

This practical assignment contains the complete workflow of an **image classification project** using deep learning techniques applied to the *Lung and Colon Cancer Histopathological Images (LC25000)* dataset. The following steps were performed systematically:

1. Dataset Acquisition

- Selected the *Lung and Colon Cancer Histopathological Images (LC25000)* dataset from Kaggle.
<https://www.kaggle.com/datasets/andrewmvd/lung-and-colon-cancer-histopathological-images>
- Downloaded the dataset archive and uploaded it to Google Drive for use in Google Colab.

2. Environment Setup

- Mounted Google Drive and created structured folders for input, processing, and outputs.
- Implemented flexible path handling to automatically detect dataset locations.
- Copied the dataset from Drive to Colab's local environment for faster access.

3. Data Preparation & Pre-processing

- Extracted and verified image folders (colon and lung sub-categories).
- Automatically detected class directories and confirmed five classes: `colon_aca`, `colon_n`, `lung_aca`, `lung_n`, `lung_scc`.
- Resized all 25,000 images uniformly to **256 × 256 pixels**, skipping corrupted files safely.
- Confirmed final count: 24,998 valid images successfully processed.

4. Dataset Splitting

- Divided the dataset into three subsets:
 - **Training:** 70%
 - **Validation:** 10%
 - **Testing:** 20%
- Ensured balanced distribution across all classes to avoid bias.

5. Data Augmentation

- Applied random horizontal and vertical flips, small rotations, and normalization.
- Enhanced model generalization and reduced the risk of overfitting.

6. Model Selection & Training

- Used **EfficientNet-B0**, a state-of-the-art convolutional neural network pre-trained on ImageNet.
- Replaced the final classification layer to match five target classes.
- Trained with **mixed precision (AMP)** for speed and lower GPU memory usage.
- Applied **AdamW optimizer**, **cosine learning-rate scheduler**, and **early stopping** to prevent overfitting.
- Implemented a two-phase strategy: first froze the backbone, then unfroze for fine-tuning.
- Achieved rapid convergence and high accuracy within approximately 16 minutes.

7. Model Evaluation

- Loaded the best checkpoint and evaluated performance on the test set.
- Computed overall metrics:
 - **Accuracy:** 0.9988
 - **Precision / Recall / F1-score:** 0.9988 (weighted)
- Generated a detailed per-class classification report and confusion matrices (raw and normalized).

8. Results Archiving & Synchronization

- Saved all key artifacts:
 - Trained weights (`best_efficientnet_b0.pth`, `efficientnet_b0_finetuned.pth`)
 - Training history (`train_history.json`)
 - Test metrics (`test_metrics_rich.npz`)
 - Per-class results (`per_class_metrics.csv`)
- Synced all files to Google Drive inside `MyDrive/Practical_Image_Processing/artifacts_option1`.
- Created a single compressed archive (`artifacts_option1.zip`) for easy submission.

9. Final Report Generation

- Compiled training curves, performance metrics, and confusion matrices into an HTML report.

- Exported the report as **PDF** using *Print* → *Save as PDF* for official submission.

0) Mount Google Drive & set paths

- Put your dataset ZIP or folder inside the chosen Drive folder (default: `MyDrive/Practical_Image_Processing`)
- We will copy from Drive to **/content** for fast processing

```
from google.colab import drive
drive.mount('/content/drive')

import os, glob, zipfile, shutil
from pathlib import Path

# === Adjust this if your Drive folder is different ===
drive_root = '/content/drive/MyDrive/Practical_Image_Processing'
Path(drive_root).mkdir(parents=True, exist_ok=True)

# Local working root (fast storage)
local_root = '/content/Practical_Image_Processing'
Path(local_root).mkdir(parents=True, exist_ok=True)

print("Drive root:", drive_root)
print("Local root:", local_root)
```

```
Mounted at /content/drive
Drive root: /content/drive/MyDrive/Practical_Image_Processing
Local root: /content/Practical_Image_Processing
```

1) Copy dataset from Drive → Local (/content)

We support two input cases inside **drive_root**:

- `Archive.zip` (or any `*.zip`) → will be extracted locally
- A folder named `dataset_raw` → will be copied locally as-is

```
import os, shutil, glob, zipfile
from pathlib import Path

# Destination local folders
local_zip_dir = os.path.join(local_root, 'zip_inbox')
local_raw_dir = os.path.join(local_root, 'dataset_raw')
os.makedirs(local_zip_dir, exist_ok=True)
os.makedirs(local_raw_dir, exist_ok=True)

# 1) If a ZIP exists in Drive, copy it locally and extract
zip_candidates = glob.glob(os.path.join(drive_root, '*.zip'))
if len(zip_candidates) > 0:
    zip_path = zip_candidates[0] # take first zip found
    print("📁 Found ZIP in Drive:", os.path.basename(zip_path))
    local_zip_path = os.path.join(local_zip_dir, os.path.basename(zip_path))
    shutil.copy2(zip_path, local_zip_path)
    print("📁 Copied ZIP to local:", local_zip_path)

    with zipfile.ZipFile(local_zip_path, 'r') as z:
        z.extractall(local_raw_dir)
    print("✅ Extracted ZIP locally to:", local_raw_dir)

# 2) If a raw folder exists in Drive, copy it locally
drive_raw_dir = os.path.join(drive_root, 'dataset_raw')
if os.path.exists(drive_raw_dir) and len(os.listdir(drive_raw_dir)) > 0:
    print("📁 Found dataset_raw folder in Drive. Copying to local...")
    shutil.copytree(drive_raw_dir, local_raw_dir, dirs_exist_ok=True)
    print("✅ Copied dataset_raw to local:", local_raw_dir)

# Sanity check
if not os.path.exists(local_raw_dir) or len(os.listdir(local_raw_dir)) == 0:
    raise FileNotFoundError(
        "No data found locally. Please place either a ZIP (e.g., Archive.zip) or a 'dataset_raw' folder inside:\n"
```

```

        f"{drive_root}"
    )

# Show a preview
for root, dirs, files in os.walk(local_raw_dir):
    print(root, "->", len(files), "files, subfolders:", dirs)
    # stop early after first informative line
    break

