```
# ================================================================
# 📦 PHASE 1: Validate Paths & Load Sample Image
# ================================================================

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

# ----------------------------------------------------------------
# STEP 1 — Set Correct Base Path
# ----------------------------------------------------------------

import os

base_path = "/content/drive/MyDrive/Skin Cancer Images"
folders = ["imgs_part_1", "imgs_part_2", "imgs_part_3"]
metadata_path = os.path.join(base_path, "metadata.csv")

print("Base Path:", base_path)
print("Metadata Path:", metadata_path)

# ----------------------------------------------------------------
# STEP 2 — Confirm Folders & Print Status
# ----------------------------------------------------------------

for folder in folders:
    path = os.path.join(base_path, folder)
    print(f"{folder} → Exists:", os.path.exists(path))

# Check metadata.csv
print("metadata.csv → Exists:", os.path.exists(metadata_path))

# ----------------------------------------------------------------
# STEP 3 — Load a Sample Image to Verify Readability
# ----------------------------------------------------------------

import matplotlib.pyplot as plt
import cv2

sample_folder = os.path.join(base_path, "imgs_part_1")
sample_filename = os.listdir(sample_folder)[0]
sample_image_path = os.path.join(sample_folder, sample_filename)

# Load image using CV2
img = cv2.imread(sample_image_path)

if img is None:
    print("❌ ERROR: Image could not be loaded. Check file path.")
else:
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(img_rgb)
    plt.axis('off')
    plt.title(f"Sample Loaded Successfully: {sample_filename}")
    plt.show()
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Base Path: /content/drive/MyDrive/Skin Cancer Images
Metadata Path: /content/drive/MyDrive/Skin Cancer Images/metadata.csv
imgs_part_1 → Exists: True
imgs_part_2 → Exists: True
imgs_part_3 → Exists: True
metadata.csv → Exists: True
```

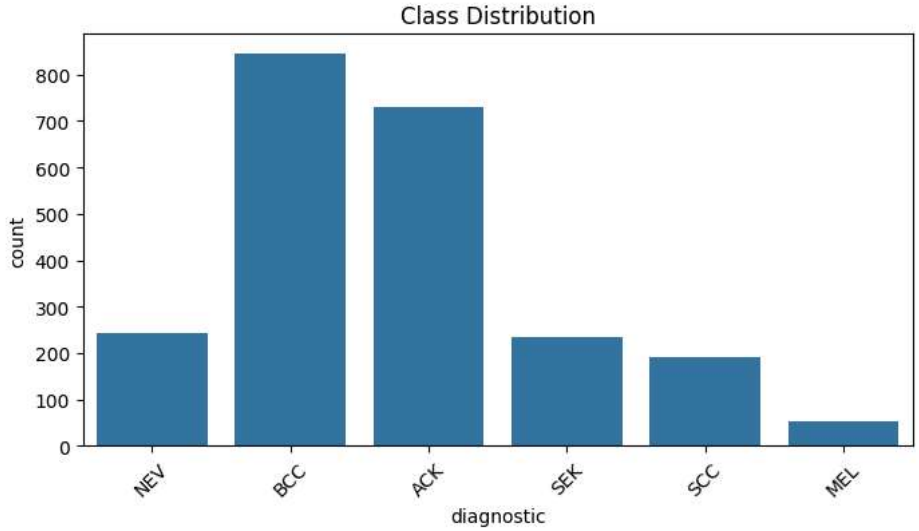


```
# ============================================================
# 📦 PHASE 2 — Corrected Preprocessing & Data Pipeline
# ============================================================

import os
import pandas as pd
import numpy as np
import cv2
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
import seaborn as sns

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

# ============================================================
# STEP 1 — Define Correct Paths
# ============================================================

BASE = "/content/drive/MyDrive/Skin Cancer Images"   # <-- EXACT FOLDER
IMG_DIRS = [
    os.path.join(BASE, "imgs_part_1"),
    os.path.join(BASE, "imgs_part_2"),
    os.path.join(BASE, "imgs_part_3")
]

metadata_path = "/content/drive/MyDrive/Skin Cancer Images/metadata.csv"

df = pd.read_csv(metadata_path)
print("Loaded metadata:", df.shape)

# Remove .png extension from metadata IDs
df["img_core"] = df["img_id"].apply(lambda x: os.path.splitext(x)[0])

# ============================================================
# STEP 2 — Build Dictionary for ALL Images
# ============================================================

all_images = {}

for folder in IMG_DIRS:
    for fname in os.listdir(folder):

        # file may be .jpg, .jpeg, .png
        core_name = os.path.splitext(fname)[0]

        full_path = os.path.join(folder, fname)
        all_images[core_name] = full_path

print("Total images indexed:", len(all_images))

