SWINBURNE UNIVERSITY OF TECHNOLOGY

APPLIED MACHINE LEARNING - COS30082

ASSIGNMENT - IMAGE CLASSIFICATION

Instructor:
Mr. Minh HOANG

Student:
Huynh Trung Chien
104848770

Submission Date:
26/10/2025

Table of Content

COS30082 - Applied Machine Learning

Assignment Report

1. Introduction

This report illustrates my work for Assignment 1 - Image Classification of the course COS30082 - Applied Machine Learning. The reports focuses on discussing the methodology used to minimize the risk of overfitting, describing the model architectures, the loss function, hyperparameters, and other details. Besides that, the model performances will also be compared and discussed with clear justifications.

2. Methodology

2.1 Dataset Description

The dataset used in this assignment is the "Deep Learning Practice - Image Classification" obtained from Kaggle, which can be accessed through this link (https://www.kaggle.com/competitions/deep-learning-practice-image-classification/overview).
The dataset contains two folder: train and test. The training set contains 10,000 images and for 10 classes with each subfolder representing one class and having 1,000 examples. In this assignment, the test dataset will not be used because the images are unlabeled. Therefore, the train, validation, and test sets will be split from the train folder only.

2.2 Data Preprocessing

Through out this assignment, the dataset will be split into train, validation, and test sets with the sizes of 0.7, 0.2, and 0.1, respectively *(Figure 2)*. All the images will be resized into (384 x 384) because by analyzing the images from the dataset, I figured out that many objects inside the images are relatively small. Therefore, increasing the image size could potentially increase the accuracy. To analyze the difference between different input sizes, I also trained a ViT-224 model with (224 x 224) size later to compare the performance with ViT-384 (384 x 384) .

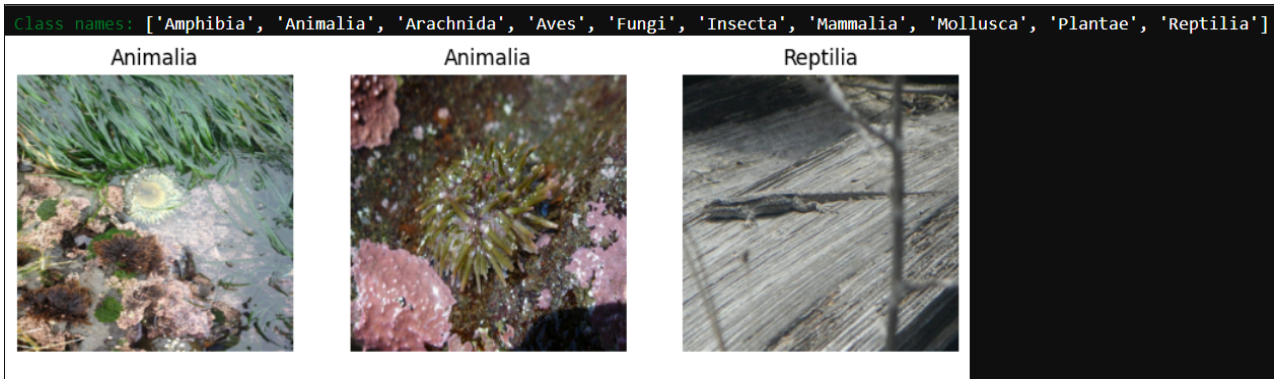

Figure 1: Example images

```
[6]: # Define AUTOTUNE for performance optimization
AUTOTUNE = tf.data.AUTOTUNE

# Path to dataset (KaggleHub downloaded folder)
data_dir = hunhtrungchin_animal_dataset_new_path + "/train/"

# Initial train/validation split (70% train, 30% temp) ===
train_ds = image_dataset_from_directory(
    data_dir,
    validation_split=0.3,
    subset="training",
    seed=42,
    image_size=(384, 384),
    batch_size=32
)

temp_ds = image_dataset_from_directory(
    data_dir,
    validation_split=0.3,
    subset="validation",
    seed=42,
    image_size=(384, 384),
    batch_size=32
)

class_names = train_ds.class_names

# Split temp dataset (30%) into validation (20%) and test (10%)
val_size = 2 / 3   # 20% of total
test_size = 1 / 3  # 10% of total

temp_size = tf.data.experimental.cardinality(temp_ds).numpy()
val_ds = temp_ds.take(int(temp_size * val_size))
test_ds = temp_ds.skip(int(temp_size * val_size))

Found 10008 files belonging to 10 classes.
Using 7006 files for training.
Found 10008 files belonging to 10 classes.
Using 3002 files for validation.
```

Figure 2: Train, test, split

2.3 Data Augmentation

Data augmentation plays a crucial role in improving the performance and generalization of deep learning models, especially in image classification tasks. Augmentation techniques increase the robustness the model by diversify the training data [3]. Some common transformations and augmentation techniques are random rotations, flips, zooming, translation, brightness adjustments, and so on. These techniques normally reduce overfitting because they prevent the models from memorizing specific training examples, while forcing the model to learn meaningful features. Additionally, the training set only contains approximately 7,000 images, so augmentation is particularly valuable because it helps to increase the dataset size.

Specifically for this assignment, I tried two versions of data augmentation, one of which relies on TensorFlow built-in preprocessing layers, while the other one got the idea from this paper [1].

a. Data Augmentation with TensorFlow

The first simple data augmentation layer was implemented using TensorFlow's **tf.keras.Sequential** API *(Figure 3)*, and this layer will later be integrated directly to image tensors during training. This augmentation pipeline includes several layers to increase the generalization of the model. To be more specific, the first layer is **RandomFlip**, which flips

the images horizontally to simulate different viewing angles, while **RandomRotation** and **RandomTranslation** slightly rotate and shift the images. The **RandomZoom** layer zooms in and out with reflection fill mode, ensuring all important features remain visible. Finally, the **RandomContrast** and **RandomBrightness** layers will help to adjust the lighting and color intensity. By adding this augmentation pipelin into training, each image will be slightly different across epochs, increasing the dataset size without requiring additional data, leading to the increase in robustness of the model.

```
# Create a data_augmentation layer for augmenting the data
data_augmentation= tf.keras.Sequential([
    tf.keras.layers.Input(shape=(384, 384, 3)),
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
    tf.keras.layers.RandomTranslation(height_factor=0.1, width_factor=0.1),
    tf.keras.layers.RandomZoom(height_factor=(-0.2, 0.1), fill_mode='reflect'),
    tf.keras.layers.RandomContrast(0.2),
    tf.keras.layers.RandomBrightness(factor=0.2, value_range=(0, 255)),
])
```

Figure 3: Standard data augmentation pipeline

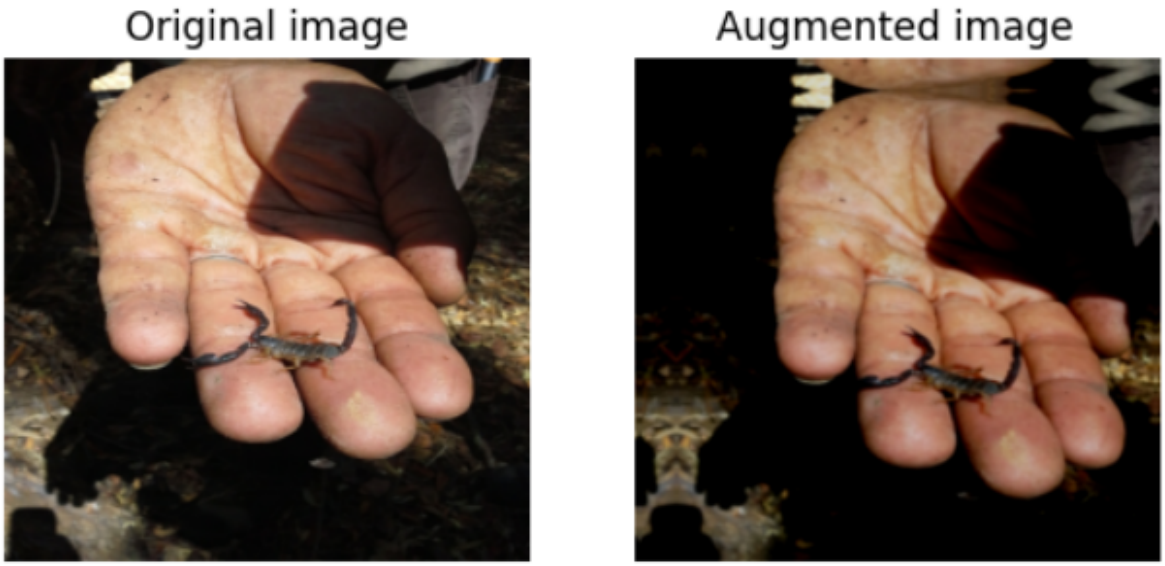

Figure 4: Image before and after applying standard data augmentation

b. Advanced Data Augmentation

Besides applying standard augmentations, I also applied advanced augmentation techniques which go beyond standard flips, rotations and zooms to simulate more real-world situations, and increase the generalization of the model. In the referenced study, the author proposed three advanced augmentation techniques along with the standard one:

- **Pairwise Channel Transfer**: This is a augmentation technique, where the color information from the source image (red, green, or blue channel) will be transferred to the corresponding channel of the target image. This technique is not only applied for the RGB channels, but can also be applied for the HSV (Hue - Saturation - Value) channels. I applied both RGB and HSV pairwise channel transformations for the dataset.

Figure 5: RGB pairwise channel transformation (Target - Source - Augmented)

Figure 6: HSV pairwise channel transforamation (Target - Source - Augmented)

- **Novel Occlusion**: This augmentation method places the random objects as the occlusion in the target image, where the random objects are randomly chosen from the dataset. First, the randomly image is picked from the dataset, then it is resized before being placed in the target image. For this specific dataset, I only placed the occlusion of size (75 x 75) in the corners to avoid the occlusion hiding the objects in the original images.

Figure 7: Novel Occlusion transformation (Target - Source - Augmented)

- **Masking Strategies**: This augmentation technique applies vertical/horizontal, circular or checkered masks to the image. This helps the model become resilient to missing information or parts of the objects being hidden.

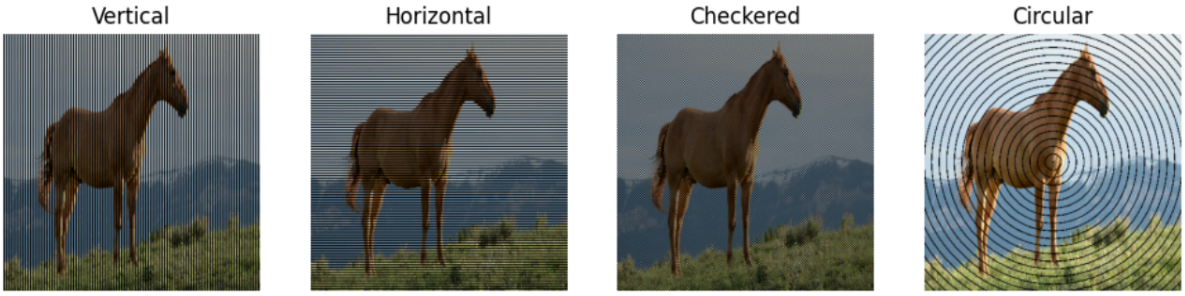

Figure 8: Images applied novel masking

- **Standard Augmentation**: I also applied the standard augmentation to also increase the diversity.

Figure 9 : Standard augmentation

After creating 5 augmented datasets, I then combined all of them to just one dataset for later training, this dataset will help to reduce overfitting and increase the overall accuracy.

2.4 Models

In this assignment, I chose five models for training, including baseline model, EfficientNetV2_B0 with classical data augmentation, EfficientNetV2_B0 with advanced data augmentation, ViT-224, and ViT-384.

a. Baseline model

The baseline model was created using **Sequential API**, this model was used to test the initial accuracy and for comparison purposes only. Therefore, the performance of this model is not expected to be high.

```
# Create a simple CNN model
baseline_model = models.Sequential( # Using Sequential API for simplicity
    [
        layers.Input(shape=(384, 384, 3)),
        data_augmentation,
        layers.Rescaling(1./255),

        # --- Block 1 ---
        layers.Conv2D(32, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.Conv2D(32, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.25),

        # --- Block 2 ---
        layers.Conv2D(64, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.Conv2D(64, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.3),

        # --- Block 3 ---
        layers.Conv2D(128, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.Conv2D(128, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.4),

        # --- Block 4 ---
        layers.Conv2D(256, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.Conv2D(256, (3, 3), padding='same', activation='relu'),
        layers.BatchNormalization(),
        layers.MaxPooling2D((2, 2)),
        layers.Dropout(0.4),
```

```
# Add Dense layers after flatten, with the final dense layers to give
baseline_model.add(layers.Dense(128, activation='relu'))
baseline_model.add(layers.BatchNormalization())
baseline_model.add(layers.Dropout(0.5))
baseline_model.add(layers.Dense(10, activation='softmax'))
```

Figure 10: Baseline model

When training the baseline model, I used **EarlyStopping** to avoid overfitting, and **ReduceLROnPlateau** to increase the accuracy of the model. For optimization, I used **Adam** optimizers with learning rate of **0.001**, while the loss was **SparseCategoricalCrossentropy**.

4.2 - Compile and train the model

```
# Create early stopping to optimize training time and reduce overfitting
early_stop = EarlyStopping(
    monitor="val_loss",   # Stop if validation loss stops improving
    patience=15,          # Allow some patience
    restore_best_weights=True,  # Roll back to the best model
    verbose=1
)


# Create reducing LR on PLateau
reduce_lr = ReduceLROnPlateau(
    monitor="val_loss",   # Watch validation loss
    factor=0.5,           # Reduce LR by half
    patience=5,           # Wait 5 epochs before reducing
    min_lr=1e-6,          # Don't let LR go below this
    verbose=1
)
```

```
# Create optimizer SGD and train the model
opt = tf.keras.optimizers.Adam(learning_rate=0.001)
loss = tf.keras.losses.SparseCategoricalCrossentropy()

baseline_model.compile(optimizer=opt, loss=loss, metrics=['accuracy'])

baseline_history = baseline_model.fit(
    train_ds,
    epochs=20,
    validation_data=val_ds,
    callbacks=[early_stop, reduce_lr], # Include early_stop and reduce_lr
    verbose=1
)
```

Figure 11: Baseline model training

b. EfficientNetV2_B0 with classical data augmentation

For this assignment, I also used the EfficientNetV2_B0 with classical data augmentation techniques that had been done before. This model uses the pretrained ImageNet weights and is fine-tuned on the train dataset. Classical augmentation techniques are applied in this model, making the model more robust and reduce overfitting. The goal of this model is to evaluate the model's performance under standard augmentation, while establishing a baseline for comparison with advanced augmentation techniques.

1.1 - Create the base model

```
# Base model (EfficientNetV2B0)
efficientnet_model_v1 = tf.keras.applications.EfficientNetV2B0(
    include_top=False,
    weights="imagenet",
    input_shape=(384, 384, 3)
)

efficientnet_model_v1.trainable = False  # freeze base model for feature extraction
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/efficientnet_v2/efficientnetv2-b0_notop.h5
24274472/24274472 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

Figure 12: Create the base model

1.1 - Create the base model

```
# Base model (EfficientNetV2B0)
efficientnet_model_v1 = tf.keras.applications.EfficientNetV2B0(
    include_top=False,
    weights="imagenet",
    input_shape=(384, 384, 3)
)

efficientnet_model_v1.trainable = False  # freeze base model for feature extraction
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/efficientnet_v2/efficientnetv2-b0_notop.h5
24274472/24274472 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

1.2 - Build the model

```
# Build the model with data augmentation Layer
inputs = layers.Input(shape=(384, 384, 3))
x = data_augmentation(inputs)  # Apply data augmentation
x = tf.keras.applications.efficientnet.preprocess_input(x)
x = efficientnet_model_v1(x, training=False)

# Multi-stage pooling (capture both average)
x = layers.GlobalAveragePooling2D()(x)

# Dense block with high regularization
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.5)(x)

x = layers.Dense(512, activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.4)(x)

x = layers.Dense(256, activation="relu")(x)
x = layers.BatchNormalization()(x)
x = layers.Dropout(0.3)(x)

outputs = layers.Dense(len(class_names), activation="softmax")(x)
model_v1 = Model(inputs, outputs)
```

Figure 13: Load the model and add classification layer

For feature extraction, I used the **SGD optimizer** for my efficientnet_v1 model with the **learning_rate = 0.001**, and the **momentum of 0.9**. I also included **EarlyStopping** and **ReduceLROnPlateau** for this model.