```

```

📦 Found ZIP in Drive: archive.zip
➡ Copied ZIP to local: /content/Practical_Image_Processing/zip_inbox/archive.zip
✅ Extracted ZIP locally to: /content/Practical_Image_Processing/dataset_raw
📁 Found dataset_raw folder in Drive. Copying to local...
✅ Copied dataset_raw to local: /content/Practical_Image_Processing/dataset_raw
/content/Practical_Image_Processing/dataset_raw -> 0 files, subfolders: ['lung_colon_image_set']

```

2) Detect class root (auto-detect where the image class folders live)

```

# === Step 2 (diagnostic): show tree and smart-detect where classes live ===
import os
from collections import defaultdict

local_raw_dir = os.path.join(local_root, 'dataset_raw')

# Pretty print 2 levels of the folder tree for debugging
def show_tree(root, max_levels=2):
    print(f"📁 Tree under: {root}")
    for dirpath, dirnames, filenames in os.walk(root):
        level = dirpath[len(root):].count(os.sep)
        if level > max_levels:
            continue
        print("    " * level + f"- {os.path.basename(dirpath) or os.path.basename(root)} "
              f"-> {len(filenames)} files, subfolders: {dirnames[:6]}")
    print()

show_tree(local_raw_dir, max_levels=3)

# Heuristic: if there's exactly one top-level folder, descend into it (common with LC25000 zips)
entries = [os.path.join(local_raw_dir, d) for d in os.listdir(local_raw_dir)]
subdirs = [d for d in entries if os.path.isdir(d)]
if len(subdirs) == 1:
    class_root = subdirs[0]
else:
    class_root = local_raw_dir

print("🔍 Initial class_root guess:", class_root)

```

```

📁 Tree under: /content/Practical_Image_Processing/dataset_raw
- dataset_raw -> 0 files, subfolders: ['lung_colon_image_set']
- lung_colon_image_set -> 0 files, subfolders: ['lung_image_sets', 'colon_image_sets']
- lung_image_sets -> 0 files, subfolders: ['lung_scc', 'lung_n', 'lung_aca']
  - lung_scc -> 5000 files, subfolders: []
  - lung_n -> 5000 files, subfolders: []
  - lung_aca -> 5000 files, subfolders: []
- colon_image_sets -> 0 files, subfolders: ['colon_aca', 'colon_n']
  - colon_aca -> 5000 files, subfolders: []
  - colon_n -> 5000 files, subfolders: []

🔍 Initial class_root guess: /content/Practical_Image_Processing/dataset_raw/lung_colon_image_set

```

3) Resize images to 256×256 (locally in /content)

```

# === Step 3: Robust recursive resize (find classes automatically) ===
from PIL import Image
import os
import glob

# We'll search all images under class_root recursively
IMG_EXT = ('.png', '.jpg', '.jpeg', '.bmp', '.tiff', '.tif', '.webp')

```

```

# Find all image files recursively
all_imgs = [p for p in glob.glob(os.path.join(class_root, "**", "**"), recursive=True)
             if os.path.isfile(p) and p.lower().endswith(IMG_EXT)]

print(f"🔍 Found {len(all_imgs)} image files (recursive search)")

# Infer class as the *immediate parent directory* of each image
# e.g. .../lung_colon_image_set/colon_aca/xxx.jpg -> class = "colon_aca"
def parent_dir(path):
    return os.path.basename(os.path.dirname(path))

# Build class -> list of images
by_class = {}
for img in all_imgs:
    cls = parent_dir(img)
    # Skip if parent is the container folder (e.g., 'lung_colon_image_set') and not a true class
    # Heuristic: treat as class only if it has at least one image inside
    by_class.setdefault(cls, []).append(img)

# Filter out obviously wrong "classes" (those whose parent folder also has only subfolders and no images)
# If a "class" has very few images AND its parent-of-parent looks like the real class folder, we'll keep anyway.
# In practice, LC25000 has class folders like colon_aca, colon_benign, lung_aca, lung_benign.

# Create output dir
local_resized_dir = os.path.join(local_root, "dataset_256")
os.makedirs(local_resized_dir, exist_ok=True)

# Only keep classes that have at least some images
valid_classes = {c: imgs for c, imgs in by_class.items() if len(imgs) > 0}
print("📁 Detected classes:", list(valid_classes.keys()))

# Resize
resize_size = (256, 256)
count, failed = 0, 0

for cls, img_paths in valid_classes.items():
    out_dir = os.path.join(local_resized_dir, cls)
    os.makedirs(out_dir, exist_ok=True)
    for src in img_paths:
        base = os.path.splitext(os.path.basename(src))[0] + ".jpg"
        dst = os.path.join(out_dir, base)
        try:
            with Image.open(src) as im:
                im = im.convert("RGB").resize(resize_size)
                im.save(dst, "JPEG", quality=90)
            count += 1
        except Exception as e:
            failed += 1

print(f"✅ Resized {count} images; ❌ failed: {failed}")
print("📁 Resized dataset:", local_resized_dir)

