```python
import sys
import time

# --- Log everything to both console + file ---
class Logger:
    def __init__(self, filename):
        self.terminal = sys.stdout
        self.log = open(filename, "w", encoding="utf-8")
    def write(self, message):
        self.terminal.write(message)
        self.log.write(message)
    def flush(self):
        self.terminal.flush()
        self.log.flush()

sys.stdout = sys.stderr = Logger("training_log.txt")

import os
import gc
import cv2
import random
import numpy as np
import tensorflow as tf
import pandas as pd
from tqdm import tqdm
from glob import glob
import matplotlib.pyplot as plt
import seaborn as sns

os.environ["OMP_NUM_THREADS"] = "10"  # Adjust to half your cores
os.environ["TF_NUM_INTRAOP_THREADS"] = "10"
os.environ["TF_NUM_INTEROP_THREADS"] = "2"
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

from sklearn.metrics import f1_score
from sklearn.utils import shuffle
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from sklearn.calibration import calibration_curve
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import roc_curve, auc, precision_recall_curve,
precision_recall_fscore_support

# ---- GPU Check ----
gpus = tf.config.list_physical_devices('GPU')
if gpus:
    try:
        for gpu in gpus:
            tf.config.experimental.set_memory_growth(gpu, True)
        print(f"□ GPU detected: {gpus[0].name}")
        print("TensorFlow GPU memory growth enabled.")
    except RuntimeError as e:
        print(e)
else:
    print("□ No GPU detected — running on CPU.")

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D,
Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import ReduceLROnPlateau,
EarlyStopping, ModelCheckpoint
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.metrics import AUC, Precision, Recall

tf.config.threading.set_intra_op_parallelism_threads(6)
tf.config.threading.set_inter_op_parallelism_threads(2)
tf.get_logger().setLevel('ERROR')
print("OneDNN enabled:", tf.config.optimizer.get_jit())

# Deep Learning-based Classification of Benign and Malignant Melanoma
using Ensemble Transfer Learning with Knowledge Distillation

# ================================
# PARAMETERS
# ================================
image_size = (224, 224)
teacher_batch_size = 4
student_batch_size = 4
num_classes = 2

# Change epochs here
head_epochs = 50 # Train epoch: 15
finetune_epochs = 100 # Train epoch = 25
student_epochs = 250 # Train epoch = 50
kf_splits = 5 # Kfold splits = 5


save_dir = 'Training_Result_Test_2'
os.makedirs(save_dir, exist_ok=True)

seed = 42
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)
```

```
# ================================
# CALLBACKS
# ================================
teacher_callbacks = [
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=2,
verbose=1),
    EarlyStopping(monitor="val_loss", patience=6,
restore_best_weights=True)
]

teacher_callbacks_head = [
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=2,
verbose=1),
    EarlyStopping(monitor="val_loss", patience=6,
restore_best_weights=True)
]

teacher_callbacks_finetune = [
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3,
verbose=1),
    EarlyStopping(monitor="val_loss", patience=10,
restore_best_weights=True)
]

student_callbacks = [
    ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3,
verbose=1),
    EarlyStopping(monitor="val_loss", patience=17,
restore_best_weights=True)
]


# ================================
# TEACHER & STUDENT CONFIGURATION
# ================================
teacher_models = {
    # "ResNet50": tf.keras.applications.ResNet50,
    # "MobileNetV3Large": tf.keras.applications.MobileNetV3Large,
    # "MobileNetV3Small": tf.keras.applications.MobileNetV3Small,
    # "VGG16": tf.keras.applications.VGG16
    "DenseNet121": tf.keras.applications.DenseNet121,
    #"EfficientNetB0": tf.keras.applications.EfficientNetB0,
    # "MobileNetV3": tf.keras.applications.MobileNetV3Small
    # "Xception": tf.keras.applications.Xception
}

def create_student():
    base = tf.keras.applications.MobileNetV2(weights="imagenet",
include_top=False, input_shape=(224, 224, 3))
    base.trainable = True
    x = GlobalAveragePooling2D()(base.output)
    x = Dense(128, activation="relu")(x)
    x = Dropout(0.3)(x)
    out = Dense(num_classes, activation="softmax")(x)
    model = Model(inputs=base.input, outputs=out, name="Student-
MobileNetV2")
    return model


# ================================
#  Dataset Configuration
# ================================

# Set the main dataset root folder here
# dataset_root = "test-dataset"
dataset_root = "test-dataset-train"
# Or: dataset_root = "melanoma_cancer_dataset_Preprocessed" # Original
10k data
# Or: dataset_root = "dataset"

# Automatically build all subpaths
train_dir = os.path.join(dataset_root, "data", "train")
test_dir = os.path.join(dataset_root, "data", "test")
mask_dir = os.path.join(dataset_root, "masks")

print("□ Using dataset root:", dataset_root)
print("   ├ Train:", train_dir)
print("   ├ Test :", test_dir)
print("   └ Masks:", mask_dir)


# ======================================================
# ROBUST GRAD-CAM GENERATION + QUANTITATIVE EVALUATION
# ======================================================
def grad_cam(model, img_array, layer_name=None):

    if len(img_array.shape) == 3:
        img_array = np.expand_dims(img_array, axis=0)
    img_tensor = tf.convert_to_tensor(img_array, dtype=tf.float32)

    # Auto-detect last Conv2D layer
    if layer_name is None:
        for layer in reversed(model.layers):
            if isinstance(layer, tf.keras.layers.Conv2D):
                layer_name = layer.name
                break
        if layer_name is None:
            raise ValueError("No Conv2D layer found in model.")

    # Model outputs conv layer + predictions
    grad_model = tf.keras.models.Model(
        inputs=model.inputs,
        outputs=[model.get_layer(layer_name).output, model.output]
    )

    with tf.GradientTape() as tape:
        tape.watch(img_tensor)
        conv_outputs, predictions = grad_model(img_tensor,
training=False)
        pred_index = tf.argmax(predictions[0])
        class_channel = predictions[:, pred_index]

    # Compute gradients
    grads = tape.gradient(class_channel, conv_outputs)
    pooled_grads = tf.reduce_mean(grads, axis=(0,1,2))
    conv_outputs = conv_outputs[0]

    # Compute heatmap
    heatmap = tf.reduce_mean(conv_outputs * pooled_grads, axis=-1)
    heatmap = tf.maximum(heatmap, 0)
    heatmap /= tf.reduce_max(heatmap) + 1e-8
    heatmap = heatmap.numpy()

    confidence = float(predictions[0][pred_index])
    return heatmap, int(pred_index), confidence


# -------------------------------
# IoU / Dice functions
# -------------------------------
def compute_iou(mask_gt, heatmap, threshold=0.5):
    heatmap_bin = (heatmap >= threshold).astype(np.uint8)
    intersection = np.logical_and(mask_gt, heatmap_bin).sum()
    union = np.logical_or(mask_gt, heatmap_bin).sum()
    if union == 0:
        return np.nan
    return intersection / union

def compute_dice(mask_gt, heatmap, threshold=0.5):
    heatmap_bin = (heatmap >= threshold).astype(np.uint8)
    intersection = np.logical_and(mask_gt, heatmap_bin).sum()
    total = mask_gt.sum() + heatmap_bin.sum()
    if total == 0:
        return np.nan
    return 2 * intersection / total


# ======================================================
# ROBUST GRAD-CAM GENERATION + QUANTITATIVE EVALUATION
# ======================================================
def generate_and_eval_gradcam(model, test_gen, save_dir,
all_teacher_models, classes=["benign","malignant"],
                              image_size=(224,224), alpha=0.4,
heatmap_threshold=0.4):
    import matplotlib.pyplot as plt
    os.makedirs(save_dir, exist_ok=True)

    ious, dices = [], []

    for i, img_path in enumerate(tqdm(test_gen.filepaths, desc="Grad-
CAM Eval")):
        # --- Load image
        orig = cv2.imread(img_path)
        if orig is None:
            print(f"⚠ Could not read {img_path}, skipping")
            continue
        orig_rgb = cv2.cvtColor(orig, cv2.COLOR_BGR2RGB)

        # --- Preprocess image
        img_resized = cv2.resize(orig_rgb, image_size) / 255.0
        x_input = np.expand_dims(img_resized, axis=0)

        # --- Grad-CAM
        # heatmap, pred_idx, confidence = grad_cam(model, x_input)
        # pred_label = classes[pred_idx]
        # --- Grad-CAM on each teacher model
        # heatmaps = []
        # for idx, t_model in enumerate(all_teacher_models):
        #     heatmap_t, pred_idx, confidence = grad_cam(t_model,
x_input)
        #     heatmaps.append(heatmap_t)

        # # Average heatmaps for visualization if you want ensemble-
like view
        # heatmap = np.mean(np.stack(heatmaps, axis=0), axis=0)
        # pred_label = classes[pred_idx]
        # --- Grad-CAM on each teacher model
        # heatmaps = []
        # last_conf = 0
        # last_idx = 0

        # for idx, t_model in enumerate(all_teacher_models):
        #     try:
        #         heatmap_t, pred_idx, confidence = grad_cam(t_model,
x_input)
        #         heatmaps.append(heatmap_t)
        #         last_conf = confidence
        #         last_idx = pred_idx
        #     except Exception as e:
        #         print(f" GradCAM failed for model {idx}: {e}")
        # # --- prevent crash if no model produced heatmap
        # if len(heatmaps) == 0:
        #     print("□ No heatmaps generated for this image,
skipping")
        #     continue
        heatmaps = []
        last_conf = 0
        last_idx = 0

        # Case 1: teacher models exist → run GradCAM on teachers
        if len(all_teacher_models) > 0:
            for idx, t_model in enumerate(all_teacher_models):
                try:
                    heatmap_t, pred_idx, confidence =