1.3 - Compile the model ¶

```
# Create optimizer SGD and train the model
opt = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
loss = tf.keras.losses.SparseCategoricalCrossentropy()

model_v1.compile(optimizer=opt, loss=loss, metrics=['accuracy'])
```

1.4 - Feature Extraction

```
# Create early_stopping and learning rate decay
early_stop = EarlyStopping(monitor="val_loss", patience=8, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3, min_lr=1e-6)

# Train the model (Feature Extraction Phase)
fe_history_v1 = model_v1.fit(
    train_ds,
    validation_data=val_ds,
    epochs=10,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 14: Compile the efficientnet_v1 model and run feature extraction

For fine tuning, I use the **Adam optimizers** with the **learning_rate of 1e-5**, and teh loss is **SparseCategoricalCrossentropy.**

1.5 - Fine tuning

```
# Fine-tuning phase
efficientnet_model_v1.trainable = True
for layer in efficientnet_model_v1.layers[:-70]:  # freeze all but top 70 layers
    layer.trainable = False

# Compile again with a lower LR for fine-tuning
model_v1.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

efficientnet_model_v1_history = model_v1.fit(
    train_ds,
    validation_data=val_ds,
    epochs=50,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 15: Fine tuning the efficientnet_v1 model

c. *EfficientNetV2_B0 with advanced data augmentation*

Besides training the EfficientNetV2_B0 with the standard augmendation, I also trained the model with the same hyperparameters, but with the advanced dataset to compare the performance with the use of classical dataset.

2.4 - Feature Extraction

```
# Create early_stopping and learning rate decay
early_stop = EarlyStopping(monitor="val_loss", patience=8, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3, min_lr=1e-6)

# Train the model (Feature Extraction Phase)
fe_history_v2 = model.fit(
    all_aug_ds_32,
    validation_data=val_ds,
    epochs=10,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 16: Feature extraction for efficientnet_v2 with advanced augmented dataset

2.5 - Fine tuning ¶

```
# Fine-tune the model
efficientnet_model_v2.trainable = True
for layer in efficientnet_model_v2.layers[:-70]:
    layer.trainable = False

# Recompile the full model with lower learning rate
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

# Fine-tune training
fine_tune_history_v2 = model.fit(
    all_aug_ds_32,
    validation_data=val_ds,
    epochs=50,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 17: Fine tuning for effcientnet_v2 with advanced augmented dataset

d. ViT-224

The Vision Transformer (ViT-224) model was implemented using a patch size of 16×16 and an input resolution of 224×224 pixels. This configuration allows the model to learn visual features by treating images as sequences of patches, leveraging self-attention mechanisms to capture global context. ViT-224 was pretrained on ImageNet and fine-tuned on the target dataset with advanced data augmentation techniques. This model will be used to compare the difference in performance between (224 x 224) inputs and (384 x 384) inputs from ViT-384.

There are three training phases for training this model. The first step was training classification head, followed by training the final three blocks, while the final step was training full model.

3.2 - Build the base ViT-224 model

```
vit_backbone_224 = keras_hub.models.ViTBackbone.from_preset("vit_base_patch16_224_imagenet")
vit_backbone_224.trainable = False  # Freeze for initial training

inputs = tf.keras.Input(shape=(IMG_SIZE_224, IMG_SIZE_224, 3))
x = vit_backbone_224(inputs)
x = layers.Lambda(lambda t: t[:, 0, :])(x)  # extract CLS token
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.4)(x)
outputs = layers.Dense(len(class_names), activation="softmax", dtype="float32")(x)

vit_model_224 = tf.keras.Model(inputs, outputs)
```

3.3 - Train Classification Head

```
vit_model_224.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

# Create callbacks
early_stop = EarlyStopping(monitor="val_loss", patience=8, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3, min_lr=1e-6)

print("\nPhase 1 (ViT-224): Training classifier head...")
vit_224_phase1_history = vit_model_224.fit(
    all_aug_ds_224,
    validation_data=val_ds_224,
    epochs=10,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 18: Loading ViT-224 model and training classification head

3.4 - Fine Tuning Final Blocks

```
vit_backbone_224.trainable = True

# Freeze all layers first
for layer in vit_backbone_224.layers:
    layer.trainable = False

# Selectively unfreeze the last 3 transformer blocks
num_blocks = 12  # ViT-Base has 12 blocks
blocks_to_unfreeze = 3  # Unfreeze last 3 blocks

for layer in vit_backbone_224.layers:
    if "transformer" in layer.name:
        try:
            block_num = int(layer.name.split("_")[-1])
            if block_num >= (num_blocks - blocks_to_unfreeze):
                layer.trainable = True
                print(f"Unfreezing: {layer.name}")
        except:
            pass

# Also unfreeze the layer normalization at the end
for layer in vit_backbone_224.layers:
    if "layer_norm" in layer.name.lower() and layer == vit_backbone_224.layers[-1]:
        layer.trainable = True
        print(f"Unfreezing: {layer.name}")

vit_model_224.compile(
    optimizer=tf.keras.optimizers.Adam(1e-5),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

print(f"\n Phase 2 (ViT-224): Fine-tuning last {blocks_to_unfreeze} transformer blocks...")
print(f" Trainable params: {vit_model_224.count_params():,}")
vit_224_phase2_history = vit_model_224.fit(
    all_aug_ds_224,
    validation_data=val_ds_224,
    epochs=5,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 19 : Fine tuning the final three blocks of the ViT-224

3.5 - Fine Tuning all model

```
print("\n Phase 3 (ViT-224): Gentle full model fine-tuning...")
for layer in vit_backbone_224.layers:
    layer.trainable = True

vit_model_224.compile(
    optimizer=tf.keras.optimizers.Adam(5e-6),  # Very low LR
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

vit_224_phase3_history = vit_model_224.fit(
    all_aug_ds_224,
    validation_data=val_ds_224,
    epochs=5,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 20 : Fine tuning all model ViT-224

e. ViT-384

The Vision Transformer (ViT-384) extends the ViT architecture to a higher input resolution of 384×384 pixels, allowing the model to capture finer spatial details. This larger input size increases the number of patches, providing a richer representation of the image while also requiring more computational resources. The model was initialized with ImageNet-pretrained weights and fine-tuned using advanced augmentation techniques to analyze how input resolution affect the performance. ViT-384 serves as a high-capacity transformer model for comparison with both CNN and lower-resolution ViT variants. All the configurations and parameters for this model will be the same as the ViT-224 model.

4.2 - Build the base ViT-384 model ¶

```
vit_backbone_384 = keras_hub.models.ViTBackbone.from_preset("vit_base_patch16_384_imagenet")
vit_backbone_384.trainable = False  # Freeze for initial training

inputs = tf.keras.Input(shape=(IMG_SIZE_384, IMG_SIZE_384, 3))
x = vit_backbone_384(inputs)
x = layers.Lambda(lambda t: t[:, 0, :])(x)  # extract CLS token
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.4)(x)
outputs = layers.Dense(len(class_names), activation="softmax", dtype="float32")(x)

vit_model_384 = tf.keras.Model(inputs, outputs)
```

```
Downloading from https://www.kaggle.com/api/v1/models/keras/vit/keras/vit_base_patch16_384_imagenet/2/download/config.json...
100%|██████████| 593/593 [00:00<00:00, 1.21MB/s]
Downloading from https://www.kaggle.com/api/v1/models/keras/vit/keras/vit_base_patch16_384_imagenet/2/download/model.weights.h5...
100%|██████████| 329M/329M [00:21<00:00, 16.1MB/s]
```

4.3 - Train Classification Head

