Answer & Codes : For Task CNNs 2

Task 1: Dataset Loading

Project Description:

- Get the Flowers Multiclass Dataset from Kaggle.
- Unzip and load it.
- If not split, divide into training (70%), validation (10%), and test (20%) sets.

Corresponding Code Section:

```
print("## 1. Dataset Loading & Setup ##")
print("Accessing dataset via kagglehub...")

train_dir = None
validation_dir = None
test_dir = None
class_names = None
num_classes = 0

try:
    # Step 1: Download dataset using Kaggle Hub
    dataset_base_path_from_hub = kagglehub.dataset_download("alsaniipe/flowers-multiclass-dataset
    print(f"Dataset path from kagglehub: {dataset_base_path_from_hub}")

    # Step 2 & 3: Check for pre-split structure (train, validation, test)
    # The script assumes the dataset from Kaggle Hub might already be structured.
    # 'flower_photos' is a common top-level directory in this dataset.
    potential_pre_split_root = os.path.join(dataset_base_path_from_hub, 'flowers', 'flowers', 'flower_pho
    print(f"Checking for pre-split structure in: {potential_pre_split_root}")

    if os.path.isdir(potential_pre_split_root):
        contents = os.listdir(potential_pre_split_root)
        print(f"Contents of {potential_pre_split_root}: {contents}")

        has_train = 'train' in contents and os.path.isdir(os.path.join(potential_pre_split_root, 'train'))
        has_test = 'test' in contents and os.path.isdir(os.path.join(potential_pre_split_root, 'test'))
        has_validation = 'validation' in contents and os.path.isdir(os.path.join(potential_pre_split_root, 'va

        if has_train and has_validation:
            print("SUCCESS: Detected pre-split 'train' and 'validation' directories.")
```

```
        train_dir = os.path.join(potential_pre_split_root, 'train')
        validation_dir = os.path.join(potential_pre_split_root, 'validation')
        if has_test:
            test_dir = os.path.join(potential_pre_split_root, 'test')
            print(f"Test directory found: {test_dir}")
        else:
            # If a test set isn't provided, it won't be used for a separate final evaluation by default.
            # The script later uses the validation set for evaluation if the test set is missing.
            print("Warning: 'test' directory not found in pre-split dataset. Evaluation on a separate test s

        # Determine class names and number of classes from the training directory structure
        if os.path.isdir(train_dir):
            class_names = sorted([d for d in os.listdir(train_dir) if os.path.isdir(os.path.join(train_dir, d))]
            num_classes = len(class_names)
            if num_classes > 0:
                print(f"Found {num_classes} classes in '{train_dir}': {class_names}")

                # Plot some sample images from the training set
                sample_image_paths = []
                sample_labels = []
                # Take samples from up to 2 classes, 4 images per class
                for cls_name in random.sample(class_names, min(len(class_names), 2)): # Corrected line
                    cls_path = os.path.join(train_dir, cls_name)
                    images_in_cls = [os.path.join(cls_path, img) for img in os.listdir(cls_path) if img.lower().
                    sample_image_paths.extend(random.sample(images_in_cls, min(len(images_in_cls), 4))
                    sample_labels.extend([cls_name] * min(len(images_in_cls), 4))
                if sample_image_paths:
                    plot_sample_images(sample_image_paths, sample_labels, title="Sample Training Image
                else:
                    print("Could not find sample images to plot.")
            else:
                print(f"Error: The 'train' directory '{train_dir}' is empty or does not contain class subdirect
                exit()
        else:
            print(f"Error: Identified 'train' directory '{train_dir}' is not a valid directory.")
            exit()
    else:
        # This handles the case where the expected 'train' and 'validation' subdirectories are missing.
        # Your project asks to split if not pre-split. This script currently exits if they aren't found.
        # To implement manual splitting, you'd list all image files per class from a single root,
        # then use sklearn.model_selection.train_test_split twice to create train, val, and test lists,
        # and then organize these files into new directories or use custom data loaders.
        print(f"CRITICAL Error: Dataset at {potential_pre_split_root} does not have 'train' and 'validatio
        print("If the dataset is not pre-split, you would need to implement manual splitting here.")
        exit()
```

```
    else:
        print(f"Error: The expected base directory for pre-split data '{potential_pre_split_root}' was not fo
        exit()
except Exception as e:
    print(f"General error during dataset loading and setup: {e}")
    exit()

if not train_dir or not validation_dir or num_classes == 0:
    print("CRITICAL ERROR: train_dir, validation_dir, or class discovery failed. Cannot proceed.")
    exit()

print("\nUsing pre-split dataset directories:")
print(f"Training directory: {train_dir}")
print(f"Validation directory: {validation_dir}")
if test_dir: print(f"Test directory: {test_dir}")
print(f"Number of classes: {num_classes}")
print("---")

# Helper function used in this section:
# def plot_sample_images(...): ...
```

Explanation:

1. **Kaggle Hub Download:** `kagglehub.dataset_download("alsaniipe/flowers-multiclass-datasets")` directly fetches the dataset.
2. **Path Setup & Pre-split Check:** The script then constructs a path (`potential_pre_split_root`) where it expects the class folders or `train` / `validation` / `test` subfolders. It explicitly checks for `train`, `validation`, and optionally `test` directories.
    - **Your Task 1, Step 3 Note:** This script *relies* on the dataset being pre-split into `train`, `validation` (and optionally `test`) directories within the downloaded structure. If the Kaggle dataset `alsaniipe/flowers-multiclass-datasets` doesn't have this structure, the script will exit. To fulfill the "divide if not split" requirement, you would need to add logic to:
        1. Identify a single directory containing all class folders.
        2. Collect all image paths and their corresponding labels.
        3. Use `sklearn.model_selection.train_test_split` (or similar) to divide these paths into training, validation, and test sets.
        4. Either move/copy files to new `train/val/test` directories or adapt the `ImageDataGenerator` to work with these lists of file paths directly (which is more complex with `flow_from_directory`). The current script assumes the simpler case of pre-split directories.
3. **Class Discovery:** It determines `class_names` and `num_classes` by listing subdirectories in the `train_dir`.
4. **Sample Images:** The `plot_sample_images` function is called to display a few images from the training set, giving a visual confirmation of the data.

Task 2: Image Preprocessing

Project Description:

- Resize images to 224×224.
- Normalize pixel values to [0,1].
- Choose label format (one-hot encoding or label encoding) and match with loss function.

Corresponding Code Section (primarily within `ImageDataGenerator` setup):

```
# Constants defined at the beginning of the script:
IMG_WIDTH, IMG_HEIGHT = 224, 224 # For resizing

# ... (Dataset Loading code) ...

print("\n## 2. Image Preprocessing & 3. Data Augmentation ##")

# train_datagen handles preprocessing for training data
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Task 2: Normalize pixel values to [0,1]
    # ... (Augmentation parameters for Task 3 are also here)
    rotation_range=20,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    brightness_range=[0.8, 1.2],
    fill_mode='nearest'
)

# validation_datagen handles preprocessing for validation data (only normalization)
validation_datagen = ImageDataGenerator(rescale=1./255) # Task 2: Normalize

if test_dir:
    # test_datagen handles preprocessing for test data (only normalization)
    test_datagen = ImageDataGenerator(rescale=1./255) # Task 2: Normalize

# ... (Visualization of augmentation) ...

# train_generator applies the preprocessing (and augmentation)
train_generator = train_datagen.flow_from_directory(
    train_dir,
```

```
    target_size=(IMG_WIDTH, IMG_HEIGHT), # Task 2: Resize images
    batch_size=BATCH_SIZE,
    class_mode='categorical', # Task 2: Labels - One-hot encoding
    shuffle=True
)

# validation_generator applies the preprocessing
validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(IMG_WIDTH, IMG_HEIGHT), # Task 2: Resize images
    batch_size=BATCH_SIZE,
    class_mode='categorical', # Task 2: Labels - One-hot encoding
    shuffle=False # Important: no shuffle for validation/test
)

if test_dir:
    test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(IMG_WIDTH, IMG_HEIGHT), # Task 2: Resize images
        batch_size=BATCH_SIZE,
        class_mode='categorical', # Task 2: Labels - One-hot encoding
        shuffle=False
    )
else:
    test_generator = None # Explicitly set to None if no test_dir

print(f"\nClass indices from training generator: {train_generator.class_indices}")
# ... (Generator summary printouts) ...
```

Explanation:

1. **Resize:** The `target_size=(IMG_WIDTH, IMG_HEIGHT)` argument (where `IMG_WIDTH, IMG_HEIGHT = 224, 224`) in `flow_from_directory` for all generators ensures all images are resized to 224×224 pixels.
2. **Normalize:** The `rescale=1./255` argument in `ImageDataGenerator` scales pixel values from the [0, 255] range to the [0, 1] range. This is applied to training, validation, and test data.
3. **Labels:**
    - `class_mode='categorical'` in `flow_from_directory` tells the generator to convert class labels (derived from subdirectory names) into **one-hot encoded vectors**.
    - The corresponding loss function used later in model compilation is `categorical_crossentropy`, which is appropriate for one-hot encoded labels.
    - The script does not experiment with label encoding (`class_mode='sparse'`). If you were to use label encoding, you'd typically use `sparse_categorical_crossentropy` as the loss function.

