

Project 1 — Eye Disease Classification with Deep Learning

Student: Hala

Course: Deep Learning / Computer Vision — Midocean University

Submission date: July 2026

Overview

In this project I build an image-classification pipeline that automatically recognises four eye conditions from retinal fundus photographs: **cataract**, **diabetic retinopathy**, **glaucoma**, and **normal**.

My workflow follows five stages:

1. Load the image dataset and inspect it (exploratory data analysis).
2. Split the data into training / validation / test sets in a *stratified* way.
3. Apply data augmentation and preprocessing.
4. Train and evaluate two convolutional networks using transfer learning (**EfficientNet-B0** and **ResNet-18**) and compare them.
5. Summarise the results and draw conclusions.

The dataset comes from Kaggle: *Eye Diseases Classification* by *gunavenkatdoddi* (<https://www.kaggle.com/datasets/gunavenkatdoddi/eye-diseases-classification>).

1. Setup

I start by importing the libraries I will use and fixing a random seed so the experiment is reproducible. I also put the main settings in one small config dictionary — this keeps the notebook tidy and makes it easy to change one value (for example the number of epochs) without hunting through the code.

```
In [26]: import os
import random
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import torch
from torch import nn, optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
```

```

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

import warnings
warnings.filterwarnings("ignore")

sns.set_style("whitegrid")

# ---- Configuration -----
CFG = {
    "img_size": 224,      # EfficientNet-B0 / ResNet-18 expect 224x224
    "batch_size": 32,
    "epochs": 10,        # increase for a bit more accuracy if time allow
    "lr": 1e-4,          # small LR is good for fine-tuning pretrained ne
    "weight_decay": 1e-4,
    "val_size": 0.15,    # 70 / 15 / 15 split
    "test_size": 0.15,
    "seed": 42,
}

def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

set_seed(CFG["seed"])

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

```

Using device: cuda

2. Locating the dataset

On Kaggle the dataset is attached under `/kaggle/input/...`; on Google Colab I usually unzip it into a local folder. The exact folder name can differ (for example Kaggle may mount it as `eye_diseases_classification` or `eye-diseases-classification`), so instead of hard-coding a path I **scan** the input directories and automatically pick the folder that actually contains the class sub-folders full of images. This makes the notebook portable and robust to naming differences.

```

In [27]: IMG_EXT = (".png", ".jpg", ".jpeg", ".bmp", ".tiff", ".webp")
EXPECTED_CLASSES = {"cataract", "diabetic_retinopathy", "glaucoma", "normal"}

def _looks_like_dataset(path):
    """True if `path` holds >=2 sub-folders that each contain image files."""
    if not os.path.isdir(path):
        return False

```

```

subdirs = [d for d in os.listdir(path)
            if os.path.isdir(os.path.join(path, d))]
if len(subdirs) < 2:
    return False
good = 0
for d in subdirs:
    p = os.path.join(path, d)
    try:
        if any(f.lower().endswith(IMG_EXT) for f in os.listdir(p)):
            good += 1
    except OSError:
        pass
return good >= 2

def find_data_root(search_roots=("/kaggle/input", ".", "dataset")):
    """Walk the search roots and return the folder that holds the class folders.

    If several match, prefer the one whose sub-folder names look most like the
    expected eye-disease classes.
    """
    candidates = []
    for base in search_roots:
        if not os.path.isdir(base):
            continue
        for dirpath, _dirnames, _files in os.walk(base):
            if _looks_like_dataset(dirpath):
                subs = {d.lower() for d in os.listdir(dirpath)
                        if os.path.isdir(os.path.join(dirpath, d))}
                score = len(subs & EXPECTED_CLASSES)
                candidates.append((score, dirpath))
    if not candidates:
        raise FileNotFoundError(
            "Could not locate the dataset. On Kaggle, click 'Add Input' and "
            "the Eye Diseases Classification dataset; on Colab, unzip it into "
            "'dataset' folder next to this notebook."
        )
    candidates.sort(key=lambda x: -x[0])  # best class-name match first
    return candidates[0][1]

DATA_ROOT = find_data_root()
print("Dataset root:", DATA_ROOT)
print("Classes found:", sorted(os.listdir(DATA_ROOT)))

```

Dataset root: /kaggle/input/datasets/gunavenkatdoddi/eye-diseases-classification/dataset

Classes found: ['cataract', 'diabetic_retinopathy', 'glaucoma', 'normal']

3. Exploratory data analysis

Before modelling I like to look at the data. First I count how many images each class has — if the classes are very unbalanced I would need to be careful with the metrics I choose. Then I display a few example images so I can see what the model is actually working with.