# ============================================================
# STEP 3 — Match Metadata with Actual Image Files
# ============================================================

df["img_path"] = df["img_core"].apply(lambda x: all_images.get(x, None))
matched = df["img_path"].notna().sum()

print("Metadata entries with matched images:", matched)

df = df[df["img_path"].notna()].reset_index(drop=True)
print("Cleaned metadata:", df.shape)

# ============================================================
# STEP 4 — Visualize Class Distribution
# ============================================================
```

```
plt.figure(figsize=(8,4))
sns.countplot(x=df["diagnostic"])
plt.title("Class Distribution")
plt.xticks(rotation=45)
plt.show()

# ============================================================
# STEP 5 — Encode Labels
# ============================================================

label_encoder = LabelEncoder()
df["label_encoded"] = label_encoder.fit_transform(df["diagnostic"])

class_names = label_encoder.classes_
print("\nClass Mapping:")
for i, c in enumerate(class_names):
    print(f"{i} → {c}")

# ============================================================
# STEP 6 — Train / Val / Test Split
# ============================================================

train_df, test_df = train_test_split(
    df, test_size=0.20, stratify=df["label_encoded"], random_state=42)

val_df, test_df = train_test_split(
    test_df, test_size=0.50, stratify=test_df["label_encoded"], random_state=42)

print("\nSplit Sizes:")
print("Train:", train_df.shape)
print("Val:", val_df.shape)
print("Test:", test_df.shape)

# ============================================================
# STEP 7 — Image Loader
# ============================================================

IMG_SIZE = 224

def load_image(path):
    img = cv2.imread(path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
    return img / 255.0

# ============================================================
# STEP 8 — Build Datasets
# ============================================================

def build_dataset(df_subset):
    X, y = [], []
    for _, row in tqdm(df_subset.iterrows(), total=len(df_subset)):
        X.append(load_image(row["img_path"]))
        y.append(row["label_encoded"])
    return np.array(X), np.array(y)

print("\nLoading Train Set...")
X_train, y_train = build_dataset(train_df)

print("\nLoading Validation Set...")
X_val, y_val = build_dataset(val_df)

print("\nLoading Test Set...")
X_test, y_test = build_dataset(test_df)

print("\nDataset Shapes:")
print(X_train.shape, y_train.shape)
print(X_val.shape, y_val.shape)
print(X_test.shape, y_test.shape)

# ============================================================
# STEP 9 — Save Preprocessed Data
# ============================================================

SAVE_DIR = "/content/drive/MyDrive/AML2_Project_Preprocessed"
os.makedirs(SAVE_DIR, exist_ok=True)
```

```
np.save(os.path.join(SAVE_DIR, "X_train.npy"), X_train)
np.save(os.path.join(SAVE_DIR, "y_train.npy"), y_train)
np.save(os.path.join(SAVE_DIR, "X_val.npy"), X_val)
np.save(os.path.join(SAVE_DIR, "y_val.npy"), y_val)
np.save(os.path.join(SAVE_DIR, "X_test.npy"), X_test)
np.save(os.path.join(SAVE_DIR, "y_test.npy"), y_test)

print("\n✅ Preprocessed data saved successfully!")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True
Loaded metadata: (2298, 26)
Total images indexed: 2298
Metadata entries with matched images: 2298
Cleaned metadata: (2298, 28)
```




```
Class Mapping:
0 → ACK
1 → BCC
2 → MEL
3 → NEV
4 → SCC
5 → SEK

Split Sizes:
Train: (1838, 29)
Val: (230, 29)
Test: (230, 29)

Loading Train Set...