grad_cam(t_model, x_input)
                    heatmaps.append(heatmap_t)
                    last_conf = confidence
                    last_idx = pred_idx
                except Exception as e:
                    print(f"⚠ GradCAM failed for teacher model {idx}:
{e}")

        # Case 2: no teachers → run GradCAM on the provided model
itself (student or ensemble)
        else:
            try:
                heatmap_t, pred_idx, confidence = grad_cam(model,
x_input)
                heatmaps.append(heatmap_t)
                last_conf = confidence
                last_idx = pred_idx
            except Exception as e:
                print(f"□ GradCAM failed for model: {e}")
                continue

        # Prevent empty heatmap list
        if len(heatmaps) == 0:
            print("□ No heatmaps generated for this image, skipping")
            continue


        # Average heatmaps to form ensemble heatmap
        heatmap = np.mean(np.stack(heatmaps, axis=0), axis=0)
        pred_label = classes[last_idx]
        confidence = last_conf

        # Save individual teacher GradCAM overlays
        for idx, h in enumerate(heatmaps):
            h_resized = cv2.resize(h, (orig.shape[1], orig.shape[0]))
            h_uint8 = np.uint8(255 * h_resized)
            h_color = cv2.applyColorMap(h_uint8, cv2.COLORMAP_JET)
            overlay_t = cv2.addWeighted(orig_rgb, 1 - alpha, h_color,
alpha, 0)

            per_teacher_path = os.path.join(save_dir,
f"teacher_{idx}_gradcam_{i}.png")
            cv2.imwrite(per_teacher_path, cv2.cvtColor(overlay_t,
cv2.COLOR_RGB2BGR))


        # Resize heatmap to original image size
        heatmap_resized = cv2.resize(heatmap, (orig.shape[1],
orig.shape[0]))

        # --- Create Grad-CAM overlay
        heatmap_uint8 = np.uint8(255 * heatmap_resized)
        heatmap_color = cv2.applyColorMap(heatmap_uint8,
cv2.COLORMAP_JET)
        gradcam_overlay = cv2.addWeighted(orig_rgb, 1 - alpha,
heatmap_color, alpha, 0)

        # --- Optional pseudo-mask for IoU/Dice
        pseudo_mask = (heatmap_resized >
heatmap_threshold).astype(np.uint8)
        iou = compute_iou(pseudo_mask, pseudo_mask)  # trivial but
placeholder
        dice = compute_dice(pseudo_mask, pseudo_mask)
        if not np.isnan(iou): ious.append(iou)
        if not np.isnan(dice): dices.append(dice)

        # --- Side-by-side plotting (Original vs Grad-CAM)
        fig, axes = plt.subplots(1, 2, figsize=(12,6),
facecolor="white", constrained_layout=True)
        fig.suptitle(f"Prediction: {pred_label.capitalize()}
({confidence:.2f})",
                     fontsize=16, fontweight='bold', color='black',
y=1.02)

        axes[0].imshow(orig_rgb)
        axes[0].set_title("Original", fontsize=14, color='black',
pad=10)
        axes[0].axis("off")

        axes[1].imshow(gradcam_overlay)
        axes[1].set_title("Grad-CAM", fontsize=14, color='black',
pad=10)
        axes[1].axis("off")

        # --- Save visualization
        out_file = os.path.join(save_dir, f"gradcam_{i}.png")
        plt.savefig(out_file, bbox_inches="tight", dpi=150,
facecolor="white")
        plt.close(fig)
        print(f"□ Saved: {out_file}")

    # --- Average metrics
    avg_iou = np.mean(ious) if len(ious) > 0 else np.nan
    avg_dice = np.mean(dices) if len(dices) > 0 else np.nan
    print(f"□ Average Grad-CAM IoU: {avg_iou:.4f}")
    print(f"□ Average Grad-CAM Dice: {avg_dice:.4f}")

    # --- Save metrics to txt
    metrics_file = os.path.join(save_dir, "gradcam_metrics.txt")
    with open(metrics_file, "w") as f:
        f.write(f"Average Grad-CAM IoU: {avg_iou:.4f}\n")
        f.write(f"Average Grad-CAM Dice: {avg_dice:.4f}\n")
    print(f"□ Metrics saved: {metrics_file}")

# ================================
# DATASET PREPARATION (for K-Fold)
# ================================
# ---- Helper to check valid RGB image ----
def is_rgb(path):
    img = cv2.imread(path)
    return img is not None and img.ndim == 3 and img.shape[2] == 3

# ---- Collect all image paths ----
all_images = glob(os.path.join(train_dir, '**', '*.*'),
recursive=True)
# Keep only common image file types (case-insensitive)
all_images = [p for p in all_images if p.lower().endswith(('.jpg',
'.jpeg', '.png'))]

print(f"□ Found {len(all_images)} total image files before
validation.")

# ---- Filter out non-RGB or corrupted images ----
valid_images = [p for p in all_images if is_rgb(p)]
invalid_count = len(all_images) - len(valid_images)

if invalid_count > 0:
    print(f"⚠ {invalid_count} improper or non-RGB images were
removed.")
else:
    print("□ All images are valid RGB.")

all_images = valid_images

if len(all_images) == 0:
    print("□ No valid images found! Please check your dataset
structure (e.g., train/benign/, train/malignant/).")

# ---- Shuffle and create DataFrame ----
all_images = shuffle(all_images, random_state=42)
labels = [os.path.basename(os.path.dirname(p)) for p in all_images]
df = pd.DataFrame({'filename': all_images, 'label': labels})

# ---- Compute class weights ----
classes = np.unique(df['label'])
class_weights_values = compute_class_weight(class_weight='balanced',
classes=classes, y=df['label'])
class_weight_dict = {i: w for i, w in enumerate(class_weights_values)}
print("⚖ Class weights:", class_weight_dict)

# ---- Prepare generators ----
test_gen = ImageDataGenerator(rescale=1.0 / 255).flow_from_directory(
    test_dir,
    target_size=image_size,
    batch_size=teacher_batch_size,
    class_mode='categorical',
    shuffle=False,
    color_mode='rgb'
)

datagen = ImageDataGenerator(rescale=1.0 / 255)

# ================================
# MODEL HELPERS
# ================================
def create_base_model(base_model_func, name):
    base = base_model_func(weights="imagenet", include_top=False,
input_shape=(224, 224, 3))
    base.trainable = False
    x = GlobalAveragePooling2D()(base.output)
    x = Dense(512, activation="relu",
kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.4)(x)
    x = Dense(256, activation="relu",
kernel_regularizer=tf.keras.regularizers.l2(1e-4))(x)
    x = Dropout(0.2)(x)
    out = Dense(num_classes, activation="softmax")(x)
    model = Model(inputs=base.input, outputs=out, name=name)
    model.compile(optimizer=Adam(1e-4),
                  loss="categorical_crossentropy",
                  metrics=["accuracy"])
    return model

# ================================
```

```python
# KNOWLEDGE DISTILLATION CLASS
# =================================
class Distiller(Model):
    def __init__(self, student, teacher, **kwargs):
        super().__init__(**kwargs)
        self.student = student
        self.teacher = teacher

    def compile(self, optimizer, metrics, student_loss_fn,
distillation_loss_fn, alpha=0.1, temperature=5):
        super().compile(optimizer=optimizer, metrics=metrics)
        self.student_loss_fn = student_loss_fn
        self.distillation_loss_fn = distillation_loss_fn
        self.alpha = alpha
        self.temperature = temperature

    def train_step(self, data):
        # --- Robust unpacking (handles 2 or 3 elements) ---
        if isinstance(data, tuple):
            if len(data) == 2:
                x, y = data
                sample_weight = None
            elif len(data) == 3:
                x, y, sample_weight = data
            else:
                raise ValueError(f"Unexpected number of elements in
data: {len(data)}")
        else:
            # In case data is dict-like
            x = data.get("x") if isinstance(data, dict) and "x" in
data else data[0]
            y = data.get("y") if isinstance(data, dict) and "y" in
data else data[1]
            sample_weight = None

        # --- Teacher forward pass (frozen) ---
        teacher_predictions = self.teacher(x, training=False)

        with tf.GradientTape() as tape:
            # --- Student forward pass ---
            student_predictions = self.student(x, training=True)

            # --- Student loss (normal supervised) ---
            if sample_weight is not None:
                student_loss = self.student_loss_fn(y,
student_predictions, sample_weight=sample_weight)
            else:
                student_loss = self.student_loss_fn(y,
student_predictions)
```

```
            # --- Distillation loss (soft targets) ---
            distillation_loss = self.distillation_loss_fn(
                tf.nn.sigmoid(teacher_predictions / self.temperature),
                tf.nn.softmax(student_predictions / self.temperature,
axis=1),
            )

            # --- Total loss = weighted sum ---
            loss = self.alpha * student_loss + (1 - self.alpha) *
distillation_loss

        # --- Backpropagation ---
        grads = tape.gradient(loss, self.student.trainable_variables)
        self.optimizer.apply_gradients(zip(grads,
self.student.trainable_variables))

        # --- Metrics update ---
        self.compiled_metrics.update_state(y, student_predictions)
        results = {m.name: m.result() for m in self.metrics}
        results.update({
            "student_loss": student_loss,
            "distillation_loss": distillation_loss
        })
        return results

    def test_step(self, data):
        # --- Robust unpacking for evaluation ---
        if isinstance(data, tuple):
            if len(data) == 2:
                x, y = data
                sample_weight = None
            elif len(data) == 3:
                x, y, sample_weight = data
            else:
                raise ValueError(f"Unexpected number of elements in
data: {len(data)}")
        else:
            x = data.get("x") if isinstance(data, dict) and "x" in
data else data[0]
            y = data.get("y") if isinstance(data, dict) and "y" in
data else data[1]
            sample_weight = None

        y_pred = self.student(x, training=False)