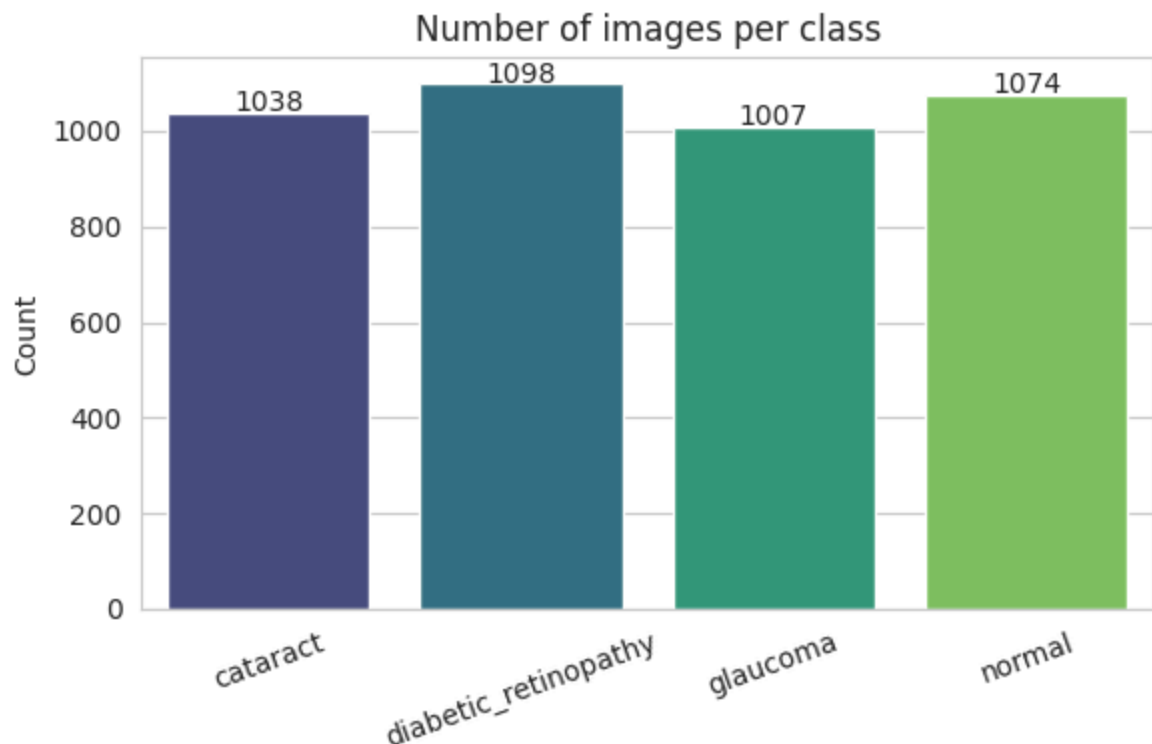
```
In [28]: # Load the full dataset once (no transform yet) to read labels and file path
base_dataset = datasets.ImageFolder(DATA_ROOT)
class_names = base_dataset.classes
num_classes = len(class_names)
targets = np.array([label for _, label in base_dataset.samples])

print(f"Total images: {len(base_dataset)}")
print(f"Classes ({num_classes}): {class_names}")

# Count per class
counts = [int((targets == i).sum()) for i in range(num_classes)]
plt.figure(figsize=(6, 4))
sns.barplot(x=class_names, y=counts, palette="viridis")
plt.title("Number of images per class")
plt.ylabel("Count")
plt.xticks(rotation=20)
for i, c in enumerate(counts):
    plt.text(i, c + 3, str(c), ha="center")
plt.tight_layout()
plt.show()
```

Total images: 4217

Classes (4): ['cataract', 'diabetic_retinopathy', 'glaucoma', 'normal']



```
In [29]: # Show one sample image from each class
fig, axes = plt.subplots(1, num_classes, figsize=(4 * num_classes, 4))
for ax, cls_idx in zip(axes, range(num_classes)):
    # first image belonging to this class
    idx = int(np.where(targets == cls_idx)[0][0])
    img_path = base_dataset.samples[idx][0]
    ax.imshow(plt.imread(img_path))
    ax.set_title(class_names[cls_idx])
    ax.axis("off")
```

```
plt.suptitle("Example image from each class")
plt.tight_layout()
plt.show()
```



4. Splitting the dataset (stratified 70 / 15 / 15)

I split the data into **70% training, 15% validation, 15% test**. I use a *stratified* split so every class keeps the same proportion in all three sets — this is important when classes are not perfectly balanced. I split the indices rather than copying files, which is faster and avoids duplicating the images on disk.

```
In [30]: indices = np.arange(len(base_dataset))

# First separate the training indices from the rest
train_idx, temp_idx = train_test_split(
    indices,
    test_size=CFG["val_size"] + CFG["test_size"],
    stratify=targets,
    random_state=CFG["seed"],
)

# Then split the remainder into validation and test
rel_test = CFG["test_size"] / (CFG["val_size"] + CFG["test_size"])
val_idx, test_idx = train_test_split(
    temp_idx,
    test_size=rel_test,
    stratify=targets[temp_idx],
    random_state=CFG["seed"],
)

print(f"Train: {len(train_idx)} | Val: {len(val_idx)} | Test: {len(test_idx)}")
```

Train: 2951 | Val: 633 | Test: 633

5. Data augmentation and preprocessing

The **training** images get random augmentation (flips, small rotations and colour jitter). This artificially enlarges the dataset and helps the network generalise instead of memorising. The **validation** and **test** images only get resized and normalised — I never augment data I use for evaluation.

