

✓ Image Classification Assignment

Submitted by: **ELSEDEEG MOHAMEDELBASHER ABDALLA AHMED**

Student ID: **251002546**

This notebook implements the complete Image Classification workflow required in the practical assignment.

The main steps are:

1. **Download the image dataset** (from Kaggle) and upload it to Google Drive.
2. **Splitting the dataset** into training, validation, and test sets.
3. **Applying data augmentation** on the training data.
4. **Training and testing models** using a CNN (EfficientNetB0).
5. **Converting the notebook (`.ipynb`) to PDF** for submission.

Google Drive is used to store:

- The original dataset (ZIP file)
- The extracted images
- The split train/val/test folders
- The trained model checkpoints
- The exported PDF report

✓ Step 1 — Mount Google Drive & Create Project Folder Structure

In this step, we:

- Mount Google Drive in the Colab environment.
- Define the main project directories.
- Create all required folders if they do not already exist.

The folder structure will be:

```
MyDrive/
├── Image_Project/
│   ├── data/
│   │   ├── eye_diseases_classification.zip
│   │   └── extracted/
│   └──
```

```
|— splitted/
|   |— train/
|   |— val/
|   └— test/
|
|— models/
|— output/
└— results/
```

The dataset ZIP file **must be uploaded manually** to:

[/content/drive/MyDrive/Image_Project/data/eye_diseases_classification.zip](#)

```
from google.colab import drive
import os

# Mount Google Drive
drive.mount('/content/drive')

# ----- main folders -----
BASE_DIR = "/content/drive/MyDrive/Image_Project"
DATA_DIR = os.path.join(BASE_DIR, "data")

EXTRACT_ROOT = os.path.join(DATA_DIR, "extracted") # هنا يتم فك الضغط
DATASET_DIR = os.path.join(EXTRACT_ROOT, "dataset") # هنا توجد الكلاسات فعلياً

SPLIT_DIR = os.path.join(BASE_DIR, "splitted")
OUTPUT_DIR = os.path.join(BASE_DIR, "output")
MODELS_DIR = os.path.join(BASE_DIR, "models")
RESULTS_DIR = os.path.join(BASE_DIR, "results")

required_folders = [
    BASE_DIR, DATA_DIR, EXTRACT_ROOT,
    SPLIT_DIR, OUTPUT_DIR, MODELS_DIR, RESULTS_DIR
]

for folder in required_folders:
    os.makedirs(folder, exist_ok=True)
    print(f"📁 Ready: {folder}")

print("\n🎉 Project folder structure is ready.")
print("➡ Make sure the dataset ZIP file is placed at:")
print("    /content/drive/MyDrive/Image_Project/data/eye_diseases_classification.zip")
```

```
Mounted at /content/drive
📁 Ready: /content/drive/MyDrive/Image_Project
📁 Ready: /content/drive/MyDrive/Image_Project/data
📁 Ready: /content/drive/MyDrive/Image_Project/data/extracted
```

```
Ready: /content/drive/MyDrive/Image_Project/splitted
Ready: /content/drive/MyDrive/Image_Project/output
Ready: /content/drive/MyDrive/Image_Project/models
Ready: /content/drive/MyDrive/Image_Project/results
```

🔧 Project folder structure is ready.
- Make sure the dataset ZIP file is placed at:
/content/drive/MyDrive/Image_Project/data/eye_diseases_classification.zip

📦 Step 2 — Extract the Image Dataset from ZIP

In this step, we assume that:

- Here i have already **downloaded** the image dataset from Kaggle.
- Here i have uploaded the file as:

eye_diseases_classification.zip

into the path:

/content/drive/MyDrive/Image_Project/data/

Now I will:

1. Check if the ZIP file exists.
2. Extract all images into:

/content/drive/MyDrive/Image_Project/data/extracted/

```
import zipfile

dataset_zip_path = os.path.join(DATA_DIR, "eye_diseases_classification.zip")
extract_path = EXTRACT_ROOT # انتبه: Her EXTRACT_ROOT Not DATASET_DIR

if os.path.exists(dataset_zip_path):
    print("✅ Dataset ZIP file found at:", dataset_zip_path)
    print("📦 Extracting dataset... please wait.")
    with zipfile.ZipFile(dataset_zip_path, "r") as zip_ref:
        zip_ref.extractall(extract_path)
    print("🎉 Extraction completed successfully!")
    print("📦 Extracted to:", extract_path)
else:
    print("❌ ERROR: Dataset ZIP file not found!")
    print("→ Please upload 'eye_diseases_classification.zip' to:")
    print("    /content/drive/MyDrive/Image_Project/data/")
```

```
✅ Dataset ZIP file found at: /content/drive/MyDrive/Image_Project/data/eye_diseases_classification.zip
📦 Extracting dataset... please wait.
🎉 Extraction completed successfully!
📦 Extracted to: /content/drive/MyDrive/Image_Project/data/extracted
```

Double-click (or enter) to edit

✓ Step 3 — Explore the Extracted Dataset

In this step, I:

- List the class folders available in the extracted dataset.
- Print the number of images in each class.

