

Anwar Abaalwaki

```
In [ ]: %matplotlib inline
```

```
In [ ]: import torch
import sklearn
import pandas
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

print("All libraries working ")
import sys
print(sys.executable)
import matplotlib.pyplot as plt
import seaborn as sns
import torch
import numpy as np
import pandas as pd

print("Everything is ready ")
```

```
In [ ]: !pip install tensorflow --upgrade --user

import tensorflow as tf
```

```
In [ ]: import os
import numpy as np
import warnings
import torch
from torchvision import datasets, transforms as T, models
from torch import nn, optim
from torch.utils.data import DataLoader
from tqdm import tqdm
from sklearn.metrics import (confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score, classification_report, roc_curve, auc)
import matplotlib.pyplot as plt
import seaborn as sns
import torchvision
import sklearn
import matplotlib
import seaborn
import pandas as pd
import numpy as np
import tqdm
import sys
print(sys.executable)
warnings.filterwarnings("ignore")
```

```
In [ ]: import torch
import numpy as np
import pandas as pd
```

```
print("Everything is ready ")
```

```
In [ ]: import tensorflow as tf

print(tf.__version__)
```

```
In [ ]: !pip install tensorflow --upgrade --user
```

```
In [ ]: import os

kaggle_dir = os.path.join(os.path.expanduser("~"), ".kaggle")
os.makedirs(kaggle_dir, exist_ok=True)

source = os.path.join(os.getcwd(), "kaggle.json")

if os.path.exists(source):
    destination = os.path.join(kaggle_dir, "kaggle.json")
    os.replace(source, destination)
    print("kaggle.json moved successfully ")
```

```
In [ ]: import zipfile

with zipfile.ZipFile("eye_disease_dataset.zip", 'r') as zip_ref:
    zip_ref.extractall("dataset_folder")
```

```
In [ ]: !pip install numpy pandas matplotlib seaborn pillow opencv-python
```

```
In [ ]: import os

dataset_path = "dataset_folder/dataset"

print(os.listdir(dataset_path))
```

```
In [ ]: import os

dataset_path = "dataset_folder/dataset"

for folder in os.listdir(dataset_path):
    folder_path = os.path.join(dataset_path, folder)

    if os.path.isdir(folder_path):
        print("Class Folder:", folder)

        images = os.listdir(folder_path)
        print("Number of images:", len(images))
        print("-" * 40)
```

```
In [ ]: from PIL import Image
import os

input_root = 'dataset_folder/dataset'
output_root = 'output'
```

```

resize_size = (256, 256)

os.makedirs(output_root, exist_ok=True)

for class_name in os.listdir(input_root):

    class_path = os.path.join(input_root, class_name)

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

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

    for image_name in os.listdir(class_path):

        if image_name.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.t

            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

            try:
                with Image.open(input_image_path) as img:
                    img = img.convert('RGB')
                    img = img.resize(resize_size)
                    img.save(output_image_path, 'JPEG')

                print(f"Processed: {input_image_path}")

            except Exception as e:
                print(f"Failed: {input_image_path} → {e}")

```

Splitting dataset

```

In [ ]: import os
import shutil
from sklearn.model_selection import train_test_split
original_dataset_dir = 'dataset_folder/dataset'
base_dir = 'new dataset'
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 dir_path in [train_dir, validation_dir, test_dir]:
    os.makedirs(dir_path, exist_ok=True)
train_ratio = 0.7
validation_ratio = 0.1
test_ratio = 0.2
def split_and_copy_files(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, temp_images = train_test_split(
    all_images,
    test_size=(1 - train_ratio),
    random_state=42
)

validation_images, test_images = train_test_split(
    temp_images,
    test_size=test_ratio/(validation_ratio + test_ratio),
    random_state=42
)

for images, folder_name in zip(
    [train_images, validation_images, test_images],
    ['train', 'validation', 'test']
):

    dest_dir = os.path.join(
        dest_base_dir,
        folder_name,
        os.path.basename(class_dir)
    )

    os.makedirs(dest_dir, exist_ok=True)

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

for class_dir in os.listdir(original_dataset_dir):

    full_class_dir = os.path.join(original_dataset_dir, class_dir)

    if os.path.isdir(full_class_dir):
        split_and_copy_files(full_class_dir, base_dir)

print("Dataset splitting completed ")