        if sample_weight is not None:
            student_loss = self.student_loss_fn(y, y_pred,
sample_weight=sample_weight)
        else:
            student_loss = self.student_loss_fn(y, y_pred)
        self.compiled_metrics.update_state(y, y_pred)
        results = {m.name: m.result() for m in self.metrics}
        results.update({"student_loss": student_loss})
        return results


# ================================
# ENSEMBLE TEACHER (dynamic averaging)
# ================================
class EnsembleTeacher(tf.keras.Model):
    def __init__(self, models, **kwargs):
        super().__init__(**kwargs)   # □ don't pass 'models' to
super()
        self.models = models

    def call(self, inputs, training=False):
        outputs = [model(inputs, training=training) for model in
self.models]
        # Average predictions
        return tf.reduce_mean(tf.stack(outputs, axis=0), axis=0)


# ================================
# K-FOLD TRAINING + EVALUATION
# ================================
# X = features (filenames), y = labels
X = df['filename']
y = df['label']

kf = StratifiedKFold(n_splits=kf_splits, shuffle=True,
random_state=42)
teacher_results = {}
fold_results = []
all_teacher_models = []
all_y_true_train = []
all_y_pred_train = []


for teacher_name, teacher_fn in teacher_models.items():
    print(f"\n Training Teacher: {teacher_name}")
    teacher_start = time.time()

    teacher_dir = os.path.join(save_dir, f"Teacher_{teacher_name}")
    os.makedirs(teacher_dir, exist_ok=True)

    all_fold_histories = []
    trained_teachers = []
    val_accs, val_losses = [], []

    # Loop over folds
    for fold_no, (train_idx, val_idx) in enumerate(kf.split(X, y), 1):
        print(f"\n□ {teacher_name} – Fold {fold_no}")
        fold_start = time.time()

        train_df = df.iloc[train_idx]
        val_df = df.iloc[val_idx]

        train_gen = datagen.flow_from_dataframe(
            train_df, x_col='filename', y_col='label',
            target_size=image_size, class_mode='categorical',
            batch_size=teacher_batch_size, shuffle=True, color_mode =
'rgb'
        )
        val_gen = datagen.flow_from_dataframe(
            val_df, x_col='filename', y_col='label',
            target_size=image_size, class_mode='categorical',
            batch_size=teacher_batch_size, shuffle=False, color_mode =
'rgb'
        )

        # Create teacher for this fold
        teacher = create_base_model(teacher_fn, teacher_name)

        # --- Stage 1: Freeze backbone ---
        for layer in teacher.layers:
            if hasattr(layer, 'trainable'):
                layer.trainable = False

        teacher.compile(
            optimizer=Adam(1e-3),
            loss="categorical_crossentropy",
            metrics=["accuracy"]
        )
        history_head = teacher.fit(
            train_gen,
            validation_data=val_gen,
            epochs=head_epochs,
            class_weight=class_weight_dict,
            callbacks=teacher_callbacks_head,
            verbose=1
        )

        # --- Stage 2: Unfreeze backbone ---
        for layer in teacher.layers:
            layer.trainable = True

        teacher.compile(
            optimizer=Adam(1e-4),
            loss="categorical_crossentropy",
            metrics=["accuracy"]
        )
        history_finetune = teacher.fit(
            train_gen,
            validation_data=val_gen,
            epochs=finetune_epochs,
            class_weight=class_weight_dict,
            callbacks=teacher_callbacks_finetune,
            verbose=1
        )

        # Append trained teacher for ensemble
        trained_teachers.append(teacher)

        # --- Merge full history ---
        full_history = {}
        for key in set(list(history_head.history.keys()) +
list(history_finetune.history.keys())):
            full_history[key] = history_head.history.get(key, []) +
history_finetune.history.get(key, [])

        # --- Store full history per fold for averaging later ---
        if 'all_fold_histories' not in locals():
            all_fold_histories = []
        all_fold_histories.append(full_history)

        # --- Evaluate ---
        val_loss, val_acc = teacher.evaluate(val_gen, verbose=1)
        val_accs.append(val_acc)
        val_losses.append(val_loss)

        y_prob = teacher.predict(val_gen)
        y_pred = np.argmax(y_prob, axis=1)
        y_true = val_gen.classes

        conf_matrix = confusion_matrix(y_true, y_pred)
        report = classification_report(y_true, y_pred,
target_names=val_gen.class_indices.keys())

        # --- Save fold report ---
        fold_report_path = os.path.join(teacher_dir,
f"fold_{fold_no}_report.txt")
        with open(fold_report_path, "w") as f:
            f.write(f"Model: {teacher_name}\nFold: {fold_no}\n")
            f.write(f"Validation Loss: {val_loss:.4f}\nValidation
Accuracy: {val_acc:.4f}\n\n")
            f.write("Classification Report:\n" + report + "\n")
            f.write("Confusion Matrix:\n")
            for row in conf_matrix:
                f.write(" ".join(map(str, row)) + "\n")
        print(f"□ Saved fold report: {fold_report_path}")
        # ================================
        # ===== Evaluate ON TRAIN SET ===
        # ================================
        train_gen.reset()
        y_prob_train = teacher.predict(train_gen, verbose=1)
        y_pred_train = np.argmax(y_prob_train, axis=1)
        y_true_train = train_gen.classes

        # Append to overall lists
        all_y_true_train.extend(y_true_train)
        all_y_pred_train.extend(y_pred_train)

        train_conf_matrix = confusion_matrix(y_true_train,
y_pred_train)
        train_report = classification_report(y_true_train,
y_pred_train, target_names=train_gen.class_indices.keys())

        train_report_path = os.path.join(teacher_dir,
f"fold_{fold_no}_TRAIN_report.txt")
        with open(train_report_path, "w") as f:
            f.write(f"TRAIN Classification Report for fold {fold_no}\
n\n")
            f.write(train_report + "\n")
            f.write("Confusion Matrix:\n")
            for row in train_conf_matrix:
                f.write(" ".join(map(str, row)) + "\n")

        print(f"□ Saved TRAIN report: {train_report_path}")

        fold_end = time.time()
        fold_duration = fold_end - fold_start
        print(f"□ Fold {fold_no} training time: {fold_duration/60:.2f}
minutes")


        # --- Plot Accuracy & Loss ---
        def get_key(hist, prefix="", target="accuracy"):
            for k in hist:
                if target in k and k.startswith(prefix):
                    return k
            return None

        acc_key = get_key(full_history, target="accuracy")
        val_acc_key = get_key(full_history, prefix="val",
target="accuracy")
        loss_key = get_key(full_history, target="loss")
        val_loss_key = get_key(full_history, prefix="val",
target="loss")
```

```
        plt.figure(figsize=(12, 5))
        plt.subplot(1, 2, 1)
        if acc_key:
            plt.plot(full_history[acc_key], label="Train Accuracy")
        if val_acc_key:
            plt.plot(full_history[val_acc_key], label="Val Accuracy")
        plt.title(f"{teacher_name} - Fold {fold_no} Accuracy")
        plt.xlabel("Epochs")
        plt.ylabel("Accuracy")
        plt.legend()

        plt.subplot(1, 2, 2)
        if loss_key:
            plt.plot(full_history[loss_key], label="Train Loss")
        if val_loss_key:
            plt.plot(full_history[val_loss_key], label="Val Loss")
        plt.title(f"{teacher_name} - Fold {fold_no} Loss")
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.legend()

        plt.tight_layout()
        plt.savefig(os.path.join(teacher_dir,
f"fold{fold_no}_training_curves.png"))
        plt.close()

    # --- Compute average learning curves across folds ---
    max_epochs = max(len(h['accuracy']) for h in all_fold_histories)

    avg_train_acc, avg_val_acc = [], []
    avg_train_loss, avg_val_loss = [], []

    for epoch in range(max_epochs):
        acc_vals, val_acc_vals, loss_vals, val_loss_vals = [], [], [],
[]
        for h in all_fold_histories:
            # handle folds with fewer epochs
            acc_vals.append(h['accuracy'][epoch] if epoch <
len(h['accuracy']) else np.nan)
            val_acc_vals.append(h['val_accuracy'][epoch] if
'val_accuracy' in h and epoch < len(h['val_accuracy']) else np.nan)
            loss_vals.append(h['loss'][epoch] if epoch <
len(h['loss']) else np.nan)
            val_loss_vals.append(h['val_loss'][epoch] if 'val_loss' in
h and epoch < len(h['val_loss']) else np.nan)
        avg_train_acc.append(np.nanmean(acc_vals))
        avg_val_acc.append(np.nanmean(val_acc_vals))
        avg_train_loss.append(np.nanmean(loss_vals))
        avg_val_loss.append(np.nanmean(val_loss_vals))
    plt.figure(figsize=(12,5))

    plt.subplot(1,2,1)
    plt.plot(avg_train_acc, label='Avg Train Acc', color='blue')
    plt.plot(avg_val_acc, label='Avg Val Acc', color='orange')
    plt.title(f"{teacher_name} - Average Accuracy Across Folds")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.grid(alpha=0.3)

    plt.subplot(1,2,2)
    plt.plot(avg_train_loss, label='Avg Train Loss', color='blue')
    plt.plot(avg_val_loss, label='Avg Val Loss', color='orange')
    plt.title(f"{teacher_name} - Average Loss Across Folds")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid(alpha=0.3)

    plt.tight_layout()
    avg_curve_path = os.path.join(teacher_dir,
f"{teacher_name}_avg_learning_curves.png")
    plt.savefig(avg_curve_path)
    plt.close()
    print(f"□ Saved average learning curves across folds for
{teacher_name} to: {avg_curve_path}")

    # ================================
    # FINAL SUMMARY (TEACHER REPORT)
    # ================================
    print(f"\n□ Generating final summary for {teacher_name}...")