100%|██████████| 1838/1838 [19:21<00:00,  1.58it/s]

Loading Validation Set...
100%|██████████| 230/230 [03:08<00:00,  1.22it/s]

Loading Test Set...
100%|██████████| 230/230 [02:48<00:00,  1.36it/s]

Dataset Shapes:
(1838, 224, 224, 3) (1838,)
(230, 224, 224, 3) (230,)
(230, 224, 224, 3) (230,)

✅ Preprocessed data saved successfully!
```

```
Start coding or generate with AI.
```

```
# ============================================================
# 📦 PHASE 3: Model Training, Evaluation & Grad-CAM (EfficientNetB0)
# ============================================================

import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report, confusion_matrix

import tensorflow as tf
from tensorflow.keras import layers, models

# ------------------------------------------------------------------
# 1. Mount Drive and set paths
# ------------------------------------------------------------------
from google.colab import drive
drive.mount('/content/drive')

BASE_DIR      = "/content/drive/MyDrive"
PREP_DIR      = os.path.join(BASE_DIR, "AML2_Project_Preprocessed")   # Phase 2 output
RESULTS_DIR   = os.path.join(BASE_DIR, "AML2_Project_Results")
MODELS_DIR    = os.path.join(BASE_DIR, "AML2_Project_Models")
META_PATH     = os.path.join(BASE_DIR, "Skin Cancer Images", "metadata.csv")

os.makedirs(RESULTS_DIR, exist_ok=True)
os.makedirs(MODELS_DIR,  exist_ok=True)

# ------------------------------------------------------------------
# 2. Load preprocessed NumPy arrays
# ------------------------------------------------------------------
X_train = np.load(os.path.join(PREP_DIR, "X_train.npy"))
y_train = np.load(os.path.join(PREP_DIR, "y_train.npy"))
X_val   = np.load(os.path.join(PREP_DIR, "X_val.npy"))
y_val   = np.load(os.path.join(PREP_DIR, "y_val.npy"))
X_test  = np.load(os.path.join(PREP_DIR, "X_test.npy"))
y_test  = np.load(os.path.join(PREP_DIR, "y_test.npy"))

# Ensure correct dtypes
X_train = X_train.astype("float32")
X_val   = X_val.astype("float32")
X_test  = X_test.astype("float32")
y_train = y_train.astype("int32")
y_val   = y_val.astype("int32")
y_test  = y_test.astype("int32")

print("Shapes:")
print("X_train:", X_train.shape, "y_train:", y_train.shape)
print("X_val:  ", X_val.shape,   "y_val:  ", y_val.shape)
print("X_test: ", X_test.shape,  "y_test: ", y_test.shape)

# ------------------------------------------------------------------
# 3. Recover class names (same encoding as Phase 2)
# ------------------------------------------------------------------
meta_df = pd.read_csv(META_PATH)
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
le.fit(meta_df["diagnostic"])
class_names = list(le.classes_)
num_classes = len(class_names)

print("\nClass mapping:")
for i, c in enumerate(class_names):
    print(f"{i} → {c}")
print("\nnum_classes:", num_classes)

# ------------------------------------------------------------------
# 4. Build tf.data pipelines
# ------------------------------------------------------------------
BATCH_SIZE = 32
AUTOTUNE   = tf.data.AUTOTUNE

def make_ds(X, y, shuffle=False):
    ds = tf.data.Dataset.from_tensor_slices((X, y))
    if shuffle:
        ds = ds.shuffle(buffer_size=len(X), seed=42)
    ds = ds.batch(BATCH_SIZE).prefetch(AUTOTUNE)
    return ds

train_ds = make_ds(X_train, y_train, shuffle=True)
val_ds   = make_ds(X_val,   y_val,   shuffle=False)
test_ds  = make_ds(X_test,  y_test,  shuffle=False)

# ------------------------------------------------------------------
```

```