```

```

🔍 Found 25000 image files (recursive search)
📁 Detected classes: ['lung_scc', 'lung_n', 'lung_aca', 'colon_aca', 'colon_n']
✅ Resized 24998 images; ❌ failed: 2
📁 Resized dataset: /content/Practical_Image_Processing/dataset_256

```

4) Split into Train/Val/Test (70/10/20) — locally

```

# === Step 4 (robust): Recursive split into Train/Val/Test (70/10/20) ===
import os, shutil, glob
from collections import defaultdict
from sklearn.model_selection import train_test_split

# where Step 3 saved the resized images
local_resized_dir = os.path.join(local_root, "dataset_256")

# where we will put the split sets (locally, fast)
local_split_dir = os.path.join(local_root, "dataset_split")
train_dir = os.path.join(local_split_dir, "train")
val_dir = os.path.join(local_split_dir, "val")
test_dir = os.path.join(local_split_dir, "test")
for d in [train_dir, val_dir, test_dir]:

```

```

os.makedirs(d, exist_ok=True)

# image extensions to consider
IMG_EXT = ('.png', '.jpg', '.jpeg', '.bmp', '.tiff', '.tif', '.webp')

# 1) recursively find class folders (folders that actually contain image files)
class_to_files = defaultdict(list)
for dirpath, dirnames, filenames in os.walk(local_resized_dir):
    # keep only image files
    imgs = [f for f in filenames if f.lower().endswith(IMG_EXT)]
    if len(imgs) > 0:
        # treat this folder as a class folder
        cls_name = os.path.basename(dirpath) # e.g. colon_aca, lung_scc, ...
        full_paths = [os.path.join(dirpath, f) for f in imgs]
        class_to_files[cls_name].extend(full_paths)

# safety check
total_imgs = sum(len(v) for v in class_to_files.values())
if total_imgs == 0:
    raise RuntimeError(
        f"No images found under {local_resized_dir}. "
        "Make sure Step 3 (resize) succeeded and produced images."
    )

# 2) split ratios
train_ratio, val_ratio, test_ratio = 0.7, 0.1, 0.2

def safe_split(files):
    """Split list of files into train/val/test with edge-case handling."""
    if len(files) < 3:
        # too few: put all in train
        return files, [], []
    # first split off (val+test)
    train_files, temp_files = train_test_split(
        files, test_size=(1 - train_ratio), random_state=42, shuffle=True
    )
    # split temp into val/test by relative ratio
    if len(temp_files) == 0:
        return train_files, [], []
    test_size = test_ratio / (val_ratio + test_ratio) if (val_ratio + test_ratio) > 0 else 1.0
    val_files, test_files = train_test_split(
        temp_files, test_size=test_size, random_state=42, shuffle=True
    )
    return train_files, val_files, test_files

# 3) perform split & copy
summary = []
for cls, files in sorted(class_to_files.items()):
    tr, va, te = safe_split(files)

    # destination subfolders
    d_tr = os.path.join(train_dir, cls); os.makedirs(d_tr, exist_ok=True)
    d_va = os.path.join(val_dir, cls); os.makedirs(d_va, exist_ok=True)
    d_te = os.path.join(test_dir, cls); os.makedirs(d_te, exist_ok=True)

    for src in tr: shutil.copy2(src, d_tr)
    for src in va: shutil.copy2(src, d_va)
    for src in te: shutil.copy2(src, d_te)

    summary.append((cls, len(files), len(tr), len(va), len(te)))

print("✅ Split complete at:", local_split_dir)
print("\nPer-class counts (total / train / val / test):")
for cls, tot, trc, vac, tec in summary:
    print(f"  - {cls:<12} {tot:5d} | train {trc:5d} | val {vac:5d} | test {tec:5d}")

# quick overall check
overall_train = sum(s[2] for s in summary)
overall_val = sum(s[3] for s in summary)
overall_test = sum(s[4] for s in summary)
print(f"\nOverall -> train: {overall_train}, val: {overall_val}, test: {overall_test}, total: {overall_train+overall_val+overall_test}")

```

✅ Split complete at: /content/Practical_Image_Processing/dataset_split

Per-class counts (total / train / val / test):
 - colon_aca 5000 | train 3499 | val 500 | test 1001

- colon_n	5000		train	3499		val	500		test	1001
- lung_aca	4998		train	3498		val	500		test	1000
- lung_n	5000		train	3499		val	500		test	1001
- lung_scc	5000		train	3499		val	500		test	1001