```
vit_model_384.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

# Create callbacks
early_stop = EarlyStopping(monitor="val_loss", patience=8, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3, min_lr=1e-6)

print("\nPhase 1 (ViT-384): Training classifier head...")
vit_384_phase1_history = vit_model_384.fit(
    all_aug_ds_384,
    validation_data=val_ds_384,
    epochs=10,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 21: Training ViT-384 in classification head

4.4 - Fine-tuning Final Blocks

```
vit_backbone_384.trainable = True

# Freeze all layers first
for layer in vit_backbone_384.layers:
    layer.trainable = False

# Selectively unfreeze the last 3 transformer blocks
num_blocks = 12  # ViT-Base has 12 blocks
blocks_to_unfreeze = 3  # Unfreeze last 3 blocks

for layer in vit_backbone_384.layers:
    if "transformer" in layer.name:
        try:
            block_num = int(layer.name.split("_")[-1])
            if block_num >= (num_blocks - blocks_to_unfreeze):
                layer.trainable = True
                print(f"Unfreezing: {layer.name}")
        except:
            pass

# Also unfreeze the layer normalization at the end
for layer in vit_backbone_384.layers:
    if "layer_norm" in layer.name.lower() and layer == vit_backbone_384.layers[-1]:
        layer.trainable = True
        print(f"Unfreezing: {layer.name}")

vit_model_384.compile(
    optimizer=tf.keras.optimizers.Adam(1e-5),
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

print(f"\nPhase 2 (ViT-384): Fine-tuning last {blocks_to_unfreeze} transformer blocks...")
print(f"Trainable params: {vit_model_384.count_params():,}")
vit_384_phase2_history = vit_model_384.fit(
    all_aug_ds_384,
    validation_data=val_ds_384,
    epochs=3,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 22: Fine tuning the final three blocks of ViT-384

4.5 - Fine Tuning all model ¶

```
print("\nPhase 3 (ViT-384): Gentle full model fine-tuning...")
for layer in vit_backbone_384.layers:
    layer.trainable = True

vit_model_384.compile(
    optimizer=tf.keras.optimizers.Adam(5e-6),  # Very low LR
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)

vit_384_phase3_history = vit_model_384.fit(
    all_aug_ds_384,
    validation_data=val_ds_384,
    epochs=5,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

Figure 23: Fine tuning the full ViT-384 model

2.7 Overfitting Mitigation

In my work, the main strategy that I used to minimize overfitting and ensure better generalization was extensive data augmentation. This technique helps to expand the dataset and train the model to handle real-time situations with a wider variety of image transformations.

Moreover, regularization methods were also integrated into the model architectures. Dropout layers were added to randomly deactivate neurons during training. Early stopping was another important method to prevent overfitting. The training process was monitored on the validation loss, and training was automatically stopped once the model performance stopped improving. ReduceLROnPlateau was also applied to fine-tune the optimization process, allowing the model to converge more smoothly.

Finally, both EfficientNetV2_B0 and Vision Transformer models were initialized with pretrained ImageNet weights. This transfer learning approach provides a strong prior knowledge of general image features, allowing the models to converge faster and avoid overfitting on the smaller dataset used in this assignment.

2.8 Training Setup

All experiments in this assignment were done using Google Colab with an NVIDIA A100 GPU, providing sufficient computational power for both convolutional and transformer-based models. The implementation was done using TensorFlow and Keras frameworks, ensuring efficient training and fine-tuning across different architectures.

For the convolutional models (Baseline and EfficientNetV2_B0), a batch size of 32 was used, while for the Vision Transformer (ViT-224 and ViT-384) models, a smaller batch size of 16 was chosen due to higher memory requirements at larger input resolutions. While the baseline and EfficientNetV2_B0 were trained on 50 epochs, the Vision Transformer models were only trained on 5 - 10 epochs to reduce the training time and save resources.

The learning rate was set to 0.001 during the feature extraction phase and reduced to 1e-5 during fine-tuning for a better convergence. Optimization was set either the Adam or SGD optimizer depending on the model configuration, and the loss function used for all models was Sparse Categorical Crossentropy.

4. Results and Discussion

4.1 Evaluation Metrics

In this assignment, the model performance was evaluated using two metrics, specifically **Top-1 Accuracy** and **Average Accuracy per Class**.

Top-1 Accuracy measures the class with the highest probability that matches the true label. This is the most commonly used metric in image classification tasks and indicates how correctly the model is at identifying the majority of images [2].

On the other hand, **Average Accuracy per Class** calculates the accuracy for each class individually and then calculate the mean of these values. This metric aims to identify whether the model performs equally well across all categories or is biased toward certain classes. It is particularly useful in datasets where class imbalance or visual complexity varies between categories.

In addition to these two main metrics, other performance indicators such as the confusion matrix and loss curves were also analyzed to further understand model behavior, identify misclassified samples, and evaluate the generalization capability of each architecture.

4.2 Model Comparison

Figure shows the performance comparison summary across all models with ViT-384 being the best one, while baseline model performed poorly.

```
================================================================================================
MODEL COMPARISON SUMMARY
================================================================================================

Model                          Top-1 Acc    Avg Class Acc   Misclassified
------------------------------------------------------------------------------------------------
ViT-384                           94.40%         94.45%         57/1018
ViT-224                           90.77%         90.75%         94/1018
EfficientNetV2-B0 (v2)            89.00%         89.12%        112/1018
EfficientNetV2-B0 (v1)            86.35%         86.42%        139/1018
Baseline Model                    29.67%         30.26%        716/1018
------------------------------------------------------------------------------------------------
```

Figure 24: Models performance comparison summary

a. Baseline model

The baseline model achieved a **Top-1 Accuracy of 29.67%** *(Figure 25)* and an **Average Accuracy per Class of 30.26%**. This result shows that the baseline model only predicts 1 out of 3 correctly for the top predicted class. This relatively low accuracy number also indicated that the model failed to handle unseen data.

The **Average Accuracy per Class (30.26%)** *(Figure 25)* being close to the overall Top-1 Accuracy shows that the model's performance was consistent across different classes. However, this also highlights that the performance of the model is not good across all classes, implying that the model architecture is not good, and data augmentation is not diverse enough.

```
============================================================
MODEL EVALUATION METRICS
============================================================
Top-1 Accuracy: 29.67%
  (How often the top predicted class matches the true label)

Average Accuracy per Class: 30.26%
  (Mean of individual class accuracies - handles class imbalance)
============================================================
```

Figure 25: Top-1 Accuracy and Average Accuracy per Class for baseline model

The below figures further evaluate the performance of the baseline models, with Figure 26 showing the Per-Class Accuracy, Figure 27 inllustrating the Classification Report, while Figure 28 indicating the Confusion Matrix.

```
PER-CLASS ACCURACY:
--------------------------------
Amphibia             :  19.79%
Animalia             :  45.45%
Arachnida            :  21.65%
Aves                 :  48.94%
Fungi                :  18.35%
Insecta              :  19.13%
Mammalia             :  32.41%
Mollusca             :  26.67%
Plantae              :  47.06%
Reptilia             :  23.16%
```