Task 3: Data Augmentation

Project Description:

- Add variety to the training set only.
- Suggested transformations: random rotations, horizontal flips, random crops/shifts, brightness/contrast changes.

Corresponding Code Section:

```
print("\n## 2. Image Preprocessing & 3. Data Augmentation ##")

# train_datagen includes augmentation parameters
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,          # Random rotations between -20 and 20 degrees
    width_shift_range=0.1,      # Random horizontal shifts (10% of width)
    height_shift_range=0.1,     # Random vertical shifts (10% of height)
    shear_range=0.1,            # Shear transformations
    zoom_range=0.1,             # Random zoom (up to 10%)
    horizontal_flip=True,       # Horizontal flips (50% chance)
    brightness_range=[0.8, 1.2],  # Slight brightness changes
    fill_mode='nearest'         # How to fill newly created pixels
)

# Validation and test data generators DO NOT have augmentation
validation_datagen = ImageDataGenerator(rescale=1./255)
if test_dir:
    test_datagen = ImageDataGenerator(rescale=1./255)

print("\nVisualizing Data Augmentation Effects...")
if sample_image_paths: # Use one of the previously sampled images
    plot_augmented_images(train_datagen, sample_image_paths[0], n_augmentations=7)
else:
    # Fallback if sample_image_paths was empty for some reason
    try:
        first_class_path = os.path.join(train_dir, class_names[0])
        any_image_name = [img for img in os.listdir(first_class_path) if img.lower().endswith(('.png', '.jpg'
        any_image_path = os.path.join(first_class_path, any_image_name)
        plot_augmented_images(train_datagen, any_image_path, n_augmentations=7)
    except Exception as e:
        print(f"Could not visualize augmentation, no sample image found: {e}")

# train_generator will apply these augmentations on the fly
train_generator = train_datagen.flow_from_directory(
```

```
    train_dir,
    target_size=(IMG_WIDTH, IMG_HEIGHT),
    batch_size=BATCH_SIZE,
    class_mode='categorical',
    shuffle=True # Shuffle is important for training data
)

# Helper function used in this section:
# def plot_augmented_images(datagen, original_image_path, ...): ...
```

Explanation:

1. **Augmentation Parameters:** The `ImageDataGenerator` for `train_datagen` is initialized with several parameters to perform data augmentation:
   - `rotation_range=20`: Rotates images randomly by up to 20 degrees.
   - `width_shift_range=0.1` and `height_shift_range=0.1`: Shifts images horizontally or vertically by up to 10% of their dimension.
   - `shear_range=0.1`: Applies shear transformations.
   - `zoom_range=0.1`: Randomly zooms into images.
   - `horizontal_flip=True`: Randomly flips images horizontally.
   - `brightness_range=[0.8, 1.2]`: Randomly alters image brightness.
   - `fill_mode='nearest'`: Specifies how to fill in pixels that might be created after a rotation or shift.
2. **Training Set Only:** Crucially, these augmentation parameters are **only** applied to `train_datagen`. The `validation_datagen` and `test_datagen` only perform rescaling. This is correct practice, as augmentation should not be applied to validation or test sets.
3. **Visualization:** The `plot_augmented_images` function is called to take an original image and display several augmented versions of it, demonstrating the effect of the `train_datagen` transformations.
4. **Applied by Generator:** The `train_generator` then uses this `train_datagen` to load and augment images in batches during model training.

Task 4: Building a VGG CNN from Scratch

Project Description:

- Create a VGG-style CNN (stacked 3×3 conv, max-pooling, fully connected).
- Define, train on augmented data, and evaluate on validation.

Corresponding Code Section:

```

# Helper function for plotting training history (used here and for ResNet)
# def plot_training_history(history, title_prefix): ...
```

```
print("\n## 4. Building a VGG CNN from Scratch ##")
def build_vgg_style_model(input_shape, num_classes_vgg): # num_classes_vgg will be num_classes
    model = Sequential([
        Input(shape=input_shape), # Explicit Input layer
        # Block 1
        Conv2D(32, (3,3), activation='relu', padding='same'),
        Conv2D(32, (3,3), activation='relu', padding='same'),
        MaxPooling2D((2,2)),
        # Block 2
        Conv2D(64, (3,3), activation='relu', padding='same'),
        Conv2D(64, (3,3), activation='relu', padding='same'),
        MaxPooling2D((2,2)),
        # Block 3
        Conv2D(128, (3,3), activation='relu', padding='same'),
        Conv2D(128, (3,3), activation='relu', padding='same'),
        MaxPooling2D((2,2)),
        # Classifier
        Flatten(),
        Dense(512, activation='relu'),
        Dropout(0.5), # Regularization
        Dense(num_classes_vgg, activation='softmax') # Output layer for multi-class classification
    ])
    return model

# Build the VGG-style model
vgg_model = build_vgg_style_model((IMG_WIDTH, IMG_HEIGHT, 3), num_classes)

# Compile the model
vgg_model.compile(optimizer=Adam(learning_rate=LEARNING_RATE_VGG),
                  loss='categorical_crossentropy', # Matches one-hot labels
                  metrics=['accuracy'])
vgg_model.summary() # Display model architecture

# Callbacks for training
early_stopping_vgg = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True, verb
model_checkpoint_vgg = ModelCheckpoint('best_vgg_model.keras', save_best_only=True, monitor='v

val_acc_vgg, val_loss_vgg = 0.0, float('inf') # Initialize for storing best results
if validation_generator.samples > 0 and train_generator.samples > 0 :
    print("\nTraining VGG-style model...")
    # Train the model
    history_vgg = vgg_model.fit(
        train_generator,
        epochs=EPOCHS_VGG, # e.g., 50
```

```
        validation_data=validation_generator,
        callbacks=[early_stopping_vgg, model_checkpoint_vgg]
    )
    # Plot training history
    plot_training_history(history_vgg, "VGG-Style CNN")

    # Evaluate on the validation set (using best weights restored by ModelCheckpoint or EarlyStopping
    print("Loading best weights for VGG model for evaluation...")
    if os.path.exists('best_vgg_model.keras'):
        vgg_model.load_weights('best_vgg_model.keras') # Load the best saved model
    else:
        print("Warning: best_vgg_model.keras not found. Using model from end of training.")
    # Perform evaluation on the validation set
    val_loss_vgg, val_acc_vgg = vgg_model.evaluate(validation_generator)
    print(f"VGG Model - Final Validation Accuracy: {val_acc_vgg:.4f}, Validation Loss: {val_loss_vgg:.4f}
else:
    print("Skipping VGG training as training or validation set is empty.")
print("---")
```

Explanation:

1. **Model Definition (`build_vgg_style_model`):**
    - A `Sequential` model is created.
    - It follows the VGG pattern:
        - Multiple blocks of two `Conv2D` layers (3×3 filters, 'relu' activation, 'same' padding) followed by a `MaxPooling2D` layer. The filter counts increase (32, 64, 128).
        - A `Flatten` layer to prepare for the dense layers.
        - A `Dense` layer (512 units, 'relu') with `Dropout` for regularization.
        - An output `Dense` layer with `num_classes` units and 'softmax' activation for multi-class probability outputs.
2. **Compilation:**
    - `optimizer=Adam(learning_rate=LEARNING_RATE_VGG)`: Uses the Adam optimizer with a specified learning rate.
    - `loss='categorical_crossentropy'`: Appropriate for the one-hot encoded labels.
    - `metrics=['accuracy']`: Tracks accuracy during training.
3. **Training (`vgg_model.fit`):**
    - The model is trained using `train_generator` (augmented data) and validated on `validation_generator`.
    - `EPOCHS_VGG` defines the maximum number of training epochs.
    - **Callbacks:**

- `EarlyStopping`: Monitors `val_loss` and stops training if it doesn't improve for `patience` epochs, restoring the weights from the best epoch. This helps prevent overfitting (as mentioned in your "Additional Notes").
- `ModelCheckpoint`: Saves the model weights only when `val_loss` improves, ensuring the `best_vgg_model.keras` file contains the best performing model on the validation set.

4. **Evaluation:**
   - After training (or if EarlyStopping stops it), the script loads the best saved weights from `best_vgg_model.keras`.
   - `vgg_model.evaluate(validation_generator)` calculates the loss and accuracy on the validation set.
   - `plot_training_history` visualizes the training and validation accuracy/loss over epochs.

Task 5: Fine-Tuning a Pretrained ResNet50

Project Description:

- Load ResNet50 pretrained on ImageNet.
- Stage 1: Freeze base, replace classifier, train head.
- Stage 2: Unfreeze last convolutional block, train with classifier.
- Stage 3: Unfreeze all layers, train whole network.
- Evaluate each method on validation.