Overall -> train: 17494, val: 2500, test: 5004, total: 24998

5) Data augmentation & loaders (local paths)

```
# Step 5 - Data augmentation & loaders (with extra checks)
import torch
from torch.utils.data import DataLoader
from torchvision import transforms as T, datasets
import os

train_data_dir = os.path.join(local_root, "dataset_split", "train")
val_data_dir   = os.path.join(local_root, "dataset_split", "val")
test_data_dir  = os.path.join(local_root, "dataset_split", "test")

batch_size = 32 # if you hit OOM, try 16

train_tfms = 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]),
])

eval_tfms = T.Compose([
    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(train_data_dir, transform=train_tfms)
val_data   = datasets.ImageFolder(val_data_dir,   transform=eval_tfms)
test_data  = datasets.ImageFolder(test_data_dir,  transform=eval_tfms)

num_workers = 2 # you can try 4 if Colab is stable
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True,
                           num_workers=num_workers, pin_memory=True, persistent_workers=False)
val_loader   = DataLoader(val_data,   batch_size=batch_size, shuffle=False,
                           num_workers=num_workers, pin_memory=True, persistent_workers=False)
test_loader  = DataLoader(test_data,  batch_size=batch_size, shuffle=False,
                           num_workers=num_workers, pin_memory=True, persistent_workers=False)

dataloaders = {"train": train_loader, "val": val_loader}
dataset_sizes = {"train": len(train_data), "val": len(val_data)}
classes = train_data.classes
num_classes = len(classes)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print("Classes:", classes)
print("num_classes:", num_classes)
print("GPU available:", torch.cuda.is_available())
print("Split sizes -> train:", len(train_data), "val:", len(val_data), "test:", len(test_data))
```

```
Classes: ['colon_aca', 'colon_n', 'lung_aca', 'lung_n', 'lung_scc']
num_classes: 5
GPU available: True
Split sizes -> train: 17494 val: 2500 test: 5004
```

6) Train EfficientNet-B0 (local)

```
# Step 6 - Train EfficientNet-B0 (fast + robust)
# - AMP mixed precision
# - Early stopping on val accuracy
# - Save best checkpoint
# - Optional freeze->unfreeze of backbone
```

```

import os, time, copy, math, json
import torch
from torch import nn, optim
from torchvision import models
from torch.cuda.amp import autocast, GradScaler
from tqdm import tqdm
import numpy as np
import random

# Reproducibility
SEED = 42
random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED); torch.cuda.manual_seed_all(SEED)
torch.backends.cudnn.benchmark = True # speed on fixed image sizes

NUM_EPOCHS = 10 # you can increase to 12-15 if time allows
FREEZE_EPOCHS = 2 # freeze backbone for the first N epochs, then unfreeze
BASE_LR = 1e-3
WEIGHT_DECAY = 1e-4
PATIENCE = 3 # early stop patience (epochs without val_acc improvement)
BEST_CKPT = os.path.join(local_root, "best_efficientnet_b0.pth")
HISTORY_JSON = os.path.join(local_root, "train_history.json")

# Model
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)
model = model.to(device)

# Optionally freeze backbone at start
def set_backbone_requires_grad(m, requires_grad: bool):
    for name, p in m.named_parameters():
        if not name.startswith("classifier"):
            p.requires_grad = requires_grad

set_backbone_requires_grad(model, False) # freeze backbone initially

criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()),
                           lr=BASE_LR, weight_decay=WEIGHT_DECAY)

# Cosine decay with warmup-like lower bound
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=NUM_EPOCHS, eta_min=BASE_LR*0.1)

scaler = GradScaler(enabled=torch.cuda.is_available())

history = {"epoch": [], "train_loss": [], "train_acc": [], "val_loss": [], "val_acc": [], "lr": []}
best_model_wts = copy.deepcopy(model.state_dict())
best_val_acc = 0.0
epochs_no_improve = 0

def run_epoch(phase):
    is_train = (phase == "train")
    model.train(mode=is_train)

    running_loss, running_correcets, seen = 0.0, 0, 0
    loader = dataloaders[phase]
    pbar = tqdm(loader, desc=f"{phase.title()}")

    for inputs, labels in pbar:
        inputs, labels = inputs.to(device), labels.to(device)

        if is_train:
            optimizer.zero_grad(set_to_none=True)

        with autocast(enabled=torch.cuda.is_available()):
            outputs = model(inputs)
            loss = criterion(outputs, labels)

        if is_train:
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()

        _, preds = torch.max(outputs, 1)
        bs = inputs.size(0)
        running_loss += loss.item() * bs

```

```

        running_corrects += torch.sum(preds == labels).item()
        seen += bs

    avg_loss = running_loss / max(seen, 1)
    avg_acc = running_corrects / max(seen, 1)
    pbar.set_postfix({"loss": f"{avg_loss:.4f}", "acc": f"{avg_acc:.4f}"})

    epoch_loss = running_loss / max(seen, 1)
    epoch_acc = running_corrects / max(seen, 1)
    return epoch_loss, epoch_acc

start_time = time.time()
for epoch in range(1, NUM_EPOCHS + 1):
    print(f"\nEpoch {epoch}/{NUM_EPOCHS}")
    print("-" * 30)

    # Unfreeze backbone after a few warmup epochs
    if epoch == FREEZE_EPOCHS + 1:
        set_backbone_requires_grad(model, True)
        optimizer = optim.AdamW(model.parameters(), lr=BASE_LR, weight_decay=WEIGHT_DECAY)
        print("🔥 Unfroze backbone and reset optimizer.")

    train_loss, train_acc = run_epoch("train")
    val_loss, val_acc = run_epoch("val")

    scheduler.step()

    # Log
    history["epoch"].append(epoch)
    history["train_loss"].append(train_loss)
    history["train_acc"].append(train_acc)
    history["val_loss"].append(val_loss)
    history["val_acc"].append(val_acc)
    history["lr"].append(optimizer.param_groups[0]["lr"])

    print(f"Train - loss: {train_loss:.4f} | acc: {train_acc:.4f}")
    print(f"Val - loss: {val_loss:.4f} | acc: {val_acc:.4f}")
    print(f"LR: {optimizer.param_groups[0]['lr']:.6f}")

    # Checkpoint on improvement
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_model_wts = copy.deepcopy(model.state_dict())
        torch.save(best_model_wts, BEST_CKPT)
        print(f"✅ New best val_acc: {best_val_acc:.4f} - saved to {BEST_CKPT}")
        epochs_no_improve = 0
    else:
        epochs_no_improve += 1
        print(f"No improvement for {epochs_no_improve} epoch(s).")
        if epochs_no_improve >= PATIENCE:
            print("❌ Early stopping triggered.")
            break

elapsed = time.time() - start_time
print(f"\n🕒 Total training time: {elapsed/60:.1f} min")

# Load best and save final weights too
model.load_state_dict(best_model_wts)
final_weights = os.path.join(local_root, "efficientnet_b0_finetuned.pth")
torch.save(model.state_dict(), final_weights)
print("📁 Best weights saved to:", BEST_CKPT)
print("📁 Final weights (same as best) saved to:", final_weights)

# Persist history for plots/reports
with open(HISTORY_JSON, "w") as f:
    json.dump(history, f, indent=2)
print("📄 Training history saved to:", HISTORY_JSON)