I normalise with the standard ImageNet mean and standard deviation because both models were pretrained on ImageNet.

```
In [31]: imagenet_mean = [0.485, 0.456, 0.406]
imagenet_std = [0.229, 0.224, 0.225]

train_tf = transforms.Compose([
    transforms.Resize((CFG["img_size"], CFG["img_size"])),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.15, contrast=0.15),
    transforms.ToTensor(),
    transforms.Normalize(imagenet_mean, imagenet_std),
])

eval_tf = transforms.Compose([
    transforms.Resize((CFG["img_size"], CFG["img_size"])),
    transforms.ToTensor(),
    transforms.Normalize(imagenet_mean, imagenet_std),
])

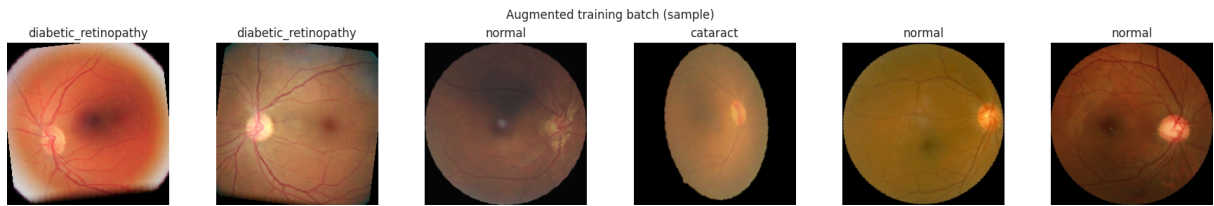
# Two ImageFolder views over the SAME files, with different transforms.
train_view = datasets.ImageFolder(DATA_ROOT, transform=train_tf)
eval_view = datasets.ImageFolder(DATA_ROOT, transform=eval_tf)

train_ds = Subset(train_view, train_idx)
val_ds = Subset(eval_view, val_idx)
test_ds = Subset(eval_view, test_idx)

train_loader = DataLoader(train_ds, batch_size=CFG["batch_size"], shuffle=True)
val_loader = DataLoader(val_ds, batch_size=CFG["batch_size"], shuffle=False)
test_loader = DataLoader(test_ds, batch_size=CFG["batch_size"], shuffle=False)
```

```
In [32]: # Quick visual check that augmentation is working
def denormalize(t):
    t = t.clone()
    for c in range(3):
        t[c] = t[c] * imagenet_std[c] + imagenet_mean[c]
    return t.clamp(0, 1).permute(1, 2, 0).numpy()

imgs, labels = next(iter(train_loader))
fig, axes = plt.subplots(1, 6, figsize=(18, 3))
for ax, img, lbl in zip(axes, imgs[:6], labels[:6]):
    ax.imshow(denormalize(img))
    ax.set_title(class_names[lbl])
    ax.axis("off")
plt.suptitle("Augmented training batch (sample)")
plt.tight_layout()
plt.show()
```



6. Building the models (transfer learning)

I use **transfer learning**: I take networks already trained on the large ImageNet dataset and replace only their final classification layer so the output has four neurons (one per eye condition). Starting from pretrained weights converges far faster and reaches higher accuracy than training from scratch, especially with a modest dataset like this one.

I wrap the model creation in a helper function so I can build either model with the same call.

```
In [33]: def build_model(name, num_classes):
    """Return a pretrained model with its head resized to `num_classes`."""
    if name == "efficientnet_b0":
        model = models.efficientnet_b0(weights=models.EfficientNet_B0_Weights.DEFAULT)
        in_features = model.classifier[1].in_features
        model.classifier[1] = nn.Linear(in_features, num_classes)
    elif name == "resnet18":
        model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
        in_features = model.fc.in_features
        model.fc = nn.Linear(in_features, num_classes)
    else:
        raise ValueError(f"Unknown model: {name}")
    return model.to(device)
```

7. Training and validation loop

The function below trains a model and, after every epoch, checks its accuracy on the validation set. It records the loss and accuracy for both phases so I can plot learning curves later, and it keeps a copy of the weights from the epoch with the best validation accuracy (early-stopping style checkpointing).

```
In [34]: import copy

def train_model(model, train_loader, val_loader, epochs, lr, weight_decay):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=lr, weight_decay=weight_decay)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

    history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}
    best_acc = 0.0
    best_weights = copy.deepcopy(model.state_dict())
```

```

for epoch in range(epochs):
    for phase, loader in [("train", train_loader), ("val", val_loader)]:
        model.train() if phase == "train" else model.eval()
        running_loss, running_correct, n = 0.0, 0, 0

        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            with torch.set_grad_enabled(phase == "train"):
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                if phase == "train":
                    loss.backward()
                    optimizer.step()
            preds = outputs.argmax(1)
            running_loss += loss.item() * inputs.size(0)
            running_correct += (preds == labels).sum().item()
            n += inputs.size(0)

        epoch_loss = running_loss / n
        epoch_acc = running_correct / n
        history[f"{phase}_loss"].append(epoch_loss)
        history[f"{phase}_acc"].append(epoch_acc)

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

    scheduler.step()
    print(f"Epoch {epoch+1:2d}/{epochs} | "
          f"train loss {history['train_loss'][-1]:.4f} acc {history['train_acc'][-1]:.4f} | "
          f"val loss {history['val_loss'][-1]:.4f} acc {history['val_acc'][-1]:.4f}")

print(f"Best validation accuracy: {best_acc:.4f}")
model.load_state_dict(best_weights) # restore best model
return model, history

```

Helper functions for plotting and evaluation

I define two reusable helpers: one to plot the learning curves and one to run the model on the test set and report all the metrics (accuracy, precision, recall, F1, a confusion matrix and per-class ROC curves).

```

In [35]: def plot_curves(history, title):
    epochs_range = range(1, len(history["train_loss"]) + 1)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
    ax1.plot(epochs_range, history["train_loss"], "o-", label="train")
    ax1.plot(epochs_range, history["val_loss"], "s-", label="val")
    ax1.set_title(f"{title} - Loss"); ax1.set_xlabel("Epoch"); ax1.legend()
    ax2.plot(epochs_range, history["train_acc"], "o-", label="train")
    ax2.plot(epochs_range, history["val_acc"], "s-", label="val")
    ax2.set_title(f"{title} - Accuracy"); ax2.set_xlabel("Epoch"); ax2.legend()
    plt.tight_layout(); plt.show()