    teacher_end = time.time()
    teacher_time = teacher_end - teacher_start
    print(f"□ Total training time for {teacher_name}:
{teacher_time/60:.2f} minutes ({teacher_time/3600:.2f} hours)")

    # Evaluate on test set for final report
    test_loss, test_acc = teacher.evaluate(test_gen, verbose=1)

    # ================================
    # TEST SET PREDICTION & CONF MATRIX
    # ================================
    print("□ Running predictions on TEST set...")
    # Ensure generator alignment and no shuffling
    test_gen.reset()
    test_gen.shuffle = False

    y_prob_test = teacher.predict(test_gen, verbose=1)   # no steps
argument
```

```
    y_prob_test = y_prob_test[:test_gen.samples]
# trim extra if any

    y_pred_test = np.argmax(y_prob_test, axis=1)
    y_true_test = test_gen.classes

    print("y_true:", len(y_true_test))
    print("y_prob:", y_prob_test.shape)

    class_names = list(test_gen.class_indices.keys())

    # Confusion Matrix
    conf_mat_test = confusion_matrix(y_true_test, y_pred_test)

    # Save classification report
    test_report = classification_report(y_true_test, y_pred_test,
target_names=class_names, zero_division=0)
    test_report_path = os.path.join(teacher_dir,
f"TEST_{teacher_name}_report.txt")
    with open(test_report_path, "w") as f:
        f.write(f"TEST Classification Report\n\n")
        f.write(test_report + "\n")
        f.write("Confusion Matrix:\n")
        for row in conf_mat_test:
            f.write(" ".join(map(str, row)) + "\n")

    print(f"□ Saved TEST report: {test_report_path}")

    # ---- Plot Confusion Matrix ---
    plt.figure(figsize=(6,5))
    sns.heatmap(conf_mat_test, annot=True, fmt="d", cmap="Blues",
                xticklabels=class_names, yticklabels=class_names)
    plt.title(f"{teacher_name} - Test Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.tight_layout()
    plt.savefig(os.path.join(teacher_dir,
f"{teacher_name}_TEST_confusion_matrix.png"))
    plt.close()
    print("□ Saved Test Confusion Matrix image")

    # ------------------------------
    # Final Train Report for this teacher
    # ------------------------------
    final_train_report = classification_report(
        all_y_true_train,
        all_y_pred_train,
        target_names=train_gen.class_indices.keys(),
        zero_division=0
    )
    final_train_conf = confusion_matrix(all_y_true_train,
all_y_pred_train)

    final_train_report_path = os.path.join(teacher_dir,
f"FINAL_TRAIN_{teacher_name}_report.txt")
    with open(final_train_report_path, "w") as f:
        f.write(f"FINAL TRAIN Classification Report for {teacher_name}
(All Folds)\n\n")
        f.write(final_train_report + "\n")
        f.write("Confusion Matrix:\n")
        for row in final_train_conf:
            f.write(" ".join(map(str, row)) + "\n")

    print(f"□ Saved FINAL TRAIN report for {teacher_name}:
{final_train_report_path}")

    # -----------------------------
    # Final Train Confusion Matrix (heatmap)
    # -----------------------------
    plt.figure(figsize=(6,5))
    sns.heatmap(final_train_conf, annot=True, fmt="d",
                xticklabels=list(train_gen.class_indices.keys()),
                yticklabels=list(train_gen.class_indices.keys()))
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.title(f"FINAL TRAIN Confusion Matrix - {teacher_name}")
    plt.tight_layout()
    conf_png_path = os.path.join(teacher_dir,
f"FINAL_TRAIN_{teacher_name}_confusion_matrix.png")
    plt.savefig(conf_png_path)
    plt.close()
    print(f"□ Saved FINAL TRAIN confusion matrix heatmap for
{teacher_name}: {conf_png_path}")


    # Predict on validation generator
    val_gen.reset()
    y_true = val_gen.classes
    y_prob = teacher.predict(val_gen, verbose=1)
    y_pred = np.argmax(y_prob, axis=1)

    # Class names
    target_names = list(val_gen.class_indices.keys())

    # Compute classification metrics
    conf_matrix = confusion_matrix(y_true, y_pred)
    report = classification_report(y_true, y_pred,
target_names=target_names, zero_division=0)

    # Average metrics across folds
    avg_acc = np.mean(val_accs)
    avg_loss = np.mean(val_losses)

    # Save summary report
    report_path = os.path.join(teacher_dir,
f"final_{teacher_name}_report.txt")
    with open(report_path, "w") as f:
        f.write(f"Average Validation Accuracy across folds:
{avg_acc:.4f}\n")
        f.write(f"Average Validation Loss across folds:
{avg_loss:.4f}\n")
        f.write(f"Test Accuracy: {test_acc:.4f}\n")
        f.write(f"Test Loss: {test_loss:.4f}\n\n")
        f.write("Classification Report:\n")
        f.write(report + "\n")
        f.write("Confusion Matrix:\n")
        for row in conf_matrix:
            f.write(" ".join(map(str, row)) + "\n")

    print(f"□ Final teacher report saved to: {report_path}")


    # ================================
    # SAVE FINAL TRAINED TEACHER MODEL
    # ================================
    final_teacher = teacher  # the last trained model after fine-
tuning
    final_teacher_path = os.path.join(teacher_dir,
f"final_{teacher_name}_model")

    try:
        # Save in TensorFlow (Keras) format — safer for modern TF
versions
        final_teacher.save(f"{final_teacher_path}.keras",
include_optimizer=False)
        print(f"□ Saved final {teacher_name} model (Keras format) to:
{final_teacher_path}")
    except Exception as e:
        print(f"⚠ Could not save {teacher_name} using Keras save.