Corresponding Code Section:

```
# ... (Constants like EPOCHS_RESNET_HEAD, LEARNING_RATE_RESNET_HEAD, etc.) ...

print("\n## 5. Fine-Tuning a Pretrained ResNet50 ##")
# Initialize metrics for each stage
val_acc_resnet_s1, val_loss_resnet_s1 = 0.0, float('inf')
val_acc_resnet_s2, val_loss_resnet_s2 = 0.0, float('inf')
val_acc_resnet_s3, val_loss_resnet_s3 = 0.0, float('inf')

if validation_generator.samples > 0 and train_generator.samples > 0 :
    print("\n### ResNet - Stage 1: Freeze Base, Train Head ###")
    # Load ResNet50 base model (without top classification layer)
    base_model_resnet_s1 = ResNet50(weights='imagenet', include_top=False, input_shape=(IMG_WID
    base_model_resnet_s1.trainable = False # Freeze all layers in the base model

    # Build the new model on top of the frozen base
    inputs_s1 = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))
    x_s1 = base_model_resnet_s1(inputs_s1, training=False) # training=False as base is frozen
    x_s1 = GlobalAveragePooling2D()(x_s1) # Pool features
    x_s1 = Dense(256, activation='relu')(x_s1) # New dense layer
```

```
x_s1 = Dropout(0.5)(x_s1) # Regularization
outputs_s1 = Dense(num_classes, activation='softmax')(x_s1) # New output layer
resnet_model_s1 = Model(inputs_s1, outputs_s1)

resnet_model_s1.compile(optimizer=Adam(learning_rate=LEARNING_RATE_RESNET_HEAD),
                        loss='categorical_crossentropy', metrics=['accuracy'])
resnet_model_s1.summary()
history_resnet_s1 = resnet_model_s1.fit(
    train_generator, epochs=EPOCHS_RESNET_HEAD, validation_data=validation_generator,
    callbacks=[EarlyStopping(monitor='val_loss', patience=7, restore_best_weights=True, verbose=1
               ModelCheckpoint('best_resnet_s1_model.keras', save_best_only=True, monitor='val_loss')
plot_training_history(history_resnet_s1, "ResNet S1 (Head Trained)")
if os.path.exists('best_resnet_s1_model.keras'): resnet_model_s1.load_weights('best_resnet_s1_moc
val_loss_resnet_s1, val_acc_resnet_s1 = resnet_model_s1.evaluate(validation_generator)
print(f"ResNet Model (Stage 1) - Val Acc: {val_acc_resnet_s1:.4f}, Val Loss: {val_loss_resnet_s1:.4f}

print("\n### ResNet - Stage 2: Unfreeze Last Conv Block ###")
if os.path.exists('best_resnet_s1_model.keras'):
    # It seems there's a slight mix-up in variable reuse vs loading a fresh base model for stage 2.
    # The intention is to continue from the trained head of stage 1.
    # A cleaner way for stage 2 is to take resnet_model_s1 (which has the trained head)
    # and make its base_model part partially trainable.

    # The script currently reloads best_resnet_s1_model.keras into resnet_model_s1 (implicitly if alrea
    # Then it sets trainability on the base_model_resnet_s1 which is part of resnet_model_s1.
    # Let's trace the script's logic:
    # resnet_model_s1 already exists and has its weights loaded from best_resnet_s1_model.keras

    # The script intends to modify the trainability of layers within base_model_resnet_s1 which is *pa
    base_model_to_modify = None
    for layer in resnet_model_s1.layers:
        if layer.name == base_model_resnet_s1.name: # Find the ResNet50 base layer within resnet_m
            base_model_to_modify = layer
            break

    if base_model_to_modify:
        base_model_to_modify.trainable = True # Allow layers within the base to be trainable

        fine_tune_at_layer_name = 'conv5_block1_1_conv' # Start of the last major block in ResNet50
        try:
            # Find the index of this layer *within the ResNet50 base model's layers*
            fine_tune_from_index = [l.name for l in base_model_to_modify.layers].index(fine_tune_at_lay
            for layer_in_base in base_model_to_modify.layers[:fine_tune_from_index]:
                layer_in_base.trainable = False
            print(f"Unfreezing ResNet base from layer: {fine_tune_at_layer_name} (index {fine_tune_fror
```

```
        except ValueError:
            print(f"Warning: Layer {fine_tune_at_layer_name} not found. Unfreezing top ~30 layers of Re
            # Fallback: unfreeze approximately the last block. ResNet50 has many layers.
            # A common strategy is to unfreeze layers from 'conv5_block1_out' or similar.
            # This fallback unfreezes more than just one layer, effectively a block.
            num_layers_in_base = len(base_model_to_modify.layers)
            for i, layer_in_base in enumerate(base_model_to_modify.layers):
                if i < num_layers_in_base - 30 : # Arbitrary number, usually related to a block
                    layer_in_base.trainable = False
                else:
                    layer_in_base.trainable = True
    else:
        print("Error: Could not find the ResNet base model layer in resnet_model_s1 for stage 2.")
        # Skip stage 2 and 3 if base model isn't found

    # Re-compile the *entire* resnet_model_s1 with the new trainability settings and a lower learning
    resnet_model_s1.compile(optimizer=Adam(learning_rate=LEARNING_RATE_RESNET_BLOCK), # L
                            loss='categorical_crossentropy', metrics=['accuracy'])
    resnet_model_s2_final = resnet_model_s1 # Using the modified resnet_model_s1
    resnet_model_s2_final.summary() # Will show updated trainable params
    history_resnet_s2 = resnet_model_s2_final.fit(
        train_generator, epochs=EPOCHS_RESNET_FINE_TUNE_BLOCK, validation_data=validation_ge
        callbacks=[EarlyStopping(monitor='val_loss', patience=7, restore_best_weights=True, verbose
                   ModelCheckpoint('best_resnet_s2_model.keras', save_best_only=True, monitor='val_los
    plot_training_history(history_resnet_s2, "ResNet S2 (Block Fine-tuned)")
    if os.path.exists('best_resnet_s2_model.keras'): resnet_model_s2_final.load_weights('best_resne
    val_loss_resnet_s2, val_acc_resnet_s2 = resnet_model_s2_final.evaluate(validation_generator)
    print(f"ResNet Model (Stage 2) - Val Acc: {val_acc_resnet_s2:.4f}, Val Loss: {val_loss_resnet_s2:

    print("\n### ResNet - Stage 3: Unfreeze All Layers ###")
    # Continue with resnet_model_s2_final (which is resnet_model_s1 with some layers unfozen)
    base_model_to_modify_s3 = None
    for layer in resnet_model_s2_final.layers:
        if layer.name == base_model_resnet_s1.name: # Find the ResNet50 base layer again
            base_model_to_modify_s3 = layer
            break

    if base_model_to_modify_s3:
        # Unfreeze all layers within the found base model
        for layer_in_base in base_model_to_modify_s3.layers:
            layer_in_base.trainable = True
        print(f"Unfreezing all layers in ResNet base: {base_model_to_modify_s3.name}")
    else:
         print("Error: Could not find the ResNet base model layer in resnet_model_s2_final for stage 3."
         # Skip stage 3 if base model isn't found
```

```
        resnet_model_s2_final.compile(optimizer=Adam(learning_rate=LEARNING_RATE_RESNET_ALL), #
                      loss='categorical_crossentropy', metrics=['accuracy'])
        resnet_model_s3_final = resnet_model_s2_final # Continue modifying the same model instance
        resnet_model_s3_final.summary()
        history_resnet_s3 = resnet_model_s3_final.fit(
          train_generator, epochs=EPOCHS_RESNET_FINE_TUNE_ALL, validation_data=validation_gener
          callbacks=[EarlyStopping(monitor='val_loss', patience=7, restore_best_weights=True, verbose
                ModelCheckpoint('best_resnet_s3_model.keras', save_best_only=True, monitor='val_los
        plot_training_history(history_resnet_s3, "ResNet S3 (All Fine-tuned)")
        if os.path.exists('best_resnet_s3_model.keras'): resnet_model_s3_final.load_weights('best_resne
        val_loss_resnet_s3, val_acc_resnet_s3 = resnet_model_s3_final.evaluate(validation_generator)
        print(f"ResNet Model (Stage 3) - Val Acc: {val_acc_resnet_s3:.4f}, Val Loss: {val_loss_resnet_s3:
    else:
        print("Skipping ResNet Stage 2 and 3 as Stage 1 model checkpoint ('best_resnet_s1_model.keras
else:
    print("Skipping ResNet training as training or validation set is empty.")
print("---")
```

Explanation:

- **Stage 1: Freeze Base, Train Head**
  1. `ResNet50(weights='imagenet', include_top=False, ...)` loads the ResNet50 architecture with weights pretrained on ImageNet, excluding its final classification layer.
  2. `base_model_resnet_s1.trainable = False` freezes all weights in the convolutional base.
  3. A new classification head is added: `GlobalAveragePooling2D`, a `Dense` layer (256 units, relu), `Dropout`, and the final `Dense` output layer (softmax).
  4. This new model (`resnet_model_s1`) is compiled (with `LEARNING_RATE_RESNET_HEAD`) and trained. Only the weights of the new head are updated.
  5. Evaluated on the validation set and best model saved as `best_resnet_s1_model.keras`.
- **Stage 2: Unfreeze Last Conv Block**
  1. The script intends to continue from `resnet_model_s1` (with its trained head).
  2. It locates the `ResNet50` base model *within* `resnet_model_s1`.
  3. `base_model_to_modify.trainable = True` makes the layers *within* this base model potentially trainable.
  4. It then selectively makes layers *before* `fine_tune_at_layer_name` (e.g., 'conv5_block1_1_conv', which is the start of the 5th convolutional block) non-trainable. This means 'conv5_block1_1_conv' and subsequent layers in the base model *are* trainable, along with the head.
  5. The *entire* `resnet_model_s1` is re-compiled with a lower learning rate (`LEARNING_RATE_RESNET_BLOCK`).
  6. Training continues, fine-tuning the weights of the unfozen block(s) and the head.
  7. Evaluated and best model saved as `best_resnet_s2_model.keras`.