This helps me to understand the dataset structure and check that the extraction is correct.

```
# Step 3 – Explore the Extracted Dataset
# -----

import os    # ← Important

# List classes and count images per class from DATASET_DIR
classes = []
class_counts = {}

for entry in sorted(os.listdir(DATASET_DIR)):
    class_path = os.path.join(DATASET_DIR, entry)
    if os.path.isdir(class_path):
        classes.append(entry)
        num_files = len([
            f for f in os.listdir(class_path)
            if os.path.isfile(os.path.join(class_path, f))
        ])
        class_counts[entry] = num_files

print("Available classes:", classes)

print("\nNumber of images per class:")
for cls, cnt in class_counts.items():
    print(f"  {cls}: {cnt}")
```

```
Available classes: ['cataract', 'diabetic_retinopathy', 'glaucoma', 'normal']
```

```
Number of images per class:
cataract: 1038
diabetic_retinopathy: 1098
glaucoma: 1007
normal: 1074
```

Step 4 — Split Dataset into Train / Validation / Test Sets

The assignment requires splitting the dataset into:

- **Training set** — used to train the model
- **Validation set** — used to tune hyperparameters and monitor overfitting
- **Test set** — used for final evaluation

I will use the following split ratios:

- Train: **70%**
- Validation: **10%**
- Test: **20%**

The split will be performed **class-wise**, and the images will be copied into:

```
splitted/  
├─ train/<class_name>/  
├─ val/<class_name>/  
└─ test/<class_name>/
```

```
# -----  
# Step 4 – Unified Cell: Define Paths + Prepare Folders + Split Dataset  
# -----  
  
from google.colab import drive  
import os  
import shutil  
import random  
  
# Mount Drive  
drive.mount('/content/drive')  
  
# ----- main folders -----  
BASE_DIR = "/content/drive/MyDrive/Image_Project"  
DATA_DIR = os.path.join(BASE_DIR, "data")  
  
# Folder of the extracted ZIP  
EXTRACT_ROOT = os.path.join(DATA_DIR, "extracted")  
  
# Folder that contains the class folders  
DATASET_DIR = os.path.join(EXTRACT_ROOT, "dataset")  
  
# Target folders for splitting  
SPLIT_DIR = os.path.join(BASE_DIR, "splitted")  
OUTPUT_DIR = os.path.join(BASE_DIR, "output")  
MODELS_DIR = os.path.join(BASE_DIR, "models")
```

```

RESULTS_DIR = os.path.join(BASE_DIR, "results")

required_folders = [
    BASE_DIR, DATA_DIR, EXTRACT_ROOT,
    SPLIT_DIR, OUTPUT_DIR, MODELS_DIR, RESULTS_DIR
]

# Create required folders
for folder in required_folders:
    os.makedirs(folder, exist_ok=True)
    print("📁 Ready:", folder)

# Seed for reproducibility
random.seed(42)

# Split ratios
split_ratios = {
    "train": 0.7,
    "val": 0.1,
    "test": 0.2
}

# Prepare train/val/test main folders
for split_name in ["train", "val", "test"]:
    split_root = os.path.join(SPLIT_DIR, split_name)
    os.makedirs(split_root, exist_ok=True)

print("\n📁 Train/Val/Test folders created successfully.")

# Split class folders
for cls in classes:
    src_dir = os.path.join(DATASET_DIR, cls)
    if not os.path.isdir(src_dir):
        continue

    images = [
        f for f in os.listdir(src_dir)
        if os.path.isfile(os.path.join(src_dir, f))
    ]

    random.shuffle(images)

    n_total = len(images)
    n_train = int(n_total * split_ratios["train"])
    n_val = int(n_total * split_ratios["val"])
    n_test = n_total - n_train - n_val

    splits = {
        "train": images[:n_train],
        "val": images[n_train:n_train + n_val],

```

```

        "test": images[n_train + n_val:]
    }

    for split_name, file_list in splits.items():
        dest_dir = os.path.join(SPLIT_DIR, split_name, cls)
        os.makedirs(dest_dir, exist_ok=True)

        for fname in file_list:
            src_path = os.path.join(src_dir, fname)
            dst_path = os.path.join(dest_dir, fname)
            shutil.copy2(src_path, dst_path)

print("\n✅ Dataset successfully split into train / val / test inside:")
print("    ", SPLIT_DIR)

```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

📁 Ready: /content/drive/MyDrive/Image_Project
 📁 Ready: /content/drive/MyDrive/Image_Project/data
 📁 Ready: /content/drive/MyDrive/Image_Project/data/extracted
 📁 Ready: /content/drive/MyDrive/Image_Project/splitted
 📁 Ready: /content/drive/MyDrive/Image_Project/output
 📁 Ready: /content/drive/MyDrive/Image_Project/models
 📁 Ready: /content/drive/MyDrive/Image_Project/results

📁 Train/Val/Test folders created successfully.