# 5. Data augmentation
# ------------------------------------------------------------------
data_augmentation = tf.keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.1),
    ],
    name="data_augmentation",
)

# ------------------------------------------------------------------
# 6. Build EfficientNetB0 model (transfer learning)
# ------------------------------------------------------------------
IMG_SIZE = 224

base_model = tf.keras.applications.EfficientNetB0(
    include_top=False,
    weights="imagenet",
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    pooling="avg",
    name="efficientnetb0"  # keep this name for Grad-CAM
)
base_model.trainable = False  # freeze for Phase 3

inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3), name="input_image")
x = data_augmentation(inputs)
# images are already in [0,1]; EfficientNet expects [0,255] + preprocess
x = tf.keras.applications.efficientnet.preprocess_input(x * 255.0)
x = base_model(x, training=False)
x = layers.Dropout(0.3)(x)
outputs = layers.Dense(num_classes, activation="softmax", name="predictions")(x)

model = models.Model(inputs, outputs, name="EfficientNetB0_skin")

model.summary()

# ------------------------------------------------------------------
# 7. Class weights to handle imbalance
# ------------------------------------------------------------------
class_weights_arr = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_train),
    y=y_train
)

class_weights = {i: class_weights_arr[i] for i in range(num_classes)}
print("\nClass weights:")
print(class_weights)

# ------------------------------------------------------------------
# 8. Compile and train (5 epochs) — FIXED VERSION
# ------------------------------------------------------------------

EPOCHS = 5

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]   # ← Precision/Recall REMOVED (they crash on multiclass)
)

history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=EPOCHS,
    class_weight=class_weights,
    verbose=1,
)



# ------------------------------------------------------------------
# 9. Save model
# ------------------------------------------------------------------
model_path = os.path.join(MODELS_DIR, "efficientnetb0_phase3.h5")
model.save(model_path)
print(f"\n✅ Model saved to: {model_path}")
```

```
# ------------------------------------------------------------
# 10. Plot training curves
# ------------------------------------------------------------
def plot_history(hist, save_path_prefix):
    hist_dict = hist.history

    # Accuracy
    plt.figure(figsize=(6,4))
    plt.plot(hist_dict["accuracy"], label="Train Acc")
    plt.plot(hist_dict["val_accuracy"], label="Val Acc")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title("Training vs Validation Accuracy")
    plt.legend()
    plt.tight_layout()
    plt.savefig(save_path_prefix + "_accuracy.png", dpi=300)
    plt.show()

    # Loss
    plt.figure(figsize=(6,4))
    plt.plot(hist_dict["loss"], label="Train Loss")
    plt.plot(hist_dict["val_loss"], label="Val Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training vs Validation Loss")
    plt.legend()
    plt.tight_layout()
    plt.savefig(save_path_prefix + "_loss.png", dpi=300)
    plt.show()

plot_history(history, os.path.join(RESULTS_DIR, "efficientnetb0_phase3"))

# ------------------------------------------------------------
# 11. Evaluation on test set + confusion matrix
# ------------------------------------------------------------
print("\nEvaluating on test set...")