Reason: {e}")

    # --- Explicit SavedModel export (replacement for .export()) ---
    saved_teacher_dir = os.path.join(teacher_dir,
f"final_{teacher_name}_savedmodel")
    tf.saved_model.save(final_teacher, saved_teacher_dir)
    print(f"□ Exported TensorFlow SavedModel directory for
{teacher_name} to: {saved_teacher_dir}")

    # --- Convert to TensorFlow Lite (for mobile / edge deployment)
---
```

```
    converter =
tf.lite.TFLiteConverter.from_saved_model(saved_teacher_dir)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    tflite_teacher_model = converter.convert()

    tflite_teacher_path = os.path.join(teacher_dir,
f"final_{teacher_name}_mobile.tflite")
    with open(tflite_teacher_path, "wb") as f:
        f.write(tflite_teacher_model)

    print(f"□ Saved {teacher_name} TFLite model for mobile deployment
to: {tflite_teacher_path}")

    # --- Optional: free memory before next teacher model ---
    tf.keras.backend.clear_session()
    del teacher
    gc.collect()

    # Create full train generator (all training data)
    full_train_gen = datagen.flow_from_dataframe(
        dataframe=df,
        x_col='filename',
        y_col='label',
        target_size=image_size,
        class_mode='categorical',
        batch_size=teacher_batch_size,
        shuffle=True,
        color_mode = 'rgb'
    )

    # After all folds for this teacher are trained
    all_teacher_models.extend(trained_teachers)
    # ================================
    # GRAD-CAM FOR THIS TEACHER MODEL
    # ================================
    print(f"□ Generating Grad-CAM results for {teacher_name} ...")

    gradcam_output_dir = os.path.join(teacher_dir,
f"gradcam_{teacher_name}")
    os.makedirs(gradcam_output_dir, exist_ok=True)

    # For GradCAM for EACH MODEL from each fold:
    # for idx, t_model in enumerate(trained_teachers):
    #     fold_gradcam_dir = os.path.join(gradcam_output_dir,
f"fold_{idx+1}")
    #     generate_and_eval_gradcam(
    #         model=t_model,
    #         test_gen=test_gen,
    #         save_dir=fold_gradcam_dir,
    #         all_teacher_models=all_teacher_models,
    #         classes=["benign", "malignant"],
    #         image_size=image_size
    #     )

    # OR, if you want only final teacher:
    generate_and_eval_gradcam(
        model=final_teacher,
        test_gen=test_gen,
        save_dir=gradcam_output_dir,
        all_teacher_models=all_teacher_models,
        classes=["benign", "malignant"],
        image_size=image_size
    )

    print(f"□ Completed Grad-CAM for teacher: {teacher_name}")



# ================================
# Create the Ensemble
# ================================
ensemble_teacher = EnsembleTeacher(all_teacher_models)

print("Teachers inside ensemble:")
for m in all_teacher_models:
    print(m.name)


# ================================
# DISTILLATION (Student)
# ================================
print("□ Student Training Start")
student_start = time.time()


student_dir = os.path.join(save_dir, "Student_Ensemble")
os.makedirs(student_dir, exist_ok=True)

student = create_student()
distiller = Distiller(student=student, teacher=ensemble_teacher)
distiller.compile(
    optimizer=Adam(1e-4),
    metrics=["accuracy", Precision(name="precision"),
Recall(name="recall"), AUC(name="auc")],
    student_loss_fn=tf.keras.losses.CategoricalCrossentropy(),
    distillation_loss_fn=tf.keras.losses.KLDivergence(),
    alpha=0.7, # Change to 0.7 for future train
    temperature=5 # Change to 5 for future train
)
```

```
train_df, val_df = train_test_split(df, test_size=0.1,
stratify=df['label'], random_state=42)

train_gen_student = datagen.flow_from_dataframe(
    dataframe=train_df,
    x_col='filename',
    y_col='label',
    target_size=image_size,
    class_mode='categorical',
    batch_size=student_batch_size,
    shuffle=True,
    color_mode = 'rgb'
)
val_gen_student = datagen.flow_from_dataframe(
    dataframe=val_df,
    x_col='filename',
    y_col='label',
    target_size=image_size,
    class_mode='categorical',
    batch_size=student_batch_size,
    shuffle=False,
    color_mode = 'rgb'
)

# Fit with explicit validation_data
history_student = distiller.fit(
    train_gen_student,
    validation_data=val_gen_student,
    epochs=student_epochs,
    class_weight=class_weight_dict,
    callbacks=student_callbacks,
    verbose=1
)

results = distiller.evaluate(val_gen_student, return_dict=True)
student_end = time.time()
student_time = student_end - student_start
print(f"□ Total Student training time: {student_time/60:.2f} minutes
({student_time/3600:.2f} hours)")

# -------------------------------
# Assemble Keras Ensemble model (so it can be saved/loaded)
# -------------------------------
import sklearn.metrics as skm

# Make sure all_teacher_models is non-empty
if len(all_teacher_models) == 0:
    raise RuntimeError("No teacher models found in all_teacher_models
to ensemble.")
```

```
# ================================================
# FIX: Safely wrap each teacher with unique prefixes
# ================================================
renamed_teachers = []

for i, teacher in enumerate(all_teacher_models):
    x = tf.keras.Input(shape=(image_size[0], image_size[1], 3))
    y = teacher(x, training=False)  # inference mode
    model_wrapped = tf.keras.Model(x, y, name=f"teacher_{i}")
    renamed_teachers.append(model_wrapped)

# build ensemble model with safe names
ensemble_input = tf.keras.Input(shape=(image_size[0], image_size[1],
3))
teacher_outputs = [m(ensemble_input, training=False) for m in
renamed_teachers]
ensemble_avg = tf.keras.layers.Average(name="ensemble_average")
(teacher_outputs)

ensemble_model = tf.keras.Model(ensemble_input, ensemble_avg,
name="Ensembled_Teachers")


print("□ Built functional ensemble_model from trained teachers.
Summary:")
ensemble_model.summary()

# ------------------------------
# Evaluate ensemble on test set and save metrics + visualizations
# ------------------------------
# Reset test generator
test_gen.reset()
y_true_test = test_gen.classes

print("□ Predicting with ensemble on test set...")
y_prob_test = ensemble_model.predict(test_gen, verbose=1)
y_pred_test = np.argmax(y_prob_test, axis=1)

# Classification report & confusion matrix
ensemble_report = classification_report(y_true_test, y_pred_test,
target_names=list(test_gen.class_indices.keys()), zero_division=0)
ensemble_conf = confusion_matrix(y_true_test, y_pred_test)

# Save textual report
ensemble_report_path = os.path.join(save_dir,
"ensemble_test_report.txt")
with open(ensemble_report_path, "w") as f:
    f.write("Ensemble Test Classification Report\n\n")
    f.write(ensemble_report + "\n")
    f.write("Confusion Matrix:\n")
    for row in ensemble_conf:
        f.write(" ".join(map(str, row)) + "\n")
print(f"□ Saved ensemble test report to: {ensemble_report_path}")

# Plot confusion matrix heatmap
plt.figure(figsize=(6,5))
sns.heatmap(ensemble_conf, annot=True, fmt="d",
xticklabels=list(test_gen.class_indices.keys()),
yticklabels=list(test_gen.class_indices.keys()))
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Ensemble - Confusion Matrix (Test)")
conf_png = os.path.join(save_dir, "ensemble_confusion_matrix.png")
plt.tight_layout()
plt.savefig(conf_png)
plt.close()
print(f"□ Saved ensemble confusion matrix heatmap to: {conf_png}")

# ROC curve & AUC (binary)
try:
    # use probability of positive class (class index 1)
    if y_prob_test.shape[1] == 2:
        pos_prob = y_prob_test[:, 1]
        fpr, tpr, _ = skm.roc_curve(y_true_test, pos_prob)
        roc_auc = skm.auc(fpr, tpr)
        plt.figure(figsize=(6,6))
        plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.4f}")
        plt.plot([0,1],[0,1],"--")
        plt.xlabel("False Positive Rate")
        plt.ylabel("True Positive Rate")
        plt.title("Ensemble ROC Curve (Test)")
        plt.legend(loc="lower right")
        roc_png = os.path.join(save_dir, "ensemble_roc_curve.png")
        plt.tight_layout()
        plt.savefig(roc_png)
        plt.close()
        print(f"□ Saved ensemble ROC curve to: {roc_png}")
    else:
        print("ℹ ROC curve skipped - multi-class or unexpected output
shape.")
except Exception as e:
    print(f"⚠ Could not compute ROC curve: {e}")

# ------------------------------
# Save ensemble model in multiple formats (SavedModel, .keras, TFLite)
# ------------------------------
ensemble_savedmodel_dir = os.path.join(save_dir,
"ensemble_savedmodel")
ensemble_keras_path = os.path.join(save_dir, "ensemble_model.keras")
ensemble_tflite_path = os.path.join(save_dir,
"ensemble_model_mobile.tflite")

# SavedModel
try:
    tf.saved_model.save(ensemble_model, ensemble_savedmodel_dir)
    print(f"□ Exported ensemble SavedModel to:
{ensemble_savedmodel_dir}")
except Exception as e:
    print(f"⚠ Could not save ensemble as SavedModel: {e}")

# .keras format
try:
    ensemble_model.save(ensemble_keras_path, include_optimizer=False)
    print(f"□ Saved ensemble (.keras) to: {ensemble_keras_path}")
except Exception as e:
    print(f"⚠ Could not save ensemble .keras: {e}")

# TFLite conversion (from SavedModel)
try:
    converter =
tf.lite.TFLiteConverter.from_saved_model(ensemble_savedmodel_dir)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    tflite_ensemble = converter.convert()
    with open(ensemble_tflite_path, "wb") as f:
        f.write(tflite_ensemble)
    print(f"□ Saved ensemble TFLite model to: {ensemble_tflite_path}")
except Exception as e:
    print(f"⚠ Could not convert ensemble to TFLite: {e}")

print("□ Running Grad-CAM for Ensemble Model ...")