- **Stage 3: Unfreeze All Layers**
  1. Continuing from the model obtained in Stage 2 (`resnet_model_s2_final`).
  2. All layers within its `ResNet50` base are made trainable (`layer_in_base.trainable = True`).
  3. The model is re-compiled with an even lower learning rate (`LEARNING_RATE_RESNET_ALL`).
  4. The entire network is trained for a few more epochs.
  5. Evaluated and best model saved as `best_resnet_s3_model.keras`.
- **Common Practices:**
  - **Decreasing Learning Rates:** Using a smaller learning rate for fine-tuning later stages is crucial to avoid corrupting the pretrained features too quickly.
  - **Callbacks:** `EarlyStopping` and `ModelCheckpoint` are used in each stage.
  - **Epochs:** The script uses `EPOCHS_RESNET_HEAD`, `EPOCHS_RESNET_FINE_TUNE_BLOCK`, `EPOCHS_RESNET_FINE_TUNE_ALL` for each stage. These are set to 15 in your script, which is a bit more than the suggested 5-10, but `EarlyStopping` will likely prevent excessive training.

Task 6: Result Comparison

Project Description:

- Compare VGG and ResNet.
- Calculate metrics: accuracy, precision, recall, F1-score.
- Plot confusion matrices and ROC curves. Calculate AUC.

Corresponding Code Section:

```
print("\n## 6. Result Comparison ##")
# Determine best VGG model path
best_vgg_model_path = 'best_vgg_model.keras'

# Determine best ResNet model path based on validation accuracy from different stages
best_resnet_model_path = None
resnet_checkpoints = {
    's1': ('best_resnet_s1_model.keras', val_acc_resnet_s1),
    's2': ('best_resnet_s2_model.keras', val_acc_resnet_s2),
    's3': ('best_resnet_s3_model.keras', val_acc_resnet_s3)
}
max_val_acc_resnet = -1.0
best_resnet_stage_name = ""
for stage_name, (path, acc) in resnet_checkpoints.items():
    if os.path.exists(path) and acc > max_val_acc_resnet:
        max_val_acc_resnet = acc
        best_resnet_model_path = path
```

```
        best_resnet_stage_name = stage_name # To identify which ResNet stage was best

print(f"Best VGG model path: {best_vgg_model_path if os.path.exists(best_vgg_model_path) else 'Not
print(f"Best ResNet model path (overall from stage '{best_resnet_stage_name}' with val_acc {max_val

# Helper function for plotting evaluation metrics
def plot_evaluation_metrics(model, model_name, generator, y_true_labels):
    if generator is None or generator.samples == 0:
        print(f"Skipping evaluation for {model_name} as its generator is empty or None.")
        return None, None, None # Return tuple of Nones

    print(f"\nEvaluating {model_name} on its generator ({generator.n} samples)...")
    # Basic accuracy and loss
    loss, accuracy = model.evaluate(generator, verbose=0) # generator.n is number of samples
    print(f"{model_name} - Accuracy: {accuracy:.4f}, Loss: {loss:.4f}")

    # Predictions for detailed metrics
    y_pred_proba = model.predict(generator) # Probabilities
    y_pred_int = np.argmax(y_pred_proba, axis=1) # Integer predicted labels

    # Classification Report (Precision, Recall, F1-score)
    print(f"\n{model_name} - Classification Report:")
    # Ensure class names are sorted according to generator's class indices
    report_class_names = [name for name, index in sorted(generator.class_indices.items(), key=lambda
    print(classification_report(y_true_labels, y_pred_int, target_names=report_class_names, zero_divisi

    # Confusion Matrix
    cm = confusion_matrix(y_true_labels, y_pred_int)
    plt.figure(figsize=(max(8, num_classes), max(6, int(num_classes*0.8)))) # Dynamic figure size
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=report_class_names, yticklabels=report_class_names)
    plt.title(f'Confusion Matrix - {model_name}')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()

    # ROC Curves and AUC Scores
    y_true_binarized = label_binarize(y_true_labels, classes=list(range(num_classes)))
    fpr, tpr, roc_auc_scores = dict(), dict(), dict()

    for i in range(num_classes):
        fpr[i], tpr[i], _ = roc_curve(y_true_binarized[:, i], y_pred_proba[:, i])
        roc_auc_scores[i] = auc(fpr[i], tpr[i])
    # Micro-average ROC
    fpr["micro"], tpr["micro"], _ = roc_curve(y_true_binarized.ravel(), y_pred_proba.ravel())
```

```
    roc_auc_scores["micro"] = auc(fpr["micro"], tpr["micro"])
    # Macro-average ROC
    all_fpr = np.unique(np.concatenate([fpr[i] for i in range(num_classes)]))
    mean_tpr = np.zeros_like(all_fpr)
    for i in range(num_classes): mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])
    mean_tpr /= num_classes
    fpr["macro"] = all_fpr; tpr["macro"] = mean_tpr; roc_auc_scores["macro"] = auc(fpr["macro"], tpr["

    plt.figure(figsize=(max(10, num_classes), max(8, int(num_classes*0.8)))) # Dynamic figure size
    plt.plot(fpr["micro"], tpr["micro"], label=f'Micro-avg ROC (area={roc_auc_scores["micro"]:.2f})', co
    plt.plot(fpr["macro"], tpr["macro"], label=f'Macro-avg ROC (area={roc_auc_scores["macro"]:.2f})',
    colors = cycle(['aqua','darkorange','cornflowerblue','green','red','purple','brown','olive','gray','pink'])
    for i, color in zip(range(num_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=2, label=f'ROC {report_class_names[i]} (area={roc_auc_scor
    plt.plot([0,1],[0,1],'k--',lw=2); plt.xlim([0,1]); plt.ylim([0,1.05]); plt.xlabel('FPR'); plt.ylabel('TPR')
    plt.title(f'ROC Curves - {model_name}'); plt.legend(loc="lower right", prop={'size':8}); plt.show()

    print(f"\n{model_name} AUC Scores:")
    for i in range(num_classes): print(f"  AUC for class '{report_class_names[i]}': {roc_auc_scores[i]:.4f
    print(f"  Micro-average AUC: {roc_auc_scores['micro']:.4f}\n  Macro-average AUC: {roc_auc_score
    print("---")
    return y_pred_proba, y_pred_int, accuracy # Return predictions and accuracy


# Determine which generator to use for final evaluation (test set if available, otherwise validation)
eval_generator = validation_generator
eval_generator_name = "Validation"
if test_generator and test_generator.samples > 0:
    print("\nUsing TEST SET for final model comparison plots.")
    eval_generator = test_generator
    eval_generator_name = "Test"
elif validation_generator and validation_generator.samples > 0:
    print("\nUsing VALIDATION SET for final model comparison plots (Test set not available or empty)."
else:
    print("\nNo suitable generator for final evaluation. Skipping comparison plots.")
    eval_generator = None # Explicitly set to None

y_true_final = None
if eval_generator:
    y_true_final = eval_generator.classes # Get true labels from the generator

# Evaluate VGG Model
test_acc_vgg = None # Initialize
if os.path.exists(best_vgg_model_path) and eval_generator:
    print(f"\n--- VGG Model Evaluation ({eval_generator_name} Set) ---")
```

```
    loaded_vgg_model = tf.keras.models.load_model(best_vgg_model_path)
    _, _, test_acc_vgg = plot_evaluation_metrics(loaded_vgg_model, "VGG Model", eval_generator, y_tru
else:
    print("\nSkipping VGG Model final evaluation (model or generator not available).")

# Evaluate Best ResNet Model
test_acc_resnet = None # Initialize
if best_resnet_model_path and os.path.exists(best_resnet_model_path) and eval_generator:
    print(f"\n--- Best ResNet Model Evaluation ({eval_generator_name} Set) ---")
    loaded_resnet_model = tf.keras.models.load_model(best_resnet_model_path)
    _, _, test_acc_resnet = plot_evaluation_metrics(loaded_resnet_model, f"Best ResNet ({os.path.base
else:
    print("\nSkipping Best ResNet Model final evaluation (model or generator not available).")

print("\n--- Task Completed (Training and Local Evaluation) ---")
```