test_loss, test_acc, test_prec, test_rec = model.evaluate(test_ds, verbose=1)
print(f"\nTest Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_acc:.4f}")
print(f"Test Precision: {test_prec:.4f}")
print(f"Test Recall: {test_rec:.4f}")

# Predictions
y_pred_probs = model.predict(test_ds)
y_pred = np.argmax(y_pred_probs, axis=1)

print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=class_names, digits=3))

cm = confusion_matrix(y_test, y_pred, labels=range(num_classes))

plt.figure(figsize=(7,6))
sns.heatmap(
    cm,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=class_names,
    yticklabels=class_names,
)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix – EfficientNetB0 (Phase 3)")
plt.tight_layout()
cm_path = os.path.join(RESULTS_DIR, "confusion_matrix_phase3.png")
plt.savefig(cm_path, dpi=300)
plt.show()
print(f"Confusion matrix saved to: {cm_path}")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Shapes:
X_train: (1838, 224, 224, 3) y_train: (1838,)
X_val:   (230, 224, 224, 3) y_val:   (230,)
X_test:  (230, 224, 224, 3) y_test:  (230,)

Class mapping:
0 → ACK
1 → BCC
2 → MEL
3 → NEV
4 → SCC
5 → SEK

num_classes: 6
Model: "EfficientNetB0_skin"
```

Next steps: Explain error

Layer (type)	Output Shape	Param #
input_image (InputLayer)	(None, 224, 224, 3)	0
data_augmentation (Sequential)	(None, 224, 224, 3)	0
multiply_1 (Multiply)	(None, 224, 224, 3)	0

```
# ------------------------------------------------------------
# Detailed metrics: precision, recall, F1, confusion matrix
# ------------------------------------------------------------
from sklearn.metrics import classification_report, confusion_matrix

# Get predictions for all test samples
y_prob = model.predict(test_ds)
y_pred = np.argmax(y_prob, axis=1)

print("\nClassification report (per class):\n")
print(classification_report(y_test, y_pred, target_names=class_names))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(7,6))
sns.heatmap(
    cm,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=class_names,
    yticklabels=class_names,
)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix – EfficientNetB0 (Phase 3)")
plt.tight_layout()

cm_path = os.path.join(RESULTS_DIR, "confusion_matrix_phase3.png")
plt.savefig(cm_path, dpi=150)
plt.show()
print(f"Confusion matrix saved to: {cm_path}")

# ------------------------------------------------------------
# Preview some test images with predictions
# ------------------------------------------------------------
def show_sample_predictions(X, y_true, y_pred, class_names, n=9):
    idx = np.random.choice(len(X), size=n, replace=False)
    plt.figure(figsize=(12,10))
    for i, j in enumerate(idx):
        plt.subplot(3, 3, i+1)
        plt.imshow(X[j])
        true_label = class_names[y_true[j]]
        pred_label = class_names[y_pred[j]]
        colour = "green" if y_true[j] == y_pred[j] else "red"
        plt.title(f"True: {true_label}\nPred: {pred_label}", color=colour)
        plt.axis("off")
    plt.tight_layout()
    grid_path = os.path.join(RESULTS_DIR, "sample_predictions_phase3.png")
    plt.savefig(grid_path, dpi=150)
    plt.show()
    print(f"Sample predictions grid saved to: {grid_path}")
```

```
show_sample_predictions(X_test, y_test, y_pred, class_names, n=9)

# ------------------------------------------------------------
# Save final model
# ------------------------------------------------------------

MODEL_DIR = "/content/drive/MyDrive/AML2_Project_Models"
os.makedirs(MODEL_DIR, exist_ok=True)


final_model_path = os.path.join(MODEL_DIR, "efficientnetB0_phase3_final.keras")
model.save(final_model_path)
print(f"\n✅ Phase 3 model saved to: {final_model_path}")
```

```
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-3195340479.py in <cell line: 0>()
    200 # ------------------------------------------------------------
    201 print("\nEvaluating on test set...")
--> 202 test_loss, test_acc, test_prec, test_rec = model.evaluate(test_ds, verbose=1)
    203 print(f"\nTest Loss: {test_loss:.4f}")
    204 print(f"Test Accuracy: {test_acc:.4f}")

ValueError: not enough values to unpack (expected 4, got 2)
```

```
# ================================================================
# 📦 PHASE 4: Explainability with LIME + Integrated Gradients (IG)
# ================================================================

!pip install -q lime

import os
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from lime import lime_image
from skimage.segmentation import mark_boundaries

# ----------------------------------------------------------------
# 1. Paths and data / model loading
# ----------------------------------------------------------------
from google.colab import drive
drive.mount('/content/drive')

BASE_DIR    = "/content/drive/MyDrive"
PREP_DIR    = os.path.join(BASE_DIR, "AML2_Project_Preprocessed")
RESULTS_DIR = os.path.join(BASE_DIR, "AML2_Project_Results")
MODEL_DIR   = os.path.join(BASE_DIR, "AML2_Project_Models")

os.makedirs(RESULTS_DIR, exist_ok=True)

# Load preprocessed data
X_train = np.load(os.path.join(PREP_DIR, "X_train.npy")).astype("float32")
y_train = np.load(os.path.join(PREP_DIR, "y_train.npy")).astype("int32")
X_test  = np.load(os.path.join(PREP_DIR, "X_test.npy")).astype("float32")
y_test  = np.load(os.path.join(PREP_DIR, "y_test.npy")).astype("int32")

print("X_train:", X_train.shape, " X_test:", X_test.shape)

# Class names (same order as in Phase 2)
from sklearn.preprocessing import LabelEncoder
```

```
import pandas as pd

meta_path = os.path.join(BASE_DIR, "Skin Cancer Images", "metadata.csv")
meta_df   = pd.read_csv(meta_path)
le = LabelEncoder()
le.fit(meta_df["diagnostic"])
class_names = list(le.classes_)
num_classes = len(class_names)

print("\nClass mapping:")
for i, c in enumerate(class_names):
    print(f"{i} → {c}")

# Load trained model (Phase 3)
model_path = os.path.join(MODEL_DIR, "efficientnetB0_phase3_final.keras")
model = tf.keras.models.load_model(model_path, compile=False)
print("\n✅ Phase 3 model loaded from:", model_path)


# Small helper: prediction function used by LIME
def lime_predict(images):
    """
    LIME will pass a list/array of images in [0,1] range.