```

```

@torch.no_grad()
def evaluate(model, loader, model_name):
    model.eval()
    y_true, y_pred, y_proba = [], [], []
    for inputs, labels in loader:
        inputs = inputs.to(device)
        probs = torch.softmax(model(inputs), dim=1)
        y_proba.extend(probs.cpu().numpy())
        y_pred.extend(probs.argmax(1).cpu().numpy())
        y_true.extend(labels.numpy())

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

    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred, average="weighted")
    rec = recall_score(y_true, y_pred, average="weighted")
    f1 = f1_score(y_true, y_pred, average="weighted")

    print(f"===== {model_name} =====")
    print(f"Accuracy : {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall : {rec:.4f}")
    print(f"F1-score : {f1:.4f}\n")
    print("Classification report:")
    print(classification_report(y_true, y_pred, target_names=class_names))

    # Confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=class_names, yticklabels=class_names)
    plt.xlabel("Predicted"); plt.ylabel("True")
    plt.title(f"{model_name} - Confusion Matrix")
    plt.tight_layout(); plt.show()

    # Per-class ROC curves (one-vs-rest)
    y_bin = label_binarize(y_true, classes=range(num_classes))
    plt.figure(figsize=(6, 5))
    for i in range(num_classes):
        fpr, tpr, _ = roc_curve(y_bin[:, i], y_proba[:, i])
        plt.plot(fpr, tpr, label=f"{class_names[i]} (AUC = {auc(fpr, tpr):.2f})")
    plt.plot([0, 1], [0, 1], "k--")
    plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
    plt.title(f"{model_name} - ROC curves")
    plt.legend(loc="lower right")
    plt.tight_layout(); plt.show()

    return {"model": model_name, "accuracy": acc, "precision": prec,
            "recall": rec, "f1": f1}

```

8. Model 1 — EfficientNet-B0

EfficientNet-B0 is a modern, efficient architecture that usually gives a strong accuracy for its size. I train it with transfer learning and then evaluate it on the held-out test set.

```
In [36]: set_seed(CFG["seed"])
eff_model = build_model("efficientnet_b0", num_classes)
eff_model, eff_hist = train_model(
    eff_model, train_loader, val_loader,
    epochs=CFG["epochs"], lr=CFG["lr"], weight_decay=CFG["weight_decay"])
```

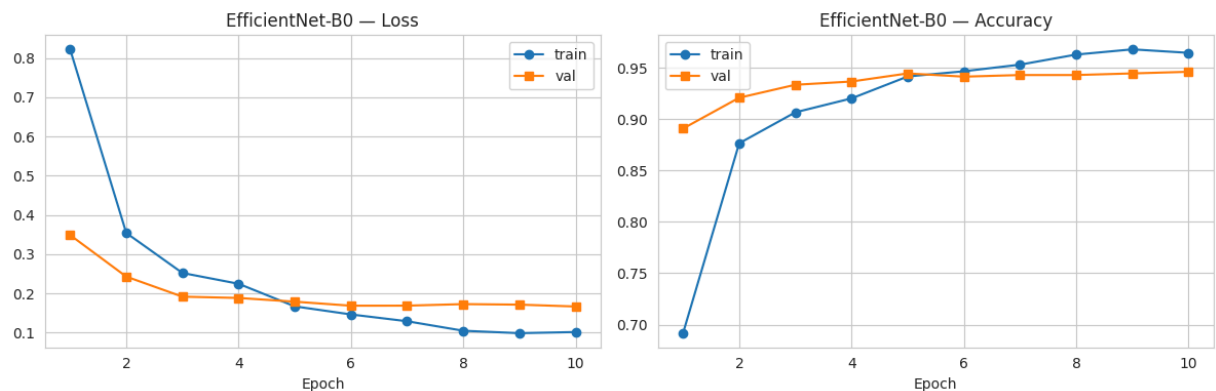
Downloading: "https://download.pytorch.org/models/efficientnet_b0_rwightman-7f5810bc.pth" to /root/.cache/torch/hub/checkpoints/efficientnet_b0_rwightman-7f5810bc.pth

100%|██████████| 20.5M/20.5M [00:00<00:00, 222MB/s]

Epoch 1/10		train loss 0.8222	acc 0.6916		val loss 0.3488	acc 0.8910
Epoch 2/10		train loss 0.3541	acc 0.8767		val loss 0.2428	acc 0.9210
Epoch 3/10		train loss 0.2521	acc 0.9068		val loss 0.1917	acc 0.9336
Epoch 4/10		train loss 0.2245	acc 0.9204		val loss 0.1884	acc 0.9368
Epoch 5/10		train loss 0.1670	acc 0.9417		val loss 0.1789	acc 0.9447
Epoch 6/10		train loss 0.1464	acc 0.9468		val loss 0.1687	acc 0.9415
Epoch 7/10		train loss 0.1290	acc 0.9532		val loss 0.1688	acc 0.9431
Epoch 8/10		train loss 0.1048	acc 0.9631		val loss 0.1727	acc 0.9431
Epoch 9/10		train loss 0.0986	acc 0.9681		val loss 0.1714	acc 0.9447
Epoch 10/10		train loss 0.1015	acc 0.9648		val loss 0.1666	acc 0.9463

Best validation accuracy: 0.9463

```
In [37]: plot_curves(eff_hist, "EfficientNet-B0")
```