```

```
Train: 100%|██████████| 547/547 [01:42<00:00, 5.32it/s, loss=0.0440, acc=0.9850]
Val: 100%|██████████| 79/79 [00:11<00:00, 7.15it/s, loss=0.0135, acc=0.9928]
Train - loss: 0.0440 | acc: 0.9850
Val - loss: 0.0135 | acc: 0.9928
LR: 0.001000
✅ New best val_acc: 0.9928 - saved to /content/Practical_Image_Processing/best_efficientnet_b0.pth
```

Epoch 5/10

```
-----
Train: 100%|██████████| 547/547 [01:42<00:00, 5.34it/s, loss=0.0225, acc=0.9932]
Val: 100%|██████████| 79/79 [00:11<00:00, 7.09it/s, loss=0.0032, acc=0.9996]
Train - loss: 0.0225 | acc: 0.9932
Val - loss: 0.0032 | acc: 0.9996
LR: 0.001000
✅ New best val_acc: 0.9996 - saved to /content/Practical_Image_Processing/best_efficientnet_b0.pth
```

Epoch 6/10

```
-----
Train: 100%|██████████| 547/547 [01:42<00:00, 5.36it/s, loss=0.0275, acc=0.9923]
Val: 100%|██████████| 79/79 [00:10<00:00, 7.24it/s, loss=0.0175, acc=0.9924]
Train - loss: 0.0275 | acc: 0.9923
Val - loss: 0.0175 | acc: 0.9924
LR: 0.001000
No improvement for 1 epoch(s).
```

Epoch 7/10

```
-----
Train: 100%|██████████| 547/547 [01:42<00:00, 5.36it/s, loss=0.0241, acc=0.9927]
Val: 100%|██████████| 79/79 [00:11<00:00, 7.18it/s, loss=0.0083, acc=0.9964]
Train - loss: 0.0241 | acc: 0.9927
Val - loss: 0.0083 | acc: 0.9964
LR: 0.001000
No improvement for 2 epoch(s).
```

Epoch 8/10

```
-----
Train: 100%|██████████| 547/547 [01:42<00:00, 5.35it/s, loss=0.0244, acc=0.9915]
Val: 100%|██████████| 79/79 [00:11<00:00, 7.13it/s, loss=0.0064, acc=0.9988]Train - loss: 0.0244 | acc: 0.9915
Val - loss: 0.0064 | acc: 0.9988
LR: 0.001000
No improvement for 3 epoch(s).
■ Early stopping triggered.
```

```
🕒 Total training time: 16.3 min
💾 Best weights saved to: /content/Practical_Image_Processing/best_efficientnet_b0.pth
💾 Final weights (same as best) saved to: /content/Practical_Image_Processing/efficientnet_b0_finetuned.pth
📄 Training history saved to: /content/Practical_Image_Processing/train_history.json
```

7) Evaluate on Test set (local) & save metrics

Step 7 - Evaluate on Test set (load best weights, rich metrics, save artifacts)

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

import torch
from torchvision.utils import make_grid

# 1) Load best checkpoint before evaluating
best_ckpt = os.path.join(local_root, "best_efficientnet_b0.pth")
if os.path.exists(best_ckpt):
    model.load_state_dict(torch.load(best_ckpt, map_location=device))
    print("✅ Loaded best checkpoint:", best_ckpt)
else:
    print("⚠️ Best checkpoint not found, using current model in memory.")