```

```
In [ ]: torch.utils.data.random_split
```

```
In [ ]: from torchvision import datasets

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

classes = get_classes("new dataset/test")

print(classes)
print(len(classes))

```

```
In [ ]: import os

train_dir = "new dataset/train"
val_dir = "new dataset/validation"
test_dir = "new dataset/test"

```

Getting name of classes in dataset

```
In [ ]: from torchvision import datasets
def get_classes(data_dir):
    dataset = datasets.ImageFolder(data_dir)
    return dataset.classes
classes = get_classes(train_dir)
print(classes)
print("Number of classes:", len(classes))
```

```
In [ ]: from torchvision import transforms as T
IMG_SIZE = 224
train_transform = T.Compose([T.Resize((IMG_SIZE, IMG_SIZE)), T.RandomHorizontalFlip()])
val_transform = T.Compose([T.Resize((IMG_SIZE, IMG_SIZE)), T.ToTensor()])
test_transform = T.Compose([T.Resize((IMG_SIZE, IMG_SIZE)), T.ToTensor()])
```

```
In [ ]: train_dataset = datasets.ImageFolder(train_dir, transform=train_transform)
val_dataset = datasets.ImageFolder(val_dir, transform=val_transform)
test_dataset = datasets.ImageFolder(test_dir, transform=test_transform)
```

```
In [ ]: from torch.utils.data import DataLoader
BATCH_SIZE = 32
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

```
In [ ]: import torch
import torch.nn as nn
from torchvision import models
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = models.resnet18(pretrained=True)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, len(classes))
model = model.to(device)
```

```
In [ ]: criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
```

```
In [ ]: EPOCHS = 10

for epoch in range(EPOCHS):

    model.train()
    train_loss = 0

    for images, labels in train_loader:

        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = model(images)
        loss = criterion(outputs, labels)
```

```

        loss.backward()
        optimizer.step()

        train_loss += loss.item()

    print(f"Epoch {epoch+1}, Loss: {train_loss/len(train_loader)}")

```

```

In [ ]: from sklearn.metrics import accuracy_score, classification_report

model.eval()

all_preds = []
all_labels = []

with torch.no_grad():
    for images, labels in test_loader:

        images = images.to(device)

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

        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.numpy())

print("Accuracy:", accuracy_score(all_labels, all_preds))
print(classification_report(all_labels, all_preds))

```

```

In [ ]: import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(all_labels, all_preds)

plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d')
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

```

```

In [ ]: from torchvision import datasets

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

classes = get_classes("new dataset/test")

print(classes)
print(len(classes))

```

Image augmentation and preprocessing

```

In [ ]: from torchvision import datasets, transforms as T

```

```

from torch.utils.data import DataLoader
import os
import torch

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

    if train:
        # Data augmentation for training
        transform = T.Compose([
            T.RandomHorizontalFlip(),
            T.RandomVerticalFlip(),
            T.RandomRotation(10),
            T.Resize((299, 299)),
            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=4
        )

        return train_loader, len(train_data)

    else:
        # Validation and test transformations
        transform = T.Compose([
            T.Resize((299, 299)),
            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=4

```

```

    )

    test_loader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=4
    )

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

# Dataset paths and loaders
dataset_path = "new dataset"
batch_size = 32

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

```

Data Augmentation

```
In [ ]: from torchvision import transforms as T
```

```

train_transform = T.Compose([
    T.Resize((224,224)),
    T.RandomHorizontalFlip(),
    T.RandomRotation(15),
    T.ToTensor(),
    T.Normalize([0.485,0.456,0.406],
                [0.229,0.224,0.225])
])

```

```
In [ ]: torchvision.transforms
```

EfficientNet_B0

```
In [ ]: from torchvision.models import efficientnet_b0
import torch.nn as nn

model_eff = efficientnet_b0(pretrained=True)
```

```

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

model_eff = model_eff.to(device)

```

```

In [ ]: import numpy as np
import torch
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm

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)

            output = model(x)

            # Softmax probabilities
            probabilities = torch.softmax(output, dim=1)

            # True labels
            y_true.extend(y.numpy())

            # Store probabilities
            y_proba.extend(probabilities.cpu().numpy())

            # Predicted labels
            pred = torch.argmax(probabilities, axis=1).cpu().numpy()
            y_pred.extend(pred)

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

    # Metrics
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')

    cm = confusion_matrix(y_true, y_pred)

    print("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))