    Our model was trained on [0,1] images (internal EfficientNet preprocessing),
    so we simply cast to float32 and call model.predict().
    """
    images = np.array(images).astype("float32")
    preds = model.predict(images, verbose=0)
    return preds


# ============================================================
# 2. LIME – Local explanations for an individual image
# ============================================================

explainer = lime_image.LimeImageExplainer()

# Pick a random test image (we'll reuse the same idx for IG)
idx = np.random.randint(0, len(X_test))
image = X_test[idx]
true_label_idx = y_test[idx]
true_label = class_names[true_label_idx]

print(f"\n🔍 LIME explanation for test index {idx}, true label = {true_label}")

# Run LIME
lime_exp = explainer.explain_instance(
    image=image,
    classifier_fn=lime_predict,
    top_labels=1,
    hide_color=0,
    num_samples=1000   # more samples → smoother explanation, but slower
)

# Get the top predicted label from LIME
top_label = lime_exp.top_labels[0]

# Positive evidence only (regions pushing towards the predicted class)
temp, mask = lime_exp.get_image_and_mask(
    label=top_label,
    positive_only=True,
    num_features=8,      # number of superpixels to show
    hide_rest=False
)

plt.figure(figsize=(10, 4))

# Original image
plt.subplot(1, 2, 1)
plt.imshow(image)
plt.axis("off")
plt.title(f"Original\nTrue: {true_label}")

# LIME explanation
plt.subplot(1, 2, 2)
plt.imshow(mark_boundaries(temp / max(temp.max(), 1e-8), mask))
pred_label = class_names[top_label]
```

```
plt.axis("off")
plt.title(f"LIME – Evidence for: {pred_label}")

plt.tight_layout()
lime_path = os.path.join(RESULTS_DIR, f"lime_explanation_idx_{idx}.png")
plt.savefig(lime_path, dpi=200)
plt.show()
print(f"💾 LIME explanation saved to: {lime_path}")


# ============================================================
# 3. Integrated Gradients (IG) – Pixel-level attributions
# ============================================================

IMG_H, IMG_W, IMG_C = image.shape

def integrated_gradients(
    input_image,
    target_class_index,
    baseline=None,
    m_steps=50,
):
    """
    Compute Integrated Gradients for a single image and target class.

    Args:
        input_image: (H, W, C) image in [0,1].
        target_class_index: int, index of class for which IG is computed.
        baseline: baseline image (H, W, C). If None, uses black image.
        m_steps: number of interpolation steps between baseline and image.

    Returns:
        ig_attributions: (H, W, C) attributions.