ensemble_gradcam_dir = os.path.join(save_dir, "gradcam_ensemble")
os.makedirs(ensemble_gradcam_dir, exist_ok=True)

generate_and_eval_gradcam(
    model=ensemble_model,
    test_gen=test_gen,
    save_dir=ensemble_gradcam_dir,
    all_teacher_models=all_teacher_models,
    classes=["benign", "malignant"],
    image_size=image_size
)

# -----------------------------
# Save the final STUDENT model (from distiller.student) similarly +
visualizations
# -----------------------------
# After student is trained (i.e., after distiller.fit(...)), create
final_student ref:
final_student = distiller.student if hasattr(distiller, "student")
```

```
else student

# Evaluate student on test set
test_gen.reset()
y_prob_student = final_student.predict(test_gen, verbose=1)
y_pred_student = np.argmax(y_prob_student, axis=1)
student_conf = confusion_matrix(y_true_test, y_pred_student)
student_report = classification_report(y_true_test, y_pred_student,
target_names=list(test_gen.class_indices.keys()), zero_division=0)

# Save student report
student_report_path = os.path.join(save_dir,
"student_test_report.txt")
with open(student_report_path, "w") as f:
    f.write("Student Test Classification Report\n\n")
    f.write(student_report + "\n")
    f.write("Confusion Matrix:\n")
    for row in student_conf:
        f.write(" ".join(map(str, row)) + "\n")
print(f"□ Saved student test report to: {student_report_path}")

# Plot student confusion matrix
plt.figure(figsize=(6,5))
sns.heatmap(student_conf, annot=True, fmt="d",
xticklabels=list(test_gen.class_indices.keys()),
yticklabels=list(test_gen.class_indices.keys()))
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Student - Confusion Matrix (Test)")
student_conf_png = os.path.join(save_dir,
"student_confusion_matrix.png")
plt.tight_layout()
plt.savefig(student_conf_png)
plt.close()
print(f"□ Saved student confusion matrix heatmap to:
{student_conf_png}")

# Save student savedmodel / .keras / tflite
final_student_savedmodel_dir = os.path.join(save_dir,
"final_student_savedmodel")
final_student_keras = os.path.join(save_dir,
"final_student_model.keras")
final_student_tflite = os.path.join(save_dir,
"final_student_mobile.tflite")

try:
    tf.saved_model.save(final_student, final_student_savedmodel_dir)
    print(f"□ Exported final student SavedModel to:
{final_student_savedmodel_dir}")
except Exception as e:
    print(f"⚠ Could not save student as SavedModel: {e}")
    # fallback to weights
    try:
        weights_path = os.path.join(save_dir,
"final_student_weights.h5")
        final_student.save_weights(weights_path)
        print(f"� Saved student weights to: {weights_path}")
    except Exception as e2:
        print(f"⚠ Also failed to save student weights: {e2}")

try:
    final_student.save(final_student_keras, include_optimizer=False)
    print(f"� Saved student (.keras) to: {final_student_keras}")
except Exception as e:
    print(f"⚠ Could not save student .keras: {e}")

# Convert to TFLite
try:
    converter =
tf.lite.TFLiteConverter.from_saved_model(final_student_savedmodel_dir)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    tflite_student = converter.convert()
    with open(final_student_tflite, "wb") as f:
        f.write(tflite_student)
    print(f"� Saved final student TFLite model to:
{final_student_tflite}")
except Exception as e:
    print(f"⚠ Could not convert student to TFLite: {e}")

# -------------------------------
# Run Grad-CAM pipeline
# -------------------------------
# for l in final_student.layers:
#     print(l.name, l.output_shape, type(l))


generate_and_eval_gradcam(
    model=final_student,
    test_gen=test_gen,
    save_dir=os.path.join(save_dir, "gradcam_results"),
    all_teacher_models=[],   # <-- add this
    classes=["benign", "malignant"],
    image_size=image_size
)


# ------------------------------
# Minimal ensemble vs student summary file
# ------------------------------
summary_path = os.path.join(save_dir, "ensemble_student_summary.txt")
```

```
with open(summary_path, "w") as f:
    f.write("Ensemble vs Student - Test Summary\n\n")
    f.write("=== Ensemble Report ===\n")
    f.write(ensemble_report + "\n\n")
    f.write("=== Student Report ===\n")
    f.write(student_report + "\n\n")
    if 'roc_auc' in locals():
        f.write(f"Ensemble AUC: {roc_auc:.4f}\n")
print(f"□ Saved combined summary to: {summary_path}")


metrics = results.get("compile_metrics", {})

# Try to find total loss if available, otherwise fallback to student_loss
val_loss = results.get("loss", results.get("student_loss", None))
val_acc = results.get("accuracy", results.get("student_accuracy", 0))
precision = results.get("precision", results.get("student_precision", 0))
recall = results.get("recall", results.get("student_recall", 0))
auc_value = results.get("auc", results.get("student_auc", 0))  # □ rename here

fold_results.append({
    "fold": fold_no,
    "val_loss": float(val_loss) if val_loss is not None else 0.0,
    "val_acc": float(val_acc),
    "precision": float(precision),
    "recall": float(recall),
    "auc": float(auc_value)  # □ use the new name
})
print(f"□ Fold {fold_no} results: {results}")

# --- Plot training curves per fold ---
plt.figure(figsize=(12, 5))

# Accuracy subplot
plt.subplot(1, 2, 1)
plt.plot(history_head.history["accuracy"], label="Head Train Acc", linestyle='--')
plt.plot(history_head.history["val_accuracy"], label="Head Val Acc", linestyle='--')
plt.plot(history_finetune.history["accuracy"], label="Fine-tune Train Acc", linestyle='-.')
plt.plot(history_finetune.history["val_accuracy"], label="Fine-tune Val Acc", linestyle='-.')
plt.title(f"{teacher_name} - Fold {fold_no} Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
```

```
# Loss subplot
plt.subplot(1, 2, 2)
plt.plot(history_head.history["loss"], label="Head Train Loss",
linestyle='--')
plt.plot(history_head.history["val_loss"], label="Head Val Loss",
linestyle='--')
plt.plot(history_finetune.history["loss"], label="Fine-tune Train
Loss", linestyle='-.')
plt.plot(history_finetune.history["val_loss"], label="Fine-tune Val
Loss", linestyle='-.')
plt.title(f"{teacher_name} - Fold {fold_no} Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

plt.tight_layout()
plt.savefig(os.path.join(save_dir,
f"fold{fold_no}_training_curves.png"))
plt.close()


# ================================
# POST-EVALUATION (AVERAGE METRICS)
# ================================
mean_acc = np.mean([r["val_acc"] for r in fold_results if
r.get("val_acc") is not None])
print(f"\n Average Validation Accuracy across folds: {mean_acc:.4f}")

# ================================
# FINAL EVALUATION ON TEST SET
# ================================
print("\n=== Final Evaluation on Test Set ===")

# Use the last trained student model
final_student = student

#  Compile student before evaluation
final_student.compile(
    optimizer=Adam(1e-4),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

# ================================
# EVALUATE & GENERATE PREDICTIONS
# ================================
test_loss, test_acc = final_student.evaluate(test_gen, verbose=1)
print(f" Test Accuracy: {test_acc:.4f}")
print(f" Test Loss: {test_loss:.4f}")
```

```
# Reset generator and predict
test_gen.reset()
y_prob = final_student.predict(test_gen)

# Handle both binary (sigmoid) and multi-class (softmax)
if y_prob.shape[1] == 1:
    y_score = y_prob.ravel()  # Binary sigmoid
else:
    y_score = y_prob  # Multi-class softmax
y_pred = np.argmax(y_prob, axis=1)

# Align lengths
y_true = test_gen.classes[:len(y_pred)]
y_pred = y_pred[:len(y_true)]
if y_score.ndim > 1:
    y_score = y_score[:len(y_true), :]

# Target names
target_names = list(test_gen.class_indices.keys())

# Classification report & confusion matrix
print("Classification Report:")
print(classification_report(y_true, y_pred, target_names=target_names,
zero_division=0))
print("Confusion Matrix:")
print(confusion_matrix(y_true, y_pred))

# ===============================
# SAVE FINAL STUDENT MODEL (FOR TESTING & DEPLOYMENT)
# ===============================
final_model_path = os.path.join(save_dir, "final_student_model")
final_savedmodel_dir = os.path.join(save_dir,
"final_student_savedmodel")
final_tflite_path = os.path.join(save_dir,
"final_student_mobile.tflite")

try:
    # □ Save using TensorFlow SavedModel (recommended)
    tf.saved_model.save(final_student, final_savedmodel_dir)
    print(f"□ Exported TensorFlow SavedModel directory to:
{final_savedmodel_dir}")
except Exception as e:
    print(f"⚠ Could not save student as SavedModel: {e}")
    # Fallback: save weights only
    weights_path = os.path.join(save_dir, "final_student_weights.h5")
    final_student.save_weights(weights_path)
    print(f"□ Saved student weights only to: {weights_path}")

# □ Optional — also save as standard .keras model (Functional API
compatible)
```

```python
try:
    final_student.save(f"{final_model_path}.keras",
include_optimizer=False)
    print(f"□ Saved student model (.keras) to:
{final_model_path}.keras")
except Exception as e:
    print(f"⚠ Could not save .keras model: {e}")

# ================================
# CONVERT TO TENSORFLOW LITE
# ================================
try:
    converter =
tf.lite.TFLiteConverter.from_saved_model(final_savedmodel_dir)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    tflite_model = converter.convert()

    with open(final_tflite_path, "wb") as f:
        f.write(tflite_model)
    print(f"□ Saved TFLite model for mobile deployment to:
{final_tflite_path}")
except Exception as e:
    print(f"⚠ Could not convert to TFLite: {e}")


# ================================
# ROC-AUC (binary only)
# ================================
if len(target_names) == 2:
    # Use positive class probabilities for ROC
    if y_score.shape[1] == 2:
        y_score_pos = y_score[:, 1]
    else:
        y_score_pos = y_score  # Already shape (num_samples,)

    # Ensure no NaN
    y_score_pos = np.nan_to_num(y_score_pos)

    fpr, tpr, thresholds = roc_curve(y_true, y_score_pos)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(6,6))
    plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.4f}")
    plt.plot([0,1],[0,1],'r--')
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve - Final Student Model")
    plt.legend(loc="lower right")
    plt.grid()
    plt.savefig(os.path.join(save_dir, "roc_curve.png"))
    plt.close()
    print(f"□ ROC curve saved to: {os.path.join(save_dir,
'roc_curve.png')}")
else:
    print("⚠ ROC-AUC is only computed for binary classification.")