# Confusion Matrix Plot
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d',
            xticklabels=classes,
            yticklabels=classes,
            cmap='Blues')

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

return accuracy, precision, recall, f1

```

```

In [ ]: evaluate_model(
        "efficientnet_b0",
        model,
        test_loader,
        classes
    )

```

Resnet18

```

In [ ]: import torch
import torch.nn as nn
from torchvision import models
import torch.optim as optim

# Load EfficientNet-B0 model
model = models.efficientnet_b0(pretrained=True)

# Modify classifier layer for number of classes
model.classifier[1] = nn.Linear(
    model.classifier[1].in_features,
    num_classes
)

model = model.to(device)

# Loss function
criterion = nn.CrossEntropyLoss().to(device)

# Optimizer
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
scheduler = optim.lr_scheduler.StepLR(

```

```

optimizer,
step_size=3,
gamma=0.97
)

def train_model(model, criterion, optimizer, scheduler, num_epochs=20):

    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
            running_corrects = 0

            for inputs, labels in 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}")

        return model

```

```

In [ ]: model = train_model(
        model,

```

```

        criterion,
        optimizer,
        scheduler,
        num_epochs=20
    )

```

```

In [ ]: from torchvision import models

model_resnet = models.resnet18(pretrained=True)

model_resnet.fc = nn.Linear(
    model_resnet.fc.in_features,
    num_classes
)

model_resnet = model_resnet.to(device)

```

DenseNet121

```

In [ ]: model_densenet = models.densenet121(pretrained=True)

model_densenet.classifier = nn.Linear(
    model_densenet.classifier.in_features,
    num_classes
)

model_densenet = model_densenet.to(device)

```

```

In [ ]: from torchvision import models
import torch.nn as nn
import torch.optim as optim

model = models.densenet121(pretrained=True)

model.classifier = nn.Linear(
    model.classifier.in_features,
    num_classes
)

model = model.to(device)

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

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

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

model = train_model(
    model,

```

```

        criterion,
        optimizer,
        scheduler,
        num_epochs=5
    )

```

```

In [ ]: evaluate_model(
        "Densenet121",
        model,
        test_loader,
        classes
    )

```

VGG16

```

In [ ]: model_vgg = models.vgg16(pretrained=True)

model_vgg.classifier[6] = nn.Linear(
    model_vgg.classifier[6].in_features,
    num_classes
)

model_vgg = model_vgg.to(device)

```

Training Function

```

In [ ]: device = torch.device("cpu")

```

```

In [ ]: def train_model(model, dataloaders, criterion, optimizer, epochs=10):

    model.to(device)

    for epoch in range(epochs):

        model.train()
        running_loss = 0

        for images, labels in dataloaders["train"]:

            images = images.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            outputs = model(images)
            loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        print(f"Epoch {epoch+1} Loss = {running_loss/len(dataloaders['train'])}")

```

Testing Function

```
In [ ]: def evaluate_model(model, loader):

    model.eval()

    correct = 0
    total = 0

    with torch.no_grad():

        for images, labels in loader:

            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)

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

            total += labels.size(0)
            correct += (predicted.cpu() == labels.cpu()).sum().item()

    print("Test Accuracy =", correct/total)
```

```
In [ ]: import matplotlib.pyplot as plt

def evaluate_model(model, loader):

    model.eval()

    correct = 0
    total = 0

    with torch.no_grad():

        for images, labels in loader:

            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)

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

            total += labels.size(0)
            correct += (predicted.cpu() == labels.cpu()).sum().item()

    accuracy = correct / total

    print("Test Accuracy =", accuracy)

    plt.figure(figsize=(5,4))
    plt.bar(["Accuracy"], [accuracy])
    plt.ylim(0,1)
```

```
plt.title("Test Accuracy Visualization")  
plt.show()
```

```
In [ ]: print(classification_report(all_labels, all_preds))
```

```
In [ ]: plt.savefig("image.png", dpi=300)
```

```
In [ ]: !jupyter nbconvert --to webpdf Anwar_Eye_Disease_AI_Project-Copy1.ipynb
```