    """
    if baseline is None:
        baseline = np.zeros_like(input_image).astype("float32")

    # Ensure float32
    input_image = input_image.astype("float32")
    baseline = baseline.astype("float32")

    # Generate scaled inputs
    interpolated_images = [
        baseline + (float(k) / m_steps) * (input_image - baseline)
        for k in range(1, m_steps + 1)
    ]
    interpolated_images = np.stack(interpolated_images, axis=0)  # (m_steps, H, W, C)

    # Compute gradients for each interpolated image
    interpolated_tensor = tf.convert_to_tensor(interpolated_images)

    with tf.GradientTape() as tape:
        tape.watch(interpolated_tensor)
        preds = model(interpolated_tensor)                    # (m_steps, num_classes)
        probs = preds[:, target_class_index]                  # (m_steps,)

    grads = tape.gradient(probs, interpolated_tensor).numpy() # (m_steps, H, W, C)

    # Average gradients across steps
    avg_grads = grads.mean(axis=0)                            # (H, W, C)

    # Integrated gradients: (input - baseline) * average_gradient
    ig = (input_image - baseline) * avg_grads                 # (H, W, C)

    return ig


# ---- Run IG for the same image used in LIME ----

# Get predicted class for this image
pred_probs = model.predict(image[None, ...], verbose=0)[0]
pred_class_idx = int(np.argmax(pred_probs))
pred_class_name = class_names[pred_class_idx]

print(f"\n🧮 Integrated Gradients for test index {idx}")
print(f"True label: {true_label} | Predicted: {pred_class_name} (p={pred_probs[pred_class_idx]:.3f})")

# Compute IG
```

```
ig_attributions = integrated_gradients(
    input_image=image,
    target_class_index=pred_class_idx,
    baseline=None,   # black baseline
    m_steps=50
)  # (H, W, C)

# Convert to a single-channel heatmap by aggregating over colour channels
ig_abs = np.abs(ig_attributions).mean(axis=-1)  # (H, W)

# Normalise to [0,1] for display
ig_min, ig_max = ig_abs.min(), ig_abs.max()
heatmap = (ig_abs - ig_min) / (ig_max - ig_min + 1e-8)

# ------------------------------------------------
# Plot IG heatmap and overlay
# ------------------------------------------------
plt.figure(figsize=(12, 4))

# Heatmap alone
plt.subplot(1, 3, 1)
plt.imshow(heatmap, cmap="inferno")
plt.axis("off")
plt.title("Integrated Gradients\nHeatmap")

# Original image
plt.subplot(1, 3, 2)
plt.imshow(image)
plt.axis("off")
plt.title("Original Image")

# Overlay
plt.subplot(1, 3, 3)
plt.imshow(image)
plt.imshow(heatmap, cmap="inferno", alpha=0.5)
plt.axis("off")
plt.title(f"IG Overlay\nPred: {pred_class_name}")

plt.tight_layout()
ig_path = os.path.join(RESULTS_DIR, f"ig_explanation_idx_{idx}.png")
plt.savefig(ig_path, dpi=200)
plt.show()

print(f"💾 Integrated Gradients explanation saved to: {ig_path}")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
X_train: (1838, 224, 224, 3)  X_test: (230, 224, 224, 3)

Class mapping:
0 → ACK
1 → BCC
2 → MEL
3 → NEV
4 → SCC
```

```
# ================================================================
# 📌 PHASE 5 — Gradio Interface for Skin Lesion Classification
# ================================================================

!pip install -q gradio lime

import gradio as gr
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import cv2
import os
from lime import lime_image
from skimage.segmentation import mark_boundaries
from google.colab import drive

# ------------------------------------------------
# 1️⃣ MOUNT DRIVE & LOAD MODEL + CLASS LABELS
# ------------------------------------------------
drive.mount('/content/drive')

BASE_DIR = "/content/drive/MyDrive"
MODEL_PATH = os.path.join(BASE_DIR, "AML2_Project_Models", "efficientnetB0_phase3_final.keras")

# Load EfficientNetB0 Phase3 model
model = tf.keras.models.load_model(MODEL_PATH, compile=False)
IMG_SIZE = 224

print("✅ Loaded model:", MODEL_PATH)

# Load class names from metadata
import pandas as pd
from sklearn.preprocessing import LabelEncoder

meta_path = os.path.join(BASE_DIR, "Skin Cancer Images", "metadata.csv")
meta_df = pd.read_csv(meta_path)

le = LabelEncoder()
le.fit(meta_df["diagnostic"])
class_names = list(le.classes_)

print("Class Names:", class_names)


# ------------------------------------------------
# 2️⃣ IMAGE PREPROCESSING FUNCTION
# ------------------------------------------------
def preprocess_image(image):
    """Resize and normalize image to match EfficientNet input."""