Explanation:

1. **Select Best Models:**
   - The path to the best VGG model (`best_vgg_model.keras`) is directly used.
   - For ResNet, it iterates through the saved checkpoints (`best_resnet_s1_model.keras`, etc.) and their corresponding validation accuracies achieved during training (`val_acc_resnet_s1`, etc.) to find the ResNet model that performed best on the validation set. This overall best ResNet model path is stored in `best_resnet_model_path`.
2. **Select Evaluation Set:**
   - It prioritizes the `test_generator` for evaluation if it exists and has samples.
   - If the test set is not available or empty, it falls back to using the `validation_generator`.
   - The true labels (`y_true_final`) are extracted from the chosen generator's `.classes` attribute.
3. `plot_evaluation_metrics` **Function:** This comprehensive function does the following for a given model and generator:
   - **Basic Evaluation:** Calculates and prints overall accuracy and loss using `model.evaluate()`.
   - **Predictions:** Gets predicted probabilities (`y_pred_proba`) and integer class predictions (`y_pred_int`).
   - **Classification Report:** Uses `sklearn.metrics.classification_report` to display precision, recall, F1-score, and support for each class.
   - **Confusion Matrix:** Uses `sklearn.metrics.confusion_matrix` and `seaborn.heatmap` to plot the confusion matrix.
   - **ROC Curves and AUC:**
     - Binarizes the true labels using `sklearn.preprocessing.label_binarize`.
     - Calculates False Positive Rate (FPR), True Positive Rate (TPR), and AUC for each class using `sklearn.metrics.roc_curve` and `sklearn.metrics.auc`.

- Calculates micro-average and macro-average ROC curves and AUCs.
- Plots all these ROC curves using `matplotlib.pyplot`.
- Prints the AUC scores for each class and the micro/macro averages.

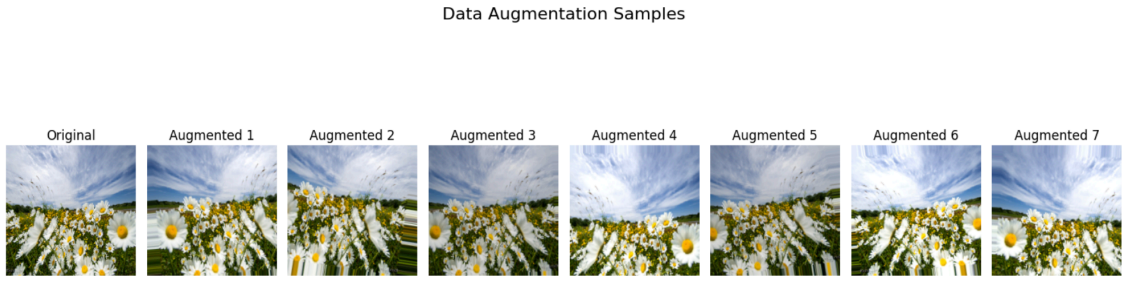
4. **Model Evaluation:**
   - The `plot_evaluation_metrics` function is called for the best VGG model and the overall best ResNet model, using the selected evaluation generator (`eval_generator`). This provides a side-by-side comparison of their performance on the same unseen (or validation) data.

Results :

```
## 1. Dataset Loading & Setup ##
Accessing dataset via kagglehub...
Dataset path from kagglehub: /kaggle/input/flowers-multiclass-datasets
Checking for pre-split structure in: /kaggle/input/flowers-multiclass-datasets/flowers/flowers/flower_photos
Contents of /kaggle/input/flowers-multiclass-datasets/flowers/flowers/flower_photos: ['validation', 'test', 'train']
SUCCESS: Detected pre-split 'train' and 'validation' directories.
Test directory found: /kaggle/input/flowers-multiclass-datasets/flowers/flowers/flower_photos/test
Found 5 classes in '/kaggle/input/flowers-multiclass-datasets/flowers/flowers/flower_photos/train': ['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```


```
Using pre-split dataset directories:
Training directory: /kaggle/input/flowers-multiclass-datasets/flowers/flowers/flower_photos/train
Validation directory: /kaggle/input/flowers-multiclass-datasets/flowers/flowers/flower_photos/validation
Test directory: /kaggle/input/flowers-multiclass-datasets/flowers/flowers/flower_photos/test
Number of classes: 5
---
```

```
## 2. Image Preprocessing & 3. Data Augmentation ##

Visualizing Data Augmentation Effects...
```




```
Found 3540 images belonging to 5 classes.
Found 80 images belonging to 5 classes.
Found 50 images belonging to 5 classes.

Class indices from training generator: {'daisy': 0, 'dandelion': 1, 'roses': 2, 'sunflowers': 3, 'tulips': 4}
Training generator found 3540 images belonging to 5 classes.
Validation generator found 80 images belonging to 5 classes.
Test generator found 50 images belonging to 5 classes.
```

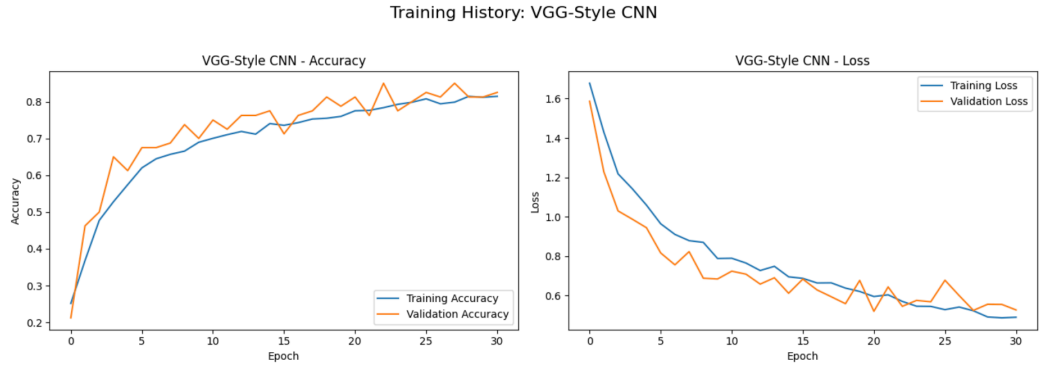
4. Building a VGG CNN from Scratch

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 224, 224, 32)	896
conv2d_1 (Conv2D)	(None, 224, 224, 32)	9,248
max_pooling2d (MaxPooling2D)	(None, 112, 112, 32)	0
conv2d_2 (Conv2D)	(None, 112, 112, 64)	18,496
conv2d_3 (Conv2D)	(None, 112, 112, 64)	36,928
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 64)	0
conv2d_4 (Conv2D)	(None, 56, 56, 128)	73,856
conv2d_5 (Conv2D)	(None, 56, 56, 128)	147,584
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 128)	0
flatten (Flatten)	(None, 100352)	0
dense (Dense)	(None, 512)	51,380,736
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 5)	2,565

Total params: 51,670,309 (197.11 MB)
Trainable params: 51,670,309 (197.11 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 118s 2s/step - accuracy: 0.2327 - loss: 1.8424 - val_accuracy: 0.2125 - val_loss: 1.5865
Epoch 2/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 99s 2s/step - accuracy: 0.3354 - loss: 1.4808 - val_accuracy: 0.4625 - val_loss: 1.2280
Epoch 3/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.4717 - loss: 1.2267 - val_accuracy: 0.5000 - val_loss: 1.0296
Epoch 4/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 78s 2s/step - accuracy: 0.5269 - loss: 1.1216 - val_accuracy: 0.6500 - val_loss: 0.9877
Epoch 5/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 84s 2s/step - accuracy: 0.5653 - loss: 1.0813 - val_accuracy: 0.6125 - val_loss: 0.9442
Epoch 6/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 86s 2s/step - accuracy: 0.6125 - loss: 0.9898 - val_accuracy: 0.6750 - val_loss: 0.8161
Epoch 7/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 74s 2s/step - accuracy: 0.6305 - loss: 0.9356 - val_accuracy: 0.6750 - val_loss: 0.7557
Epoch 8/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 61s 2s/step - accuracy: 0.6672 - loss: 0.8728 - val_accuracy: 0.6875 - val_loss: 0.8228
Epoch 9/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 99s 2s/step - accuracy: 0.6729 - loss: 0.8551 - val_accuracy: 0.7375 - val_loss: 0.6878
Epoch 10/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 81s 2s/step - accuracy: 0.6908 - loss: 0.7846 - val_accuracy: 0.7000 - val_loss: 0.6839
Epoch 11/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 67s 2s/step - accuracy: 0.7073 - loss: 0.7642 - val_accuracy: 0.7500 - val_loss: 0.7235
Epoch 12/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 61s 2s/step - accuracy: 0.7099 - loss: 0.7576 - val_accuracy: 0.7250 - val_loss: 0.7084
Epoch 13/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 97s 2s/step - accuracy: 0.7137 - loss: 0.7301 - val_accuracy: 0.7625 - val_loss: 0.6579
Epoch 14/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 61s 2s/step - accuracy: 0.7089 - loss: 0.7432 - val_accuracy: 0.7625 - val_loss: 0.6900
Epoch 15/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 135s 3s/step - accuracy: 0.7466 - loss: 0.6914 - val_accuracy: 0.7750 - val_loss: 0.6114
Epoch 16/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 61s 2s/step - accuracy: 0.7451 - loss: 0.6851 - val_accuracy: 0.7125 - val_loss: 0.6833
Epoch 17/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.7455 - loss: 0.6451 - val_accuracy: 0.7625 - val_loss: 0.6284
Epoch 18/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 99s 3s/step - accuracy: 0.7580 - loss: 0.6437 - val_accuracy: 0.7750 - val_loss: 0.5937
Epoch 19/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 73s 2s/step - accuracy: 0.7543 - loss: 0.6476 - val_accuracy: 0.8125 - val_loss: 0.5584
Epoch 20/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 61s 2s/step - accuracy: 0.7666 - loss: 0.5996 - val_accuracy: 0.7875 - val_loss: 0.6764
Epoch 21/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 129s 3s/step - accuracy: 0.7631 - loss: 0.6156 - val_accuracy: 0.8125 - val_loss: 0.5198
Epoch 22/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 62s 2s/step - accuracy: 0.7757 - loss: 0.5851 - val_accuracy: 0.7625 - val_loss: 0.6436
Epoch 23/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.7835 - loss: 0.5718 - val_accuracy: 0.8500 - val_loss: 0.5450
Epoch 24/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 61s 2s/step - accuracy: 0.7956 - loss: 0.5327 - val_accuracy: 0.7750 - val_loss: 0.5751
Epoch 25/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.8085 - loss: 0.5281 - val_accuracy: 0.8000 - val_loss: 0.5682
Epoch 26/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 81s 2s/step - accuracy: 0.8053 - loss: 0.5229 - val_accuracy: 0.8250 - val_loss: 0.6775
Epoch 27/50
36/36 ━━━━━━━━━━━━━━━━━━━━ 62s 2s/step - accuracy: 0.7830 - loss: 0.5634 - val_accuracy: 0.8125 - val_loss: 0.5996
Epoch 28/50
```