# 2) Collect predictions
y_true, y_pred = [], []
model.eval()
with torch.no_grad():
    for x, y in test_loader:
        x = x.to(device)
        logits = model(x)
        preds = torch.argmax(logits, dim=1).cpu().numpy()
        y_pred.extend(preds)
        y_true.extend(y.numpy())
```

```

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

# 3) Metrics (overall + per-class)
acc = accuracy_score(y_true, y_pred)
prec_w = precision_score(y_true, y_pred, average='weighted', zero_division=0)
rec_w = recall_score(y_true, y_pred, average='weighted', zero_division=0)
f1_w = f1_score(y_true, y_pred, average='weighted', zero_division=0)

print(f"\nOverall:")
print(f" Accuracy: {acc:.4f}")
print(f" Precision: {prec_w:.4f} (weighted)")
print(f" Recall: {rec_w:.4f} (weighted)")
print(f" F1-score: {f1_w:.4f} (weighted)\n")

print("Per-class report:\n")
report_str = classification_report(y_true, y_pred, target_names=classes, zero_division=0)
print(report_str)

# 4) Confusion matrix (+ normalized)
cm = confusion_matrix(y_true, y_pred)
cm_norm = cm.astype(float) / cm.sum(axis=1, keepdims=True).clip(min=1)

fig = plt.figure(figsize=(7,6))
plt.imshow(cm, interpolation='nearest')
plt.title('Confusion Matrix (counts)')
plt.colorbar()
ticks = np.arange(len(classes))
plt.xticks(ticks, classes, rotation=45, ha='right')
plt.yticks(ticks, classes)
plt.ylabel('True')
plt.xlabel('Predicted')
plt.tight_layout()
plt.show()

fig = plt.figure(figsize=(7,6))
plt.imshow(cm_norm, interpolation='nearest')
plt.title('Confusion Matrix (row-normalized)')
plt.colorbar()
ticks = np.arange(len(classes))
plt.xticks(ticks, classes, rotation=45, ha='right')
plt.yticks(ticks, classes)
plt.ylabel('True')
plt.xlabel('Predicted')
plt.tight_layout()
plt.show()

# 5) Save artifacts locally
metrics_npz = os.path.join(local_root, "test_metrics_rich.npz")
np.savez(metrics_npz,
         acc=acc, prec_w=prec_w, rec_w=rec_w, f1_w=f1_w,
         cm=cm, cm_norm=cm_norm, classes=np.array(classes, dtype=object),
         y_true=y_true, y_pred=y_pred)
print(f"💾 Saved metrics ->", metrics_npz)

# Optional: also save a CSV-friendly per-class table
import pandas as pd
from sklearn.metrics import precision_recall_fscore_support
prec_c, rec_c, f1_c, support_c = precision_recall_fscore_support(
    y_true, y_pred, labels=list(range(len(classes))), zero_division=0
)
df_cls = pd.DataFrame({
    "class": classes,
    "precision": prec_c,
    "recall": rec_c,
    "f1": f1_c,
    "support": support_c
})
csv_path = os.path.join(local_root, "per_class_metrics.csv")
df_cls.to_csv(csv_path, index=False)
print(f"💾 Saved per-class CSV ->", csv_path)

# (Optional) Preview a few misclassifications for your report
show_miscls = False
if show_miscls:
    wrong_idx = np.where(y_true != y_pred)[0][:16]
    if len(wrong_idx) > 0:

```

```

imgs, titles = [], []
# re-iterate once to fetch those specific images
seen = 0
for x, y in test_loader:
    bs = x.size(0)
    x_dev = x.to(device)
    with torch.no_grad():
        pr = torch.argmax(model(x_dev), 1).cpu().numpy()
    y_np = y.numpy()
    for i in range(bs):
        if seen in wrong_idx:
            imgs.append(x[i])
            titles.append(f"T:{classes[y_np[i]]}\nP:{classes[pr[i]]}")
            if len(imgs) == 16:
                break
        seen += 1
    if len(imgs) == 16:
        break
if imgs:
    grid = make_grid(imgs, nrow=4, pad_value=1.0)
    plt.figure(figsize=(8,8))
    plt.imshow(np.transpose(grid.numpy(), (1,2,0)))
    plt.title("Sample Misclassifications")
    plt.axis('off')
    plt.show()

```


✓ Loaded best checkpoint: /content/Practical_Image_Processing/best_efficientnet_b0.pth

Overall:

Accuracy: 0.9988
Precision: 0.9988 (weighted)
Recall: 0.9988 (weighted)
F1-score: 0.9988 (weighted)

Per-class report:

8) Sync artifacts back to Drive

This copies trained weights, metrics, and the split dataset back to your Drive folder.

Step 8 – Sync artifacts back to Drive (clean + zipped)

```
import os, shutil, glob, json, math
from pathlib import Path
import shutil
```

---- where to put things on Drive ----

```
drive_artifacts = os.path.join(drive_root, "artifacts_option1")
os.makedirs(drive_artifacts, exist_ok=True)
```

---- what to copy from local (/content) ----

```
to_copy = [
    os.path.join(local_root, "best_efficientnet_b0.pth"),
    os.path.join(local_root, "efficientnet_b0_finetuned.pth"),
    os.path.join(local_root, "train_history.json"),
    os.path.join(local_root, "test_metrics_rich.npz"),
    os.path.join(local_root, "per_class_metrics.csv"),
]
```

optional HTML (if you ran the export cell)

```
htmls = glob.glob("/content/*.html")
to_copy += htmls
```

try to also copy the notebook itself (first *.ipynb in /content)

```
ipynbs = glob.glob("/content/*.ipynb")
if ipynbs:
    to_copy.append(ipynbs[0])
```

---- copy files ----

```
def human(n):
    for unit in ["B", "KB", "MB", "GB"]:
        if n < 1024: return f"{n:.1f} {unit}"
        n /= 1024
    return f"{n:.1f} TB"
```

print("📁 Copying key artifacts to Drive:")