    img = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (IMG_SIZE, IMG_SIZE))
    img = img.astype("float32") / 255.0
    return img


# ------------------------------------------------
# 3️⃣ LIME EXPLANATION FUNCTION
# ------------------------------------------------
lime_explainer = lime_image.LimeImageExplainer()

def generate_lime(image_np):
    explanation = lime_explainer.explain_instance(
        image_np,
        classifier_fn=lambda imgs: model.predict(np.array(imgs)),
        top_labels=1,
        hide_color=0,
        num_samples=800
    )
```

```
    top_label = explanation.top_labels[0]
    temp, mask = explanation.get_image_and_mask(
        label=top_label, positive_only=True, num_features=8, hide_rest=False
    )

    lime_image_out = mark_boundaries(temp / temp.max(), mask)
    return lime_image_out


# ------------------------------------------------
# 4️⃣ INTEGRATED GRADIENTS FUNCTION
# ------------------------------------------------
def integrated_gradients(model, img, baseline=None, steps=50):

    if baseline is None:
        baseline = np.zeros_like(img)

    img = tf.cast(img, tf.float32)
    baseline = tf.cast(baseline, tf.float32)

    img = tf.expand_dims(img, axis=0)
    baseline = tf.expand_dims(baseline, axis=0)

    interpolated_imgs = [
        baseline + (float(i) / steps) * (img - baseline) for i in range(steps + 1)
    ]
    interpolated_imgs = tf.concat(interpolated_imgs, axis=0)

    with tf.GradientTape() as tape:
        tape.watch(interpolated_imgs)
        preds = model(interpolated_imgs)

    grads = tape.gradient(preds, interpolated_imgs)
    avg_grads = tf.reduce_mean(grads, axis=0)

    integrated_grads = (img - baseline) * avg_grads
    return tf.squeeze(integrated_grads).numpy()


def compute_ig_heatmap(image_np):
    ig = integrated_gradients(model, image_np)
    ig = np.abs(ig).mean(axis=-1)

    ig = (ig - ig.min()) / (ig.max() - ig.min() + 1e-8)
    heatmap = cv2.applyColorMap((ig * 255).astype(np.uint8), cv2.COLORMAP_JET)
    heatmap = cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB)
    return heatmap


# ------------------------------------------------
# 5️⃣ MAIN PREDICTION + EXPLANATION FUNCTION
# ------------------------------------------------
def predict(image):

    # Preprocess
    img_np = preprocess_image(image)

    # Predict class
    preds = model.predict(np.expand_dims(img_np, axis=0))[0]
    top_idx = np.argmax(preds)
    predicted_class = class_names[top_idx]
    confidence = preds[top_idx]

    # LIME
    lime_img = generate_lime(img_np)

    # IG heatmap
    ig_img = compute_ig_heatmap(img_np)

    return (
        f"Prediction: {predicted_class}\nConfidence: {confidence:.2f}",
        lime_img,
        ig_img
    )
```

```
# ------------------------------------------------
# 6️⃣ BUILD GRADIO UI
# ------------------------------------------------
interface = gr.Interface(
    fn=predict,
    inputs=gr.Image(type="numpy", label="Upload Skin Lesion Image"),
    outputs=[
        gr.Textbox(label="Model Prediction"),
        gr.Image(label="LIME Explanation"),
        gr.Image(label="Integrated Gradients Heatmap")
    ],
    title="🩺 AI-Powered Skin Lesion Classification",
    description="""
This tool classifies skin lesion images using **EfficientNetB0** and explains predictions using:

- **LIME (Local Interpretable Model-Agnostic Explanations)**
- **Integrated Gradients (Attribution heatmap)**

Upload a dermatoscopic image to begin.
""",
    examples=None
)

interface.launch(debug=True)
```