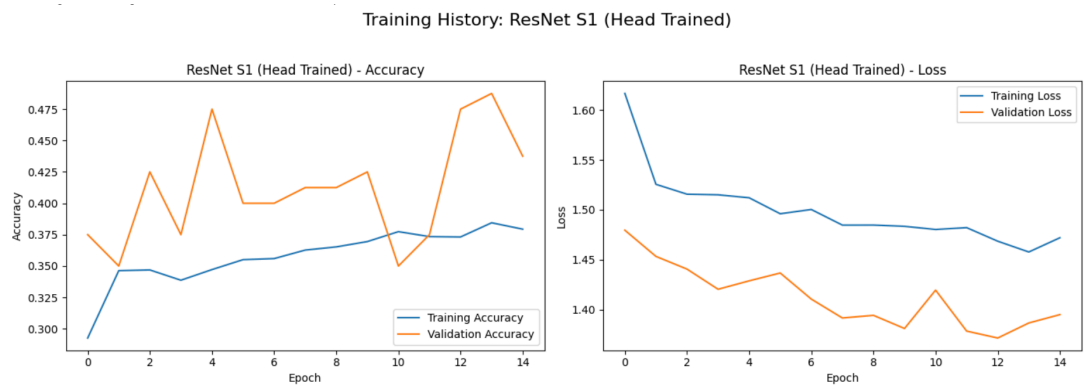



```
Loading best weights for VGG model for evaluation...
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 431ms/step - accuracy: 0.8125 - loss: 0.5198
VGG Model - Final Validation Accuracy: 0.8125, Validation Loss: 0.5198
---
```

```
## 5. Fine-Tuning a Pretrained ResNet50 ##

### ResNet - Stage 1: Freeze Base, Train Head ###
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 5s 0us/step
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 224, 224, 3)	0
resnet50 (Functional)	(None, 7, 7, 2048)	23,587,712
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dense_2 (Dense)	(None, 256)	524,544
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 5)	1,285

```
Total params: 24,113,541 (91.99 MB)
Trainable params: 525,829 (2.01 MB)
Non-trainable params: 23,587,712 (89.98 MB)
Epoch 1/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 90s 2s/step - accuracy: 0.2599 - loss: 1.7151 - val_accuracy: 0.3750 - val_loss: 1.4796
Epoch 2/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 59s 2s/step - accuracy: 0.3340 - loss: 1.5310 - val_accuracy: 0.3500 - val_loss: 1.4533
Epoch 3/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 60s 2s/step - accuracy: 0.3509 - loss: 1.5093 - val_accuracy: 0.4250 - val_loss: 1.4406
Epoch 4/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.3310 - loss: 1.5192 - val_accuracy: 0.3750 - val_loss: 1.4204
Epoch 5/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 80s 2s/step - accuracy: 0.3427 - loss: 1.5159 - val_accuracy: 0.4750 - val_loss: 1.4288
Epoch 6/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.3630 - loss: 1.4918 - val_accuracy: 0.4000 - val_loss: 1.4367
Epoch 7/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 60s 2s/step - accuracy: 0.3386 - loss: 1.5143 - val_accuracy: 0.4000 - val_loss: 1.4105
Epoch 8/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.3632 - loss: 1.4912 - val_accuracy: 0.4125 - val_loss: 1.3916
Epoch 9/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 80s 2s/step - accuracy: 0.3611 - loss: 1.4761 - val_accuracy: 0.4125 - val_loss: 1.3942
Epoch 10/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 60s 2s/step - accuracy: 0.3725 - loss: 1.4822 - val_accuracy: 0.4250 - val_loss: 1.3810
Epoch 11/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 58s 2s/step - accuracy: 0.3806 - loss: 1.4733 - val_accuracy: 0.3500 - val_loss: 1.4195
Epoch 12/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 58s 2s/step - accuracy: 0.3604 - loss: 1.4926 - val_accuracy: 0.3750 - val_loss: 1.3784
Epoch 13/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.3807 - loss: 1.4729 - val_accuracy: 0.4750 - val_loss: 1.3715
Epoch 14/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.3877 - loss: 1.4576 - val_accuracy: 0.4875 - val_loss: 1.3866
Epoch 15/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 81s 2s/step - accuracy: 0.3826 - loss: 1.4624 - val_accuracy: 0.4375 - val_loss: 1.3949
Restoring model weights from the end of the best epoch: 13
```

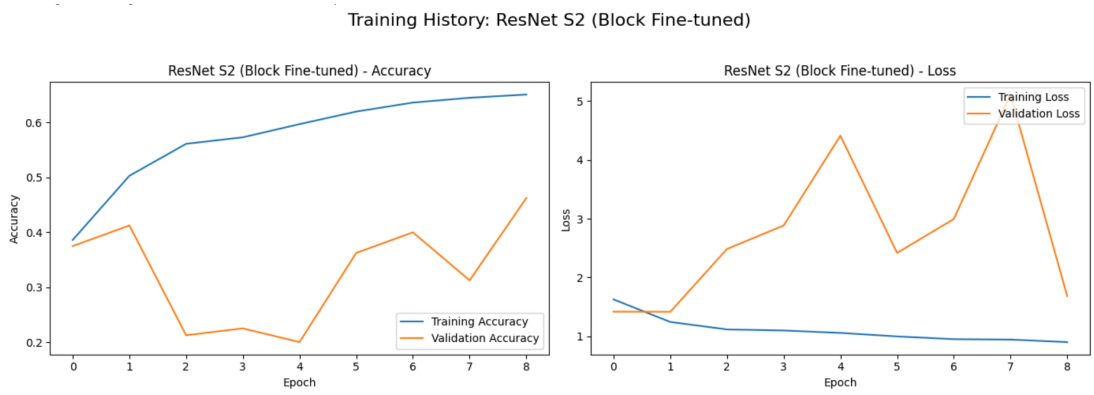



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 549ms/step - accuracy: 0.4750 - loss: 1.3715
ResNet Model (Stage 1) - Val Acc: 0.4750, Val Loss: 1.3715
```

```
### ResNet - Stage 2: Unfreeze Last Conv Block ###
Unfreezing ResNet base from layer: conv5_block1_1_conv (index 143)
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 224, 224, 3)	0
resnet50 (Functional)	(None, 7, 7, 2048)	23,587,712
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dense_2 (Dense)	(None, 256)	524,544
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 5)	1,285