Figure 26: Per-Class accuracy for baseline model

```
CLASSIFICATION REPORT:
--------------------------------------------------------------
              precision    recall  f1-score   support

    Amphibia      0.306     0.198     0.241        96
    Animalia      0.247     0.455     0.320        99
   Arachnida      0.256     0.216     0.235        97
        Aves      0.418     0.489     0.451        94
       Fungi      0.339     0.183     0.238       109
     Insecta      0.379     0.191     0.254       115
    Mammalia      0.324     0.324     0.324       108
    Mollusca      0.215     0.267     0.238       120
     Plantae      0.360     0.471     0.408        85
    Reptilia      0.227     0.232     0.229        95

    accuracy                          0.297      1018
   macro avg      0.307     0.303     0.294      1018
weighted avg      0.306     0.297     0.290      1018
```

Figure 27: Classification Report for baseline model

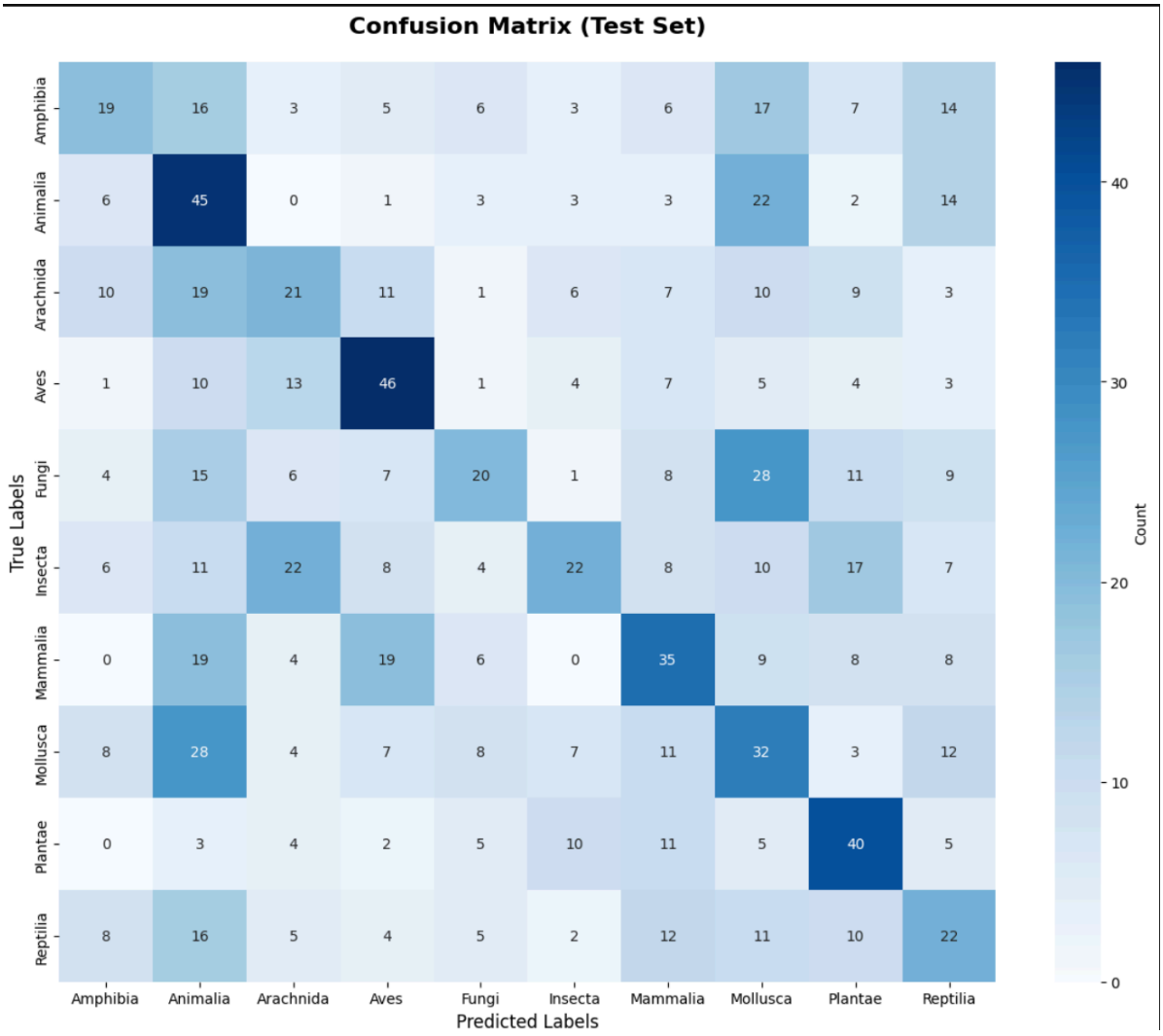

Figure 28: Confusion matrix for baseline model

b. EfficientNetV2_B0 with classical data augmentation

The EfficientNetV2_B0 model trained with classical data augmentation achieved a **Top-1 Accuracy of 86.35%** and an **Average Accuracy per Class of 86.42%**. This represents a big improvement over the baseline model, showing that the architecture of the pretrained model on the imageNet dataset is much better.

The close gap between the Top-1 Accuracy and Average Accuracy per Class illustrates that the model performed well across all classes, with minimal bias toward specific categories. This demonstrates that the well-pretrained model not only improved overall accuracy but also helped mitigate class imbalance and overfitting issues observed in the baseline.

```
============================================================
MODEL EVALUATION METRICS
============================================================
Top-1 Accuracy: 86.35%
  (How often the top predicted class matches the true label)

Average Accuracy per Class: 86.42%
  (Mean of individual class accuracies - handles class imbalance)
============================================================
```

Figure 29: Top-1 Accuracy and Average Accuracy per Class for EfficientNetV2_B0 with classical augmentation

The figures below provides more information about the performance of the EfficientNetV2_B0 model trained with classical data augmentation technique.

```
PER-CLASS ACCURACY:
------------------------------------------------------------
Amphibia             :  92.47%
Animalia             :  81.63%
Arachnida            :  89.69%
Aves                 :  93.62%
Fungi                :  87.50%
Insecta              :  85.84%
Mammalia             :  92.73%
Mollusca             :  79.13%
Plantae              :  76.60%
Reptilia             :  85.00%
```

Figure 30: Per-Class Accuracy for EfficientNetB2_V0 with classical augmentation

```
CLASSIFICATION REPORT:
------------------------------------------------------------
              precision    recall  f1-score   support

    Amphibia      0.748     0.925     0.827        93
    Animalia      0.800     0.816     0.808        98
   Arachnida      0.897     0.897     0.897        97
        Aves      0.936     0.936     0.936        94
       Fungi      0.919     0.875     0.897       104
     Insecta      0.874     0.858     0.866       113
    Mammalia      0.879     0.927     0.903       110
    Mollusca      0.798     0.791     0.795       115
     Plantae      0.911     0.766     0.832        94
    Reptilia      0.914     0.850     0.881       100

    accuracy                          0.863      1018
   macro avg      0.868     0.864     0.864      1018
weighted avg      0.867     0.863     0.864      1018
```

Figure 31: Classification Report for EfficientNetB2_V0 with classical augmentation