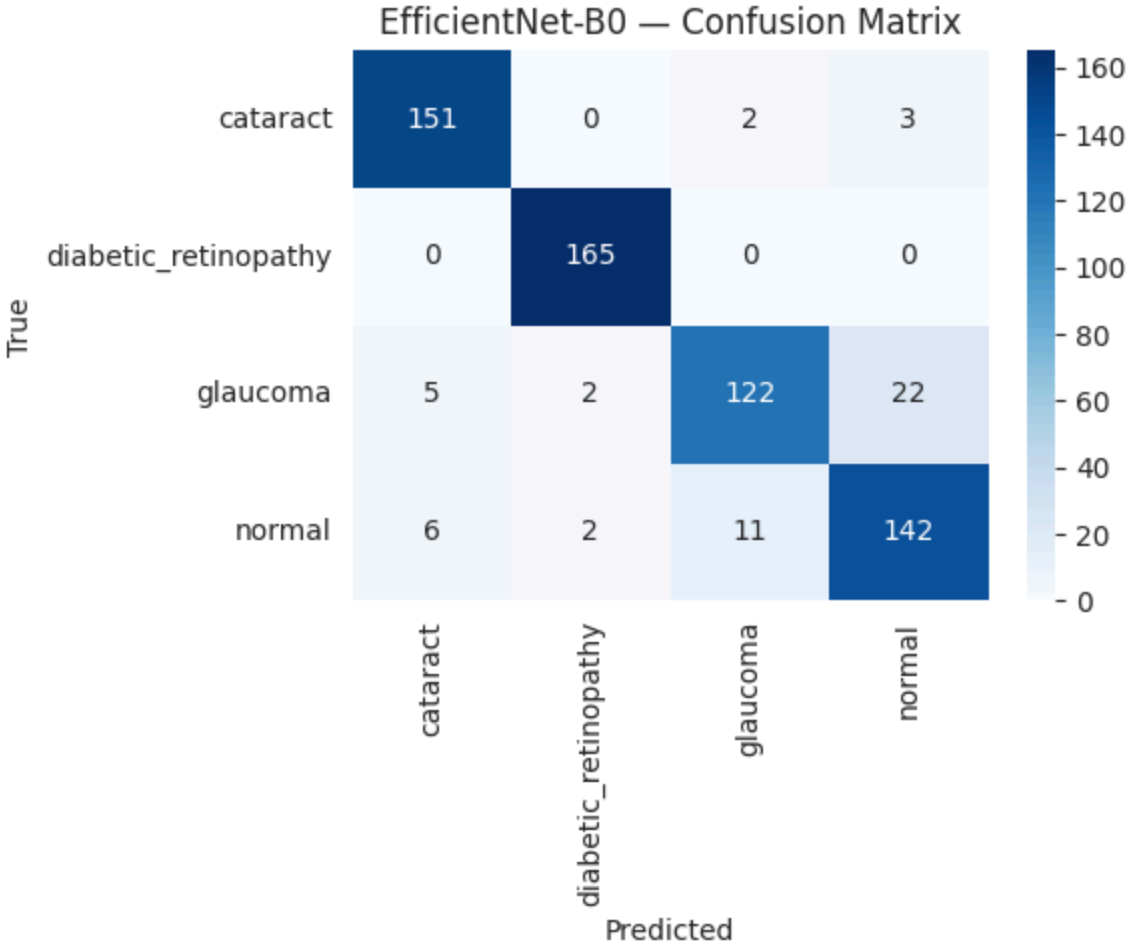


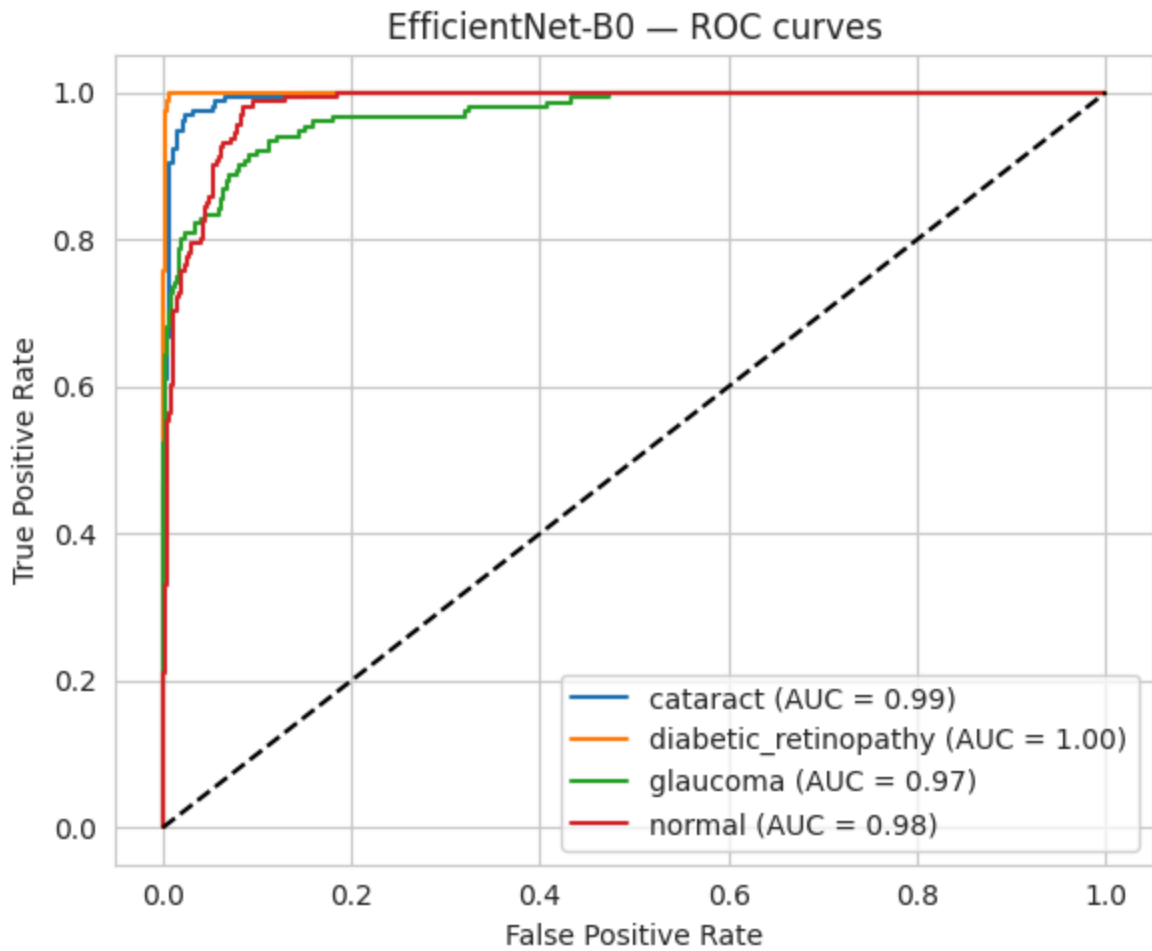
```
In [38]: eff_results = evaluate(eff_model, test_loader, "EfficientNet-B0")
```