```
 Total params: 24,113,541 (91.99 MB)
 Trainable params: 15,501,829 (59.13 MB)
 Non-trainable params: 8,611,712 (32.85 MB)
Epoch 1/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 102s 2s/step - accuracy: 0.3383 - loss: 2.1533 - val_accuracy: 0.3750 - val_loss: 1.4202
Epoch 2/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.4905 - loss: 1.2800 - val_accuracy: 0.4125 - val_loss: 1.4180
Epoch 3/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 58s 2s/step - accuracy: 0.5556 - loss: 1.1141 - val_accuracy: 0.2125 - val_loss: 2.4862
Epoch 4/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 81s 2s/step - accuracy: 0.5752 - loss: 1.1069 - val_accuracy: 0.2250 - val_loss: 2.8859
Epoch 5/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 60s 2s/step - accuracy: 0.5992 - loss: 1.0566 - val_accuracy: 0.2000 - val_loss: 4.4137
Epoch 6/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 81s 2s/step - accuracy: 0.6124 - loss: 0.9956 - val_accuracy: 0.3625 - val_loss: 2.4198
Epoch 7/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.6314 - loss: 0.9603 - val_accuracy: 0.4000 - val_loss: 2.9958
Epoch 8/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.6477 - loss: 0.9434 - val_accuracy: 0.3125 - val_loss: 5.1137
Epoch 9/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 58s 2s/step - accuracy: 0.6391 - loss: 0.9096 - val_accuracy: 0.4625 - val_loss: 1.6859
Epoch 9: early stopping
Restoring model weights from the end of the best epoch: 2.
```

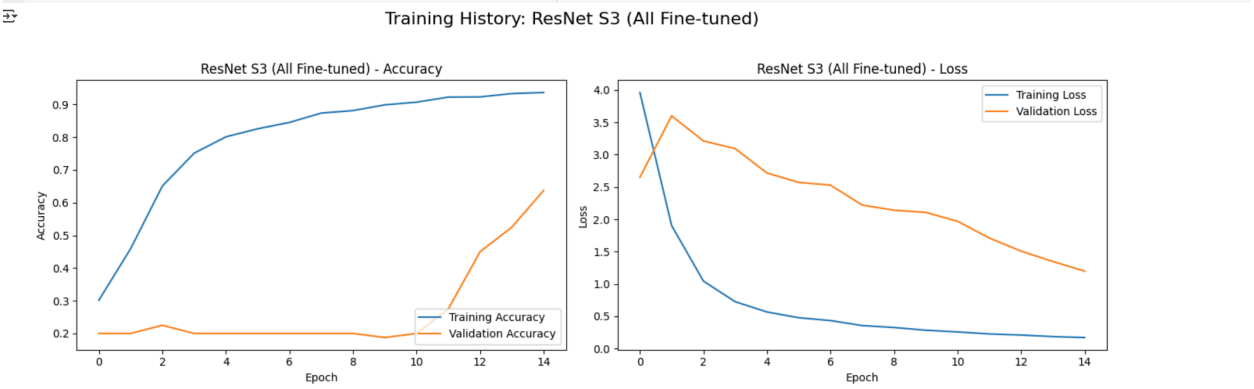



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 508ms/step - accuracy: 0.4125 - loss: 1.4180
ResNet Model (Stage 2) - Val Acc: 0.4125, Val Loss: 1.4180
```

```
### ResNet - Stage 3: Unfreeze All Layers ###
Model: "functional_1"
```

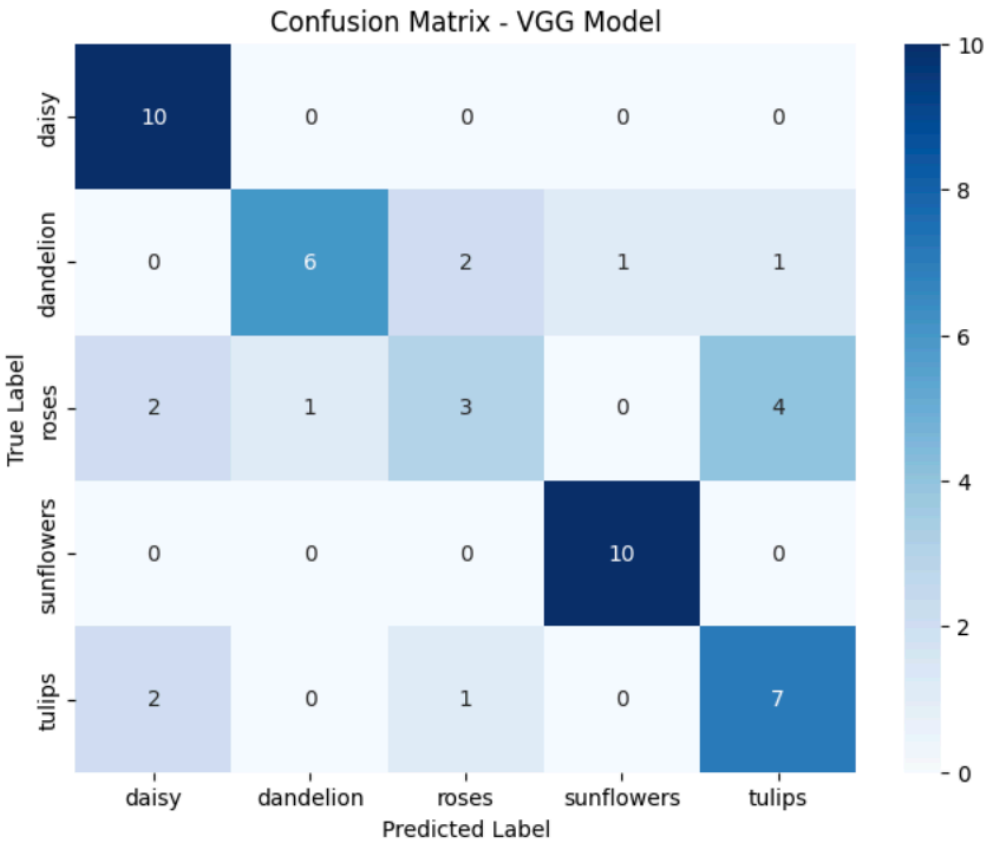
Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 224, 224, 3)	0
resnet50 (Functional)	(None, 7, 7, 2048)	23,587,712
global_average_pooling2d (GlobalAveragePooling2D)	(None, 2048)	0
dense_2 (Dense)	(None, 256)	524,544
dropout_1 (Dropout)	(None, 256)	0
dense_3 (Dense)	(None, 5)	1,285

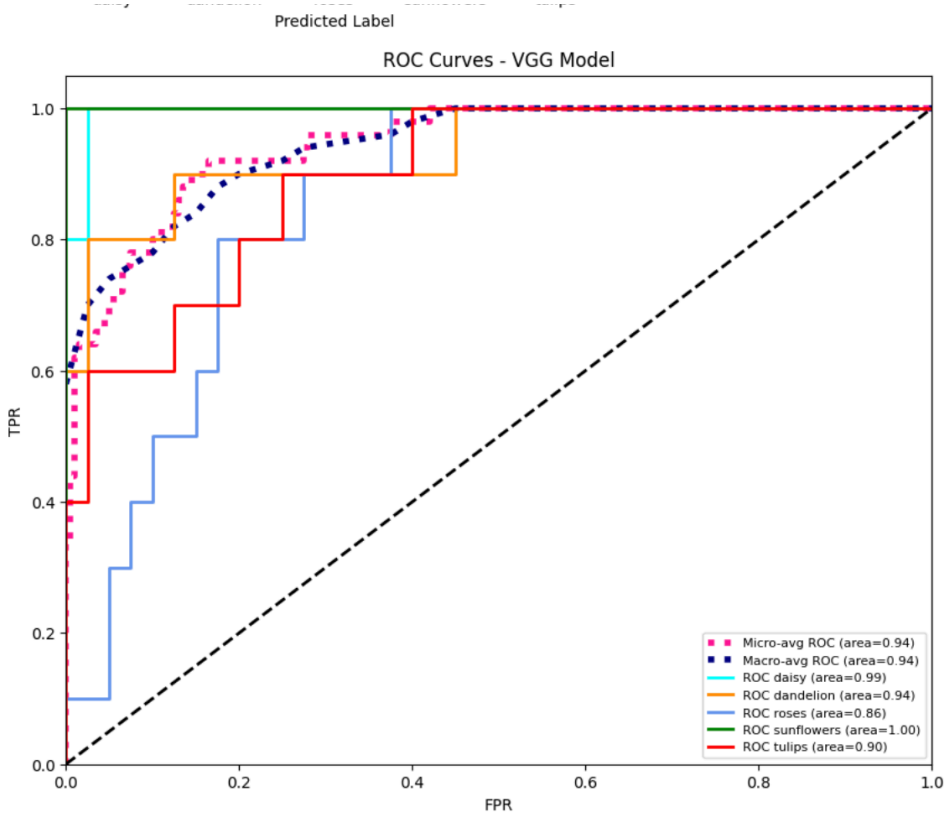
```
Total params: 24,113,541 (91.99 MB)
Trainable params: 24,060,421 (91.78 MB)
Non-trainable params: 53,120 (207.50 KB)
Epoch 1/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 196s 4s/step - accuracy: 0.2523 - loss: 4.8363 - val_accuracy: 0.2000 - val_loss: 2.6496
Epoch 2/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 65s 2s/step - accuracy: 0.4008 - loss: 2.2479 - val_accuracy: 0.2000 - val_loss: 3.6002
Epoch 3/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 82s 2s/step - accuracy: 0.6026 - loss: 1.1916 - val_accuracy: 0.2250 - val_loss: 3.2122
Epoch 4/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.7376 - loss: 0.7574 - val_accuracy: 0.2000 - val_loss: 3.0936
Epoch 5/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 81s 2s/step - accuracy: 0.7964 - loss: 0.5798 - val_accuracy: 0.2000 - val_loss: 2.7163
Epoch 6/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 96s 3s/step - accuracy: 0.8205 - loss: 0.4979 - val_accuracy: 0.2000 - val_loss: 2.5707
Epoch 7/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 119s 2s/step - accuracy: 0.8485 - loss: 0.4275 - val_accuracy: 0.2000 - val_loss: 2.5291
Epoch 8/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 74s 2s/step - accuracy: 0.8769 - loss: 0.3624 - val_accuracy: 0.2000 - val_loss: 2.2202
Epoch 9/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 110s 3s/step - accuracy: 0.8781 - loss: 0.3336 - val_accuracy: 0.2000 - val_loss: 2.1413
Epoch 10/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 143s 3s/step - accuracy: 0.8963 - loss: 0.2875 - val_accuracy: 0.1875 - val_loss: 2.1069
Epoch 11/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 105s 3s/step - accuracy: 0.9126 - loss: 0.2422 - val_accuracy: 0.2000 - val_loss: 1.9703
Epoch 12/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 111s 2s/step - accuracy: 0.9165 - loss: 0.2383 - val_accuracy: 0.2750 - val_loss: 1.7098
Epoch 13/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 76s 2s/step - accuracy: 0.9254 - loss: 0.2056 - val_accuracy: 0.4500 - val_loss: 1.5088
Epoch 14/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 81s 2s/step - accuracy: 0.9309 - loss: 0.1880 - val_accuracy: 0.5250 - val_loss: 1.3474
Epoch 15/15
36/36 ━━━━━━━━━━━━━━━━━━━━ 106s 3s/step - accuracy: 0.9369 - loss: 0.1676 - val_accuracy: 0.6375 - val_loss: 1.1993
Restoring model weights from the end of the best epoch: 15.
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 533ms/step - accuracy: 0.6375 - loss: 1.1993
ResNet Model (Stage 3) - Val Acc: 0.6375, Val Loss: 1.1993
```