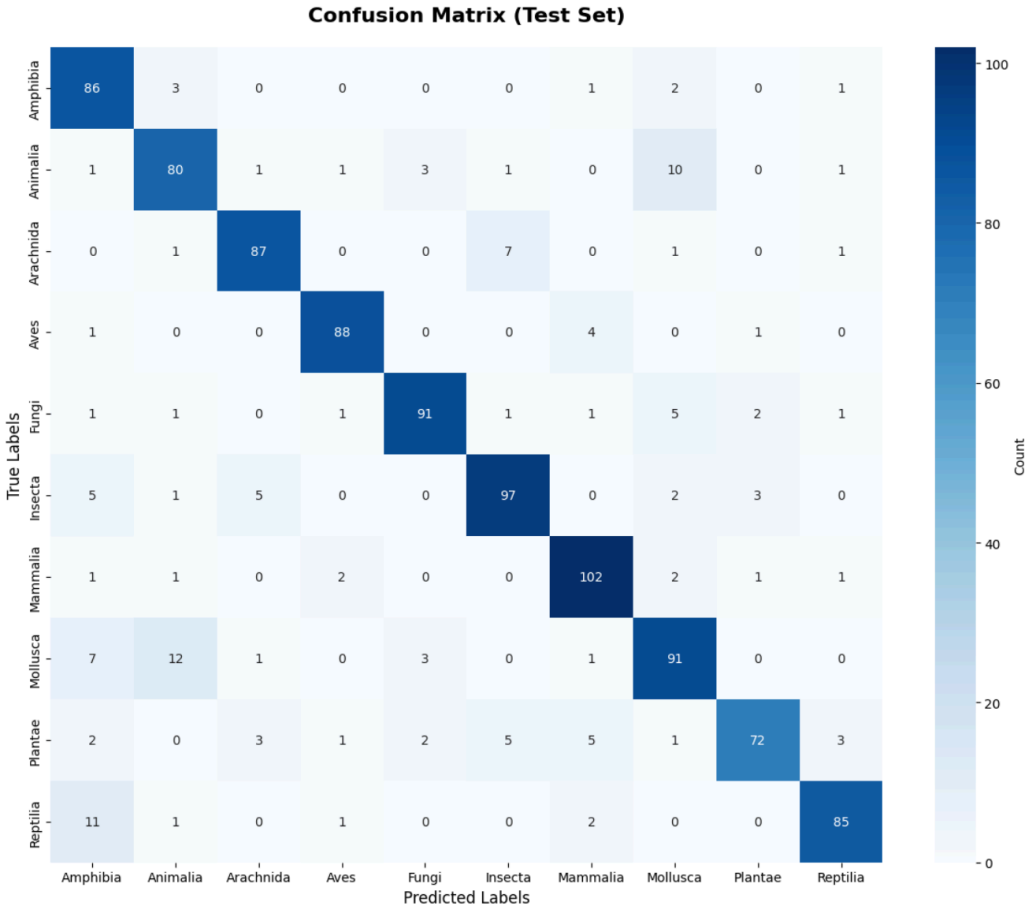

Figure 32: Confusion Matrix for EfficientNetB2_V0 with classical augmentation

c. EfficientNetV2_B0 with advanced dataset augmentation

The EfficientNetV2_B0 model trained with advanced data augmentation achieved a **Top-1 Accuracy of 89.00%** and an **Average Accuracy per Class of 89.12%**. Compared to the version trained with classical augmentation, this represents a relatively better performance improvement of nearly 3%, showing that the advanced augmentation techniques enhance the model generalization and robustness.

The close discrepancy between **Top-1 Accuracy** and **Average Accuracy per Class** also shows that the model performed well across all classes, suggesting balanced learning and small bias toward specific classesclasses. This further demonstrates that advanced augmentations not only improved overall accuracy but also contributed to good per-class performance.

```
============================================================
MODEL EVALUATION METRICS
============================================================
Top-1 Accuracy: 89.00%
  (How often the top predicted class matches the true label)

Average Accuracy per Class: 89.12%
  (Mean of individual class accuracies - handles class imbalance)
============================================================
```

Figure 33: Top-1 Accuracy and Average Accuracy per Class for EfficientNetV2_B0 with advanced augmentation

```
PER-CLASS ACCURACY:
------------------------------------------------------------
Amphibia             :  91.00%
Animalia             :  86.32%
Arachnida            :  90.53%
Aves                 :  95.35%
Fungi                :  93.33%
Insecta              :  86.55%
Mammalia             :  89.81%
Mollusca             :  86.32%
Plantae              :  85.87%
Reptilia             :  86.14%
```

Figure 34: Per-class accuracy for EfficientNetV2_B0 with advanced augmentation

```
CLASSIFICATION REPORT:
------------------------------------------------------------
              precision    recall  f1-score   support

    Amphibia      0.820     0.910     0.863       100
    Animalia      0.845     0.863     0.854        95
   Arachnida      0.945     0.905     0.925        95
        Aves      0.965     0.953     0.959        86
       Fungi      0.899     0.933     0.916       105
     Insecta      0.945     0.866     0.904       119
    Mammalia      0.915     0.898     0.907       108
    Mollusca      0.821     0.863     0.842       117
     Plantae      0.859     0.859     0.859        92
    Reptilia      0.916     0.861     0.888       101

    accuracy                          0.890      1018
   macro avg      0.893     0.891     0.891      1018
weighted avg      0.892     0.890     0.890      1018
```

Figure 35: Classification Report for EfficientNetV2_B0 with advanced augmentation

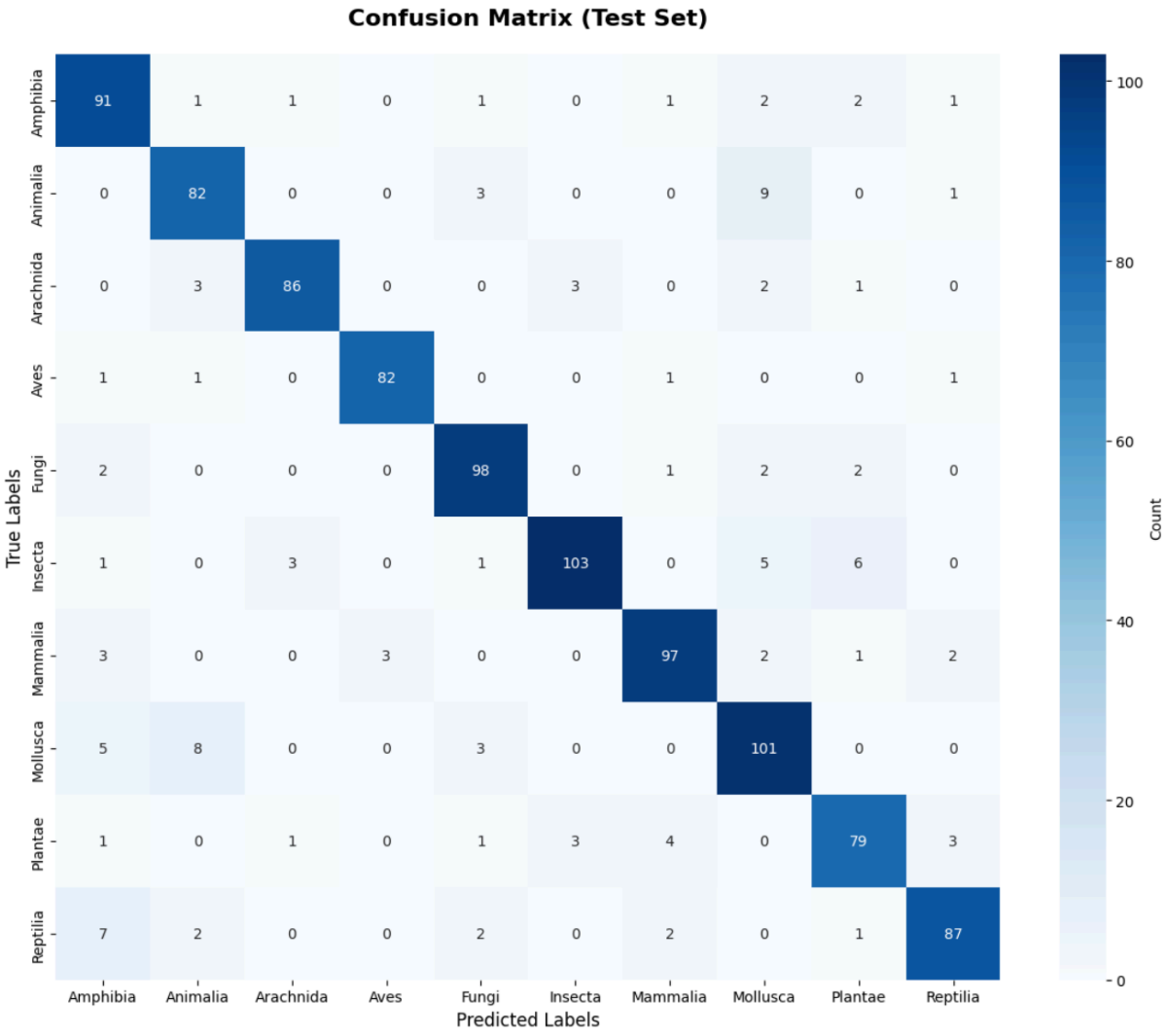

Figure 36: Confusion Matrix for EfficientNetV2B0 with advanced augmentation

d. ViT-224

The Vision Transformer (ViT-224) achieved a **Top-1 Accuracy of 90.77%** and an **Average Accuracy per Class of 90.75%**, demonstrating very good overall performance. This result is better the EfficientNetV2_B0 models, indicating that the transformer-based architecture effectively captured global contextual information through self-attention mechanisms. The small discrepancy between **Top-1** and **Average Accuracy** suggest that the model performs well across all classes.

```
============================================================
MODEL EVALUATION METRICS
============================================================
Top-1 Accuracy: 90.77%
  (How often the top predicted class matches the true label)

Average Accuracy per Class: 90.75%
  (Mean of individual class accuracies - handles class imbalance)
============================================================
```