```
for p in to_copy:
    if os.path.exists(p):
        dest = os.path.join(drive_artifacts, os.path.basename(p))
        shutil.copy2(p, dest)
        print(f" - {os.path.basename(p):35s} -> {human(os.path.getsize(dest))}")
    else:
        print(f" - (skip) {os.path.basename(p)} (not found)")
```

---- optionally copy the split dataset (can be large) ----

COPY_SPLIT = False # set True if you need full split dataset on Drive

local_split_dir = os.path.join(local_root, "dataset_split")

if COPY_SPLIT and os.path.exists(local_split_dir):

print("📁 Copying dataset_split to Drive (this may take a while)...")

shutil.copytree(local_split_dir, os.path.join(drive_artifacts, "dataset_split"), dirs_exist_ok=True)

print("✓ dataset_split copied.")

---- zip everything for easy sharing ----

zip_base = os.path.join(drive_root, "artifacts_option1")

zip_path = shutil.make_archive(zip_base, "zip", drive_artifacts)

print("\n✓ All artifacts synced to:", drive_artifacts)

print("📦 ZIP created at:", zip_path)

final listing

print("\n📁 Contents of artifacts_option1:")

for f in sorted(os.listdir(drive_artifacts)):

fp = os.path.join(drive_artifacts, f)

if os.path.isfile(fp):

```

        print(f" - {f:35s} {human(os.path.getsize(fp))}")
    else:
        print(f" - {f}/")

```

 Saved key artifacts to Practical_Image_Processing/test_metrics_rich.npz
 Saved per-class CSV to /content/Practical_Image_Processing/per_class_metrics.csv
 - best_efficientnet_b0.pth -> 15.6 MB
 - efficientnet_b0_finetuned.pth -> 15.6 MB
 - train_history.json -> 988.0 B
 - test_metrics_rich.npz -> 81.0 KB
 - per_class_metrics.csv -> 265.0 B

 All artifacts synced to: /content/drive/MyDrive/Practical_Image_Processing/artifacts_option1
 ZIP created at: /content/drive/MyDrive/Practical_Image_Processing/artifacts_option1.zip

 Contents of artifacts_option1:
 - best_efficientnet_b0.pth 15.6 MB
 - efficientnet_b0_finetuned.pth 15.6 MB
 - per_class_metrics.csv 265.0 B
 - test_metrics_rich.npz 81.0 KB
 - train_history.json 988.0 B

9) Export to PDF

Step 9 – Build a polished HTML report

```

import os, json, numpy as np, matplotlib.pyplot as plt
from datetime import datetime
from base64 import b64encode
import io

author_name = "Your Name" # <-- put your name here
course_name = "Practical Image Processing"
dataset_name = "Lung and Colon Cancer Histopathological Images (LC25000)"
run_time = datetime.now().strftime("%Y-%m-%d %H:%M")

# Paths
hist_path = os.path.join(local_root, "train_history.json")
npz_path = os.path.join(local_root, "test_metrics_rich.npz")
out_html = os.path.join(local_root, "final_report.html")

# Load artifacts
with open(hist_path, "r") as f:
    H = json.load(f)

D = np.load(npz_path, allow_pickle=True)
acc, prec_w, rec_w, f1_w = float(D["acc"]), float(D["prec_w"]), float(D["rec_w"]), float(D["f1_w"])
cm, cm_norm = D["cm"], D["cm_norm"]
classes = list(D["classes"])

# Helper: render a matplotlib fig to base64 <img>
def fig_to_base64(fig):
    buf = io.BytesIO()
    fig.savefig(buf, bbox_inches="tight")
    plt.close(fig)
    buf.seek(0)
    return "data:image/png;base64," + b64encode(buf.read()).decode("utf-8")

# Plot training curves
epochs = H["epoch"]
fig1 = plt.figure(figsize=(6,4))
plt.plot(epochs, H["train_loss"], label="train loss")
plt.plot(epochs, H["val_loss"], label="val loss")
plt.title("Loss")
plt.xlabel("epoch"); plt.ylabel("loss"); plt.legend()
loss_b64 = fig_to_base64(fig1)

fig2 = plt.figure(figsize=(6,4))
plt.plot(epochs, H["train_acc"], label="train acc")
plt.plot(epochs, H["val_acc"], label="val acc")
plt.title("Accuracy")
plt.xlabel("epoch"); plt.ylabel("accuracy"); plt.legend()
acc_b64 = fig_to_base64(fig2)

# Confusion matrices

```

```

fig3 = plt.figure(figsize=(6,5))
plt.imshow(cm, interpolation='nearest')
plt.title('Confusion Matrix (counts)')
plt.colorbar()
ticks = np.arange(len(classes))
plt.xticks(ticks, classes, rotation=45, ha='right')
plt.yticks(ticks, classes)
plt.ylabel('True'); plt.xlabel('Predicted')
cm_b64 = fig_to_base64(fig3)

fig4 = plt.figure(figsize=(6,5))
plt.imshow(cm_norm, interpolation='nearest')
plt.title('Confusion Matrix (row-normalized)')
plt.colorbar()
ticks = np.arange(len(classes))
plt.xticks(ticks, classes, rotation=45, ha='right')
plt.yticks(ticks, classes)
plt.ylabel('True'); plt.xlabel('Predicted')
cmn_b64 = fig_to_base64(fig4)