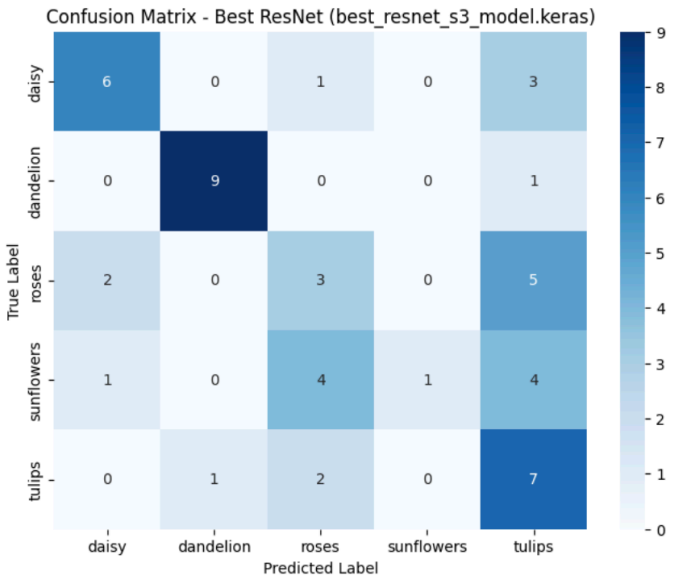
```
## 6. Result Comparison ##
Best VGG model path: best_vgg_model.keras
Best ResNet model path (overall): best_resnet_s3_model.keras

Using TEST SET for final model comparison plots.

--- VGG Model Evaluation (Test Set) ---

Evaluating VGG Model on its generator (50 samples)...
VGG Model - Accuracy: 0.7200, Loss: 0.6725
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 582ms/step

VGG Model - Classification Report:
              precision    recall  f1-score   support

       daisy       0.71      1.00      0.83        10
   dandelion       0.86      0.60      0.71        10
       roses       0.50      0.30      0.38        10
  sunflowers       0.91      1.00      0.95        10
      tulips       0.58      0.70      0.64        10

    accuracy                           0.72        50
   macro avg       0.71      0.72      0.70        50
weighted avg       0.71      0.72      0.70        50
```

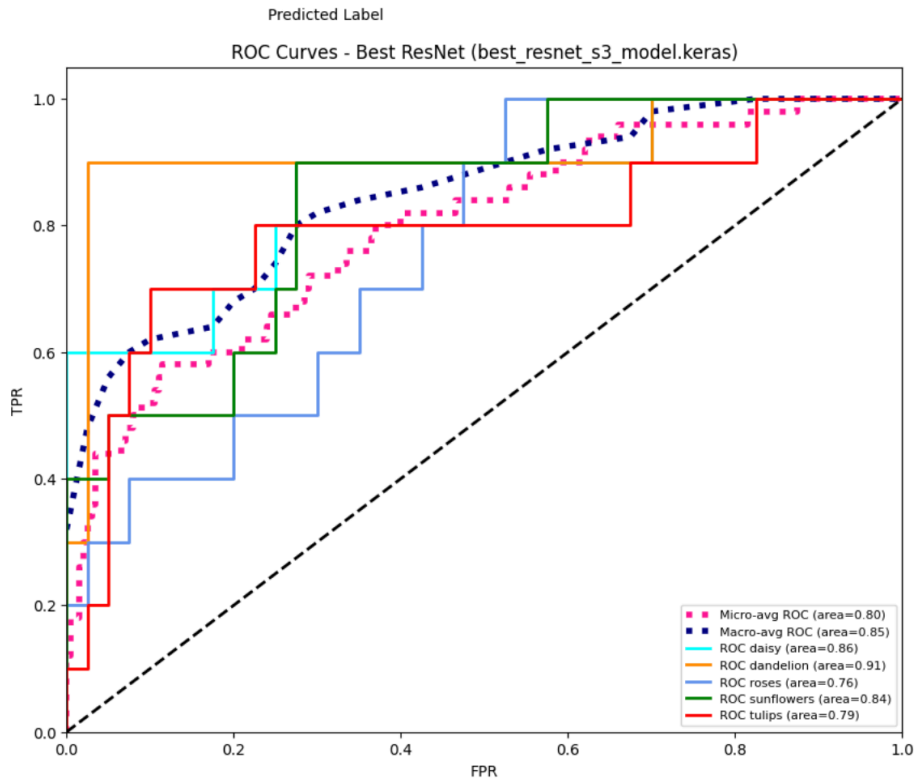

Confusion Matrix - VGG Model
True Label
daisy
dandelion
roses
sunflowers
tulips
10 0 0 0 0
0 6 2 1 1
2 1 3 0 4
0 0 0 10 0
2 0 1 0 7
daisy
dandelion
roses
sunflowers
tulips
Predicted Label
10
8
6
4
2
0




```
VGG Model AUC Scores:
  AUC for class 'daisy': 0.9950
  AUC for class 'dandelion': 0.9375
  AUC for class 'roses': 0.8575
  AUC for class 'sunflowers': 1.0000
  AUC for class 'tulips': 0.8975
  Micro-average AUC: 0.9444
  Macro-average AUC: 0.9440
---
```

```
--- Best ResNet Model Evaluation (Test Set) ---

Evaluating Best ResNet (best_resnet_s3_model.keras) on its generator (50 samples)...
Best ResNet (best_resnet_s3_model.keras) - Accuracy: 0.5200, Loss: 1.4377
1/1 ━━━━━━━━━━━━━━━━━━━━ 4s 4s/step

Best ResNet (best_resnet_s3_model.keras) - Classification Report:
              precision    recall  f1-score   support

       daisy       0.67      0.60      0.63        10
   dandelion       0.90      0.90      0.90        10
       roses       0.30      0.30      0.30        10
  sunflowers       1.00      0.10      0.18        10
      tulips       0.35      0.70      0.47        10

    accuracy                           0.52        50
   macro avg       0.64      0.52      0.50        50
weighted avg       0.64      0.52      0.50        50
```






```
Best ResNet (best_resnet_s3_model.keras) AUC Scores:
  AUC for class 'daisy': 0.8600
  AUC for class 'dandelion': 0.9150
  AUC for class 'roses': 0.7625
  AUC for class 'sunflowers': 0.8375
  AUC for class 'tulips': 0.7925
  Micro-average AUC: 0.7958
  Macro-average AUC: 0.8458
---
```