Figure 37: Top-1 Accuracy and Average Accuracy per Class for ViT-224

```
PER-CLASS ACCURACY:
------------------------------------------------------------
Amphibia             :  89.25%
Animalia             :  86.00%
Arachnida            :  87.63%
Aves                 :  95.74%
Fungi                :  92.66%
Insecta              :  89.57%
Mammalia             :  97.17%
Mollusca             :  89.43%
Plantae              :  88.76%
Reptilia             :  91.30%
```

Figure 38: Per-Class Accuracy for ViT-224

```
CLASSIFICATION REPORT:
------------------------------------------------------------
              precision    recall  f1-score   support

    Amphibia      0.865     0.892     0.878        93
    Animalia      0.869     0.860     0.864       100
   Arachnida      0.955     0.876     0.914        97
        Aves      0.947     0.957     0.952        94
       Fungi      0.953     0.927     0.940       109
     Insecta      0.904     0.896     0.900       115
    Mammalia      0.928     0.972     0.949       106
    Mollusca      0.880     0.894     0.887       123
     Plantae      0.888     0.888     0.888        89
    Reptilia      0.894     0.913     0.903        92

    accuracy                          0.908      1018
   macro avg      0.908     0.908     0.908      1018
weighted avg      0.908     0.908     0.908      1018
```

Figure 39: Classification Report for ViT-224

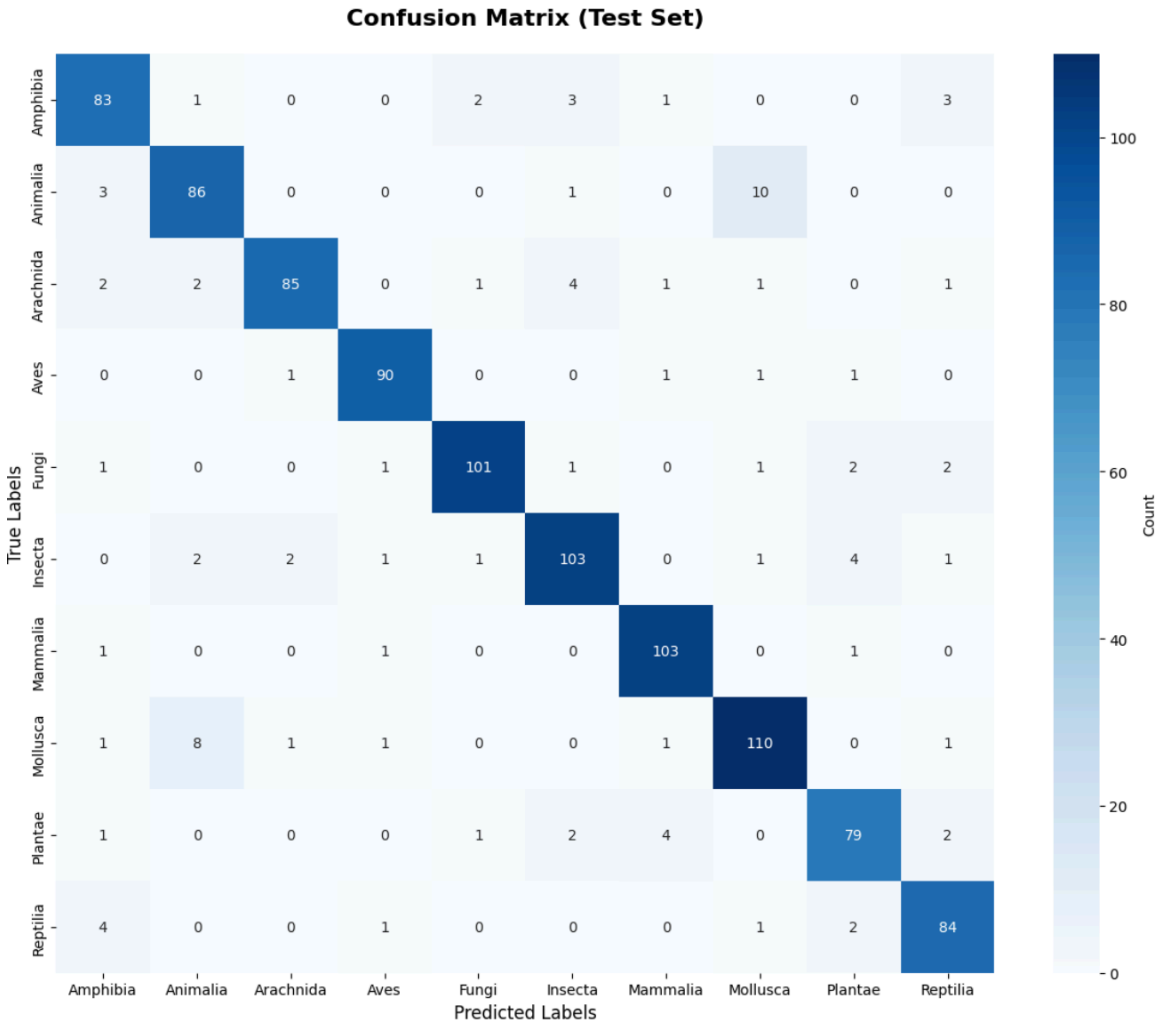

Figure 40: Confusion Matrix for ViT-224

e. ViT-384

The Vision Transformer (ViT-384) achieved a **Top-1 Accuracy of 94.40%** and an **Average Accuracy per Class of 94.45%**, making it the highest performance compared to other models.

This high improvement over ViT-224 (≈ 3.6%) and EfficientNetV2-B0 models demonstrates the effectiveness of using a higher input resolution (384×384), allowing the transformer to capture more details. Additionally, the insignificant difference between **Top-1** and **Average Accuracy per Class** again indicates balanced learning, showing that the model performs equally well across all categories without overfitting.

```
============================================================
MODEL EVALUATION METRICS
============================================================
Top-1 Accuracy: 94.40%
  (How often the top predicted class matches the true label)

Average Accuracy per Class: 94.45%
  (Mean of individual class accuracies - handles class imbalance)
============================================================
```

Figure 41: Top-1 Accuracy and Average Accuracy per Class for ViT-384

```
PER-CLASS ACCURACY:
------------------------------------------------------------
Amphibia             :  96.88%
Animalia             :  93.94%
Arachnida            :  93.00%
Aves                 :  96.59%
Fungi                :  96.26%
Insecta              :  95.08%
Mammalia             :  91.82%
Mollusca             :  90.27%
Plantae              :  92.68%
Reptilia             :  98.02%
```

Figure 42: Per-Class Accuracy of ViT-384

```
CLASSIFICATION REPORT:
------------------------------------------------------------
              precision    recall  f1-score   support

    Amphibia      0.930     0.969     0.949        96
    Animalia      0.921     0.939     0.930        99
   Arachnida      0.949     0.930     0.939       100
        Aves      0.966     0.966     0.966        88
       Fungi      0.972     0.963     0.967       107
     Insecta      0.975     0.951     0.963       122
    Mammalia      0.962     0.918     0.940       110
    Mollusca      0.919     0.903     0.911       113
     Plantae      0.927     0.927     0.927        82
    Reptilia      0.917     0.980     0.947       101

    accuracy                          0.944      1018
   macro avg      0.944     0.945     0.944      1018
weighted avg      0.944     0.944     0.944      1018
```

