proba

```

```

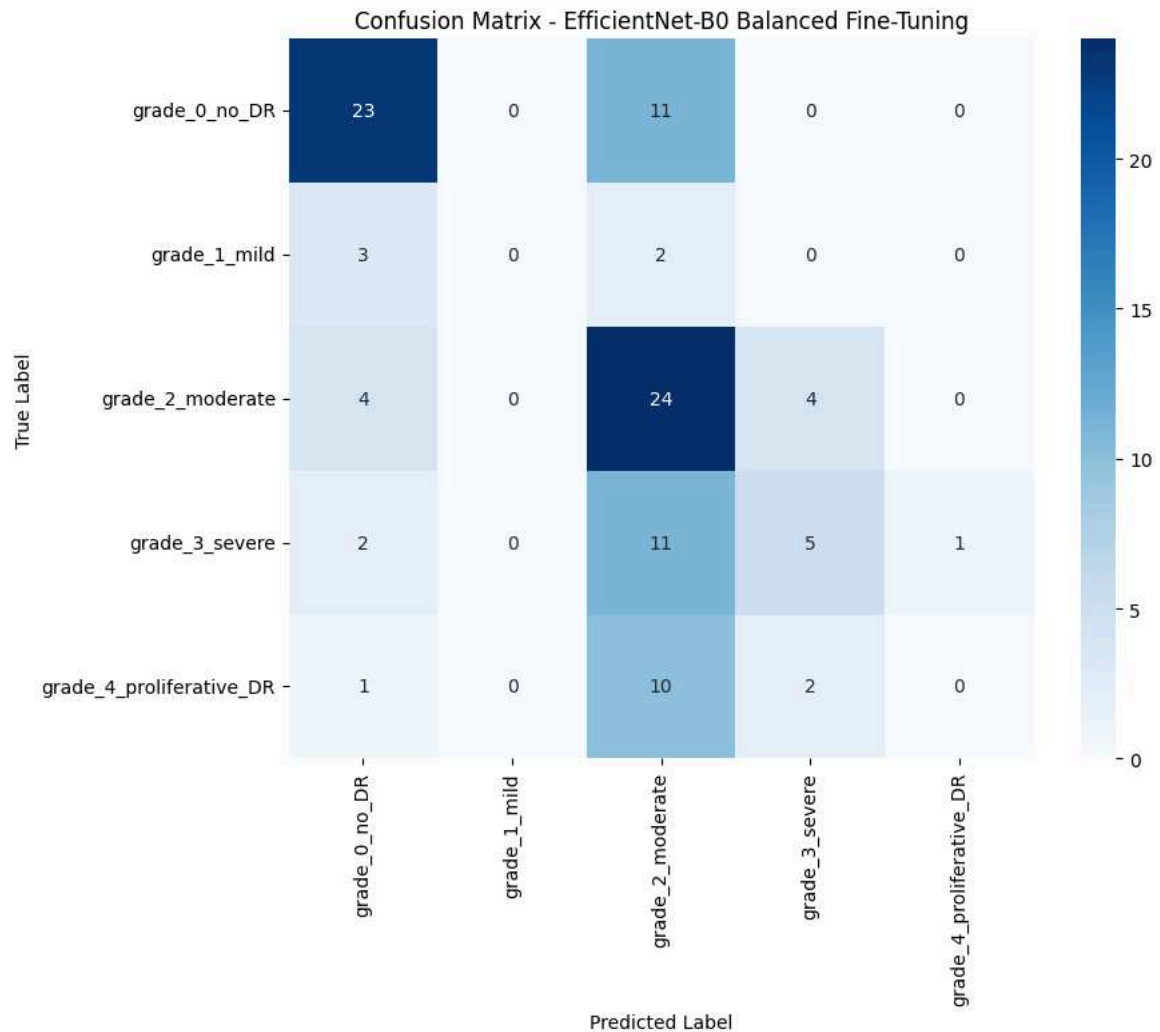
balanced_results = evaluate_model(
    "EfficientNet-B0 Balanced Fine-Tuning",
    balanced_model,
    test_loader,
    classes
)

```

100%|██████████| 7/7 [00:01<00:00, 6.72it/s]
 Model: EfficientNet-B0 Balanced Fine-Tuning
 Accuracy: 0.5048543689320388
 Precision: 0.44247293828813755
 Recall: 0.5048543689320388
 F1-score: 0.4538182872047529

Classification Report:

	precision	recall	f1-score	support
grade_0_no_DR	0.70	0.68	0.69	34
grade_1_mild	0.00	0.00	0.00	5
grade_2_moderate	0.41	0.75	0.53	32
grade_3_severe	0.45	0.26	0.33	19
grade_4_proliferative_DR	0.00	0.00	0.00	13
accuracy			0.50	103
macro avg	0.31	0.34	0.31	103
weighted avg	0.44	0.50	0.45	103



```
best_epoch = np.argmax(balanced_history["val_acc"])

best_train_acc = balanced_history["train_acc"][best_epoch]
best_val_acc = balanced_history["val_acc"][best_epoch]

gap = best_train_acc - best_val_acc

print("Best Epoch:", best_epoch + 1)
print("Best Train Accuracy:", best_train_acc)
print("Best Validation Accuracy:", best_val_acc)
print("Overfitting Gap:", gap)
print("Overfitting Percentage:", gap * 100, "%")
```

```
Best Epoch: 8
Best Train Accuracy: 0.5379939209726444
Best Validation Accuracy: 0.5357142857142857
Overfitting Gap: 0.0022796352583587254
Overfitting Percentage: 0.22796352583587254 %
```

```
balanced_results = evaluate_model(
    "EfficientNet-B0 Balanced Fine-Tuning",
```

```
balanced_model,
test_loader,
classes
)
```

100%|██████████| 7/7 [00:00<00:00, 7.62it/s]
 Model: EfficientNet-B0 Balanced Fine-Tuning
 Accuracy: 0.5048543689320388
 Precision: 0.44247293828813755
 Recall: 0.5048543689320388
 F1-score: 0.4538182872047529

Classification Report:

	precision	recall	f1-score	support
grade_0_no_DR	0.70	0.68	0.69	34
grade_1_mild	0.00	0.00	0.00	5
grade_2_moderate	0.41	0.75	0.53	32
grade_3_severe	0.45	0.26	0.33	19
grade_4_proliferative_DR	0.00	0.00	0.00	13
accuracy			0.50	103
macro avg	0.31	0.34	0.31	103
weighted avg	0.44	0.50	0.45	103