✅ Dataset successfully split into train / val / test inside:
 /content/drive/MyDrive/Image_Project/splitted

✎ Step 5 — Define Data Augmentation & DataLoaders

In this step, we:

- I prepare the data pipeline for the deep learning model. I will apply data augmentation to increase dataset diversity and improve generalization. and will also create separate DataLoaders for training, validation, and testing,

Data augmentation applied:

- Random horizontal flip
- Random rotation
- Resize to 224×224
- Normalization using ImageNet statistics

Output of this step:

- train_loader
- val_loader

- test_loader
- classes list

These loaders will be used in the model training phase.

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

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

# -----
# Data Augmentation (Training)
# -----
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])
])

# -----
# Validation/Test Transform
# -----
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])
])

# Directories
train_dir = os.path.join(SPLIT_DIR, "train")
val_dir   = os.path.join(SPLIT_DIR, "val")
test_dir  = os.path.join(SPLIT_DIR, "test")

# Datasets
train_dataset = datasets.ImageFolder(train_dir, transform=train_transform)
val_dataset   = datasets.ImageFolder(val_dir,   transform=test_transform)
test_dataset  = datasets.ImageFolder(test_dir,  transform=test_transform)

# Data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=2)
val_loader   = DataLoader(val_dataset,   batch_size=32, shuffle=False, num_workers=2)
```

```

test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=2)

# Extract class names
classes = train_dataset.classes

print("Classes:", classes)
print("Number of training samples:", len(train_dataset))
print("Number of validation samples:", len(val_dataset))
print("Number of test samples:", len(test_dataset))

```

```

Using device: cuda
Classes: ['cataract', 'diabetic_retinopathy', 'glaucoma', 'normal']
Number of training samples: 3556
Number of validation samples: 677
Number of test samples: 1323

```

✓ Step 6 — Define and Train the CNN Model (EfficientNetB0)

In this step, I:

- Load a **pre-trained EfficientNetB0** model from `torchvision`.
- Replace the final classification layer to match the number of classes in our dataset.
- Define the **loss function** (CrossEntropyLoss) and the **optimizer** (Adam).
- Train the model for a few epochs using the `train_loader`.
- Monitor the validation loss using `val_loader`.
- Save the **best model checkpoint** (lowest validation loss) to the `models/` folder.

This model will later be evaluated on the **test set** in the next step.

```

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

# -----
# Define EfficientNetB0 model
# -----
# load pre-trained weights
model = models.efficientnet_b0(weights=models.EfficientNet_B0_Weights.IMAGENET1K_V1)

# replace final classifier layer
in_features = model.classifier[1].in_features
model.classifier[1] = nn.Linear(in_features, len(classes))

model = model.to(device)

```

```

# loss function & optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

print("Model ready with", len(classes), "output classes.")

# -----
# Training loop
# -----
num_epochs = 3
best_val_loss = np.inf
best_model_path = os.path.join(MODELS_DIR, "best_efficientnet_b0.pth")

for epoch in range(1, num_epochs + 1):

    # ---- training phase ----
    model.train()
    running_loss = 0.0

    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()

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

    epoch_train_loss = running_loss / len(train_loader.dataset)

    # ---- validation phase ----
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * images.size(0)

    epoch_val_loss = val_loss / len(val_loader.dataset)

    print(f"Epoch [{epoch}/{num_epochs}] "
          f"Train Loss: {epoch_train_loss:.4f} "
          f"Val Loss: {epoch_val_loss:.4f}")

# save best model

```

```

if epoch_val_loss < best_val_loss:
    best_val_loss = epoch_val_loss
    torch.save(model.state_dict(), best_model_path)
    print("📁 Best model updated and saved to:", best_model_path)

print("\n✅ Training completed.")

```

```

Downloading: "https://download.pytorch.org/models/efficientnet_b0_rwrightman-7f5810bc.pth" to /root/.cache/torch/hub/checkpoints/efficientnet_b0_rwrightman-7f5810bc.pth
100%|██████████| 20.5M/20.5M [00:00<00:00, 159MB/s]
Model ready with 4 output classes.
Epoch [1/3] Train Loss: 0.4523 Val Loss: 0.2880
📁 Best model updated and saved to: /content/drive/MyDrive/Image_Project/models/best_efficientnet_b0.pth
Epoch [2/3] Train Loss: 0.2915 Val Loss: 0.2220
📁 Best model updated and saved to: /content/drive/MyDrive/Image_Project/models/best_efficientnet_b0.pth
Epoch [3/3] Train Loss: 0.2315 Val Loss: 0.2275