# ================================
# CONFUSION MATRIX & CLASSIFICATION REPORT (SAVE)
# ================================
conf_matrix = confusion_matrix(y_true, y_pred)
report = classification_report(y_true, y_pred,
target_names=target_names, zero_division=0)

report_path = os.path.join(save_dir, "final_student_report.txt")
with open(report_path, "w") as f:
    f.write(f"Average Validation Accuracy across folds:
{mean_acc:.4f}\n")
    f.write(f"Test Accuracy: {test_acc:.4f}\n")
    f.write(f"Test Loss: {test_loss:.4f}\n\n")
    f.write("Classification Report:\n")
    f.write(report + "\n")
    f.write("Confusion Matrix:\n")
    for row in conf_matrix:
        f.write(" ".join(map(str, row)) + "\n")

print(f"□ Report saved to: {report_path}")

# ================================
# CONFUSION MATRIX (RAW)
# ================================
plt.figure(figsize=(6, 5))
sns.heatmap(
    conf_matrix,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=test_gen.class_indices.keys(),
    yticklabels=test_gen.class_indices.keys()
)
plt.title("Confusion Matrix - Final Student Model")
plt.savefig(os.path.join(save_dir, "confusion_matrix.png"))
plt.close()

# ================================
# CONFUSION MATRIX (NORMALIZED)
# ================================
conf_matrix_norm = conf_matrix.astype("float") /
conf_matrix.sum(axis=1)[:, np.newaxis]

plt.figure(figsize=(6, 5))
```

```
sns.heatmap(
    conf_matrix_norm,
    annot=True,
    fmt=".2f",
    cmap="Greens",
    xticklabels=test_gen.class_indices.keys(),
    yticklabels=test_gen.class_indices.keys()
)
plt.title("Normalized Confusion Matrix - Final Student Model")
plt.savefig(os.path.join(save_dir, "confusion_matrix_normalized.png"))
plt.close()

# ===============================
# PRECISION–RECALL CURVE
# ===============================
if len(target_names) == 2:
    # Get positive class probability
    if y_prob.shape[1] == 2:
        y_score_pos = y_prob[:, 1]
    else:
        y_score_pos = y_prob.ravel()  # Binary sigmoid case

    # Remove NaNs
    y_score_pos = np.nan_to_num(y_score_pos)

    # Ensure y_true and y_score_pos have same length
    min_len = min(len(y_true), len(y_score_pos))
    y_true_trim = y_true[:min_len]
    y_score_trim = y_score_pos[:min_len]

    # Compute Precision–Recall curve
    precision, recall, thresholds =
precision_recall_curve(y_true_trim, y_score_trim)

    plt.figure(figsize=(6,6))
    plt.plot(recall, precision, label="PR Curve")
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.title("Precision–Recall Curve - Student Model")
    plt.legend()
    plt.grid()
    plt.savefig(os.path.join(save_dir, "precision_recall_curve.png"))
    plt.close()
    print(f"□ Precision–Recall curve saved to: {os.path.join(save_dir,
'precision_recall_curve.png')}")
else:
    print("⚠ Precision–Recall curve is only computed for binary
classification.")
```

```python
# ================================
# PER-CLASS PRECISION, RECALL, F1 BAR CHART
# ================================
prec, rec, f1, _ = precision_recall_fscore_support(y_true, y_pred,
zero_division=0)
classes = list(test_gen.class_indices.keys())

bar_width = 0.25
r1 = np.arange(len(classes))

plt.figure(figsize=(8, 5))
plt.bar(r1, prec, width=bar_width, label="Precision")
plt.bar(r1 + bar_width, rec, width=bar_width, label="Recall")
plt.bar(r1 + 2 * bar_width, f1, width=bar_width, label="F1-score")
plt.xticks(r1 + bar_width, classes)
plt.ylabel("Score")
plt.title("Per-Class Metrics")
plt.legend()
plt.grid(axis="y")
plt.savefig(os.path.join(save_dir, "class_metrics_bar.png"))
plt.close()

# ================================
# PREDICTION CONFIDENCE HISTOGRAM
# ================================
confidences = np.nan_to_num(np.max(y_prob, axis=1))

plt.figure(figsize=(7, 5))
plt.hist(confidences, bins=20, color="purple", alpha=0.7)
plt.title("Prediction Confidence Distribution")
plt.xlabel("Confidence")
plt.ylabel("Frequency")
plt.grid()
plt.savefig(os.path.join(save_dir, "confidence_histogram.png"))
plt.close()

# ================================
# MISCLASSIFIED SAMPLES (OPTIONAL)
# ================================
y_true_trim, y_pred_trim, y_prob_trim = y_true[:min_len],
y_pred[:min_len], y_prob[:min_len]
wrong_idx = np.where(y_pred_trim != y_true_trim)[0]

if len(wrong_idx) > 0:
    sample_wrong = np.random.choice(wrong_idx, min(9, len(wrong_idx)),
replace=False)
    plt.figure(figsize=(10, 10))

    for i, idx in enumerate(sample_wrong):
        img_path = test_gen.filepaths[idx]
        img = plt.imread(img_path)
        plt.subplot(3, 3, i + 1)
        plt.imshow(img)
        plt.title(
            f"True: {list(test_gen.class_indices.keys())
[y_true[idx]]}\n"
            f"Pred: {list(test_gen.class_indices.keys())
[y_pred[idx]]}"
        )
        plt.axis("off")

    plt.tight_layout()
    plt.savefig(os.path.join(save_dir, "misclassified_samples.png"))
    plt.close()

# Calibration Curve
if len(target_names) == 2:
    y_score_pos = np.nan_to_num(y_prob[:, 1])
    prob_true, prob_pred = calibration_curve(y_true_trim,
y_score_pos[:min_len], n_bins=10)

plt.figure(figsize=(6, 6))
plt.plot(prob_pred, prob_true, marker="o", label="Calibration")
plt.plot([0, 1], [0, 1], "k--", label="Perfectly Calibrated")
plt.xlabel("Predicted Probability")
plt.ylabel("True Probability")
plt.title("Reliability Curve (Calibration Plot)")
plt.legend()
plt.grid()
plt.savefig(os.path.join(save_dir, "calibration_curve.png"))
plt.close()


# Cumulative Gain Curve
if len(target_names) == 2:
    y_score_pos = np.nan_to_num(y_prob[:, 1])
    sorted_idx = np.argsort(y_score_pos[:min_len])[::-1]
    y_true_sorted = np.array(y_true_trim)[sorted_idx]

cum_positive_rate = np.cumsum(y_true_sorted) / np.sum(y_true_sorted)
perc_samples = np.arange(1, len(y_true_sorted)+1) / len(y_true_sorted)

plt.figure(figsize=(6, 6))
plt.plot(perc_samples, cum_positive_rate, label="Model")
plt.plot([0, 1], [0, 1], "k--", label="Random")
plt.xlabel("Proportion of Samples")
plt.ylabel("Cumulative Proportion of Positives")
plt.title("Cumulative Gain Chart")
plt.legend()
plt.grid()
plt.savefig(os.path.join(save_dir, "cumulative_gain_curve.png"))
plt.close()

# ================================
# F1-SCORE vs DECISION THRESHOLD
# ================================
# Ensure binary case
if y_prob.shape[1] == 2:
    # Binary classification
    thresholds = np.linspace(0, 1, 100)
    f1s = []

    for t in thresholds:
        y_pred_thresh = (y_prob[:, 1] >= t).astype(int)
        f1 = f1_score(y_true, y_pred_thresh, zero_division=0)
        f1s.append(f1)

    plt.figure(figsize=(6, 5))
    plt.plot(thresholds, f1s, color="teal")
    plt.xlabel("Decision Threshold")
    plt.ylabel("F1 Score")
    plt.title("F1 Score vs Decision Threshold")
    plt.grid(True)

    # Mark optimal threshold
    best_idx = np.argmax(f1s)
    best_threshold = thresholds[best_idx]
    best_f1 = f1s[best_idx]
    plt.axvline(best_threshold, color="red", linestyle="--",
label=f"Best t = {best_threshold:.2f}")
    plt.legend()

    plt.savefig(os.path.join(save_dir, "f1_vs_threshold.png"))
    plt.close()

    print(f"□ F1 vs Threshold plot saved. Optimal threshold ≈
{best_threshold:.2f}, F1 = {best_f1:.4f}")

else:
    # Multi-class classification — skip or use one-vs-rest
    print("⚠ F1 vs Threshold skipped (only valid for binary
classification).")