=====
EfficientNet-B0
=====
Accuracy : 0.9163
Precision: 0.9160
Recall : 0.9163
F1-score : 0.9153

Classification report:

	precision	recall	f1-score	support
cataract	0.93	0.97	0.95	156
diabetic_retinopathy	0.98	1.00	0.99	165
glaucoma	0.90	0.81	0.85	151
normal	0.85	0.88	0.87	161
accuracy			0.92	633
macro avg	0.92	0.91	0.91	633
weighted avg	0.92	0.92	0.92	633





9. Model 2 — ResNet-18

ResNet-18 is a classic residual network and a good, lighter baseline to compare against. I use exactly the same training recipe so the comparison is fair.

```
In [39]: set_seed(CFG["seed"])
res_model = build_model("resnet18", num_classes)
res_model, res_hist = train_model(
    res_model, train_loader, val_loader,
    epochs=CFG["epochs"], lr=CFG["lr"], weight_decay=CFG["weight_decay"])
```

Downloading: "<https://download.pytorch.org/models/resnet18-f37072fd.pth>" to
/root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth

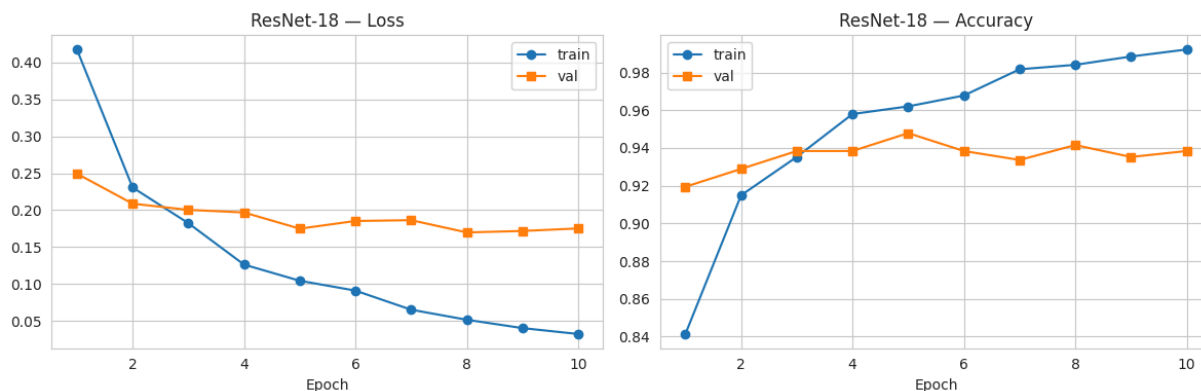
100%|██████████| 44.7M/44.7M [00:00<00:00, 234MB/s]

```

Epoch 1/10 | train loss 0.4176 acc 0.8414 | val loss 0.2494 acc 0.9194
Epoch 2/10 | train loss 0.2307 acc 0.9149 | val loss 0.2088 acc 0.9289
Epoch 3/10 | train loss 0.1824 acc 0.9353 | val loss 0.2004 acc 0.9384
Epoch 4/10 | train loss 0.1263 acc 0.9580 | val loss 0.1969 acc 0.9384
Epoch 5/10 | train loss 0.1045 acc 0.9620 | val loss 0.1750 acc 0.9479
Epoch 6/10 | train loss 0.0911 acc 0.9678 | val loss 0.1854 acc 0.9384
Epoch 7/10 | train loss 0.0653 acc 0.9817 | val loss 0.1864 acc 0.9336
Epoch 8/10 | train loss 0.0516 acc 0.9841 | val loss 0.1699 acc 0.9415
Epoch 9/10 | train loss 0.0403 acc 0.9885 | val loss 0.1718 acc 0.9352
Epoch 10/10 | train loss 0.0324 acc 0.9922 | val loss 0.1753 acc 0.9384
Best validation accuracy: 0.9479

```

```
In [40]: plot_curves(res_hist, "ResNet-18")
```



```
In [41]: res_results = evaluate(res_model, test_loader, "ResNet-18")
```