✅ Training completed.

```

✓ Step 7 — Evaluate the Trained Model on the Test Set

In this step, I evaluate the **best saved EfficientNetB0 model** on the **test set**.

And I will:

1. Load the best model weights from: `models/best_efficientnet_b0.pth`
2. Run the model on `test_loader`.
3. Compute the following metrics:
 - Accuracy
 - Precision (weighted)
 - Recall (weighted)
 - F1-score (weighted)
4. Print a detailed **classification report** for all classes.
5. Plot a **confusion matrix** to visualize the performance per class.

This completes the **Training and Testing models** requirement of the assignment.

```

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

# =====

```

```

# 1) Load best saved model
# =====
if os.path.exists(best_model_path):
    model.load_state_dict(torch.load(best_model_path, map_location=device))
    print("✅ Loaded best model from:", best_model_path)
else:
    print("⚠️ Best model file not found. Using in-memory model.")

# =====
# 2) Run evaluation
# =====
model.eval()
y_true, y_pred = [], []

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        outputs = model(images)
        preds = torch.argmax(outputs, dim=1).cpu().numpy()

        y_pred.extend(preds)
        y_true.extend(labels.numpy())

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

# =====
# 3) Compute main metrics
# =====
acc = accuracy_score(y_true, y_pred)
prec = precision_score(y_true, y_pred, average="weighted", zero_division=0)
rec = recall_score(y_true, y_pred, average="weighted", zero_division=0)
f1 = f1_score(y_true, y_pred, average="weighted", zero_division=0)

print("\n📊 Overall Test Metrics")
print(f" Accuracy : {acc:.4f}")
print(f" Precision: {prec:.4f} (weighted)")
print(f" Recall   : {rec:.4f} (weighted)")
print(f" F1-score  : {f1:.4f} (weighted)")

# =====
# 4) Per-class report
# =====
report = classification_report(
    y_true, y_pred,
    target_names=classes,
    zero_division=0,

```

```

    output_dict=True
)

print("\n📄 Detailed Classification Report:\n")
print(classification_report(y_true, y_pred, target_names=classes, zero_division=0))

# 🔥 NEW: Per-class accuracy (allowed improvement)
print("\n🎯 Per-Class Accuracy (from classification report):")
for cls in classes:
    print(f"   {cls}: {report[cls]['precision']:.4f}")

# =====
# 5) Confusion Matrix
# =====
cm = confusion_matrix(y_true, y_pred)
cm_norm = cm.astype(float) / cm.sum(axis=1, keepdims=True).clip(min=1)

# ---- Confusion Matrix (Counts) ----
plt.figure(figsize=(7, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=classes, yticklabels=classes)
plt.title("ELSEDEEG Project – Confusion Matrix (Counts)")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()

# ---- Confusion Matrix (Normalized) ----
plt.figure(figsize=(7, 6))
sns.heatmap(cm_norm, annot=True, fmt=".2f", cmap="Greens",
            xticklabels=classes, yticklabels=classes)
plt.title("ELSEDEEG Project – Confusion Matrix (Normalized)")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()

# 🔥 NEW: Per-class accuracy from normalized CM
print("\n🎯 Per-Class Test Accuracy (from normalized confusion matrix):")
for i, cls in enumerate(classes):
    correct = cm[i, i]
    total = cm[i].sum()
    acc_cls = correct / total if total > 0 else 0.0
    print(f"   {cls}: {acc_cls:.4f}")

```


✔ Loaded best model from: /content/drive/MyDrive/Image_Project/models/best_efficientnet_b0.pth
The EfficientNet-B0 model achieved a high performance with an overall test accuracy of 91.84%, demonstrating excellent generalization.

📊 Overall Test Metrics
The diabetic_retinopathy class was recognized with perfect accuracy (100%), while cataract and glaucoma also achieved strong results. The normal class showed slightly lower recall (0.81), which is expected due to visual similarity with early-stage diseases.

Accuracy : 0.9184
Recall : 0.9184 (weighted)
F1-score : 0.9175 (weighted)

📄 Detailed Classification Report:

	precision	recall	f1-score	support
cataract	0.89	0.96	0.93	333
diabetic_retinopathy	1.00	1.00	1.00	345
glaucoma	0.88	0.89	0.88	359
normal	0.90	0.81	0.85	286