# Build simple HTML
html = f"""
<!doctype html>
<html>
<head>
<meta charset="utf-8" />
<title>Assignment Report – {author_name}</title>
<style>
body {{ font-family: Arial, sans-serif; margin: 24px; }}
h1,h2,h3 {{ margin: 0.2em 0; }}
hr {{ margin: 1.2em 0; }}
.table {{ border-collapse: collapse; }}
.table td, .table th {{ border: 1px solid #ccc; padding: 6px 8px; }}
.figure {{ margin: 12px 0; }}
.small {{ color: #666; font-size: 13px; }}
.code {{ background: #f6f8fa; padding: 8px; border-radius: 6px; font-family: monospace; }}
</style>
</head>
<body>

<h1>{course_name} – Assignment Report</h1>
<div class="small">Author: <b>{author_name}</b> • Generated: {run_time}</div>
<hr/>

<h2>Dataset</h2>
<p><b>{dataset_name}</b><br/>
Classes detected: {', '.join(classes)}<br/>
Preprocessing: resize 256×256, split 70/10/20, data augmentation (flip, rotate, normalize)</p>

<h2>Model & Training</h2>
<ul>
<li>Backbone: EfficientNet-B0 (ImageNet weights)</li>
<li>Optimizer: AdamW • LR schedule: Cosine</li>
<li>Mixed Precision (AMP), Early Stopping, Best Checkpoint</li>
</ul>

<h3>Learning Curves</h3>
<div class="figure"></div>
<div class="figure"></div>

<h2>Test Metrics</h2>
<table class="table">
<tr><th>Metric</th><th>Value</th></tr>
<tr><td>Accuracy</td><td>{acc:.4f}</td></tr>
<tr><td>Precision (weighted)</td><td>{prec_w:.4f}</td></tr>
<tr><td>Recall (weighted)</td><td>{rec_w:.4f}</td></tr>
<tr><td>F1-score (weighted)</td><td>{f1_w:.4f}</td></tr>
</table>

<h3>Confusion Matrices</h3>
<div class="figure"></div>
<div class="figure"></div>

<h2>Reproducibility</h2>
<p>Artifacts saved to Drive folder: <code>MyDrive/Practical_Image_Processing/artifacts_option1</code><br/>
Files: best_efficientnet_b0.pth, efficientnet_b0_finetuned.pth, train_history.json, per_class_metrics.csv, test_metrics_rich.np

```

```
</body></html>
"""
```

```
with open(out_html, "w", encoding="utf-8") as f:
    f.write(html)
```

```
print("✅ Report written to:", out_html)
print("Tip: Open the HTML (left file browser), then: File → Print → Save as PDF.")
```

✅ Report written to: /content/Practical_Image_Processing/final_report.html
Tip: Open the HTML (left file browser), then: File → Print → Save as PDF.

```
import os, glob, shutil
```

```
local_html = "/content/Practical_Image_Processing/final_report.html"
print("Exists locally?", os.path.exists(local_html), "->", local_html)
```

```
# List the folder to be sure
print("\nContents of /content/Practical_Image_Processing:")
for p in sorted(os.listdir("/content/Practical_Image_Processing")):
    print(" -", p)
```

```
# If not found, search everywhere under /content
if not os.path.exists(local_html):
    hits = glob.glob("/content/**/*.*html", recursive=True)
    print("\nSearched for any *.html under /content:")
    for h in hits[:20]:
        print(" -", h)
    if not hits:
        print("No HTML files found - re-run the Step 9 cell to regenerate.")
```

```
# Copy to Drive so it's visible there too
drive_copy = "/content/drive/MyDrive/Practical_Image_Processing/artifacts_option1/final_report.html"
os.makedirs(os.path.dirname(drive_copy), exist_ok=True)
if os.path.exists(local_html):
    shutil.copy2(local_html, drive_copy)
    print("\n✅ Copied to Drive:", drive_copy)
else:
    print("\n⚠️ Local final_report.html not found; nothing copied. Re-run the report cell first.")
```

Exists locally? True -> /content/Practical_Image_Processing/final_report.html

```
Contents of /content/Practical_Image_Processing:
- best_efficientnet_b0.pth
- dataset_256
- dataset_raw
- dataset_split
- efficientnet_b0_finetuned.pth
- final_report.html
- per_class_metrics.csv
- test_metrics_rich.npz
- train_history.json
- zip_inbox
```

✅ Copied to Drive: /content/drive/MyDrive/Practical_Image_Processing/artifacts_option1/final_report.html

10) Checking the Model

```
# Check the model
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
import os, glob, csv, torch
from PIL import Image
from torchvision import transforms as T
from torchvision import models
from torch import nn
import numpy as np
```

```
drive_root = '/content/drive/MyDrive/Practical_Image_Processing'
folder = f'{drive_root}/Sample_Image' # predict everything in this folder
```

```
# Classes (reuse if present; else define):
try:
```

```

        classes = classes
    except NameError:
        classes = ['colon_aca', 'colon_n', 'lung_aca', 'lung_n', 'lung_scc']

num_classes = len(classes)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Model
local_root = '/content/Practical_Image_Processing'
model = models.efficientnet_b0(weights=None)
in_features = model.classifier[1].in_features
model.classifier[1] = nn.Linear(in_features, num_classes)
model.load_state_dict(torch.load(os.path.join(local_root, 'best_efficientnet_b0.pth'), map_location=device))
model.eval().to(device)

# TFMs
tfms = T.Compose([
    T.Resize((299, 299)),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# Gather images
IMG_EXT = ('.png', '.jpg', '.jpeg', '.bmp', '.tiff', '.tif', '.webp')

```