Figure 43: Classification Report of ViT-384

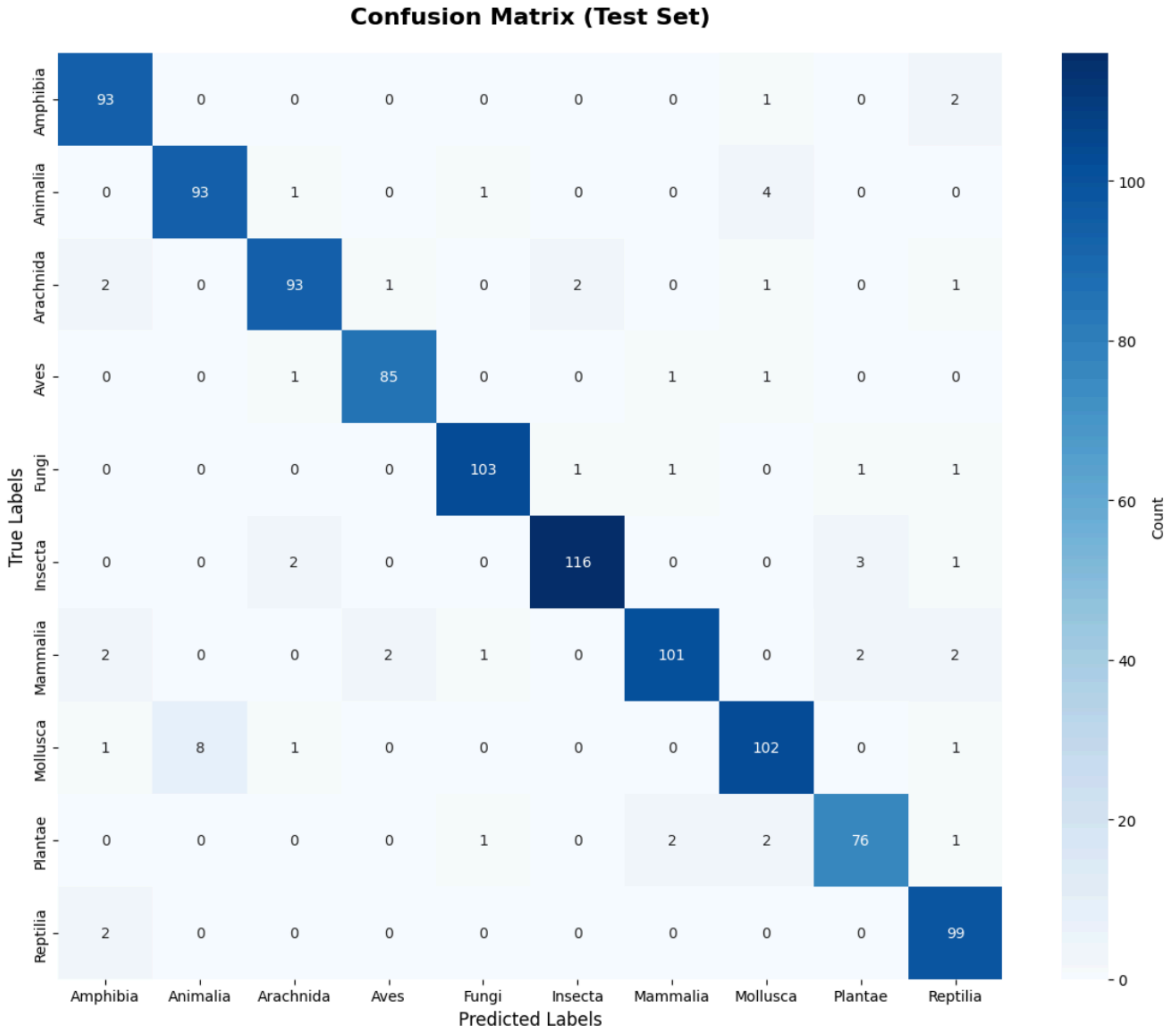

Figure 44: Confusion Matrix of ViT-384

4.3 Visualizations

Beside that, I also visualized the wrong predicted image along with confidence scores for better visualization. These images can also be used for error analysis.

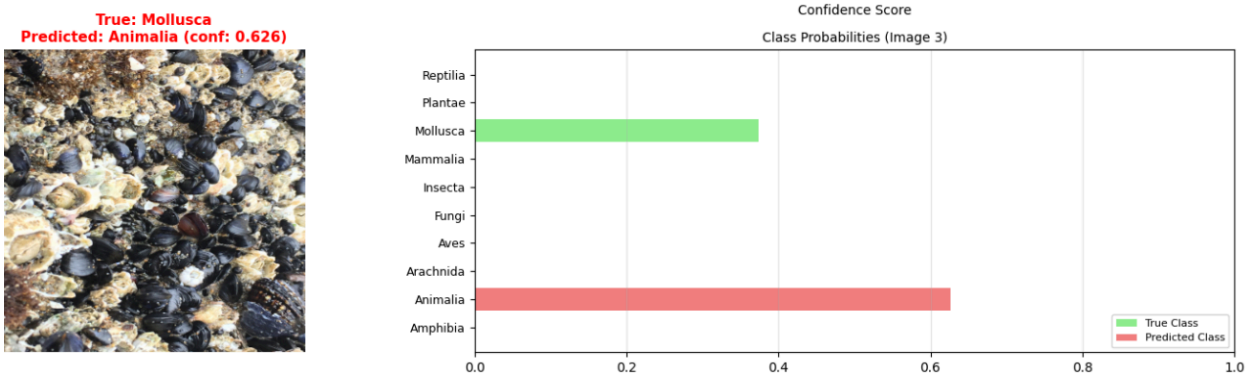

Figure 45: Example of wrong predicted image with confidence scores

4.4 Discussion

The results show clear performance improvements as models and data augmentations became more advanced. The baseline CNN achieved only 29.67% accuracy, indicating limited learning capacity without transfer learning. EfficientNetV2_B0 significantly improved to 86.35% with classical augmentations, and further to 89.00% using advanced augmentation, showing how richer data diversity helps generalization. The Vision Transformer models achieved the best performance — ViT-224 reached 90.77%, and ViT-384 achieved 94.40%, demonstrating that transformer architectures and higher input resolution capture global and fine-grained image features more effectively. Overall, combining transfer learning, strong augmentation, and transformer architectures resulted in high accuracy and balanced class performance.

5. Conclusion

In conclusion, this project clearly shows how powerful modern architectures and thoughtful data augmentation can be in image classification. Starting from a simple CNN with modest performance, each improvement — from EfficientNetV2_B0 to Vision Transformers — brought noticeable gains in accuracy and generalization. The final ViT-384 model demonstrated exceptional results, proving that combining transfer learning, advanced augmentations, and transformer-based vision models leads to more robust and reliable classifiers. Overall, this work highlights the importance of both model choice and data preparation in building high-quality deep learning systems.

6. References

[1] S. Kumar, P. Asiamah, O. Jolaoso, and U. Esiowu, “Enhancing Image Classification with Augmentation: Data Augmentation Techniques for Improved Image Classification,” arXiv.org, 2025. https://arxiv.org/abs/2502.18691

[2] “What is Top-1 Error Rate? Evaluating Model Accuracy,” *Deepchecks*, May 27, 2024. https://www.deepchecks.com/glossary/top-1-error-rate/ (accessed Oct. 25, 2025).

[3] GeeksforGeeks, “What is Data Augmentation? How Does Data Augmentation Work for Images?,” *GeeksforGeeks*, Jul. 22, 2024. https://www.geeksforgeeks.org/computer-vision/what-is-data-augmentation-how-does-data-augmentation-work-for-images/