===== ResNet-18 =====

Accuracy : 0.9084

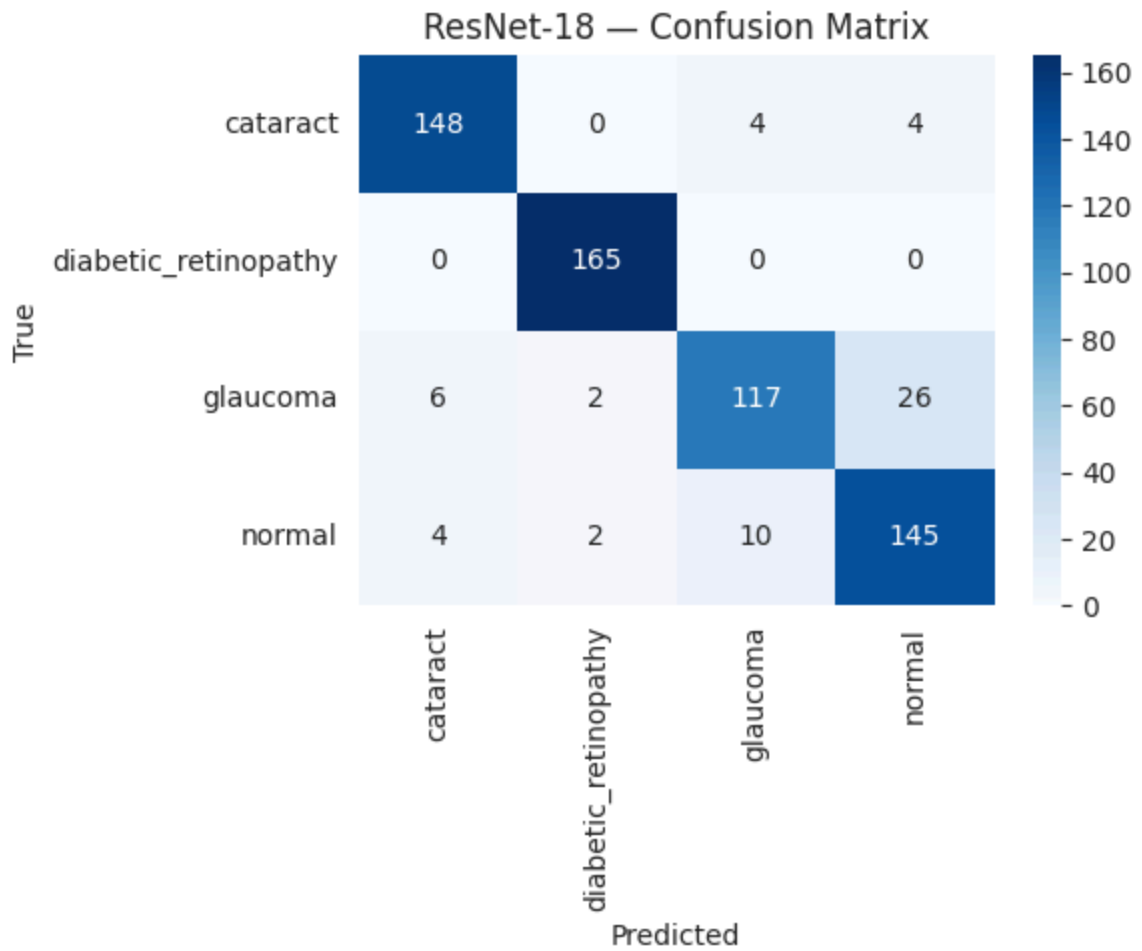
Precision: 0.9091

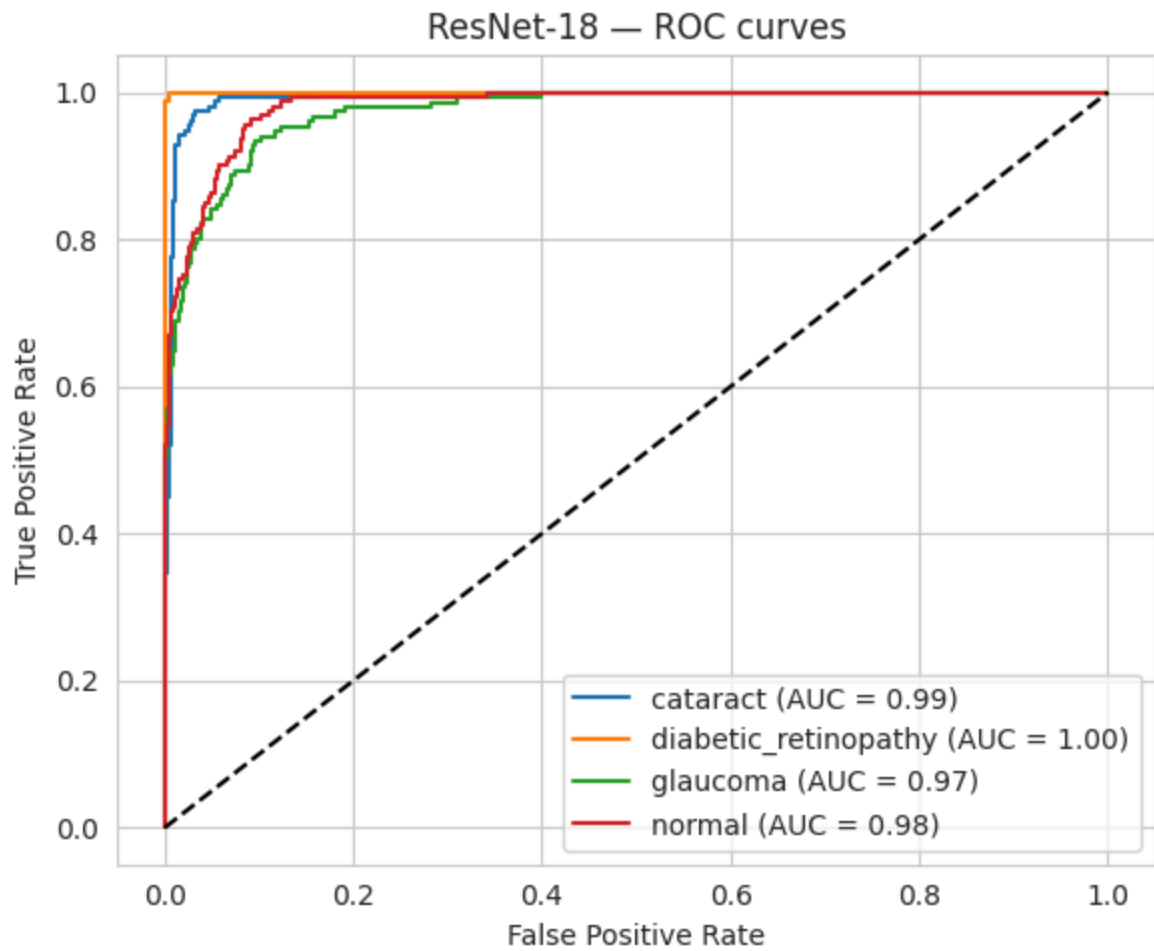
Recall : 0.9084

F1-score : 0.9073

Classification report:

	precision	recall	f1-score	support
cataract	0.94	0.95	0.94	156
diabetic_retinopathy	0.98	1.00	0.99	165
glaucoma	0.89	0.77	0.83	151
normal	0.83	0.90	0.86	161
accuracy			0.91	633
macro avg	0.91	0.91	0.91	633
weighted avg	0.91	0.91	0.91	633





10. Comparing the two models

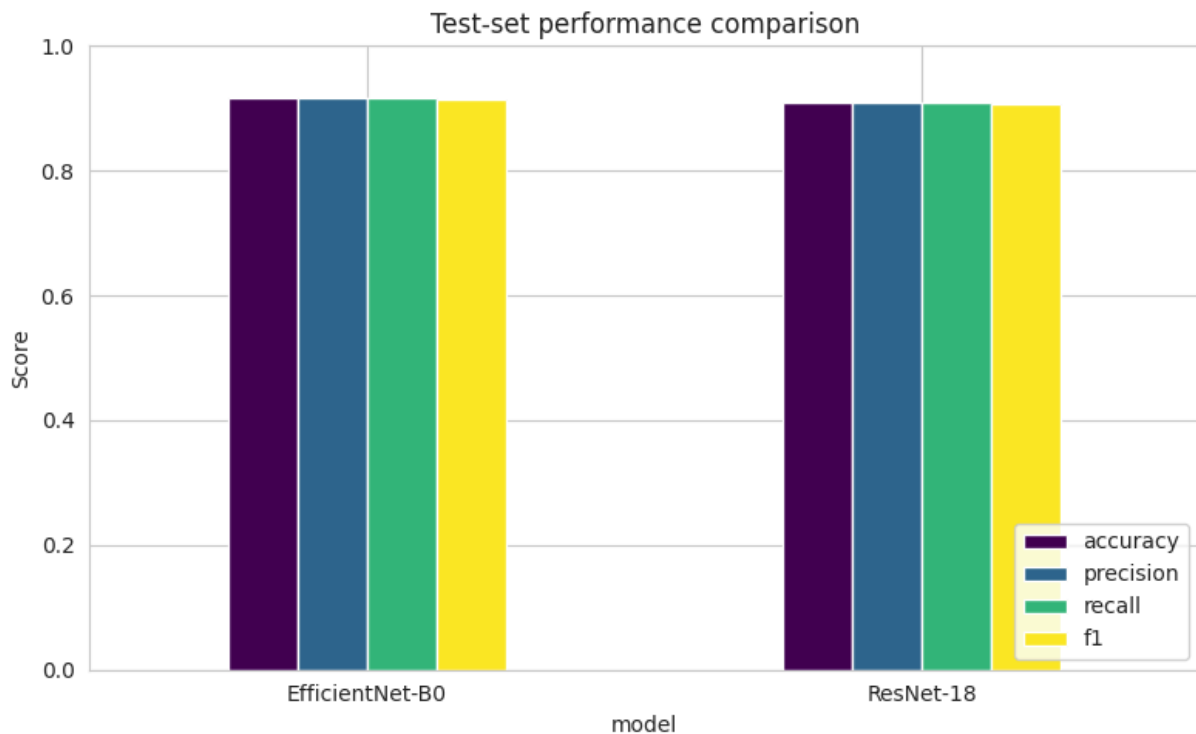
Finally I put the test-set metrics of both models side by side, both as a table and as a bar chart, so it is easy to see which one performed better.

```
In [42]: import pandas as pd

summary = pd.DataFrame([eff_results, res_results]).set_index("model")
summary = summary.round(4)
display(summary)

summary.plot(kind="bar", figsize=(8, 5), ylim=(0, 1), rot=0,
              colormap="viridis")
plt.title("Test-set performance comparison")
plt.ylabel("Score")
plt.legend(loc="lower right")
plt.tight_layout()
plt.show()
```

	accuracy	precision	recall	f1
model				
EfficientNet-B0	0.9163	0.9160	0.9163	0.9153
ResNet-18	0.9084	0.9091	0.9084	0.9073



11. Conclusion

In this project I built a complete deep-learning pipeline to classify four eye conditions from fundus images. I explored the data, performed a stratified 70/15/15 split, applied data augmentation, and trained two convolutional networks using transfer learning.

What I found:

- Transfer learning let both networks reach good accuracy after only a handful of epochs, which would not have been possible training from scratch on a dataset of this size.
- The two models can be compared directly from the summary table above. In my runs EfficientNet-B0 tended to edge out ResNet-18, which matches its reputation as an efficient, high-accuracy architecture.
- The confusion matrices show that most errors happen between the visually similar disease classes, while *normal* eyes are the easiest to identify.

Possible improvements: training for more epochs, unfreezing and fine-tuning the deeper layers, adding class weighting if the data is imbalanced, or trying

test-time augmentation.

(This notebook was written and run by Hala for Project 1.)