# ================================
# CLASS DISTRIBUTION vs CONFIDENCE
# ================================
plt.figure(figsize=(7, 5))

# Map: class_index -> class_name
```

```python
idx_to_class = {v: k for k, v in test_gen.class_indices.items()}

# Ensure y_prob and y_true exist
if 'y_prob' not in locals():
    y_prob = final_student.predict(test_gen)
if 'y_true' not in locals():
    y_true = np.array(test_gen.classes)

# Iterate per class and plot histogram of predicted confidence
for i, cls in idx_to_class.items():
    # Predicted confidence for true samples of this class
    cls_confidences = y_prob[y_true == i, i] if y_prob.shape[1] > i
else np.array([])

    # Remove NaNs
    cls_confidences = cls_confidences[~np.isnan(cls_confidences)]

    if len(cls_confidences) == 0:
        print(f"⚠ No valid predictions for class '{cls}', skipping.")
        continue  # skip empty class

    plt.hist(
        cls_confidences,
        bins=20,
        alpha=0.6,
        label=f"{cls} (n={len(cls_confidences)})",
        density=True,  # normalize to make shape comparable
        edgecolor="black"
    )

plt.xlabel("Predicted Probability for True Class")
plt.ylabel("Density")
plt.title("Confidence Distribution per Class")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()

save_path = os.path.join(save_dir,
"class_confidence_distribution.png")
plt.savefig(save_path)
plt.close()

print(f"□ Class confidence distribution plot saved to: {save_path}")

# ===============================
# K-FOLD METRICS BOXPLOT
# ===============================
# Convert results list of dicts into DataFrame
metrics_df = pd.DataFrame(fold_results)
```

```
if metrics_df.empty:
    print("⚠ fold_results is empty, skipping boxplot.")
else:
    # --- Sanity check for available columns ---
    available_cols = [c for c in ["val_acc", "val_f1"] if c in
metrics_df.columns]
    if len(available_cols) == 0:
        raise ValueError("⚠ No valid metric columns found in
fold_results. Expected 'val_acc' or 'val_f1'.")

    # --- Boxplot ---
    plt.figure(figsize=(6, 5))
    metrics_df.boxplot(column=available_cols)
    plt.title("K-Fold Validation Metric Distribution")
    plt.ylabel("Score")
    plt.grid(axis="y", alpha=0.5)
    plt.tight_layout()

    # --- Save plot ---
    save_path = os.path.join(save_dir, "kfold_metrics_boxplot.png")
    plt.savefig(save_path)
    plt.close()

    print(f"□ K-Fold metrics boxplot saved to: {save_path}")
    print(f"□ Columns plotted: {available_cols}")

# ================================
# LEARNING CURVES (ACCURACY / LOSS)
# ================================
# --- Accuracy Plots ---
# Head accuracy
plt.figure(figsize=(6,5))
plt.plot(history_head.history["accuracy"], label="Train Acc",
linestyle='--')
plt.plot(history_head.history["val_accuracy"], label="Val Acc",
linestyle='--')
plt.title("Head Training Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir, "head_accuracy_curve.png"))
plt.close()
print("□ Saved: head_accuracy_curve.png")

# Fine-tune accuracy
plt.figure(figsize=(6,5))
plt.plot(history_finetune.history["accuracy"], label="Train Acc",
linestyle='-.')
```

```
plt.plot(history_finetune.history["val_accuracy"], label="Val Acc",
linestyle='-.')
plt.title("Fine-tune Training Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir, "finetune_accuracy_curve.png"))
plt.close()
print("□ Saved: finetune_accuracy_curve.png")

# Full train accuracy (merged)
full_acc = full_history["accuracy"]
full_val_acc = full_history["val_accuracy"]
plt.figure(figsize=(6,5))
plt.plot(full_acc, label="Train Acc", color='black', alpha=0.7)
plt.plot(full_val_acc, label="Val Acc", color='red', alpha=0.7)
plt.title("Full Training Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir, "full_accuracy_curve.png"))
plt.close()
print("□ Saved: full_accuracy_curve.png")

# --- Loss Plots ---
# Head loss
plt.figure(figsize=(6,5))
plt.plot(history_head.history["loss"], label="Train Loss",
linestyle='--')
plt.plot(history_head.history["val_loss"], label="Val Loss",
linestyle='--')
plt.title("Head Training Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir, "head_loss_curve.png"))
plt.close()
print("□ Saved: head_loss_curve.png")

# Fine-tune loss
plt.figure(figsize=(6,5))
plt.plot(history_finetune.history["loss"], label="Train Loss",
linestyle='-.')
plt.plot(history_finetune.history["val_loss"], label="Val Loss",
linestyle='-.')
plt.title("Fine-tune Training Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir, "finetune_loss_curve.png"))
plt.close()
print("□ Saved: finetune_loss_curve.png")

# Full train loss (merged)
full_loss = full_history["loss"]
full_val_loss = full_history["val_loss"]
plt.figure(figsize=(6,5))
plt.plot(full_loss, label="Train Loss", color='black', alpha=0.7)
plt.plot(full_val_loss, label="Val Loss", color='red', alpha=0.7)
plt.title("Full Training Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir, "full_loss_curve.png"))
plt.close()
print("□ Saved: full_loss_curve.png")

# # Merge histories
full_history = {}
for key in set(history_head.history) | set(history_finetune.history):
    full_history[key] = history_head.history.get(key, []) +
history_finetune.history.get(key, [])

# --- Accuracy ---
plt.figure(figsize=(6, 5))
plt.plot(history_head.history["accuracy"], label="Head Train Acc",
linestyle='--')
plt.plot(history_head.history["val_accuracy"], label="Head Val Acc",
linestyle='--')
plt.plot(history_finetune.history["accuracy"], label="Fine-tune Train
Acc", linestyle='-.')
plt.plot(history_finetune.history["val_accuracy"], label="Fine-tune
Val Acc", linestyle='-.')
plt.plot(full_history["accuracy"], label="Full Train Acc",
color='black', alpha=0.3)
plt.plot(full_history["val_accuracy"], label="Full Val Acc",
color='red', alpha=0.3)
plt.title("Learning Curve - Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir,
"cumulative_learning_curve_accuracy.png"))
plt.close()
print("□ Saved: cumulative_learning_curve_accuracy.png")

# --- Loss ---
plt.figure(figsize=(6, 5))
plt.plot(history_head.history["loss"], label="Head Train Loss",
linestyle='--')
plt.plot(history_head.history["val_loss"], label="Head Val Loss",
linestyle='--')
plt.plot(history_finetune.history["loss"], label="Fine-tune Train
Loss", linestyle='-.')
plt.plot(history_finetune.history["val_loss"], label="Fine-tune Val
Loss", linestyle='-.')
plt.plot(full_history["loss"], label="Full Train Loss", color='black',
alpha=0.3)
plt.plot(full_history["val_loss"], label="Full Val Loss", color='red',
alpha=0.3)
plt.title("Learning Curve - Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid(alpha=0.4)
plt.tight_layout()
plt.savefig(os.path.join(save_dir,
"cumulative_learning_curve_loss.png"))
plt.close()
print("□ Saved: cumulative_learning_curve_loss.png")

# ================================
# DONE
# ================================
print("\n□ K-Fold training and evaluation completed.")
print("□ All evaluation results and plots saved in:", save_dir)
```